EPOCH

A History of Computing

STÉPHANE FOSSE

EPOCH

A history of computing

Stéphane FOSSE

January 2026

ISBN: 978-29-59208-97-3
ISBN (PDF): 978-29-59208-98-0
ISBN (EPUB): 978-29-59208-99-7

Copyright © 2026 - Stéphane Fosse
Auto-édition - https://fosse.fr
Première édition: Janvier 2026
Dépôt légal: Janvier 2026

English translation produced with the assistance of artificial intelligence and revised by the author.

This work is licensed under the Creative Commons Attribution Share-Alike 4.0 International License. You can view a copy of this license at the end, or at https://creativecommons.org/licenses/by-sa/4.0/deed.en

You are free to share and adapt this book provided you credit me and redistribute your modifications under the same license (CC BY-SA 4.0).

Humans are allergic to change. They love to say, "We've always done it this way." I try to fight that. That's why I have a clock on my wall that runs counterclockwise.

Grace Hopper

Contents

Foreword **xvii**

Before 1930 **1**

Binary System . . . 4
Punch Card . . . 6
Charles Babbage . . . 7
Boolean Algebra . . . 9
Submarine Cable . . . 11
Herman Hollerith . . . 13

1930 **15**

Vannevar Bush's Differential Analyzer . . . 17
Gödel's Incompleteness Theorem . . . 18
The Turing Machine . . . 20
The Z1 Mechanical Computer . . . 22
Lambda Calculus . . . 23
Shannon's Logic Circuit . . . 25
The Complex Number Calculator . . . 26

1940 **29**

Atanasoff-Berry Computer . . . 31
Artificial Neural Networks . . . 33
Colossus . . . 36
Harvard Mark I . . . 37
ENIAC . . . 38
Delay Line Memory . . . 41
Monte Carlo Method . . . 42
Williams Tube . . . 44
Association for Computing Machinery . . . 46

Transistor 47
Cybernetics 49
Magnetic Drum 51
Claude Shannon's Information Theory 53
Short Code 54
EDSAC 55
Application Programming Interface 57
Whirlwind I 59
1950 **61**
Hamming Code 64
Magnetic Core Memory 66
Magnetic Tape 67
Ferranti Mark I 68
LEO I 70
UNIVAC I 72
A-0 Compiler 74
IBM 701 76
IBM 702 78
Assembly Language 80
IBM 650 82
IBM 704 83
Dartmouth Conference 85
IBM 350 RAMAC 87
IPL 89
TX-0 91
FORTRAN 93
Integrated Circuit 95
ALGOL 96
Lisp 98
DEC PDP-1 100
IBM 1401 102
COBOL 104
Machine Learning 106
Backus-Naur Form 108

1960 **111**
Quicksort . . . 113
Bull Gamma 60 . . . 115
APL . . . 116
Simula . . . 118
SNOBOL . . . 120
Spacewar! . . . 122
ASCII . . . 124
Bell 103 Modem . . . 126
Music V . . . 128
Sketchpad . . . 129
GECOS . . . 131
IBM System/360 . . . 133
BASIC . . . 135
PL/I . . . 137
IBM CP-40 . . . 139
Electronic Mail . . . 141
Multics . . . 143
DEC PDP-8 . . . 145
CDC 6600 . . . 146
ELIZA . . . 148
Information Management System . . . 150
LOGO . . . 152
Read Only Memory . . . 154
NLS . . . 155
ARPANET . . . 157
B . . . 159
The First RFC . . . 161
RAM . . . 162
UNIX . . . 164
1970 **167**
ALOHAnet . . . 170
FORTH . . . 172
Pascal . . . 174
DEC PDP-11 . . . 175
B-tree . . . 177

Floppy Disk 179
ed 181
File Transfer Protocol 183
Intel 4004 185
Intel 8008 187
C 189
Prolog 191
Smalltalk 194
IBM VM/370 196
Ethernet 197
TCP/IP 199
Telnet 201
The SQL Language 203
System R. 205
Intel 8080 207
CP/M 209
Diffie-Hellman 211
Altair 8800 213
X.25 215
Cray-1 217
UUCP 219
Apple I 221
vi 223
DEC VAX-11 225
RSA 227
Apple II 229
Commodore P.E.T 230
Tandy TRS-80 232
Berkeley Software Distribution 235
Space Invaders 237
TEX 238
WordStar 240
dBASE 242
Ada. 244
Usenet. 246
VisiCalc 248

1980 **253**

Apple III 255
Intel 8086/8088 258
Smalltalk-80 260
UDP 261
Commodore VIC-20 263
WordPerfect 266
ICMP 268
Acorn BBC Micro 269
IBM PC-5150 272
MS-DOS 273
Osborne 1 275
LZW 277
AutoCAD 279
CD-ROM 281
Intel 80286 283
SMTP 285
Commodore 64 287
Sinclair ZX Spectrum 289
Thomson TO/MO 291
PostScript 293
SunOS 295
Agat 297
Apple Lisa 299
Coleco Adam 301
Compaq Portable 304
GNU Project 306
DNS 308
C++ 310
Objective-C 312
IPX/SPX 314
SMB 316
Lotus 1-2-3 318
Microsoft Word 320
MSX 321
Novell NetWare 324

Oric-1 326
Amstrad CPC 464 328
Apple Macintosh 329
HP-UX 331
NFS 333
POP3 335
IBM PCjr 338
GNU Emacs 340
Microsoft Paint 341
X Window System 343
Intel 80386 345
AppleTalk 347
NTP 349
Microsoft Excel 352
Microsoft Windows 353
Amstrad PCW. 355
Atari 520 ST 357
Toshiba T1100. 359
Aldus PageMaker. 360
IBM AIX. 362
Eiffel 364
Erlang. 366
IMAP 368
NSFNet 369
SCSI 371
Bash 373
GIF. 375
Perl. 377
Self 379
Tcl 381
Acorn Archimedes 383
Apple Macintosh II 385
Commodore Amiga 500 387
Sharp X68000 389
OS/2 390
Unicode 392

FrameMaker 394
IBM AS/400 396
Internet Relay Chat 397
SNMP 400
ZIP 402
Intel 80486 403
Intel i960 405
World Wide Web 407
Microsoft Office 409
NeXTSTEP 411
1990 **413**
Adobe Photoshop 415
IBM PS/1 417
Haskell 419
HTML 421
Gopher 423
PCMCIA 424
MP3 426
WAVE 429
Python 431
Linux 433
Visual Basic 435
DEC OSF/1 437
Blowfish 439
JPEG 441
IBM ThinkPad 444
OpenGL 447
IBM Simon 448
Adobe PDF 450
Common Gateway Interface 452
FreeBSD 454
Intel Pentium 456
Lua 458
R 460
DHCP 462
NCSA Mosaic 463

Secure Socket Layer 465
Microsoft Windows NT 3.51 467
Apple Power Macintosh 469
CD-RW 471
IBM Aptiva 472
Netscape Navigator 474
World Wide Web Consortium 476
Yahoo! 478
AltaVista 479
Apache Web Server 481
BeOS 483
DVD 484
GPU 486
Microsoft Internet Explorer 488
Java 489
JavaScript 491
PHP 493
Ruby 495
SSH 496
Lynx 498
MySQL 500
RealPlayer 502
Sun UltraSPARC 504
Wiki 505
Microsoft Windows 95 507
Microsoft DirectX 509
ICQ 510
KDE 512
Cascading Style Sheets 513
Flash 515
OpenBSD 516
PostgreSQL 518
USB 520
VPN 522
Hotmail 524
Internet Archive 525

Wi-Fi 527
Google 529
VMware 530
Microsoft Windows 98 532
XML 534
GNOME 536
Bluetooth 538
MSN Messenger 539
RSS 541
SOAP 543
Transport Layer Security 544
Napster 546
Year 2000 Bug 548
2000 **551**
C# 553
FLAC 555
JSON 557
BitTorrent 559
Apple Mac OS X 560
SHA-256 562
Microsoft .NET Framework 564
Mozilla Firefox. 566
Apple Xserve 567
Tor 569
Apple iTunes 571
Skype 572
WordPress 574
Xen. 576
Apache Cassandra 578
Google Gmail 581
Scala 582
NGINX 584
OAuth. 586
Git 587
Redis 589
YouTube 590

Amazon Web Services 592
Hadoop 594
jQuery. 596
PowerShell 598
MongoDB 599
Apple iPhone 601
Clojure 603
F# 605
Groovy 607
Android 608
Google Chrome 610
HTML5 612
Memcached 614
OpenStack 615
Bitcoin 617
CouchDB. 619
Node.js 621
Raspberry Pi 623
2010 **625**
Elasticsearch 628
HEIF 630
Rust 632
ChromeOS 633
NVMe 635
Dart 637
Kotlin 639
WebRTC 641
WebSocket 643
Go 645
Julia 647
TypeScript 649
Docker 650
SHA-3 653
Spark 655
Kubernetes 657
Swift 659

Kafka 661
GraphQL. 662
TensorFlow 664
Accelerated Mobile Pages 667
Natural Language Processing 668
WebAssembly 670
WebAuthn 671
2020 **675**
Apple M1 678
Zig 680
Mojo 682
WebGPU. 684
Epilogue **687**
License **729**

Foreword

My first encounter with a computer dates back to an ORIC-1 in 1983. Shortly after came an Apple IIc, during the very first computer literacy classes in primary school. I was fortunate. LOGO and BASIC were my first windows into programming. Back then, we bought books filled with source code that had to be copied line by line, saved on cassette, then debugged: `syntax error`. Buying compiled programs on cassette would have cost a fortune. With all these new words, I also remember wondering what the difference was between software (logiciel) and Loriciel, that French company that published video games founded in 1983. Everything was confusing.

Decades later, having become an information systems architect, I manipulate technologies daily whose origins I often knew nothing about. I knew how they worked, how to configure and assemble them. But the original context of their creation remained in the shadows. So I started asking myself the same types of questions I ask daily in my work. What problems were those who invented them facing? What needs or frustrations drove them to seek new solutions? These questions remained unanswered.

So I began reading and compiling articles, official manufacturer documents, scripts from international conferences, ISO standards, and RFCs. More than 3,750 documents scrutinized. Gradually, the volume took shape. Enough to fill a book. This compilation was no longer just a personal curiosity but had become a project of transmission.

On the internet, 38% of web pages that existed in 2013 are now inaccessible. Information is crumbling away. Recording this knowledge in a book, under a free license no less, represents for me a way to preserve what risks disappearing. *Copyleft* seemed the most appropriate response: knowledge must circulate, be shared, and grow richer through readings

and reuses.

I therefore chose to approach each technology from the angle of the problem to be solved. By addressing the context that leads to innovation, we can show that the history of computing is not a succession of abstract technical feats. It is a human story, made of concrete constraints and often pragmatic responses.

The book is built around 350 technologies, presented chronologically in the form of chronicles. Each can be read independently of the others, maintaining its unity so that readers can navigate as they wish. The idea was to make the work usable as a resource that one consults according to their interests of the moment, or simply in order by following the timeline.

This second edition is enriched with more than 250 technologies. The choices inevitably reflect my own sensibilities and my journey. Telling everything would be utopian. It was necessary to make cuts, select, give up, and there are still so many things to tell. Technologies that appeared after 2010 are too recent in my opinion for us to truly judge their historical weight. Perspective is lacking. This is why they are less represented in these pages.

This book addresses several audiences. First, enthusiasts of the history of science and technology will find material to explore. Innovation professionals, consultants, those working in research and development will see patterns of evolution that repeat themselves. Teachers and trainers in computer science will be able to draw on it as a resource for their courses. Students in engineering schools or at university will have structured learning material. Intellectually curious people attracted by tales of discoveries and inventors will not be left out. And investors or entrepreneurs in the technology sector may discover the innovation cycles that foreshadow coming disruptions.

EPOCH, in the computing sense of the term, designates a starting point from which time begins to be counted. This book aims to be a landmark in understanding our technological heritage. A landmark among others, certainly, but a landmark that establishes a history.

I

Before 1930

The Dawn of Computing

The period from 1700 to 1930 radically transformed our humanity. These two centuries, marked by industrialization and its social upheavals, laid the foundations for the computing that was about to emerge.

The 18$^{\text{th}}$ century saw Europe ignited by an unprecedented intellectual fervor. The Enlightenment imposed a worldview where science and reason took the throne. Inventors, driven by frenzied curiosity, sought to tame the universe through numbers and mechanisms. Scientific academies multiplied like buzzing hives of new ideas, weaving their networks across the old continent.

Around 1760, Great Britain was shaken by the first convulsions of the industrial revolution. Nascent factories swallowed workers whole and machines gradually replaced human hands. Industrialists, confronted with the growing complexity of their enterprises, demanded precise tools to juggle their figures. This thirst for arithmetic precision pushed inventors to devise mechanical solutions to the most laborious calculations.

The end of the century witnessed the emergence of the first mechanized devices in textile mills. Joseph-Marie Jacquard transformed weaving in 1801 with his punched cards that dictated patterns for the looms to reproduce. These seemingly innocuous perforated cardboard rectangles carried

within them the seed of future computer systems. Each hole, each solid space represented binary information that the machine interpreted without error, a striking prefiguration of digital memories.

The 19th century opened in a whirlwind of change. Empires stretched their tentacles to the four corners of the globe, trade intensified, creating a visceral need for rapid communications. The railway, that steel monster devouring space, imposed clock synchronization and rigorous schedule management. Modern states swelled their administrations, which soon drowned beneath mountains of paperwork.

George Boole, an English mathematician with an incisive mind, published his theory of logical algebra in 1847. Though seemingly of interest only to a few eccentric scholars, his work contained the key that would one day unlock the doors of computing. His binary system, reducing every proposition to true or false, patiently awaited visionary minds to seize upon it.

The electric telegraph, copper wires stretched from one continent to another, shattered the barriers of time and space after 1840. What had taken months now required only minutes. Samuel Morse imposed his code of dots and dashes, the first standardization of binary language. Submarine cables, copper serpents laid on the ocean floor, connected Europe to America as early as 1866, creating the first global network of instantaneous communication.

The adoption of universal time in 1884 at the International Meridian Conference in Washington perfectly illustrated how technology reshaped our perception of the world. The 24 time zones imposed an unprecedented temporal discipline on humanity. The entire world set its watches to Greenwich, subject to the gentle tyranny of synchronized clocks.

The final decades of the 19th century saw companies grow into bureaucratic behemoths. Banks, insurance companies, and department stores collapsed beneath registers and files. The 1890 American census illustrated this problem: the federal administration took seven years to process the data from the previous census. Herman Hollerith, a brilliant young engineer, designed his electromagnetic tabulating machine. Thanks to it, the results of the 1890 census were available in only six weeks. His Tabulating Machine Company, founded in 1896, would later become IBM.

The Great War radically transformed the relationship with information technologies. In the shadow of the trenches, mathematicians and linguists broke enemy codes. In France, Lieutenant Georges Painvin deciphered the German ADFGVX code in 1918, thwarting the Spring Offensive and perhaps saving Paris. Ballistic calculations, necessary for long-range artillery like the German Paris-Geschütz, demanded unprecedented precision. The

war had shown that victory belonged to those who mastered information.

The interwar period saw the birth of new work organization methods. Frederick Taylor dissected workers' movements with a stopwatch while Henry Ford imposed his assembly line. The office followed the factory in this race toward rationalization. Burroughs accounting machines, National cash registers, Monroe calculators invaded administrations. The noise of keys, the chime of totals, the sliding of carriages now punctuated the daily life of white-collar workers.

The 1920s saw the explosion of telecommunications. Wireless telegraphy (TSF) carried voices and music into every home. The BBC (British Broadcasting Corporation) broadcast its first regular programs in 1922, followed by numerous stations around the world. The telephone, Graham Bell's 1876 invention, left behind its status as a gadget for the wealthy and began its conquest of households. Switchboard operators, predominantly women, became the first human interfaces between machines and their users, a striking metaphor for future human-machine relationships.

Social transformations accompanied these technological developments and mass literacy changed the relationship with knowledge. In France, the Jules Ferry laws of 1881-1882 imposed compulsory education. In the United States, the literacy rate exceeded 80% as early as 1870. This educated population demanded ever more books, newspapers, and information. Knowledge became a commodity that now circulated at the speed of electricity.

Galloping urbanization created unprecedented organizational challenges. Paris grew from 500,000 to 3 million inhabitants between 1800 and 1930. New York exploded from 60,000 to 7 million souls during this period. These sprawling metropolises demanded colossal infrastructure. Baron Haussmann redesigned Paris, imposing his straight avenues on the old medieval quarters. In New York, engineer John Augustus Roebling threw his suspension bridge between Manhattan and Brooklyn in 1883. These technical feats required diabolically precise calculations and methodical resource management.

The world of work metamorphosed under these intersecting influences. The tertiary sector swelled to dominate the American economy by the 1920s. Information workers (accountants, stenographers and operators) formed a new social class. Women flooded into these professions, altering gender relations in society. The typist, bent over her Remington, became an emblematic figure of this bureaucratic modernity.

In 1929, on the eve of the stock market crash that would shake the global economy, computing remained officially unborn. Yet all the ingredients for its future explosion were assembled. The need for calcula-

tion intensified in every field. The necessary mathematical theories had matured. Nascent electronics promised extraordinary performance. The ground was ready for the advent of programmable calculators that would soon transform the world more radically than the steam engine had done a century earlier.

Binary System

The historical journey of the binary system reveals how this mathematical idea evolved into the foundation of modern computing. Thomas Harriot, an English mathematician, sketched around 1600 the first outlines of a numerical representation based solely on 0 and 1. His research on combinations led him to observe that any number could be expressed as a sum of powers of 2. These works, never published, remained in obscurity until the 1920s.

A century later, Gottfried Wilhelm Leibniz rediscovered this system by chance. In the 1670s, while working intensively on number division and prime numbers, binary notation appeared to him as an elegant solution for certain mathematical calculations. Initially, he used it simply to illustrate his theorems without imagining its practical use. In his 1697 letter to the Duke of Brunswick, Leibniz proposed creating a commemorative medal and developed a theological reading of binary where 1 symbolizes being and 0 represents nothingness. This medal never came to fruition, but the letter launched a series of publications on this system.

In 1703, in the *Mémoires de l'Académie Royale des Sciences*, Leibniz published his "Explanation of Binary Arithmetic". This text outlined the principles of the system and demonstrated how to perform basic operations while acknowledging the cumbersomeness of these long sequences of 0s and 1s for everyday use.

The 18^{th} century saw mathematicians like Jean Bernoulli and Leonard Euler take up the binary system. Euler notably used it to study the properties of numbers of the form $2^n + 1$. Other researchers established connections between this system and other number bases such as octal and hexadecimal.

The practical application of binary emerged in the 19^{th} century with the advent of telecommunications. Morse's telegraph, patented in 1837, relied on two electrical states, presence or absence of current, to transmit information. Its code combined dots and dashes, foreshadowing the future use of binary in digital communications. Émile Baudot took this a

step further in 1870 with his explicitly binary telegraphic code. In 1901, Charles Bouton demonstrated the usefulness of binary for analyzing the game of Nim, paving the way for its application in game theory.

The revolution came with 20$^{\text{th}}$-century electronics. The Eccles-Jordan flip-flop circuit, created in 1919, gave physical existence to the 0 and 1 states. This crucial invention made the first binary memory circuits possible. In 1937, Claude Shannon demonstrated in his thesis the connection between Boolean algebra and the binary system for designing electronic switching circuits.

Early computers like ENIAC (1946) still operated in decimal, with ten vacuum tubes per digit. The 1947 Burks-Goldstine-von Neumann report changed the game by demonstrating binary's advantages: simpler circuits, increased reliability, and natural harmony with logical operations. This recommendation influenced all subsequent computers.

The 1950s and 1960s saw the rise of electronic memories, strengthening binary's dominance. Magnetic core memories and later semiconductor memories naturally relied on two distinct states. Some manufacturers attempted to explore four- or sixteen-level memories in the 1970s and 1980s; Intel experimented with four-level ROMs in certain processors, but these attempts remained marginal compared to the robustness of binary storage.

Binary extended beyond computer hardware. Digital telecommunications adopted it for its noise resistance and signal regeneration capability. Optical media such as CD-ROMs and DVD-ROMs also used it. Its success stems from its conceptual simplicity, which makes circuits more reliable, its compatibility with Boolean algebra, which makes it the natural language of logical operations, and its robustness against disturbances, explained by the distinction of only two states.

Research on other numerical systems has not disappeared, however. The ternary system (with three states) was the subject of extensive studies in the Soviet Union in the 1950s. More recently, quantum computing has opened new horizons with its qubits capable of existing in a superposition of states. Despite these alternatives, the binary system remains the soul of classical computing. Its ability to encode any information using only two symbols gives it remarkable universality.

Punch Card

In 1801, Joseph-Marie Jacquard, a Lyon-based weaver, was searching for a way to automate his looms. He invented perforated cardboard cards to encode weaving patterns. Without knowing it, he had just laid the first stone of a major idea. These cards were not intended for calculations, but they already embodied the powerful concept of transforming an instruction into holes readable by a machine.

It was almost a century later, in 1890, that the U.S. Census Bureau faced a colossal problem. The population was growing at a breakneck pace and the previous census had stretched over eight years. A statistician named Herman Hollerith then proposed the bold solution of adapting Jacquard's principle to counting people. His system reduced processing time to just two and a half years. A staggering gain that did not go unnoticed at the time. The success was such that Hollerith founded his own company in 1896, the Tabulating Machine Company. It would later merge with others to give birth to a giant we all know: IBM.

At the beginning of the 20^{th} century, the punch card became established in the administrative landscape of large organizations. No more manual bookkeeping! Treasurers and accountants marveled at these machines capable of sorting, counting and adding in record time. Inventory management, payroll calculations, everything went through them.

Hollerith's first cards had only 45 columns, a dimension deemed insufficient, quickly. In 1928 at IBM, Clair D. Lake replaced circular holes with rectangular ones, thus making it possible to go from 45 to 80 columns on the same surface. An innovation that would leave a lasting mark on computing.

This 80-column standard left a surprising imprint on our current digital world. When IBM designed its first terminals in the 1960s, such as the famous 3270, each line displayed exactly 80 characters. A choice dictated by compatibility with existing equipment. This technical constraint propagated like a shock wave through the decades. Even today, many code editors limit lines to 80 characters by default, or at least offer it as an option. Terminal emulators perpetuate this tradition. Programming best practice guides recommend this limit to improve readability. The shadow of the punch card still looms over our screens.

Between the two world wars, a true mechanographic data processing industry developed. IBM dominated in the United States, while in Europe, companies like Powers (future Remington Rand) or Bull in France attempted to carve out a market share. Their business model relied on renting machines rather than selling them, ensuring stable revenue and

strict control over this strategic technology.

This new technique created new professions. Mechanographic rooms came alive under the nimble fingers of keypunch operators, mostly women, who transformed raw data into punch cards with the precision of goldsmiths. Others verified each card, while operators adjusted sorters and tabulators. These departments became the beating heart of large organizations.

The heyday of the punch card came in the 1950s-1960s. After the arrival of the first electronic computers, it remained the preferred medium for data input and storage. IBM's first models, such as the 650 or the 1401, were designed to integrate with existing punch card installations, gently accompanying the transition to the electronic era.

The history of this technology also includes its dark chapters. During World War II, punch card systems were used for census and population control in occupied Europe. Deutsche Hollerith-Maschinen Gesellschaft, IBM's German subsidiary, supplied machines used to organize deportations. A chilling reminder that any technology can also be used for terrible purposes.

The decline began in the 1970s. The arrival of interactive terminals and magnetic media gradually spelled the end for these cardboard rectangles. The punch card established the principle of binary information storage (hole or no hole, 1 or 0), clearly separated the data medium from the processing machine, and demonstrated the importance of standardization. The organizational methods developed around it influenced the design of early computers and structured the organization of work in computing centers.

Charles Babbage

Young Charles Babbage was born into a wealthy London household in 1791. His education at Cambridge, where he received rigorous mathematical training, made him aware of the limitations of manual calculation. With some classmates, he founded the Analytical Society to modernize mathematics education at the university. This first experience already reflected his desire to renew established methods.

One evening in 1821, Charles was discussing with his friend John Herschel. Both were exhausting themselves verifying mathematical tables riddled with errors. "Imagine these calculations performed by a steam

engine", Babbage suggested. This remark, tossed into the conversation, marked the beginning of an extraordinary intellectual adventure.

Babbage set to work. His first creation, the difference engine, was designed to automatically calculate polynomials using the method of finite differences. The 1822 prototype worked well enough to convince the British government to fund a more ambitious version. A real tour de force since the device not only performed operations but also printed the results, avoiding any transcription errors.

The cogwheels meshed together in a complex mechanical ballet. But London artisans, despite their expertise, could not manufacture parts with the required precision. The project became bogged down. Babbage, far from being discouraged, turned to an even bolder idea.

In 1834, his analytical engine took shape on paper. This new machine no longer followed a fixed program but could be reprogrammed for different calculations. Its architecture bore a striking resemblance to 20^{th}-century computers: a computing unit (the mill), memory (the store), and input and output devices.

Fascinated by Jacquard looms, Babbage borrowed the principle of punched cards to control his machine. These cards defined both the data and the instructions to be executed. The analytical engine incorporated everything that constitutes the essence of modern computing, namely the stored program, conditional loops, and subroutines. It compared numbers, made decisions, and executed the four basic arithmetic operations.

His meeting with Ada Lovelace gave a new dimension to the project. Daughter of Lord Byron and a brilliant mathematician, Ada translated an Italian article describing the machine. Her notes added to the translation detailed an algorithm for calculating Bernoulli numbers. This text remains etched in history as the first computer program ever written.

The analytical engine would never progress beyond the stage of plans and diagrams. Victorian England, despite its industrial might, could not manufacture a mechanism of such complexity and precision. The project's cost frightened public financiers. Yet until his death in 1871, Babbage never stopped refining his drawings, obsessed with his vision.

A hundred years later, Howard Aiken created the Harvard Mark I, drawing direct inspiration from Babbage's work. The designers of ENIAC and the first electronic computers adopted his architectural principles without always knowing their origin. Babbage had understood everything: the separation between hardware and software, the notion of debugging, code reuse. He had also anticipated the questions of reliability and performance that still haunt computing today.

In 1991, the Science Museum of London finally built a working version

of his difference engine No. 2. It functioned perfectly, proving the accuracy of Babbage's calculations and the coherence of his mechanical design. What we rediscover is not merely an inventor, but a multifaceted thinker.

His writings on industrial economics, his critiques of the British university system, his interest in applying mathematics to industry reveal a mind ahead of its time. Between the mechanical calculators of the 17^{th} century and electronic computers, Charles Babbage's work traced a visionary path, too often overlooked.

Boolean Algebra

At the beginning of the 19^{th} century, the stagnation of British mathematics contrasted sharply with continental ferment. A dispute between Newton's and Leibniz's disciples had paralyzed work across the Channel. While Europe calculated using Leibnizian differential notation, the British clung to Newtonian fluxions, which were less practical and less fruitful.

Around 1810, a few innovative minds founded the Analytical Society at Cambridge. Babbage and his colleagues broke the intellectual isolation by importing methods from the continent. This called into question the very foundations of algebra: what did negative numbers and imaginary quantities truly mean? An answer emerged in symbolic algebra. It no longer derived its legitimacy from operations on numbers, but established formal laws applicable to any symbols whatsoever.

It was in this intellectual atmosphere that George Boole forged his work. Born in 1815 in Lincoln into modest circumstances, never a university graduate, he taught himself mathematics with astonishing rigor. Initially a schoolmaster, he became professor at Queen's College Cork in 1849. A controversy between Augustus De Morgan and William Hamilton on quantification of the predicate prompted him to publish *The Mathematical Analysis of Logic* in 1847. His insight was to apply algebra to logical reasoning.

Boole transformed logic into a formal system with his algebraic notation. Literal symbols like x or y represented classes of objects, combinable through operators ($+$, $\times$). These operations followed precise rules: commutativity ($xy = yx$), distributivity ($x(u+v) = xu+xv$), and this singular property: $x^2 = x$. This last rule marked a break with ordinary numerical algebra, where only 0 and 1 satisfy it.

His system translated logical propositions into equations. Thus, all

X are Y became $x(1-y)=0$. The use of symbol 1 for the universe of discourse and 0 for the empty class constituted another discovery.

Augustus De Morgan, Boole's friend and contemporary, was developing his own logical theories in parallel. His *Formal Logic* appeared on the same day as Boole's work in 1847. De Morgan introduced the notion of variable universe of discourse, breaking with the fixed Aristotelian universe. He also formulated the famous laws that bear his name: the negation of a conjunction equals the disjunction of negations, and vice versa.

Boolean notation suffered from limitations. Boole required that addition apply only to disjoint classes, complicating the expression of certain relations. His definition of subtraction required that the subtracted class be included in the initial one.

In subsequent decades, mathematicians refined this system. Charles Sanders Peirce made decisive contributions in the 1880s, notably demonstrating that all Boolean operations could be reduced to a single one: NAND (not-and) or NOR (not-or).

Modern notation took shape gradually. Bertrand Russell introduced the symbol $\vee$ (or) in 1906, while Arend Heyting proposed $\wedge$ (and) in 1930. The expression "Boolean algebra" was first used by Henry Maurice Sheffer in 1913.

In 1936, Marshall Harvey Stone took a step toward abstraction by unifying earlier work under the concept of Boolean ring. He established the isomorphism between Boolean algebra and this structure, creating a fundamental theoretical bridge.

Practical applications of Boolean algebra exploded in the 20^{th} century. In 1938, Claude Shannon demonstrated in his master's thesis at MIT that these principles enabled the analysis and design of electrical switching circuits. This discovery created a link between mathematical logic and electronic design, founding modern computer science.

Today, Boolean algebra permeates all of computer science. It structures the design of logic circuits, formal verification of programs, and query optimization in databases. Every microprocessor relies on optimization techniques derived from this theory.

The history of Boolean algebra shows how an abstract theory, born from questions about logic, transforms an entire technological domain. It underscores the importance of mathematical notations in which an adapted symbolism reveals previously invisible relations. George Boole dreamed of a "calculus of thought" mechanizing logical reasoning. While this philosophical project did not succeed as such, his mathematical tools became the secret language of the machines that shape our world.

Submarine Cable

In the mid-XIXth century, crossing the Atlantic required at least ten days at sea. In 1840, Samuel Morse launched an idea that seemed like fantasy : connecting Europe to America with a telegraph cable at the bottom of the ocean. This almost delirious vision would transform our relationship with time and space.

A businessman named Cyrus W. Field seized on this ambition. After building his fortune in the paper industry, he took an interest in telegraphy during the 1850s. His first project, modest in scope, aimed to connect St. John's, Newfoundland to New York, taking over from engineer Frederic Newton Gisborne's work. Funding ran short. No matter, Field thought bigger and redirected everything toward a transatlantic link.

The Atlantic Telegraph Company was born in 1856 on the initiative of Field and engineers John Watkins Brett and Charles Tilston Bright, two Britons experienced in submarine telecommunications. London and Washington participated in the financing. The cable's design sparked passionate discussions among specialists. Morse, supported by Michael Faraday, argued for a thin wire limiting signal delay. William Thomson, the future Lord Kelvin, defended a thick copper core reducing electrical resistance.

The Gutta Percha Company finally manufactured this revolutionary cable. Its structure ultimately consisted of seven twisted copper wires creating a 2.1-millimeter conductor, wrapped in three layers of gutta-percha, a resin extracted from an Indonesian tree. A protective layer of tarred hemp and a metal armor completed the assembly, for a final diameter of about 1.6 centimeters.

No ship could carry the necessary 3,200 kilometers of cable alone. The operation therefore mobilized two vessels : the British HMS *Agamemnon* and the American USS *Niagara*. Loading took three weeks, already attracting journalists' attention and public curiosity.

The first attempt in 1857 was a failure. The cable broke near the Irish coast. The following year, three attempts followed in succession. On July 29, 1858, the ships met in the middle of the ocean, connected their cable sections and departed in opposite directions. On August 5, the continents were finally linked between Bull's Arm Bay in Newfoundland and Telegraph Field on the Irish island of Valentia.

On August 16, 1858, Queen Victoria and President James Buchanan exchanged messages. Transmitting the sovereign's 98 words took nearly

sixteen hours, a delay that represented a prodigious advance at the time.

Euphoria erupted on both sides of the ocean. New York organized a parade and fireworks so enthusiastic they partially set City Hall ablaze. Trinity Church celebrated the event in lower Manhattan. In London, Atlantic Telegraph Company shares doubled in value while Charles Tilston Bright received a knighthood.

Jeweler Tiffany & Co. sensed an opportunity and purchased the hundreds of kilometers of unused cable brought back by the USS *Niagara*. The company transformed them into souvenirs, ten-centimeter segments with brass ends and descriptive plates sold for fifty cents apiece. Other merchants followed with cufflinks, earrings and letter openers.

The magic was short-lived and within weeks, the cable stopped functioning, transmissions became illegible. Edward Orange Wildman Whitehouse, technical director at the Irish terminus, had applied excessive voltages up to 2000 volts, convinced that intensity needed to increase with distance. A 1985 analysis by historian Donard de Cogan would also reveal manufacturing problems, the off-center central conductor passing dangerously close to the outer metal armor. The marriage of these defects with the imperfect insulation of the gutta-percha and material impurities sealed the cable's death warrant.

During its brief existence, this first link transmitted 732 messages. Among them, the British order to cancel the deployment of two regiments to India, the local rebellion having been brought under control. This decision saved between 50,000 and 60,000 pounds sterling, roughly one-seventh of the cable's cost.

The technical failure provoked rumors and accusations of fraud. Cyrus W. Field, far from being discouraged and still supported by the British government, persevered. The Atlantic Telegraph Company finally installed a permanent link in 1866. This project, completed at a time when running water and electricity remained inaccessible to most people, testifies to an extraordinary vision.

This first electrical crossing of the Atlantic marks the history of communications. It proved the possibility of instantaneous intercontinental exchanges and heralded the architecture of the global network of submarine cables on which all of today's Internet rests.

Herman Hollerith

In the 19th century, the United States was drowning in data. The Census Bureau struggled to keep pace with an exploding population. Herman Hollerith would create a machine that addressed the challenge of processing these mountains of forms.

In 1879, fresh out of Columbia School of Mines, he joined the Census Bureau. He met Dr. John Shaw Billings, who headed the vital statistics division, and who suggested the idea of mechanizing data processing. Indeed, the stakes became obvious when looking at the 1880 census. Fifty million Americans to count. Operators laboriously processing twenty characteristics per minute with rudimentary tools. The figures were not published until 1889, barely a year before the next census. The situation was untenable.

Hollerith then left the Bureau to follow General Francis Walker to MIT. Having become an instructor in mechanical engineering, he began developing his first prototypes. He drew inspiration from Colonel Charles W. Seaton's work, who had created a machine using a roll-feed system comparable to that of a mechanical piano. But Hollerith would take a different direction.

After this academic interlude, he returned to Washington, joined the Patent Office, then established himself as an independent expert. This experience gave him the keys to protect his future inventions. He first worked on electric brakes for railroads, outperforming George Westinghouse during tests in 1887. But the examiners doubted electricity's reliability as a power source. No matter.

His data processing system relied on punched cards measuring 6.5 × 3.25 inches. The operator, reading census forms, could encode up to 17 different characteristics among 240 possible positions. Standard codes, such as those for districts, were preset and reproduced on entire stacks of cards.

The tabulating machine revealed Hollerith's ingenuity. In the upper jaw, rows of spring-loaded pins; in the lower jaw, tiny cups of mercury. The pins, connected to an electrical current, faced cups linked to 40 counters. The employee inserted a card into the press, closed the jaws, and the pins passing through the holes plunged into the mercury. The current flowed to the electromagnetic counters, advancing each relevant needle. A complete rotation of the units needle activated the hundreds needle. The card was then placed in a compartment of the sorter before moving to the next one.

The numbers spoke for themselves: 700 punched cards per day per operator, 7,000 cards processed daily by a machine user, 250 character-

istics analyzed per minute. Through relays, the system performed more sophisticated analyses, such as identifying "white men born abroad".

The major trial by fire came with the 1890 census. The Bureau ordered fifty machines to count nearly 63 million Americans. Such success that the administration would use improved versions of the system for six decades, until the arrival of UNIVAC I.

Hollerith's influence spread beyond the census. Statistical services, insurance companies, railroads-all wanted this technology that drastically reduced data processing times. In a rapidly changing industrial society, it addressed a critical need. Moreover, some technical choices he made had surprising longevity. His punched card format matched the dimensions of banknotes of the era, simply because filing cabinets adapted to them already existed. His positional coding system, designed for the 1901 agricultural census, would influence IBM and other companies through the 1970s.

The patents filed by Hollerith were the forerunners of the data processing industry. For sixty years, his machine bridged the gap between manual processing and electronic computing. It transformed how we manage information while initiating the techniques of the modern computer industry. An invention born from a very concrete problem.

II

1930

The Dawn of Automatic Computation

A world in transition awakens to the mechanization of calculation. The fertile ground of the 1930s draws nourishment from a rich intellectual heritage: Leibniz's binary system (1703), Jacquard's punched card (1801), and Boole's algebra (1854). These concepts, dormant for decades, are poised to revolutionize our relationship with mathematics.

The crash of the 1929 stock market collapse still echoes as this pivotal decade opens. The American administration is mired in its own numbers. The 1930 census illustrates this bureaucratic quagmire, where two full years were needed to manually process the forms. An eternity.

Facing the Great Depression, Roosevelt deploys his New Deal. The state expands, social programs emerge. Each citizen becomes a card, each benefit a figure to record. The administrative machinery groans under the load.

Meanwhile, the eye of the totalitarian storm descends upon Europe. Nazi Germany, with its cold classificatory obsession, uses tabulators from Dehomag, an IBM subsidiary. Lives reduced to holes in cardboard. The bureaucracy of horror takes shape.

As the 1930s draw to a close, the air grows thick with tension. Intelligence services attempt to pierce the mystery of Enigma. This German

encryption machine, with its rotating rotors, defies analysis. Cryptography will give birth to the first electronic calculators, a vital challenge.

Radio transforms information. Waves now carry voice and music across continents and oceans. In studios, technicians manipulate increasingly sophisticated equipment. Yet while information travels at the speed of light, its processing remains archaic.

Company counters become equipped. The characteristic clacking of mechanical calculating machines punctuates employees' days. IBM, Burroughs, and NCR prosper by selling these precious aids to commercial calculation.

At the heart of this ferment, flashes of genius illuminate the scientific horizon. At Cambridge, Alan Turing publishes "On Computable Numbers" in 1936. This foundational text poses the question of the theoretical limits of automatic computation. His abstract universal machine establishes the concept of the modern computer, though no one grasps its significance. Meanwhile at MIT, Claude Shannon, a brilliant 21-year-old student, links Boolean algebra to electrical circuits in his master's thesis. This fruitful union will spawn the digital electronics of the 20^{th} century.

The factory gradually automates. Ford's assembly lines represent only a first step. Industrialists seek systems capable of executing and regulating processes. Punched cards, heirs to the Jacquard loom, emerge as instruction carriers.

All scientific domains collide with the limits of manual calculation. Quantum physicists, astronomers or meteorologists, all juggle complex equations that saturate human processing capacity. Pencil and paper no longer suffice.

Aviation demands diabolical precision. A tiny error in aerodynamic calculations spells catastrophe. At the Aberdeen ballistics laboratory, female mathematicians, called "*computers*", tirelessly calculate trajectories for artillery. Meticulous work upon which human lives depend.

A new generation emerges from universities. These young scientists conceive of calculation differently: no longer as a series of isolated operations, but as a process, a logical flow. They think in algorithms before the word becomes commonplace. AT&T's Bell Labs forms an unparalleled crucible of innovation. In its corridors, mathematicians, physicists, linguists, and engineers cross paths. This fertile interdisciplinarity nourishes nascent computer science.

The race to mechanize calculation accelerates. The electromagnetic relay gradually replaces the mechanical lever. Faster, more reliable, it constitutes a first form of electronic breakthrough. Patents pile up in national patent offices.

In this fever of innovation, certain standards begin to emerge, such as IBM's punched card with its 80 columns and 12 rows. This cardboard rectangle asserts itself as the universal carrier of coded information, the direct ancestor of our digital files.

When Hitler's troops invade Poland in September 1939, no one suspects that the global conflict will serve as a technological accelerator for computing. Military urgency will soon transform mechanical calculators into electronic giants.

These 1930s, caught between the Great Depression and World War II, carry within them the seeds of our digital age. Theoretical concepts await their material embodiment. The ground is ready, the storm rumbles. Paradoxically, in this world on the brink of the abyss, the roots of the computer revolution germinate.

Vannevar Bush's Differential Analyzer

More than two centuries after Gottfried Wilhelm Leibniz's reflections on the mechanization of mathematical reasoning, a dream that had long remained in the realm of fantasy took shape in the laboratories of the Massachusetts Institute of Technology (MIT). One man, Vannevar Bush, born in 1890 in Everett, Massachusetts, undertook this colossal task during the interwar period.

He created the differential analyzer, completed in 1931. A machine resulting from a collaborative effort with Frank D. Gage, Harold L. Hazen, King E. Gould, and Samuel H. Caldwell. The idea was not entirely new, however. Sir William Thomson had suggested fifty years earlier that the integrators designed by his brother could, if connected together, solve differential equations. But the technical constraints of the Victorian era made this dream unattainable.

Bush succeeded where others had failed. His differential analyzer solved sixth-order differential equations or three second-order equations simultaneously, a genuine technical feat. The machine relied on torque amplifiers that supported considerable mechanical loads, a system of transmission rods (*bus shafts*) that connected the various units, and impressive dimensions to maximize plotting precision. A bold concept.

The heart of this invention aimed at three qualities rarely combined: extreme flexibility, mechanical robustness, and acceptable precision. Under normal conditions, the machine achieved precision of one thousandth

for each individual unit. Momentary errors naturally compensated during the integration process, recalling the behavior of a planimeter whose deviations eventually balance out.

To use this machine, several hours were needed to configure it after completing the necessary plots and determining the connection diagram. The actual solving generally took about ten minutes for each set of boundary conditions. Operators needed to acquire some experience but gained in return an intimate understanding of the differential equations they were manipulating.

Vannevar Bush and his collaborators had to overcome significant technical obstacles. Mechanical backlash and integrator slippage were complex issues. They developed an ingenious system called *lashlock* to eliminate backlash in worm gears and designed two-stage torque amplifiers producing very high torque ratios with minimal input torque.

To validate the reliability of their invention, they conducted forty rigorous tests on a complete integration unit. These tests, performed with loads varying from zero to one foot-pound of output torque, at different positions and in both directions of rotation, revealed an average deviation of only 0.032 % from the calibration constant. The maximum deviation, observed during a single test, did not exceed 0.12 %.

This machine marked the history of scientific computing. By positioning itself midway between rudimentary mechanical calculators and future electronic computers, the differential analyzer demonstrated that complex mathematical calculations could be mechanized. This breakthrough opened unprecedented perspectives for solving challenging problems in physics and engineering.

Gödel's Incompleteness Theorem

Kurt Gödel published a text in 1931 that radically changed our understanding of mathematics. His paper "On Formally Undecidable Propositions of Principia Mathematica and Related Systems" revealed an insurmountable barrier in formal mathematical systems. This revelation, the famous incompleteness theorem, would later transform theoretical computer science.

Mathematicians of the early 20^{th} century dreamed of a discipline with solid and unshakable foundations. David Hilbert embodied this quest with his program aimed at proving the complete consistency of mathematics

through formal methods. Bertrand Russell and Alfred North Whitehead had written their *Principia Mathematica* between 1910 and 1913, an ambitious attempt to reconstruct all of mathematics from elementary logical axioms. It was in this intellectual world that Gödel posed a disturbing question: do these formal systems have intrinsic limits?

His approach was brilliant. He created a correspondence between mathematical statements and numbers, called *Gödel numbering*. Each symbol, formula, and proof was assigned a unique numerical code. This trick transformed mathematical propositions into manipulable arithmetical objects. Using this system, Gödel constructed a particular mathematical proposition which, translated into ordinary language, states: "I cannot be proven in this formal system". This construction evokes the ancient liar paradox of Eubulides (5^{th} century BC), but without falling into contradiction.

The result was devastating for Hilbert's program: in any consistent formal system capable of describing elementary arithmetic, mathematical truths exist that are neither provable nor refutable within that system. His second theorem drove the point home by proving that a consistent formal system cannot demonstrate its own consistency. The edifice of certainties that mathematicians sought to build collapsed.

Five years later, Alan Turing, inspired by this work, developed the concept of the universal machine. Gödel's numerical encoding technique showed him how to represent programs as numbers, an idea that underlies our current computers. Turing established the existence of the halting problem, a computational question that no algorithm can systematically solve.

Computability theory was born from these discoveries and defines the boundaries of automatic computation. Programming languages, their compilers and interpreters bear the imprint of these theoretical results. When we attempt to formally verify computer programs, we encounter the limitations identified by Gödel.

Researchers continue to explore these complex territories. Complexity theory focuses on the resources needed to solve decidable problems. Fuzzy logic offers alternative paths in the face of the limits of classical systems. Mathematical proof assistant tools incorporate these constraints into their design.

For knowledge representation, artificial intelligence draws on Gödelian encoding techniques. Expert systems rely on these foundations while acknowledging the inherent barriers to formal reasoning. On a philosophical level, these theorems challenge us to consider the nature of thought, particularly whether there is a difference between human mathematical

intuition and the capabilities of formal systems.

Modern cryptography draws directly on the numbering methods invented by Gödel. Number theory, central to his proofs, now constitutes a pillar of computer security. Researchers working on type systems, program verification, or proof assistants navigate an intellectual space whose contours Gödel traced.

Nearly a century after their publication, the incompleteness theorems remain at the heart of fundamental computer science. They remind us of the inherent limits of formal systems and stimulate our creativity in designing new approaches.

The Turing Machine

Alan Mathison Turing, a 24-year-old British mathematician fresh from King's College, published the scientific article *On Computable Numbers, with an Application to the Entscheidungsproblem* in 1936. An intellectual upheaval that would establish the foundations of theoretical computer science.

In it, he described an abstract machine of striking simplicity: an endless tape divided into cells, a head that reads and writes symbols, a set of states and rules dictating its behavior. This minimalist construction concealed extraordinary power: it could solve any mechanically computable problem.

Turing's genius lay not so much in the complexity of his model as in his way of rethinking computation. He broke with traditional mathematical abstractions and drew inspiration from the work of human computers, those people who executed calculations by hand according to precise procedures. His machine modeled computation as a sequence of deterministic elementary actions.

This new approach settled a pressing question posed by David Hilbert in 1928, the Entscheidungsproblem. Turing demonstrated the nonexistence of a universal method for deciding whether a given mathematical formula is provable. His conclusion aligned with that of Alonzo Church, obtained independently through lambda calculus. But Turing went further: his universal machine, capable of simulating any other Turing machine, sketched the idea of a programmable computer that would shape the architecture of the first electronic calculators designed by John von Neumann.

World War II transformed these theoretical ideas into a vital concern. At Bletchley Park, Alan Turing put his intelligence to work deciphering German communications. He designed the "Bombes", electromechanical machines that automated the search for Enigma encryption keys. These 211 machines decoded up to 3,000 messages per day. This contribution shortened the conflict by approximately two years according to General Eisenhower.

With peace restored, Turing continued his research on thinking machines, first at the National Physical Laboratory, then at Manchester. He invented the first computer chess game and proposed his famous test: if a human cannot distinguish a machine's responses from those of another human during a conversation, then the machine exhibits a form of intelligence. This idea remains a reference point in artificial intelligence research.

But fate struck brutally. In 1952, police arrested Turing for homosexuality, a crime at the time in the United Kingdom. Sentenced to hormonal treatment intended to "cure" his sexual orientation, he suffered devastating effects on his health. Stripped of security clearance and banned from traveling to the United States, Turing died in 1954, presumably by suicide. He was 41 years old.

His public recognition would wait a long time, as his work at Bletchley Park remained classified. The 1970s began to lift the veil on his contributions. In 2009, Gordon Brown, British Prime Minister, offered an official apology for the treatment Turing had suffered. Queen Elizabeth II granted him a posthumous pardon in 2013. Four years later, the "Turing" law extended this pardon to all men convicted of homosexuality.

Turing's scientific legacy radiates through our digital world, as his theoretical machine remains the benchmark for understanding the limits of computation. A system is called "Turing-complete" if it equals its computational power, a criterion that has become standard for evaluating programming languages and computer architectures.

His insights on artificial intelligence continue to nourish current research. The Turing Medal, the highest distinction in computer science, bears his name. In mathematics, his work on computability and decidability opened entire fields. His final research on morphogenesis, using mathematics to explain biological patterns, testified to his boundless curiosity.

Today, his portrait adorns the 50-pound note since 2021. His name marks universities, institutes, and scientific prizes. Turing's story tells both the birth of modern computer science and the evolution of attitudes. His broken life and visionary work continue to inspire mathematicians,

computer scientists, and artificial intelligence researchers of the XXIst century.

Every contemporary digital computation bears his mark. Our world of algorithms, data, and artificial intelligence flows directly from his vision. The next time you use a computer, remember the young mathematician who, in the 1930s, was already imagining our digital future.

The Z1 Mechanical Computer

Berlin, 1936. Konrad Zuse, a 26-year-old man, works away in his parents' apartment. This civil engineer has just left his position at aircraft manufacturer Henschel, where he worked as a calculator. He has a fixed idea: to build a machine capable of automating tedious mathematical calculations. Unknowingly, he is about to write an important page in the history of computing machines.

Technical naivety works in his favor. He knows nothing of IBM's punched cards or Charles Babbage's designs from the previous century. Free from any influence, he rethinks the architecture of calculating machines from scratch. His first technical choice is the binary system, at a time when all existing machines operate in decimal. His machine will separate memory from the arithmetic unit, manipulate floating-point numbers, and store 64 values of 24 bits.

The heart of the Z1 rests on a surprising invention. Rather than using conventional gears, Zuse manufactures metal plates that slide within a frame. A movement in one direction symbolizes a 1, the absence of movement represents a 0. These plates, stacked and connected by vertical rods, form purely mechanical logic gates. The arithmetic unit is divided into two distinct sections: one processes exponents, the other the mantissas of floating-point numbers. Four calculation phases punctuate the execution of micro-instructions, each associated with a cardinal direction.

To program his invention, Zuse reuses 35mm film stock. He punches this film according to the precise code of eight holes per instruction line, two for the operation and six for memory addressing. His eight instructions cover elementary arithmetic operations, memory transfers, input and output of results. The Z1 can also recognize special cases such as zero value (exponent -64) or infinity (exponent 63). Dedicated circuits stop the machine in case of invalid operations.

However, the elegant mechanics clash with real-world constraints. The

perfect coordination of thousands of moving parts becomes a nightmare. Completed in 1938, the Z1 never functions properly. Allied bombing in 1943 reduces the machine to ashes, but the architecture will survive in the Z3, an electromechanical relay version that Zuse builds in 1941.

Forty years later, in the 1980s, Konrad Zuse begins reconstructing his firstborn for the Deutsches Technikmuseum in Berlin. This replica, completed in 1989, benefits from modern manufacturing techniques: more compact, its twelve mechanical layers contain approximately 6,000 logic gates. Despite these refinements, the machine retains the quirks of the original and requires constant monitoring.

The Z1 is remarkably at the origin of our current computers: architecture separating computation and memory (later named "von Neumann architecture"), standardized floating-point representation, binary system. These characteristics are even more impressive when one knows that Zuse had neither electronics training nor deep knowledge of formal logic.

The scarcity of materials in pre-war Germany forced Zuse to optimize every component. This minimalist approach contrasts with contemporary American projects like ENIAC or Harvard's Mark I, which benefited from considerable budgets and large teams.

While the Z1 does not constitute a universal computer in the strict sense defined by Turing (it notably lacks conditional branch instructions) it nevertheless marks a breakthrough in the evolution of calculating machines. Its binary architecture and modular design remain the foundations of modern computing. It remains an anomaly, the almost solitary work of a visionary who, through his ignorance of established conventions, reinvented the art of mechanical calculation in the 20^{th} century.

Lambda Calculus

When Alonzo Church published his work on lambda calculus in the 1930s at Princeton, he probably had no idea of the monumental impact his creation would have on the future of computing. This mathematical formalism, austere in appearance, conceals an elegance that still fascinates researchers nearly a century later.

The story truly begins with a remark by Frege in 1893: any function with multiple arguments transforms into a sequence of functions with a single argument. Take an addition function: instead of directly computing the sum of two numbers, we first build a function that expects a first

number, then returns another function that expects the second. This vision corresponds remarkably to the physical reality of computers. When the machine loads a number into memory, it is ready to associate it with any other value.

In 1924, Moses Schönfinkel discovered that two elementary functions, K and S, are sufficient to construct all others. Church later took up this idea by creating a more refined notation: the application of a function F to an argument A is simply written FA, with parentheses appearing only when necessary.

The 1930s saw major results flourish. Church and Rosser proved confluence of reductions in 1936, a fundamental mathematical property of lambda calculus, which states that the final result does not depend on the order of intermediate calculations. That same year, Alan Turing established a connection between his abstract machine and Church's formalism—both approaches compute exactly the same functions.

Lambda calculus achieves the remarkable feat of representing natural numbers without using digits. A number n is also a function that applies another function n times to an argument. This representation, called Church encoding, makes all arithmetic operations possible. Recursive functions, pillars of modern programming, are expressed through a mechanism called fixed point.

Church then formulated his famous thesis: computable functions are those definable in his formalism. This statement links an intuitive notion to a precise mathematical concept and remains impossible to prove rigorously. Kleene strengthened it in 1936 by proving the equivalence between lambda-definable functions and Gödel's general recursive functions.

The imprint of lambda calculus on modern computing proves profound. John McCarthy drew directly from it to create LISP in 1958, ancestor of an entire family of functional languages like Haskell or ML. These languages inherit the fundamental concepts of Church's formalism: functions as first-class values, evaluation by reduction, sophisticated type systems.

In the 1970s, Dana Scott constructed a rigorous mathematical semantics for lambda calculus. His work gave birth to domain theory, a powerful tool for understanding and verifying programming languages.

The influence of lambda calculus also touches compiler design. The representation of programs as trees rather than linear sequences of instructions, inspired by the structure of lambda terms, optimizes memory usage—a technique still relevant in current compilers. Dependent type theory, an extension of the simple type system of the original lambda calculus, now serves as the foundation for proof assistants like Coq or Agda, tools that formally verify the validity of programs or mathematical proofs.

The concepts of lambda calculus nourish parallel architectures, web languages like JavaScript, and modern type systems. Its minimalist philosophy—everything is a function—combined with its extraordinary expressiveness makes it a valuable instrument for thinking about computer systems.

Shannon's Logic Circuit

The design of electrical circuits in the 1930s was an almost artisanal process. Engineers working on early computers assembled relays and switches based on their technical intuition, without formalized methods. Their creations, born from personal experience, lacked a rigorous theoretical framework.

It was at MIT that a young student named Claude Shannon revolutionized this approach. In 1937, he built a bridge between two seemingly distinct worlds: switching circuits and Boolean algebra. This mathematical discipline, developed in the 19$^{\text{th}}$ century, had never been applied to electricity. Shannon's insight was that a closed circuit represents the value 1, an open circuit the value 0. Two components in series function as the `AND` logical operation, while their parallel arrangement corresponds to the `OR` operation.

His master's thesis *A Symbolic Analysis of Relay and Switching Circuits*, published in 1938, formalized this correspondence. The document did not merely present an abstract theory. Shannon detailed concrete applications: a binary adder and an electric combination lock. These examples demonstrated the power of an approach that transformed circuit design into a mathematical discipline.

The impact was immediate. Engineers now had tools to calculate the minimum number of components needed for a given function. Costs decreased, reliability improved. Instead of working through trial and error, they could verify their concepts before physical construction.

Shannon was not alone in this pursuit. In Japan, Akira Nakashima had been working since 1935 on similar concepts for the NEC company. In the USSR, Viktor Shestakov explored comparable ideas, inspired by the work of physicist Paul Ehrenfest. The convergence of this research showed that the time was ripe for this conceptual breakthrough.

The arrival of electronic computers in the 1940s and 1950s gave new dimension to Shannon's work. Mechanical relays gave way to vacuum

tubes, then to transistors. The mathematical approach adapted perfectly to these new technologies. The constant miniaturization of components made the use of formal methods indispensable.

The development of integrated circuits in the 1960s raised the question of manually designing chips containing thousands of logic gates. The principles established by Shannon then became the foundation of computer-aided design tools. This software automatically translates abstract descriptions into optimized circuits.

The semiconductor industry has continued to evolve since then, but the theoretical framework has remained stable. Today's computers, despite their dizzying complexity, still operate according to the principles identified by Shannon. His theory illustrates how mathematical abstraction can generate major technological advances.

The symbolic representation of systems proposed by Shannon also inspired the development of programming languages and formal verification methods. His influence extends to theoretical computer science, particularly automata theory and the study of algorithmic complexity.

His thesis received the Alfred Noble Prize from the American Institute of Electrical Engineers in 1940. Herman H. Goldstine later called it "one of the most important master's theses ever written", which had transformed the design of digital circuits "from an art into a science".

This scientific achievement embodies the successful fusion of mathematical theory and engineering practice. Without this vision, modern electronics would have followed a very different path. Shannon's genius was to understand that an abstract formalism from the 19^{th} century could solve the technical problems of the 20^{th}: automatic computation and information processing. Today's computers, with their billions of transistors, remain faithful to the principles he formulated. Few ideas traverse decades this way without losing their relevance.

The Complex Number Calculator

A plywood board, two relays salvaged from a dumpster, strips cut from a tobacco tin, batteries, and light bulbs. Who would have thought that these makeshift materials assembled by George R. Stibitz in his kitchen in 1937 would mark our history of computing? This mathematician at Bell Telephone Laboratories (Bell Labs) had just noticed a parallel between the positions of telephone relays and binary notation. His domestic tinkering

gave birth to a one-bit binary adder.

Stibitz's evenings turned into circuit design sessions for other arithmetic operations. In 1938, he presented his work to Thornton Fry, head of the laboratory's mathematics section. Fry raised a practical question: could these relay calculators handle complex numbers? This task was then mobilizing an army of human computers at Bell Labs.

Stibitz tackled the challenge head-on. The plans were completed by February 1938. He partnered with Sam Williams, an engineer specializing in switching systems. Their collaboration bore fruit in 1939 with the completion of the Complex Number Calculator.

The machine shone through its technical innovations. It used a binary-coded decimal system with four relays per decimal digit. It processed numbers up to eight decimal digits, with two additional internal digits to limit rounding errors. Its structure comprised two distinct calculation units: one for the real part of complex numbers, the other for the imaginary part.

The user interface constituted a revolution in itself. The machine was hidden in a closet, accessible only for maintenance. Users worked at three operator stations scattered throughout the Bell Labs building on West Street in New York. Each station had a keyboard for input and a teleprinter for displaying results. This configuration represents the first use of remote terminals in computing history.

The multiplication keys, as well as the division key, activated subroutines of about a dozen steps. These executed complex operations using the two calculation units, which worked exclusively on real numbers. This architecture already foreshadowed the modern notion of subroutines.

On September 11, 1940, the Complex Number Calculator was unveiled to the public at a meeting of the American Mathematical Society at Dartmouth College. For the occasion, a console was modified to communicate with the computer via a long-distance telephone line. Participants, including the famous Norbert Wiener (founding father of cybernetics), submitted problems on the keyboard. The data traveled to the relay equipment in New York, and the results returned to the teleprinter in less than a minute. This demonstration inaugurated remote control of a computer, heralding the data transmission that would explode in the 1960s.

The Complex Number Calculator served until 1949, with remarkable reliability. During World War II, the network design groups, its primary users, ran it almost continuously from 8 a.m. to 9 p.m., six days a week. The machine had not been designed for such a regime. Built as a demonstration model before the war, it lacked self-checking features and contact protection standard in telephone exchanges. The war prevented the con-

struction of a more robust second machine. Toward the end of the conflict, it had to be shut down for two days to replace worn relay contacts.

The legacy of the Complex Number Calculator is immense. It proved the feasibility of automatic calculations with reliable electromechanical components. Its success spawned other relay calculators at Bell Labs during the war: the Model II in 1943, capable of iterative operations, then the Models III and IV in 1944-1945, more powerful with approximately 1,400 relays and seven teleprinter units each.

These machines established fundamental principles still present in modern computing: the use of binary for calculations, the separation between calculation unit and user interface, the notion of subroutines, and remote access to computing resources. Bell Labs' relay calculators continued to operate efficiently 13 to 15 years after the war, some remaining in service well after the arrival of the first commercial electronic computers.

The Complex Number Calculator, born in a kitchen, showed that complex mathematical tasks could be automated reliably. Its influence extended to automatic telephone message accounting systems, where similar calculators processed detailed call billing. This machine represents an essential link between mechanical calculators and our computers.

III

1940

Computing in the Turmoil of War

When war broke out in 1939, no one imagined that this conflict would radically transform our relationship with calculation and information. Hitler invaded Poland, the world tilted, and in the shadows of general staffs, an equally decisive battle began: the battle of intelligence.

Bletchley Park manor, a few miles from London, housed a secret that would change history. Alan Turing designed machines capable of decoding messages encrypted by Enigma, that reputedly unbreakable system used by the Germans. Cryptanalysis, a confidential science until then, became a vital issue. The electromechanical "bombes", primitive ancestors of computers, ran day and night. Deciphering enemy communications saved thousands of lives and would have shortened the war by two years, according to some historians.

Meanwhile, the American army struggled with its ballistic calculations, too complex to be solved by hand, which slowed weapons production. At the University of Pennsylvania, a team worked relentlessly on a crazy project: ENIAC. This 30-ton machine, with its 18,000 vacuum tubes that heated like embers, consumed 200 kilowatts of electricity to calculate shell trajectories in seconds. When it was first powered on in 1946, the war had been over for months, but the era of electronic computing had just begun.

Women played a strangely forgotten role in this story. Six female mathematicians programmed ENIAC by physically reconfiguring its circuits. They invented, unknowingly, the profession of programmer. They were Kay McNulty, Betty Snyder, Marlyn Wescoff, Ruth Lichterman, Betty Jean Jennings, and Fran Bilas. Their names deserve to appear in history books alongside the engineers who assembled the machine.

The post-war period sketched a new landscape where the USA, virtually untouched by combat, now possessed 50% of the world's wealth. American industry ran at full capacity to rebuild devastated Europe. IBM's punched card machines equipped administrations and large companies, but their limitations quickly became apparent in the face of exploding calculation needs.

In the scientific world, a young Bell Labs engineer, Claude Shannon, published "A Mathematical Theory of Communication", a 1948 article that would almost go unnoticed. In it, he demonstrated that any information could be treated as a sequence of 0s and 1s. This deceptively simple idea would revolutionize our civilization.

The Cold War settled in and pushed both superpowers to invest in research. In Moscow, Sergei Lebedev worked secretly on MESM, the first Soviet computer. The Americans multiplied projects: Whirlwind at MIT, IAS at Princeton, Mark III at Harvard. John von Neumann theorized an architecture that would remain the standard for more than half a century. Memory circuits improved, vacuum tubes gave way to transistors invented in 1947.

In the innovative world of telecommunications, AT&T expanded its automatic telephone network. Undersea cables multiplied. In laboratories, digital data transmission was being experimented with. These technologies would eventually converge with computing.

Universities created their first automatic computing departments. A new vocabulary emerged: programming, algorithm, bit, byte. Commercial machines appeared. Ferranti released its Mark I in Great Britain. Konrad Zuse, in Germany, created the first European computer company. Remington Rand delivered UNIVAC I to the American Census Bureau in 1951.

Libraries and documentation centers, overwhelmed by the explosion of post-war scientific literature, sought new solutions. The universal decimal classification showed its limits. Theoretical work on automated documentation systems emerged. Vannevar Bush, in his famous 1945 article, imagined an office equipped with a "Memex", a machine capable of storing and linking documents through associations, a troubling prefiguration of the Web we know.

Computing did not yet exist as a discipline; people spoke of cybernetics, a term proposed by Norbert Wiener in 1948 to designate the science of control and communication. In France, IBM translated "*computer*" as "ordinateur", on the suggestion of Professor Jacques Perret. The term quickly took hold in the Francophone world.

Who could have predicted, in 1940, that ten years later, electronic machines would process information at dizzying speed? These early achievements, despite their monumental size and exorbitant cost, would transform every aspect of our daily lives in the following decades.

The desire to create thinking machines was nothing new. Pascal and Leibniz had built mechanical calculators in the 17^{th} century. Babbage had designed his analytical engine in the 19^{th}. But it took the urgency of war, massive state investment, and collaboration between universities and industries for the dream to become reality. The modern computer was born from this complex alchemy where military necessity, mathematical research, and electronic engineering intertwined.

In 1950, computing was barely emerging from its prehistory. Machines were enormous, fragile, costly. Only a few initiates understood their operation and possibilities. Yet the path to digital transformation was laid out. The theoretical and technical foundations were established. The first civilian applications appeared.

The adventure that was beginning would last longer than most technological revolutions. Seventy-five years later, we live in the wake of this transformation. The computer, a child of war turned universal tool, continues to modify our relationship to knowledge, work, and communication. Its history truly begins in the 1940s, when men and women, facing the challenges of a world in flames, invented thinking machines.

At the dawn of the 1950s, a page turned. Pioneers gave way to industrialists. Fundamental research stepped back before practical applications. Computing entered its commercial phase. Names like IBM, Remington Rand, Ferranti, and Bull established themselves. A new industry took flight, driven by the growing needs of a booming economy. No one suspected the scale of the upheavals to come.

Atanasoff-Berry Computer

In the winter of 1937, John Vincent Atanasoff, a physicist and mathematician at Iowa State University, was driving home on a freezing night.

During this journey, his mind suddenly lit up with an insight that would transform the world: four fundamental principles that would form the conceptual foundation of the first electronic computer. These principles were the use of electronics for calculations, the adoption of the binary system, the separation of computation from memory, and the periodic refresh of data in memory.

Born in 1903 in Hamilton, New York, Atanasoff had studied electrical engineering in Florida before turning to mathematics at Iowa State College. His doctorate in theoretical physics from MIT, completed in 1930, had confronted him with the limitations of existing mechanical calculators. These obsolete machines hindered his research on the dielectric constant of helium, a problem requiring computing power that did not yet exist.

The meeting with Clifford Berry changed everything. This brilliant young electrical engineering student brought the technical skills necessary to implement Atanasoff's ideas. Their collaboration began in 1939 with a rudimentary prototype that validated the basic concepts. This initial success earned them a modest grant of $850 to build the complete machine, the Atanasoff-Berry Computer (ABC).

This machine weighed over 300 kg and featured various innovative elements. Rotating drums covered with capacitors stored data in binary form. Vacuum tubes executed logical operations. Punched cards handled data input and output. A central clock synchronized the entire system. Unlike other calculators, the ABC did not process numbers directly but converted them to binary, a precursor to all modern computers. Its specific purpose was to solve systems of linear equations with up to 29 equations and 29 unknowns, with a remarkable precision of fifteen decimal places.

The fate of this machine took a tragic turn. Completed in 1942, the ABC was abandoned when its creators left the university to contribute to the war effort. The machine suffered from reliability issues, particularly with the punched card system. The university neglected to finalize the patent procedure initiated by Atanasoff. These unfortunate circumstances led to the dismantling of the ABC in 1948, without its contribution to computing being recognized.

The injustice deepened with the appearance of ENIAC in 1945. This machine designed by Presper Eckert and John Mauchly at the University of Pennsylvania was long celebrated as the first electronic computer. The reality was quite different. Mauchly had visited Atanasoff at Iowa State in June 1941, examined the ABC in detail, and read the 35-page manuscript describing its architecture. Many fundamental concepts from the ABC found their way into ENIAC, without attribution to their true creator. Mauchly's media savvy and generous military funding did the rest: ENIAC

stole the spotlight, Atanasoff fell into obscurity.

The truth came to light during a landmark trial. In the 1960s, Honeywell challenged the validity of the ENIAC patent held by Sperry Rand. The legal battle, which lasted from 1971 to 1973, exposed Mauchly's appropriation of Atanasoff's ideas. The judge ruled unambiguously: Eckert and Mauchly had not invented the electronic digital computer but had drawn inspiration from Atanasoff's work. The patent was invalidated, placing the invention of the computer in the public domain.

Official recognition finally came for Atanasoff. He received multiple distinctions in the following decades, including the prestigious National Medal of Technology presented by President George H. W. Bush in 1990. Books, a biography by Clark Mollenhoff in 1988, and a documentary (*Atanasoff: Forgotten Father of the Computer*, 2012) were dedicated to him. An exact replica of the ABC was built in the 1990s and displayed at Iowa State University, alongside a rare original memory drum.

Atanasoff's story illustrates how solitary genius can transform a discipline. By radically questioning existing approaches, he laid the conceptual foundations for all twentieth-century computing. Some of his concepts, such as the use of electronics or binary, became established with ENIAC and later EDVAC. Others, like the stored program, took longer to become industry standards. The ABC played a catalytic role in the evolution of electronic computing, fully justifying Atanasoff's title as the father of modern computing.

Artificial Neural Networks

In the middle of the 20^{th} century, research in artificial intelligence experienced a remarkable breakthrough with the work of Warren McCulloch and Walter Pitts. These two researchers, who seemed to have nothing in common, combined their talents to design a mathematical model of the neuron. Their proposal advanced our understanding of the brain and laid the foundation for future machine learning technologies.

In 1943, McCulloch, a seasoned neuropsychiatrist, and Pitts, a young mathematical prodigy, published a seminal article. This groundbreaking text *A Logical Calculus of the Ideas Immanent in Nervous Activity* established a correspondence between the functioning of biological neurons and logical operations. The authors relied on the all-or-nothing character of neuronal activity to model nerve cells as binary devices, either active

or inactive. This simplification enabled them to construct an equivalence with Boolean logic and its true/false values.

The backgrounds of these two researchers deserve attention. McCulloch, trained in psychiatry, had long harbored philosophical questions about the logical nature of thought and its physiological substrate. During the 1930s, he devoted himself to research in neurophysiology and collaborated notably with J. G. Dusser de Barenne at Yale, where they studied the localization of brain functions. Pitts, for his part, stood out for his precocious genius in mathematics. Without formal academic training, he developed a passion for mathematical logic from a very young age, devouring Bertrand Russell's *Principia Mathematica*. In the early 1940s, he joined Nicolas Rashevsky's team at the University of Chicago, a pioneering group in applying mathematics to biology.

The meeting between McCulloch and Pitts occurred in 1942, orchestrated by Jerome Lettvin. Despite their significant age difference, they quickly discovered their common interests. Pitts was enthusiastic about the concept of a logical machine capable of executing reasoning, an idea he connected to the work of Leibniz and Turing. McCulloch, for his part, had been attempting since the 1920s to translate neuronal activity into logical calculus. Together, they wondered whether the nervous system might function as such a logical machine.

Their 1943 article boldly combines elements of neurophysiology, philosophy, and mathematics. They present a simplified model of the neuron as a binary entity that activates or not according to the sum of its excitatory and inhibitory inputs, emitting a signal when this sum exceeds a certain threshold. By connecting these idealized neurons, they demonstrate the construction of networks using fundamental logical functions (`AND`, `OR`, `NOT`). Their approach follows an axiomatic method: from simplifying assumptions about neuronal activity, they build a logical calculus of relationships between neurons.

The authors themselves emphasize the theoretical limitations of their model. They do not seek to faithfully represent biological neurons, but rather to establish a formal framework for reasoning about hypothetical networks with known properties. They acknowledge that the activity of real neurons proves more continuous than discrete and that phenomena such as learning permanently modify the structure of networks. Their ambition lies elsewhere: to provide a mathematical tool for rigorously manipulating known networks and easily creating networks with desired properties.

This theoretical approach to biology, of which Nicolas Rashevsky was an ardent promoter, marks McCulloch and Pitts's article. A physicist

by training, Rashevsky founded a program of mathematical biophysics at Chicago in the 1930s. According to him, mathematical biology should proceed through abstractions and idealizations, similar to physics. Just as the physicist studies ideal concepts such as the material point or perfect fluid, the theoretical biologist must start from simplified models to progressively grasp the complexity of living systems. Mathematics, according to Rashevsky, offers a valuable framework for formalizing biological phenomena and deducing properties through rigorous reasoning.

It is in this spirit that McCulloch and Pitts shaped their model of logical neural networks. Their project aims not so much to describe the real nervous system as to show how a network of simple elements, properly connected, develops logical functions and, potentially, complex reasoning. Their work also connects to contemporary reflections on the logical foundations of computation, illustrated by the Turing machine. In 1936, Turing had shown how to define the process of computation by an abstract machine manipulating symbols according to precise rules. McCulloch and Pitts drew inspiration from this idea to conceive the brain as a logical machine, whose elementary components would be all-or-nothing neurons.

The impact of this article exceeded all expectations. On the theoretical level, it laid the foundations of computational neuroscience and cognitive science: the idea that complex cognitive functions arise from networks of simple units inspired numerous works in artificial intelligence and neuroscience. The article also presents a philosophical dimension by proposing a mechanistic vision of the mind, where thought would reduce to logical operations executed by neural networks. This conception had a lasting impact on debates about the nature of intelligence and its relationship to biological substrate.

The proposed model certainly has important limitations, which the authors acknowledge. It idealizes the neuron as a binary entity with a fixed threshold, whereas biological neurons possess continuous and adaptive dynamics. It assumes a fixed network structure, when neural connections constantly reconfigure through plasticity. Nevertheless, through its formal clarity, the article opened a vast field of research. It showed how a mathematical approach illuminates the functioning of the nervous system and, more broadly, the biological bases of intelligence.

This work testifies to the richness of interdisciplinary exchanges. It emerges at the crossroads of philosophical questions about the mind, experimental advances in neurophysiology, and formal developments in mathematical logic. It is through this intersection that McCulloch and Pitts arrive at a new vision, where the brain appears as a system for logical information processing. While their model remains rudimentary, it

sketches the conceptual outlines of future research on formal neural networks and their applications.

Colossus

At the heart of the secret war waged by British cryptanalysts, an electronic machine enabled the decryption of enemy messages and changed the course of history: Colossus.

Between 1943 and 1945, in the silence of the buildings at Bletchley Park in Buckinghamshire, England, Tommy Flowers led the construction of a series of calculators designed to crack the codes of the German Lorenz SZ40 machine. British cryptanalysts had nicknamed it "Tunny"—the tuna. The Germans called it *Schlüsselzusatz* or SZ. Colossus's mission was to find the initial rotor settings of Lorenz to decode intercepted messages.

Technically, Colossus embodied a remarkable achievement. The machine read 5,000 characters per second from punched tapes, performed Boolean calculations on the fly, and immediately printed the results. Its heart beat to the rhythm of 1,500 vacuum tubes, at a frequency of 5,000 pulses per second. It had binary arithmetic circuits, a primitive form of memory, and conditional branching capabilities.

This achievement would never have come to fruition without the marriage of mathematics and engineering. Max Newman and Alan Turing, through their theoretical vision, contributed to the design of this machine, even though Turing did not directly participate in the project. His work on the "universal machine" nevertheless infused the spirit of the project.

The first model became operational in early 1944. Such was its success that around ten units were built by the end of the war. Remarkably for the time, Colossus operated without breaking down. A rare reliability, inherited from the telephony expertise of the Dollis Hill team. This machine delivered usable results almost immediately, whereas the first electronic computers required many months of calibration before functioning properly.

Colossus's impact on the outcome of the conflict remains difficult to measure precisely. Accounts suggest a crucial role in anticipating German movements around the Normandy landings. The machine apparently decrypted messages signed by Hitler.

For three decades, Colossus remained shrouded in military secrecy. The American ENIAC received the honors while Colossus slept in oblivion.

It was not until the mid-1970s and the work of Brian Randell that its existence was revealed.

The exact classification of Colossus remains debated. Was it a true computer? Randell defined it as a "programmable electronic calculator for special purposes". Tommy Flowers preferred to speak of an "electronic processor". For Colossus did not compute like conventional computers. It executed a fixed program, without any real possibility of modification. Its logic, based on bit streams rather than words, distinguished it from traditional architectures. This hybrid nature represents its true strength and testifies to the richness of approaches during this foundational period. The history of machines did not follow a straight line toward the modern computer but explored different paths to address concrete problems.

Colossus's success also lies in its practical relevance. It perfectly met the needs of cryptanalysts in an urgent context. Its history reminds us that technological development is nourished by theoretical innovations, fitness for purpose, reliability, and operational efficiency.

Harvard Mark I

In the 1930s, the scientific landscape lacked tools capable of solving highly complex mathematical problems. Howard Aiken, a budding physicist at Harvard, set out to fill this gap. During 1937, he sketched the plans for a calculator that would process entire rows instead of the columns favored by machines of the era. He envisioned a machine capable of juggling positive and negative numbers, handling transcendent functions, and automatically chaining calculations.

After a rocky path with various manufacturers, Aiken finally convinced IBM. The company christened the project the *Automatic Sequence Controlled Calculator* (ASCC). March 1939 marked the signing of an agreement whereby IBM would build the machine as a gift to Harvard, which would provide facilities and personnel. Construction began two months later in IBM's laboratories in Endicott, New York. Aiken worked closely with engineer Clair D. Lake to bring this ambitious project to fruition.

The prototype emerged in January 1943 and successfully passed its initial tests. Transported disassembled to Harvard, it took up residence in the basement of the *Physics Research Laboratory.* In spring 1944, as it became operational, its use was diverted toward the war effort. Robert Campbell took command of a U.S. Navy team that included Grace Murray

Hopper and Richard Bloch. Aiken, a reserve officer, joined active service in 1941 to work on naval mines before taking command of the Mark I.

On August 7, 1944, an official ceremony brought together James Bryant Conant, president of Harvard, and Thomas J. Watson Sr., president of IBM. The event turned sour following a press release drafted under Aiken's influence that attributed all credit for the invention to himself, almost entirely erasing IBM's contribution. Watson, furious, threatened to boycott the ceremony. Aiken had to back down and acknowledge the role played by Lake, Hamilton, and Durfee in creating this machine.

The mechanical behemoth impressed with its dimensions: 15.5 meters long, 2.4 meters high, with a mass of 4.5 tons. Its interior housed electromagnetic relays, rotary decimal counters, and perforated tapes for instruction input. Data entered via punched cards and exited through modified electric typewriters. Slow compared to future electronic marvels, the Mark I compensated with exemplary reliability. It ran without interruption, day and night, producing a continuous flow of results.

Beyond the technical achievement, this machine embodied the feasibility of large-scale automatic calculators. These machines could now follow a sequence of operations from beginning to end without human intervention. The worldwide media coverage generated by its prowess swept away the last doubts about the future of such machines in the scientific sphere. The Harvard Mark I symbolizes the transition between the era of manual calculation and the dawn of modern computing.

Aiken did not stop at this first success. He developed a series of four increasingly sophisticated calculators. A visionary, he understood that these new machines called for a new generation of mathematicians capable of programming them. He persuaded Harvard to create a specific curriculum, first at the master's level and then at the doctoral level, in this nascent discipline that was computer science. The first tenured professor in this still-virgin field, he initiated what appears to have been the first academic program of its kind in the world. His teaching shaped pioneers such as Gerrit Blaauw, Frederick Brooks Jr., Kenneth Iverson, and Anthony Oettinger.

ENIAC

In 1943, the U.S. Army launched an ambitious project to accelerate its ballistic calculations. The war was in full swing, and artillery firing

tables, vital for bombing accuracy, required endless calculations. The Moore School of Electrical Engineering at the University of Pennsylvania was tasked with creating a radically new machine under the direction of John Mauchly and J. Presper Eckert.

Named ENIAC (Electronic Numerical Integrator and Computer), this machine marked a break from electromechanical calculators. Its development mobilized tremendous energy: approximately 200,000 hours of work were needed to produce, in November 1945, a technological monster. Too late to serve during the war, ENIAC nevertheless stood out as an unprecedented engineering feat.

The numbers are staggering: 30 tons of equipment spread over an area of 150 m^2, 40 large panels housing 18,000 vacuum tubes, 1,500 relays, 10,000 capacitors, and 6,000 manual switches. Unlike Atanasoff's ABC, which used binary, ENIAC directly manipulated 10-digit decimal numbers. Its memory consisted of accumulators, each storing one number, supplemented by function tables for constants.

ENIAC was programmed in a rudimentary and tedious manner. Thousands of switches had to be set and hundreds of cables plugged in, an operation that could take several days. Once configured, however, its performance was astonishing: 5,000 additions or 385 multiplications per second, shattering the records of existing machines.

On February 14, 1946, a carefully orchestrated press conference unveiled ENIAC to the world. The effect was striking. Newspapers went wild, calling the machine a "giant brain", a "magic brain", or a "meteor calculator". For the first time, the media attributed quasi-human capabilities to a machine. This anthropomorphization captured the collective imagination, despite Mauchly's attempts to temper the enthusiasm by reminding people that ENIAC simply performed calculations without truly "thinking". The myth was born, the machine capable of solving all problems had appeared.

What the public didn't know was that ENIAC drew heavily on the earlier work of John Atanasoff and Clifford Berry. As later revealed during a sensational trial, Mauchly had visited Atanasoff in 1941 and studied the ABC in detail, borrowing several fundamental concepts without ever crediting the inventors. This controversy over the paternity of the electronic computer remained buried for decades, while ENIAC reaped all the honors.

ENIAC was also the first computer programmed by women, another little-known reality. Female mathematicians and physicists, previously employed as human "computers" to perform calculations by hand, were selected for their excellence. Six of them became the first programmers

in history. They developed the initial programs and performed countless calculations, often in the shadows, their contribution remaining long ignored.

The Cold War offered ENIAC a second life. Installed at the Aberdeen Proving Ground in Maryland, it served for the originally planned ballistic tables, as well as for calculations related to the first thermonuclear bombs. Its ability to rapidly solve complex equations accelerated research in nuclear physics and other scientific fields.

Although retired from service in 1955, ENIAC left a lasting imprint on the development of computing. It stimulated research in the United States, both in hardware and theory. The lessons learned from its design directly inspired more advanced machines like EDVAC and UNIVAC. Meanwhile, figures like John von Neumann, Alan Turing, and Claude Shannon developed the theoretical foundations that would structure the discipline.

ENIAC's influence extended far beyond American borders. Its media coverage made the world aware of the potential of electronic computers. Similar projects emerged in the United Kingdom, the USSR, Germany, and Japan. Within a decade, computing transitioned from experimental to industrial. The concept of "computer" became embedded in popular culture, mixing fascination and anxiety about these "electronic brains" with mysterious capabilities.

ENIAC's story took a twist in the 1970s. A sensational trial, pitting Honeywell against Sperry Rand (holder of the ENIAC patent), revealed Mauchly's appropriation of Atanasoff's ideas. The judge ruled in 1973 that ENIAC was not the first electronic computer and invalidated its patent, thereby placing the invention in the public domain. This legal decision had critical consequences for the subsequent evolution of computing.

The cancellation of the ENIAC patent unleashed tremendous innovation potential. Deprived of exclusivity over fundamental concepts, laboratories and companies had to rely on specific technical improvements rather than legal protection of basic ideas. This situation fostered a remarkable flowering of competing machines: IBM, DEC, Control Data Corporation, and others rushed into the breach, each offering their vision of computer architecture. Without this decision, the digital landscape might have resembled an industry dominated by a few patent holders imposing their conditions on the market. Computing instead took a unique path where foundational principles remained accessible to all while specific innovations could be protected. This singular dynamic partly explains the stunning speed at which computers evolved from multi-ton monsters to personal machines, then to the mobile devices we now know.

This shift in the perception of calculating machines was ENIAC's most enduring contribution, despite its contested paternity. In short order, the public image of computers was radically transformed: from simple calculating tools, they became entities potentially capable of surpassing human intelligence. This vision, though exaggerated, guided computing research for decades to come, notably with the emergence of artificial intelligence in the 1950s.

Delay Line Memory

The first electronic computers faced the challenge of storing their data and instructions. ENIAC had only about twenty vacuum tube registers, a prohibitively expensive solution for any substantial memory.

J. Presper Eckert then conceived an innovative concept of transforming electrical signals into sound waves traveling through a mercury tube. These vibrations, captured at the other end, became electrical again before being reinjected at the input. The loop was complete. This technique multiplied memory capacity by one hundred at equal cost, with each line capable of containing up to 500 bits.

Maurice Wilkes and his Cambridge team brought this idea to life in EDSAC in 1949. Their machine utilized 32 mercury tubes storing 512 words of 36 bits. A word circulated every 48 microseconds on each of the 128 lines, while a simple addition required 864 microseconds.

This technology addressed the needs John von Neumann had identified in his description of EDVAC in 1945. He emphasized the necessity of an "internal" memory distinct from external devices such as punched cards. This memory had to preserve intermediate results, instructions, function tables, initial conditions, and results of successive iterations.

Despite its ingenuity, this technology suffered from uncertain reliability and high costs. Manufacturers preferred magnetic drum memory. IBM 650 systems were thus equipped with drums capable of storing up to 2,000 ten-digit words, with an average access time of 2.4 milliseconds, limited by the drum's rotation at 12,500 revolutions per minute.

EDVAC illustrated this transition: in 1954, a 4608-word drum memory complemented its delay lines. Drums remained present until the 1960s, primarily as secondary memory.

Delay line memory proved that large-capacity memories could be built separate from computing units, thus realizing von Neumann's architecture.

It was the founding concept of our current memory hierarchy.

The 1960s saw the advent of magnetic cores. Without moving parts, they offered truly random access with cycle times of approximately 6 microseconds—a thousand times faster than drums. Capacities exploded: the 1965 CDC 6600, clocked at 100 nanoseconds, had 128,000 words accessible in 1 microsecond.

The delay line symbolizes the delicate balances sought by pioneers between cost and performance, reliability and capacity, complexity and feasibility. It testifies to the inventiveness of early computer architects, confronted with challenges using the technical means of the 20^{th} century.

The principle of data circulation and regeneration it implemented is found, transformed, in contemporary DRAM memories. Its trajectory illustrates a recurring pattern in computing history: a technical innovation overcomes an obstacle before being supplanted by more efficient solutions.

Delay line memory represents the moment when computers acquired true working memory, distinct from computing units and permanent storage. This separation, which has become axiomatic in modern computer architecture, found its first concrete expression with this technology.

Monte Carlo Method

The history of the Monte Carlo method begins in the unique context of the Manhattan Project during World War II. Researchers at Los Alamos were grappling with nuclear physics calculations of unprecedented complexity. The lack of suitable tools stimulated their mathematical imagination.

As early as the 1930s, Enrico Fermi, an Italian physicist who became a naturalized American citizen in 1945, experimented with statistical sampling techniques to solve his neutronics equations. These early attempts remained confidential, lacking computers capable of processing large volumes of calculations.

The true birth of Monte Carlo occurred in 1946. Stanislaw Ulam, a Polish mathematician, was recovering from an illness while playing solitaire. A question nagged at him: how to precisely calculate the odds of winning a game? He hit upon the idea of simulating a vast number of games to obtain a reliable statistical estimate. This brilliant insight met with the enthusiasm of John von Neumann, who programmed the first simulations on ENIAC in 1947. For these calculations, he created a

powerful mathematical technique for generating pseudo-random numbers, called *middle-square digits*, to estimate numerical values and solve complex problems. The name "Monte Carlo" arose from a joke by Nicholas Metropolis, a playful reference to Ulam's uncle, a compulsive gambler at the Monaco casino.

Between 1946 and 1947, during an extended ENIAC outage, Enrico Fermi designed FERMIAC, a remarkable mechanical device capable of simulating neutron diffusion according to the Monte Carlo principle. This invention testifies to the immediate enthusiasm of physicists for this radically new approach.

A conceptual leap occurred in 1953 with the algorithm devised by Metropolis and his team, including physicists Marshall and Augusta Mici Rosenbluth, as well as Edward Teller. Their method exploited Markov chains to generate samples following a given distribution and explore complex spaces. In 1970, Wilfred Keith Hastings enriched this approach, resulting in the Metropolis-Hastings algorithm, a current pillar of modern computational statistics.

The 1980s saw inspired variants flourish. The simulated annealing algorithm, designed by Kirkpatrick, Gelatt, and Vecchi in 1983, adapted Monte Carlo to combinatorial optimization by drawing inspiration from the controlled cooling of metals. A year later, the Geman brothers applied Gibbs sampling to image processing, opening new horizons in computer vision.

The early 1990s marked the triumph of Markov chain Monte Carlo methods thanks to the foundational work of Gelfand and Smith. These advances revolutionized Bayesian inference and transformed everyday statistical practice. This prolific decade also saw the emergence of Peter Green's reversible jump algorithm in 1995, which enabled exploration of variable-dimension spaces, as well as the perfect sampling of Propp and Wilson in 1996, guaranteeing exactly distributed samples.

The evolution of Monte Carlo illustrates the fruitful dialogue between theory and technology. From the first calculators to today's supercomputers, each leap in power has exponentially increased the potential of these methods. Their fields of application have diversified: statistical physics, molecular chemistry, quantitative finance, machine learning, and 3D graphics with Eric Veach's Metropolis light transport in 1997.

Historical irony: this technique initially developed for nuclear weapons now serves modern medicine. Monte Carlo is used in radiotherapy to simulate with unparalleled precision the interactions between radiation and biological tissues. Its tremendous scientific impact earned it designation as one of the ten most influential algorithms of the 20^{th} century by the

Society for Industrial and Applied Mathematics.

Williams Tube

The postwar period witnessed a frantic quest to solve the thorny problem of data storage in early electronic computers. While Presper Eckert and John Mauchly toiled over mercury delay lines in Philadelphia and Jay Forrester began his research on magnetic core memory at MIT, Frederic Williams charted a different course at the Telecommunications Research Establishment in Great Malvern.

Williams's visit to the ENIAC project in 1946 gave him two key insights: the concept of the stored program and the vital need for fast electronic storage. His background in radar circuits during the world war led him, along with his doctoral student Tom Kilburn, to explore cathode ray tubes as memory devices. Computers required approximately 32,000 binary bits organized into 1,024 words of 32 bits. Electronic flip-flops would have demanded over 64,000 vacuum tubes, making any reliability impossible.

Williams and Kilburn's breakthrough rested on a subtle physical phenomenon: electron bombardment on the screen created a positive potential well. A second bombardment nearby generated secondary electrons that partially filled the first well. Detection of this filling during a subsequent scan determined whether a 1 or 0 had been written. A metal plate against the screen captured these minute charge variations.

Five variants of this principle emerged. The `dot-dash` system encoded a 0 as a dot and a 1 as a dash. The `dash-dot` inverted this logic. The `defocus-focus` and `focus-defocus` systems played with beam sharpness. The `anticipation` system exploited a predictive signal during charge destruction.

In early 1947, now at the University of Manchester, Williams and Kilburn stored their first stable bit. By autumn, they maintained 1,024 digits for several hours. Their stroke of genius was introducing a regeneration mechanism. Since the charge persisted for only a few tenths of a second, the content was constantly read and rewritten—a technique that survives in our current DRAM memories.

June 1948 saw the birth of the Manchester Baby, the nickname for the Small-Scale Experimental Machine. This prototype used three tubes: one for main storage (32 words), one for the accumulator (calculation

register), and a final one for control (program counter and current instruction). Its architecture allowed random access to memory, whereas delay lines imposed sequential access. Its instruction set was limited to seven operations: `JMP` (jump), *JRP* (relative jump), `LDN` (load negative), `STO` (store), `SUB` (subtract), `CMP` (compare), and `STP` (stop).

Two new versions emerged in 1949. The first offered 128 words of 40 bits with an electronic multiplier. It served for sophisticated calculations such as verifying the Riemann hypothesis and testing the primality of Mersenne numbers. The second combined Williams memory with a magnetic drum storing an additional 40,960 bits, thus creating a two-level memory hierarchy that foreshadowed the organization of future computers.

IBM seized this technology in 1951, acquired a license, and launched the manufacture of Williams tubes for its 701, the company's first commercial computer. This industrialization required remarkable achievements; the company created ultra-clean environments ahead of its time. Personnel wore orlon smocks to avoid textile particles. Sticky mats trapped dust from shoes. These practices anticipated those later adopted for semiconductors.

Delivered in 1951, the final version of the Manchester computer aligned eight Williams tubes each storing 1,024 bits (total memory of 10,240 bits), supplemented by a magnetic drum of 150,000 bits. It processed instructions in 1.2 milliseconds and performed multiplications in 2.16 milliseconds. Its applications touched biology, optics, crystallography, chemical engineering, and even chess.

The technical constraints of Williams tubes proved numerous. The quality of the phosphorescent screen was critical: the slightest imperfection (carbon particle or microscopic hole) compromised storage. The beam required perfect uniformity across the entire surface. The scanning speed necessitated absolute precision control.

Despite these challenges, the technology equipped several machines of the 1950s: the Ferranti Mark I (commercial successor to the Manchester machine), the IBM 701, and the MANIAC I at Los Alamos. By mid-decade, it yielded to Forrester's magnetic core memory, which was more reliable and denser. In 1998, to celebrate the fiftieth anniversary of the Manchester Baby, Chris Burton led the construction of a functional replica, shedding new light on the problems faced by the 1948 team.

Association for Computing Machinery

In 1947, a small group of pioneers, including E. C. Berkeley, R. V. D. Campbell, J. H. Curtiss, H. E. Goheen, J. W. Mauchly, T. K. Sharpless, R. Taylor, and C. B. Tompkins, decided to create an informal organization bringing together people interested in the new computing and reasoning machines. On June 25, 1947, they circulated a note proposing the creation of an *Eastern Association for Computing Machinery*.

On September 15, 1947, Columbia University hosted these 78 pioneers. Curtiss took the presidency, Mauchly the vice-presidency, and Berkeley, the driving force of the group, assumed the secretariat. Ambitions grew quickly. By January 1948, the *Eastern* qualifier disappeared in favor of a national vision.

The organization took shape around four geographical hubs: Boston, New York, Philadelphia, and Washington. The structure was articulated around two levels of exchange: local and national meetings. The sections, initially conceived as restricted activity centers, expanded rapidly. Truly local *chapters* would only appear later.

The 1950s saw outreach to other scientific communities. The ACM forged ties with the American Institute of Electrical Engineers and the Institute of Radio Engineers. These collaborations gave birth, in 1961, to the American Federation of Information Processing Societies, the voice of the United States in the international arena. Rapid growth disrupted the informal operations of the early days. In 1953, the evidence was clear: a volunteer secretary was no longer sufficient. The New York Academy of Sciences lent support to provide the necessary staff and facilities, before the association established its own headquarters in New York.

Intellectual influence was built through publications. The *Journal of the ACM* came into being in 1954, followed by the *Communications of the ACM*. These journals shaped the academic landscape, set standards of excellence, and structured a field in the midst of definition.

With the 1960s came the Special Interest Groups, true communities within the community. These specialized groups became idea laboratories, organized their own scientific meetings, distributed their bulletins, and gave rise to fertile subdisciplines.

The ACM then tackled the need for education in this new science. In 1968, under the aegis of William F. Atchison, the curriculum committee published its recommendations, a founding text for computer science education.

Honors came to recognize the great builders of this science. The Turing Award, created in 1966, established itself as the supreme distinction. Alan

J. Perlis inaugurated it, before Wilkes, McCarthy, and so many other major figures. Subsequently, other awards followed: the Distinguished Service Award (1976) and the Grace Murray Hopper Award (1971).

Faced with the proliferation of programs and concerned with contributing to social issues, the ACM created an accreditation committee in 1968. It developed a code of ethics and set up a mediation system for problems related to the computerization of society.

The 1970s saw the flourishing of new services such as national lecture series, continuing professional education, or the documentation center at headquarters. The association thus responded to the expectations of a community that was growing day by day.

As early as the 1960s, the ACM became concerned with preserving the history of computing. Grace Hopper and Robert Bemer initiated a campaign to preserve historical computing documents. These efforts led to the creation of ACM archives, initially housed at the Moore School of Electrical Engineering at the University of Pennsylvania.

Over time, the ACM evolved into a global reference. Its arsenal of publications expanded: dozens of specialized journals, thousands of conference proceedings, a digital library rich with more than 430,000 articles. Each year, more than 275 events bring together its 118,000 members from over a hundred countries. The digital library is an accessible treasure, bringing together all publications since 1951. Bibliometric analysis tools, author and institution profiles, and usage statistics enrich this scientific goldmine.

For three-quarters of a century, the ACM has shaped computer science as a discipline and as a profession. Through its publications, conferences, training programs, and awards, it serves as a compass in the digital world.

Transistor

The winter of 1947 saw the birth of an innovation that would transform our relationship with technology. At Bell Labs, three men—John Bardeen, Walter Brattain, and William Shockley—designed the transistor, a tiny device whose significance no one yet suspected. Their work would earn them the Nobel Prize in Physics in 1956.

This discovery was no accident. As early as the 1930s, Mervin Kelly, director of research at Bell Labs, had anticipated the limitations of vacuum tubes that equipped all electronic devices. These tubes, as fragile as

glass, consumed enormous amounts of energy and occupied considerable space. Kelly sought an alternative and invested in semiconductors, materials halfway between conductors and insulators, then poorly understood by the scientific community.

To explore this path, Kelly patiently built a multidisciplinary team. He recruited William Shockley and Dean Woolbridge in 1936, then James Fisk and Charles Townes three years later. Walter Brattain, already at Bell Labs since 1929, joined the venture. The team held seminars to decipher semiconductor physics. Their initial experiments with copper oxide came to nothing. At the dawn of the war, Russell Ohl proposed focusing on silicon.

The world conflict accelerated this research. Radars required frequencies too high for vacuum tubes. Germanium and silicon were ideal candidates, especially since the military funded sophisticated purification techniques for these materials. In 1945, Kelly restructured his laboratories and created a section dedicated to solid-state physics under Shockley's leadership. The team expanded to include Walter Brattain, Gerald Pearson, and the newly recruited John Bardeen.

The autumn of 1947 saw a remarkable acceleration in their work, a period Shockley would later call the "magic month". On November 17, Brattain and Ralph Gibney discovered that a semiconductor device immersed in an electrolyte produced an amplification effect. Experiments followed in rapid succession with germanium. On December 16, 1947, Bardeen and Brattain achieved significant amplification with a device featuring two closely spaced contact points. The transistor had just been born.

The story unfolded in the following weeks. Between December 1947 and February 1948, Shockley developed the theory of the junction transistor, which would prove industrially viable. He understood that amplification came from an injection of minority charge carriers and envisioned a sandwich of three semiconductor layers. On February 23, 1948, an experiment by John Shive brilliantly validated this hypothesis.

Manufacturing the first junction transistor as Shockley had imagined required two additional years of development. Gordon Teal and Morgan Sparks succeeded in April 1950, thanks to metallurgical innovations. In November 1950, Shockley published *Electrons and Holes in Semiconductors*, an instant bible for scientists in the field.

The transistor's first applications surprised its creators. AT&T's telephone networks, though responsible for the invention, only adopted this technology in the late 1950s. Transistors, however, conquered portable radios, hearing aids, and especially the military. The Minuteman missile

program (1958-1962) integrated transistors into its computer guidance system.

Transistorized computers emerged in the early 1950s. Bell Labs designed TRADIC, the first airborne transistor computer. IBM launched its transistorized calculator, the model 608, in 1957. Philco commercialized its TRANSAC computers based on its surface-barrier transistor technology. Texas Instruments, General Electric, and RCA offered transistors specifically designed for computing.

This technological transformation also revolutionized production methods. Fairchild Semiconductor, founded in 1957 by eight defectors from Shockley's company, including Gordon Moore and Robert Noyce, invented new manufacturing methods. These innovations gradually shaped what would become Silicon Valley, the global epicenter of the semiconductor industry.

Cybernetics

At the heart of the turbulence of World War II, while the great powers mobilized their scientists, a new science was born: cybernetics. This discipline, which could be described as transdisciplinary, emerged through the work of Norbert Wiener, a mathematician at MIT.

In 1940, facing the threat of German bombers, Wiener tackled an exceptionally complex technical problem: how to shoot down aircraft flying at 600km/h at dizzying altitudes? Through a trajectory predictor called the "AA predictor", designed in collaboration with Julian Bigelow.

Wiener's brilliant insight was to consider the pilot-aircraft ensemble as a single system whose behaviors obeyed certain predictable statistical laws. This approach broke radically with the traditional mechanistic view. To implement this idea, Wiener drew upon three emerging technologies: radar, servomechanisms, and analog computers.

But Wiener's contribution extended far beyond the initial military context. From this work emerged a fundamental reflection on control and communication mechanisms, both in artificial and natural systems. The term "cybernetics", from the Greek *kubernêtikê* (art of steering), was chosen to convey this unifying vision.

At the core of cybernetic thought lie several foundational concepts: feedback, information as a measurable quantity, and self-regulating systems. Wiener postulated that a cybernetic system, whether living or ar-

tificial, must necessarily possess an internal model of its environment to interact effectively with it. This model guides the extraction of relevant information through sensors, their processing according to internal rules, then action on the environment through effectors, all in an uninterrupted loop.

The publication of *Cybernetics: Or Control and Communication in the Animal and the Machine* in 1948 marked the official advent of this discipline. The work generated immediate enthusiasm in international scientific circles, as evidenced by its simultaneous publication in the United States and in France by Hermann.

Cybernetics, however, experienced divergent destinies across countries. In the United States, under the aegis of MIT and researchers like Jay Forrester, it became firmly rooted in engineering sciences, giving birth to concrete applications such as the SAGE air defense system. In France, it took more the form of theoretical and philosophical reflection, sometimes at the expense of its practical applications.

The famous Macy Conferences, held between 1946 and 1953, played a decisive role in the expansion of this discipline. These extraordinary gatherings brought together brilliant minds from diverse backgrounds: mathematicians, engineers, neurologists, psychologists, and anthropologists mingled to explore the multiple ramifications of cybernetic thought.

The postwar period saw Wiener distance himself from military research, horrified by the destruction at Hiroshima and Nagasaki. He directed his work toward civilian and medical applications, seeing in cybernetics a tool to combat social entropy.

Modern computing bears the indelible mark of this thinking. The concepts of feedback, information processing, and modeling of complex systems now constitute the conceptual foundation of our digital systems. The attention Wiener paid to human-machine interfaces finds a striking echo in our current concerns.

Cybernetics nourished several adjacent fields: information theory developed by Claude Shannon, game theory formalized by John von Neumann, and the first reflections on artificial intelligence. These parallel currents progressively built the theoretical edifice upon which 21st-century computing rests.

In the 1960s-1970s, the limitations of cybernetics became apparent. Its claim to provide a universal explanatory framework came up against the irreducible complexity of many natural and social phenomena. Its totalizing ambitions were scaled back, but its fundamental concepts retained their relevance for understanding information and control systems.

In our information-saturated digital world, where complex systems in-

terlock with one another, Wiener's insights retain remarkable freshness. While its field of application has become more precise since then, his vision of a science of control and communication has lost none of its explanatory power. The principles he identified continue to illuminate our understanding of computer systems and their interaction with human users, in an incessant dialogue between human and machine that he had foreseen as early as the 1940s.

Magnetic Drum

We can say with humility and without judgment: the computing world has known strange machines in its early days. Among them, a rotating cylinder dominated three decades of technological evolution. Gustav Tauschek, an Austrian inventor, created in 1932 an object that would change the face of data storage: the magnetic drum.

A metal cylinder coated its surface with a ferromagnetic layer capturing binary information. All around the drum, read-write heads captured and inscribed data in the form of tiny magnetized dots. The constant rotation of the cylinder provided access to stored information, in a precise and regular mechanical ballet.

The first versions of the 1940s already showed the technical limitations of the era. Recording density reached only 50 bits per inch, with about twenty tracks per inch. The adjustment required watchmaker precision, so much so that technicians adjusted the gap between heads and surface to within a thousandth of an inch, using differential screws and high-precision machining.

The IBM 650, marketed from 1954, perfectly embodied this technology. Its drum, made of a cobalt-nickel alloy, measured 4 inches in diameter by 14 inches in length. It spun at the dizzying speed of 12,500 revolutions per minute on ultra-precise ball bearings. Its recording density reached 50 magnetic dots per inch, pulsing at 128 kHz. The data organization followed a particular logic: parallel storage at the bit level within each digit, but serial for the digits of a single word. Fifty words were inscribed on one circumference of the drum, each containing ten decimal digits and a sign. The system coded the minus sign by the digit 8, and the plus sign by the digit 9. A space equivalent to one digit separated each word, and five parallel tracks encoded each value.

David Macklin, a programmer at Republic Aviation in 1957, recounts the constraints imposed by this architecture. Developers had to account

for the actual physical position of data on the drum to optimize access times. With its 2,000 addressable positions distributed across parallel tracks, the machine imposed a constant intellectual gymnastics. The programmer calculated the location of the next instruction or let the SOAP assembler handle it. On average, three or four executions per complete revolution completed the work, depending on the previous data positions.

The technology evolved significantly with the invention of the hydrodynamic bearing. This advance, developed by IBM for the SAGE air defense computer, made it possible to achieve densities comparable to magnetic tapes without resorting to the complex mechanical adjustments of the past.

The legacy of the magnetic drum survives in certain modern UNIX systems, where `/dev/drum` designates the virtual swap device. This name comes directly from the historical use of the drum as memory paging support.

The arrival of magnetic core memories signaled the gradual decline of the drum as primary memory. However, these rotating cylinders remained in use as memory extensions into the 1960s, thanks to their reliability and moderate cost compared to contemporary alternatives.

The electronic heart of the magnetic drum beat to the rhythm of about 2,000 tubes, primarily models 5955, 6211, 12AY7, 6AL5, 2D21 and 5687. Types 6211 and 5965 were similar to the 12AV7 but met IBM's specific acceptance tests. Nearly 3,600 crystal diodes completed the logic circuits. The power supply consumed 16 to 18 KVA at 208V, 60 cycles, single-phase. Selenium rectifiers provided direct current, avoiding the need for complex electronic regulation.

Beyond simple storage, the magnetic drum shaped the design of early computers and the method of writing programs. Developers delved into the physical reality of storage to optimize their code, creating a hardware-software symbiosis characteristic of the 20^{th} century. Its limitations in terms of access and capacity stimulated the search for alternatives. This work led to magnetic disks, faster and larger. IBM's RAMAC, the first commercial hard disk marketed in 1956, marked the beginning of the end for the magnetic drum as a primary storage device.

Claude Shannon's Information Theory

It is difficult to imagine our digital universe without the theoretical breakthroughs of Claude Shannon, this brilliant-minded American. In his foundational 1948 text, *A Mathematical Theory of Communication*, Shannon established the mathematical foundations that underpin all our digital communications today.

The story begins at AT&T's Bell Labs in the 1940s. Telephone and radio were no longer novelties but maturing technologies. However, these systems lacked a solid theoretical framework. Shannon tackled this conceptual void. His genius lay in the intuition to approach information as a mathematical quantity. He invented the illuminating concept of information entropy, a measure that quantifies the uncertainty of a message. Instead of remaining vague, Shannon translated information into mathematical equations and formulas. He demonstrated that the higher the entropy, the more information the message contains. This was the revelatory idea that transformed our view of the world.

His channel coding theorem constitutes another decisive breakthrough. He proved that it is possible to transmit information over a noisy channel with almost zero probability of error. The only condition is not to exceed the channel's capacity. This discovery shed new light on the theoretical limits of our communication systems.

Shannon's concepts found concrete applications from the early days of computing. His work guided the creation of robust error-correcting codes. No modern storage or transmission system would function without them. From CDs to Wi-Fi, including satellite communications, Shannon's fingerprint is visible everywhere.

Data compression owes just as much to this visionary mathematician. The Huffman and Lempel-Ziv algorithms, which allow you to send photos or store movies, draw their basic principles from Shannon's work. Without him, our hard drives would be 10 times larger and our internet connections 10 times slower.

But Shannon's influence does not stop at the borders of computer science. His theory has spilled over into multiple disciplines. In psychology, it inspired models on cerebral information processing. Biologists use it to decode DNA and understand genetic transmission. Economists have drawn tools from it to model financial markets, with quantum physics appropriating certain "Shannonian" concepts.

Over time, his theory has branched out. Quantum information theory explores the novel properties of quantum systems. Algorithmic information studies the fundamental limits of computation and complexity. These

extensions testify to the robustness of Shannon's original ideas.

More than 75 years after its initial publication, this theory remains a guiding star for computer science and telecommunications in the 21st century. It still guides researchers in cutting-edge fields such as machine learning or data science. Who would have thought that an article published when computers occupied entire rooms would continue to illuminate the era of nanometric chips?

When you send a message that travels thousands of kilometers without error, when you compress a file to send by email, when you watch a streaming movie without interruption, you directly benefit from his discoveries. If our data travels through time and space without degrading, it is thanks to his equations. Pure mathematics always ends up finding practical applications, sometimes far beyond what their creators imagined.

Short Code

Short Code was born into a world where programming resembled electrical engineering more than code writing as we conceive it today. In the late 1940s, computing was in its infancy with monumental machines filling entire rooms for computing power that now seems laughable.

The story begins in the laboratories of the Moore School of Engineering at the University of Pennsylvania. It was there that John Mauchly, a physicist who had transitioned to electronics, worked from 1941 on a military project aimed at calculating ballistic tables. Faced with the limitations of calculators, he envisioned a fully electronic machine: ENIAC.

This colossal machine, commissioned in 1946, executed 5,000 additions per second—an unprecedented technical achievement. ENIAC, however, suffered from a flaw: the absence of a stored program. Each calculation required manually modifying circuit connections, a tedious task that could take several days.

To overcome this constraint, Mauchly and his collaborator J. Presper Eckert designed EDVAC in 1944, incorporating the concept of a program stored in memory. Their departure from the university in 1946, following disagreements over intellectual property, delayed the project, which was not completed until 1951. Mauchly then proposed, in July 1949, a more accessible language than machine code: Short Code, initially named Brief Code.

Short Code stood out for its conceptual simplicity. Rather than writing directly in binary, programmers used mathematical expressions. The language supported floating-point numbers, variables, conditional jumps, and calls to subroutines stored separately. The UNIVAC manual illustrated this innovative syntax with examples such as:

```
00 IXABC            I is set to the value X+(A*B*C)
01 BVII             Go to instruction 11 if B is positive
02 HRDW             Print the values of R, D, and W
```

This approach, though rudimentary by today's standards, represented a break from previous methods. Programs remained complex, however, and had to be manually translated into machine language—the compiler did not yet exist. Tests showed that code written in Short Code executed approximately 50 times slower than an equivalent program in machine language, arousing skepticism among many specialists. Despite these limitations, Short Code marked a decisive milestone. For the first time, the algorithm took shape independently of the machine, enabling the emergence of more advanced languages such as Fortran (1957) or Algol (1958). The concepts it introduced—mathematical expressions, variables, branching, subroutines—became the foundations of modern programming.

Mauchly and Eckert's work was not limited to Short Code. After ENIAC and EDVAC, they created BINAC (Binary Automatic Computer, 1949), the first operational stored-program computer, then UNIVAC I (1951), the first commercial computer in history. William Schmitt developed a prototype of Short Code on BINAC, later adapted to UNIVAC I, and finally to UNIVAC II. By founding the Eckert-Mauchly Computer Corporation in 1946, they were the first to attempt to commercialize these extraordinary machines, whose usefulness remained mysterious to the general public.

Short Code, by partially freeing developers from hardware complexity, helped democratize programming and accelerated the growth of the computer industry. Seventy years after Mauchly's work, the quest for expressive, safe, and efficient languages, such as Rust for example, remains at the heart of concerns for computer scientists of the 21st century.

EDSAC

In the aftermath of the Second World War, as Europe tended its wounds, a project was taking shape in the laboratories of Cambridge Uni-

versity. Maurice Wilkes and his team were designing a machine that would forever mark the history of automatic computation: EDSAC.

This automatic calculator with delay line storage, built between 1947 and 1949, bore no resemblance to the computers we know today. Bulky, noisy, power-hungry, EDSAC embodied the first practical application of von Neumann's theories on computer architecture, a conceptual framework that still governs our current machines.

In the post-war British context, where resources were sorely lacking, building such a device was quite an undertaking. Yet the project was part of a rich tradition, with the codebreaking work at Bletchley Park and Turing's ACE at the National Physical Laboratory. EDSAC's distinctive feature lay in its general-purpose nature which, unlike its predecessors dedicated to specific calculations, was designed to be adaptable to all types of mathematical problems.

Its memory constituted a fascinating technical feat. Imagine tubes filled with mercury where ultrasonic pulses propagate representing through their presence or absence the famous 0s and 1s of binary language. This "delay line memory" stored 1024 words of 17 bits each. The team had also designed miniature versions of these lines for the arithmetic unit's registers, a solution as ingenious as it was elegant despite its practical drawbacks.

The microprogrammed control unit, another major innovation, physically separated the decoding of instructions from their execution. This architecture, encoded on 17 bits with a single-address format, included conditional jumps, a hardwired multiplier, and shift operations. Numbers were represented with a sign bit and a fixed point, to handle either short integers or near-double precision numbers using two memory words.

In May 1949, EDSAC performed its first public demonstrations, thus becoming the first operational computer based on von Neumann architecture. With its modest 650 instructions per second and 2 kilobytes of memory (a million times less than our current phones), the machine found its place in Cambridge's scientific research.

The 1950s saw EDSAC at the heart of scientific advances. Rosalind Franklin used it for her work on DNA structure, while researchers in astronomy, economics, or linguistics trained in its use. A genuine computing community was emerging around this pioneering machine.

On the software front, David Wheeler created the first assembler on EDSAC in 1951, replacing tedious binary codes with mnemonic instructions. A few years later, Alick Glennie's Autocode language foreshadowed high-level languages like Fortran or Algol. Libraries of reusable subroutines appeared, initiating the structured programming methods of the

future.

EDSAC inspired other British universities for their own machines, namely EDSAC 2 at Cambridge, MOSAIC at Manchester. Across the Atlantic, the University of Illinois developed ILLIAC, directly derived from plans generously shared by Wilkes and his team.

EDSAC's decommissioning in 1958, after a decade of loyal service, marked the end of an era but not that of its legacy. Without knowing it, EDSAC users were already manipulating the fundamental concepts we use daily: stored-program in memory, sequential architecture, instruction set, and subroutines.

Application Programming Interface

Application Programming Interfaces, known by the acronym API, have shaped our digital environment. Their history begins in post-war England, within the laboratories of Cambridge. Maurice Wilkes and David Wheeler, two little-known pioneers, developed a modular program library for the EDSAC computer. These British researchers stored their creations on punched tapes in a simple metal cabinet, accompanied by a catalog detailing their use. This modest arrangement constitutes the first documented trace of an API.

Wilkes and Wheeler formalized their work in 1951 in the book *The Preparation of Programs for an Electronic Digital Computer*. This book marks the conceptual birth of standardized interfaces between computer components, well before their current denomination.

The term "API" made its first scientific appearance in 1968 in the work of Ira Cotton and Frank Greatorex. Their paper "Data structures and techniques for remote computer graphics", presented at an AFIPS conference, described ways to abstract the specifics of graphics peripherals. Fortran programmers then used subroutine calls to free themselves from hardware constraints.

The 1970s saw the concept extend to data systems. In 1974, CJ Date published "The Relational and Network Approaches: Comparison of the Application Programming Interface", integrating these interfaces into the ANSI/SPARC architecture of database management systems. This scientific contribution considerably broadened their scope of application.

The network computing of the 1980s transformed developers' needs, who now faced accessing libraries on different machines. APIs evolved to

support remote procedure calls. This period saw the birth of interfaces promoting compatibility between disparate platforms.

The Internet revolution of the 1990s pushed APIs toward data exchange between applications via standardized protocols. Carl Malamud proposed in 1990 a definition that has endured through the ages: a coherent set of services made available to a programmer to accomplish specific tasks. This decade laid the groundwork for emerging web services.

The year 2000 was significant with Roy Fielding's doctoral thesis, *Architectural Styles and the Design of Network-based Software Architectures*. This work established REST as the reference protocol for communications between connected systems. Also in 2000, Salesforce presented the first modern API at the IDG Demo conference, thus inaugurating a new technological era.

During this decade, online services multiplied. Salesforce, eBay, and Amazon innovated by exploiting HTTP to disseminate machine-readable data, formatted in JSON or XML, via web APIs. From small players to industrial giants, companies massively adopted the model based on the cloud and APIs. Amazon played a pioneering role by requiring that all shared digital resources have an API.

From 2010 onward, the explosion of social networks catalyzed API evolution. Organizations sought cost-effective solutions to create adaptable applications. These interfaces simplified integration with external services such as payment processors or customer relationship systems. The arrival of Kubernetes around 2015 accelerated the transition toward distributed architectures composed of autonomous microservices, each with its own API.

The 2020 health crisis intensified our dependence on web services, multiplying the use of programming interfaces. Their role extended to connected objects and the creation of artificial intelligences. Organizations deployed their applications across different cloud providers, placing APIs at the center of communication between services.

Currently, evolution focuses on two major axes. Security is a priority given the growing vulnerabilities of distributed systems. The *zero trust* principle is becoming standard, systematically verifying authentication and authorizations, including for internal requests. Rate limiting mechanisms protect against denial-of-service attacks, while observability detects and resolves incidents.

API governance establishes in parallel rules for their design, deployment, and maintenance within organizations. The objective is to guarantee their consistency, security, and scalability in harmony with business strategy.

APIs have revolutionized the way developers create applications, with increasingly sophisticated abstractions. What began as a simple exchange between two computers in a British university laboratory today forms the backbone of 21st century digital systems, particularly in the distributed universe of cloud computing.

Whirlwind I

At the heart of MIT, the Whirlwind I project began in 1944, driven by the US Navy. Initially, the ambition was modest: to build a flight simulator for pilot training. Jay Forrester, a researcher at the servomechanics laboratory, took charge of the project the following year. The initial idea of an analog machine was quickly discarded in favor of a digital approach, which the Navy validated in 1946.

This decision radically transformed the project's scope. Instead of a simple simulator, MIT was committing to creating a universal digital computer, capable of meeting the simulator's requirements while opening up prospects for other scientific applications. The central machine became fully operational in 1951.

Whirlwind I's primary purpose shaped its design. From the outset, the team conceived it as a real-time system. This approach required ultra-fast memory. The chosen solution was cathode ray tube memory, the fastest available at that time. A 1952 report noted, however, that this memory remained "the most important factor affecting the Whirlwind I system's reliability".

To address this problem, the team developed rigorous systematic testing procedures, aimed at detecting hardware failures before they could compromise calculations. At the same time, the military's growing use of the machine demanded flawless reliability. In the tense Cold War climate, the Whirlwind, whose funding shifted from the Office of Naval Research to the US Air Force, became integrated into the American defense system.

Its production version, the AN/FSQ-7, became a component of the SAGE airborne surveillance system. Dissatisfied with the limitations of cathode ray tubes, the researchers sought alternatives. Towards the end of the 1940s, Jay Forrester and other scientists explored the use of magnetic cores. William Papian, a team member, mentioned in his notes Harvard's work on "static magnetic delay lines".

The summer of 1953 saw the installation of core memory on the Whirl-

wind. A project report highlighted two advantages: “magnetic core memory has two major advantages: (1) greater reliability with consequent reduction in maintenance time devoted to memory; (2) shorter access time (core access time is 9 microseconds versus approximately 25 microseconds for tubes), which increases the computer’s operating speed.” This innovation made the Whirlwind the first large computer equipped with this technology, which dominated the market until the 1970s.

Whirlwind I’s architecture was based on a stored-program model with 16-bit words. Its power reached 20,000 instructions per second, with random access memory of 2048 words. It used single-address instructions and fixed-point numbers in 9’s complement. Its structure was divided into three parts: a storage unit with 32 electrostatic tubes, an arithmetic element for calculations, and a control unit orchestrating the whole. The electronic components relied primarily on flip-flops to store binary states and gate tubes to direct pulses.

The central concern was speed. The initial objective of 50,000 operations per second was not achieved, but the actual 20,000 operations already represented a feat. Execution times—3 microseconds for addition, 16 for multiplication—far exceeded those of contemporary machines. The instruction set, reduced to about thirty commands, included conditional jumps making the machine self-modifiable according to results. Specialized instructions, such as automatic coordinate transformations, further accelerated calculations.

Whirlwind I’s human-machine interface was bold. Beyond the photoelectric punched tape reader and conventional typewriters, the machine had cathode ray screens—alphanumeric or vector—with photographic capture of results. This advanced system gave programmers immediate visual feedback on their work.

The programming cycle prefigured what we know today. The programmer first broke down their problem into elementary steps, translated them into instructions, which operators entered via special equipment. This code was converted to intermediate language, then to binary, before being stored on punched or magnetic tape. The controller then loaded the instructions sequentially from memory, decoded them, and sent the signals necessary for their execution.

Whirlwind I’s applications touched numerous domains: economics, magnetism, radar, mechanical design, signal processing. But its most enduring legacy is undoubtedly its advances in real-time computing applied to industrial and air traffic control. This extraordinary machine contributed to improving computer architecture and accelerated the development of computing, both hardware and software.

IV

1950

The Silent Birth of a Computed World

In the aftermath of 1945, while Europe tended to its wounds, a new world order was taking shape. The smoldering ruins of World War II gave way to an ideological confrontation between two giants: the United States and the USSR. This rivalry, less bloody but equally fierce, would shape technological development throughout the 1950–1960 decade.

Washington and Moscow poured colossal funds into their military and space programs. Bombs became more destructive, missiles more accurate, aircraft faster. Each technical advance demanded more complex calculations. Old abacuses and slide rules were no longer sufficient. Machines capable of rapidly processing equations with multiple variables, simulating nuclear explosions, or calculating ballistic trajectories in hours rather than months were needed.

The war had proven the utility of computing machines. The breaking of the Enigma cipher by the British at Bletchley Park had shown that a well-designed machine could solve problems deemed unsolvable. In the United States, the National Security Act of 1947 created the CIA and NSA. These new agencies fed on data—intercepted messages, spy reports, or aerial reconnaissance photographs. Encryption and decryption became major strategic issues.

Far from military concerns, civil society was undergoing an unprecedented transformation. The Western economy was experiencing meteoric growth. Companies were expanding, administrations growing larger. Everywhere, papers were accumulating: pay slips, purchase orders, invoices, letters, or medical records. Traditional methods of filing and archiving were showing their limits. Herman Hollerith's punched cards, already dated, no longer met modern needs.

The baby boom was transforming demographics. Millions of children were born each year, requiring more efficient public management systems. In Europe, new social welfare policies generated mountains of forms. In the United States, the Social Security Administration was overwhelmed with files. How could all these citizens, their contributions, their entitlements, be effectively tracked?

Once reserved for the wealthiest, banking services were becoming democratized. The number of accounts exploded, transactions multiplied. Tellers and accountants, armed with their ledgers and mechanical calculators, struggled to keep pace. The financial sector demanded faster and more reliable solutions.

In factories, automation was progressing by leaps and bounds. Assembly lines were adopting electronic control systems. Norbert Wiener's cybernetic theory was finding concrete applications. Feedback principles, used to guide missiles during the war, now served to regulate industrial production.

A fruitful alliance was forming between universities, industrialists, and the military. MIT, Harvard, the University of Pennsylvania, as well as private laboratories like Bell Labs or IBM, became crucibles of innovation. A new generation of researchers and engineers tackled the problems of automatic computation. This cooperation model, which President Eisenhower would later call the military-industrial complex, considerably accelerated technical progress.

The new profession of programmer was emerging. These specialists translated concrete needs and problems into instructions comprehensible to machines. Notably, many women held these positions, such as Grace Hopper, who worked on the first compiler. This marine mathematician was paving the way for more intuitive programming methods, freeing users from the complexity of machine languages.

The proliferation of computing machine manufacturers created an unexpected compatibility problem. Each constructor developed its own standards. Machines could not understand each other; programs written for one model did not work on another. The first standardization efforts appeared timidly, harbingers of an issue that would leave a lasting mark on

computing history.

The media was discovering these mysterious machines. Popular newspapers evoked with fascination or concern these "electronic brains." Hollywood seized upon the subject, sometimes presenting the computer as a marvelous assistant, sometimes as a threat to humanity. These contradictory representations shaped public perception of a still poorly understood technology.

Automation raised social concerns. Certain professions seemed threatened with extinction, and unions were alarmed by the risks of massive unemployment. In academic circles, the first debates on artificial intelligence were taking shape, particularly after the Dartmouth conference in 1956 where the term was officially adopted.

The global telephone network was expanding, and new cables were crossing the oceans, connecting continents. Communication quality was improving and costs were decreasing. Remote data transmission was becoming technically possible, opening the way to future computer networks.

Claude Shannon's theoretical work on information was finding its first practical applications. His mathematical concepts of coding and error correction made communications more reliable. The bit was gradually becoming established as the fundamental unit of information measurement.

The transistor, invented in 1947 at Bell Labs, was gradually replacing vacuum tubes. Smaller, more reliable, less power-hungry, it enabled the design of more compact devices. In the San Francisco region, young companies specializing in semiconductors were establishing themselves in what would become Silicon Valley.

The natural sciences were adopting these new computational tools. Astronomers, physicists, chemists discovered the possibilities offered by automation. Emerging disciplines like molecular biology benefited from these increased processing capabilities. Meteorology began using computing machines to improve its forecasts, with modest but promising results.

This 1950–1960 decade marked the beginning of a social transformation. Technical innovations were inextricably intertwined with economic and cultural changes. The impact on our societies remains palpable in the 21st century, as we navigate an increasingly digitized and interconnected world.

Hamming Code

Richard Wesley Hamming was working at Bell Labs on the Model V, one of the first relay computers intended for the Aberdeen Proving Ground, when he observed a challenging reality: this machine composed of 8,900 mechanical relays experienced two to three failures daily. This represented approximately one failure every two to three million operations. The problem lay not so much in detecting errors, since the self-checking circuits handled this adequately. The difficulty arose when the machine operated unattended, particularly at night and during weekends. Calculations would come to an abrupt halt. Sometimes the computer would resume work on a new problem, without however guaranteeing the continuity of previous operations.

This very concrete technical situation pushed Hamming toward deeper reflection. He sought to go beyond simple error detection to create a system capable of automatically correcting these failures. A true break from previous approaches.

The particularity of the technological context adds depth to his approach. Computers were beginning to supplant telephone exchanges as the most complex information processing systems. Yet these two architectures presented a key difference. A telephone exchange has many parallel paths relatively autonomous from one another. An error on one of these paths affects only a limited number of calls. Moreover, errors escaping automatic detection end up being reported by customers themselves. A computer, on the other hand, generally follows a single path where information passes many times through the same components before reaching the final result. A single undetected error risks invalidating all subsequent calculations.

Faced with this problematic structure, Hamming developed an innovative mathematical approach. His starting point was information encoded in binary form. This representation naturally aligned with open or closed relays, flip-flop circuits, Morse code dots and dashes, or punched tapes. His true breakthrough lies in adding supplementary bits to the data, called parity bits, which not only detect errors but locate them precisely.

The heart of Hamming code lies in a deceptively simple principle: each parity bit checks a specific set of positions in the message. Through judicious choice of these sets, one can identify exactly which position has been altered. As an example, in a seven-position code with four dedicated to information, three parity bits suffice to correct any single error. This redundancy, defined as the ratio between the total number of bits used and the minimum number necessary to transmit the information, constitutes

the price to pay for guaranteeing data integrity.

For the first time, with Hamming's publication in the *Bell System Technical Journal* in 1950, a rigorous theoretical framework addressed the question of digital system reliability. Hamming notably introduced a geometric representation of error-correcting codes, where bit sequences are points in an n-dimensional space. The distance between two points, measured by the number of positions where their coordinates differ, becomes a powerful analytical tool for evaluating the detection and correction capabilities of codes.

This geometric approach enabled Hamming to prove the optimality of his codes in several significant cases. He demonstrated that no code can detect single errors with less redundancy than his construction, and established analogous results for single-error correction and double-error detection. These theoretical demonstrations confirmed that the proposed trade-offs between redundancy and correction capability achieve the best possible balance under the assumptions considered.

Richard Wesley Hamming envisioned other applications beyond the computer domain. Three particular situations derive notable benefit from his codes: unattended operation over long periods with minimal backup equipment, large tightly interconnected systems where an isolated failure risks paralyzing the entire installation, and signal transmission in the presence of noise when its reduction proves impossible or too costly.

His vision proved prophetic. The advent of electronics has certainly made components more reliable than mechanical relays, per operation. However, the considerable increase in speed and complexity of systems has kept the question of errors at the forefront of concerns. Hamming codes and their descendants populate today's computer memories, storage devices, and communication systems.

Hamming's legacy lives on in sophisticated error-correction techniques integrated into modern technologies. More complex codes now handle varied error patterns, all resting on the foundations laid by Hamming: the controlled addition of redundancy to detect and correct errors, and rigorous mathematical analysis of the capabilities and limits of error-correcting codes. His reflection on digital reliability gave birth to a renewed vision of computer design. Error management no longer appears as a secondary issue but as an integral part of system architecture. A perspective that retains all its relevance in our time.

Magnetic Core Memory

Magnetic ferrites revolutionized the world of computing in the mid-20th century. These small ferrite rings, capable of maintaining a stable magnetic state, marked the history of computer storage for nearly twenty years. Their principle relies on an elegant application of the physical laws of magnetic hysteresis: each core could be magnetized in one of two distinct states, thus perfectly representing the binary values 0 and 1.

An Wang, a researcher at the Harvard Computation Laboratory, described this technique as early as 1949, which retained information even after the power supply was cut off. The real breakthrough lay in manipulating this magnetic state through simple electrical pulses passing through the core.

Jay Forrester transformed this theory into concrete reality. At MIT, he demonstrated in 1951 the viability of a three-dimensional storage system based on these tiny rings. William Papian, working on the Whirlwind project, and Jan Rajchman at RCA Laboratory developed architectures during the same period that would be massively adopted. Their work led to the integration of this technology into computers such as Harvard's Mark IV.

The construction of such a memory relies on a two- or three-dimensional mesh of rings, each traversed by multiple conductors: X and Y selection lines, read/write wire, and sometimes an inhibit wire. To select a specific core, currents must be applied simultaneously to the corresponding X and Y lines. This coincident current trick solves the individual addressing problem while limiting the necessary control circuits.

Notably, reading erases the core's contents. The data must therefore be rewritten immediately if it is to be preserved. Despite this constraint, the advantages of this technology won over the industry through data retention without power, rapid access in a few microseconds, excellent reliability, and remarkable storage density for the time.

Manufacturing required extreme precision. Assembly remained largely manual, with threading wires through the minuscule rings demanding patience and precision. Semi-automated tests could verify about 200 cores per hour before integration. Once the matrix was assembled, replacing a defective core was a real headache.

Burroughs Corporation explored an interesting variant with rope memory. This particular configuration of cores and wires enabled permanent information storage, ideal for preserving programs and system routines. The Apollo program's guidance computer notably benefited from this technology.

Performance continued to improve. Access times dropped to less than 2 microseconds. Output signals, initially a few millivolts, reached several hundred millivolts, greatly simplifying detection circuits. Component standardization led to cost reduction. Magnetic cores thus found their place in large systems and in industrial and embedded computers.

The advent of semiconductor memories initiated this technology's decline in the late 1960s. Benefiting from the meteoric rise of microelectronics, these new memories offered unprecedented miniaturization possibilities. The transition nevertheless took time, with the non-volatility of magnetic cores remaining an asset for certain critical applications.

The conceptual legacy of magnetic core memory remains visible in modern computing. Matrix addressing, read-with-refresh, and hierarchical memory organization are all principles that remain relevant today. These ferrite rings made faster and more reliable computers possible, contributing to the computing explosion across numerous sectors of activity.

Magnetic Tape

In 1878, an American engineer named Oberlin Smith envisioned a recording system using metallic wire. Fascinated by Thomas Edison's phonograph, he published his idea in *The Electrical World* ten years later. A stroke of genius that opened a new chapter in the art of capturing sound.

Danish inventor Valdemar Poulsen transformed this vision into tangible reality. His "Telegraphon" of 1898 recorded signals on steel wire for the first time. Proof was established that magnetic recording worked. However, without electronic amplification, the device remained a prototype with no immediate commercial future.

The true metamorphosis occurred in 1930s Germany. Fritz Pfleumer invented his famous "sound paper" in 1928, a support coated with iron oxide. This discovery attracted AEG, which partnered with BASF to create the Magnetophon in 1935. On its cellulose acetate tape coated with iron oxide, the future of magnetic storage was already being written.

World War II accelerated technical evolution. In 1940, Walter Weber discovered AC bias, a process that enhanced sound quality remarkably. Reich radio broadcasts then featured recordings of such realism that the Allies mistook them for live transmissions. After the war, John Mullin, an American officer, brought two precious Magnetophons back to the United States. His demonstrations dazzled American industry. Ampex seized the

opportunity and launched its Model 200 in 1948, the first professional tape recorder made in the USA.

The 1950s structured and standardized the technology. In the computing world, magnetic tape established itself as the reference for audio recording and reigned supreme for data archiving. Its generous capacity and moderate cost made it the ideal solution for early *mainframes*. The 9-track tape standardized data exchange between different systems. Japan awakened to this technology, with Sony at the forefront.

A leap forward occurred in 1963. Philips unveiled its compact cassette, a miniature format compared to the imposing reels of the era. The Dutch giant made a master stroke by making its license freely available. The format exploded. Japanese manufacturers perfected the technology with innovative magnetic materials and ultra-sensitive read heads.

The 1970s-1980s witnessed major innovations. Chrome and then metal particles increased recording density. Read heads became more refined through Sendust and amorphous alloys. Magnetic tape revolutionized our relationship with sound and memory. It spawned unprecedented cultural practices such as portable listening, symbolized by Sony's Walkman in 1979.

The digital era of the 1980s transformed tape without killing it. For computing, DLT (Digital Linear Tape) appeared in 1984, followed by DDS (Digital Data Storage) in 1989. These formats offered ever-greater capacities. For 21st-century data centers, it remains a preferred archival solution, with an unbeatable cost per terabyte ratio.

The history of this brown ribbon demonstrates constant adaptation. Born for sound, it conquered computer data and professional archiving. In the age of cloud computing, tape retains a trump card against cyber threats: its capacity for physical isolation (*airgap*). This bulwark against ransomware makes it the ultimate protection for critical data. This resistance to attacks, coupled with its cost-effectiveness for storing immense volumes, ensures it a lasting place in modern architectures, where it complements hard drives and cloud solutions.

Ferranti Mark I

In the winter of 1951, the University of Manchester received from Ferranti a machine that would change everything: the Mark I. This British electronic calculator beat the American UNIVAC I by a few months and

went down in history as the first commercially available computer. This technical venture would never have happened without the foresight of Sir Ben Lockspeiser. This senior official at the British Ministry of Supply had grasped the importance of connecting university research with industry.

In reality, this machine extended the work of F.C. Williams and Tom Kilburn. These researchers had already designed a working prototype at the University of Manchester. The British government, eager to capitalize on this breakthrough, entrusted Ferranti with transforming the experimental into the commercial. While the overall architecture remained faithful to the original, Ferranti's engineers added their own touch: a revised instruction set, additional index registers (the famous B-lines), and novel features like a random number generator.

The Mark I's memory reflected the technological constraints of the era. On one side, electronic memory built around Williams tubes, fast but limited to 20,480 bits. On the other, a slower magnetic drum with 655,360 bits distributed across 256 tracks of 2,560 bits each—a dual system foreshadowing the hierarchical memory architecture of modern computers.

Programming the Mark I was almost an initiation rite. Alan Turing, the legendary figure in computing and then a researcher at Manchester, wrote the first manual for this machine. His approach consisted of bare-bones machine code, with no concessions to human cognitive limitations. Meanwhile at Cambridge, David Wheeler had created a rudimentary but far more accessible assembler for the EDSAC. Two visions clashed.

The interface with the outside world relied on five-hole punched tapes. The input stream reached 400 characters per second thanks to Ferranti's photoelectric reader. For output, a more modest punch (33 characters per second) and a teletype printer translated the calculation results. A loudspeaker completed the setup, alerting the operator with sound signals when attention was needed.

Mathematicians were among the first to harness this unprecedented computing power. They used it to explore Mersenne numbers or test aspects of the Riemann hypothesis. Physicists employed it for ray tracing calculations in optics. The data processing speed opened new possibilities for research, despite reliability that sometimes left much to be desired.

The Mark I's commercial venture also tells the story of the British computer industry's struggles. From 1951 to 1957, only nine units found buyers: seven Mark I Stars and two Mark Is. The narrow British market limited growth prospects. When IBM arrived in the United Kingdom with its rental model, competition became fierce for local manufacturers.

The Mark I's history can also be read through the complex relationships between Ferranti and government bodies, notably the National Re-

search Development Corporation (NRDC). While public funding made this venture possible, the often tense negotiations between Ferranti and the NRDC sometimes slowed the project. Sir Vincent de Ferranti fiercely defended his family company's interests, seeking to minimize financial risks in this uncertain venture.

The contrast between British technical potential and its commercial realization stands out in the statistics: in 1960, the United Kingdom had 240 computers in operation, compared to 5,400 in the United States. Yet the Ferranti Mark I occupies a special place in computing's pantheon. Its multi-level memory, indexing system, and overall architecture shaped the design of subsequent computers. As the first commercial computer, it embodied that moment when university research met industrial production.

This experience also redirected Ferranti's trajectory. Facing economic realities, the company gradually abandoned the mainframe market to focus on military applications, minicomputers, and later microprocessors. In 1963, Ferranti definitively left the central computer sector.

LEO I

In the mid-20th century, J. Lyons & Co dominated the British food market. Born from the visionary ideas of the Salmon-Gluckstein family in the late 19th century, this company managed a sprawling network of elegant tearooms, popular restaurants and industrial bakeries across the entire country.

Unlike other companies, Lyons cultivated an atmosphere conducive to technical innovation. The reason was simple: every day, thousands of modest transactions accumulated, each generating a minimal margin. Financial survival therefore required almost obsessive rigor in administrative management. This particular feature had prompted management to recruit a Cambridge mathematician, John Simmons, as early as 1923, capable of bringing a scientific dimension to their working methods.

In the spring of 1947, two company executives, Oliver Standingford and Raymond Thompson, flew to America. Their initial mission concerned the study of modern office equipment, but they stumbled upon information about ENIAC. These gigantic calculating machines, then confined to scientific laboratories, aroused their curiosity. They sensed that such automated calculations could radically transform Lyons' bureaucracy. Their intuition led them to Cambridge University, where Maurice Wilkes was

working on EDSAC. Upon their return, they wrote a bold report. Not content with suggesting the adoption of computing for their company, they proposed that Lyons build its own machine. This idea, which would seem absurd today for a food company, fitted perfectly with the company's philosophy of technical independence, accustomed to designing its own industrial tools.

Management approved the project and paid out £3,000 to support Wilkes' work, securing valuable technical assistance in return. To carry out this venture, the company hired Dr John Pinkerton, a brilliant engineer recommended by Wilkes. David Caminer, head of systems at Lyons, was given the task of developing applications and programming methods tailored to the company's specific needs.

Construction of LEO I (Lyons Electronic Office) began in 1949. This colossus occupied over 450 square meters and operated using approximately 6,000 vacuum tubes distributed across 21 imposing metal cabinets. Although based on EDSAC's architecture, LEO I incorporated substantial modifications to efficiently process commercial data. The engineers paid particular attention to input-output systems, combining punched cards and paper tape for data input, supplemented by printers and punches for outputting results.

On November 29, 1951, LEO I accomplished its first regular task, the weekly evaluation of Lyons bakeries' production. This program analyzed production figures for breads, cakes and pies from twelve different sites. It automatically calculated valuation by integrating raw material costs, wages and overhead expenses. No computer had ever performed such a management task before.

This first success encouraged the team to develop other applications: inventory management, calculation of optimal tea blends, organization of deliveries to different points of sale, automated invoicing. A new milestone was reached on February 12, 1954, when LEO I processed the payroll for 1,670 bakery employees, directly producing salary checks without a backup system.

The prowess of this machine soon attracted attention from outside organizations. The British Meteorological Service used it for weather forecasts, the tax administration for calculating tax scales, and British Railways to determine distances between all its stations. The success of these services prompted Lyons to create a specialized subsidiary in 1954, LEO Computers Limited, to commercialize this technology.

This story tells a curious paradox: while traditional office equipment manufacturers and electronics giants hesitated, a chain of restaurants and tearooms was creating the world's first computer for a viable commercial

application. LEO I operated until January 1965, before giving way to the more powerful LEO II and LEO III models that equipped numerous British companies and administrations.

The LEO saga continued through a series of industrial mergers: first with English Electric in 1963, then with other companies to form International Computers Limited in 1968. The last LEO III was quietly decommissioned in 1981 in the premises of the British Post Office.

UNIVAC I

The first commercially produced computer manufactured in series came to life under the name UNIVAC I, an acronym for *Universal Automatic Computer*. Its history began with two talented engineers from the University of Pennsylvania: J. Presper Eckert and John Mauchly. These visionaries had just completed the design of ENIAC during World War II when they recognized the commercial potential of computers.

With the ENIAC blueprints barely dry, Eckert and Mauchly contacted the U.S. Census Bureau in late 1945. Their proposal was for a computer capable of rapidly processing demographic data through the innovative use of magnetic tape as a storage medium.

In the spring of 1946, the two engineers left the University of Pennsylvania, determined to capitalize on their unique expertise by entering the private sector. They first established a simple partnership before founding the Eckert-Mauchly Computer Corporation in December 1947. Beyond scientific computation, they envisioned applications for accounting, document management, and statistical analysis.

UNIVAC evolved continuously during its development phase. Initially designed as a statistical tool for the Census Bureau, the project expanded following discussions with the A.C. Nielsen Company and Prudential Insurance. These new requirements steered the machine toward alphanumeric data processing, essential for managing insurance policies, billing, and actuarial calculations.

Financial difficulties plagued the machine's development. During 1948, American Totalisator Company invested 500,000 dollars in the company in exchange for 40% of the shares. Despite this injection of capital, difficulties mounted. The situation became so critical in 1950 that Eckert and Mauchly had to sell their company to Remington Rand, which took over development and commercialization.

Technically, UNIVAC I excelled through its innovations. Its mercury memory stored 1,000 words of 12 characters each. Used for data input and output, the magnetic tapes processed 20,000 characters per second. The machine performed an addition in 525 microseconds and a multiplication in 2,150 microseconds. Its remarkable reliability relied on a self-checking system: nearly one-third of the circuits served solely to detect errors.

UNIVAC I's modular design constituted one of its strengths. Tasks not directly requiring the central processing unit were delegated to autonomous auxiliary modules, each with its own power supply. This architecture enabled parallel operations for magnetic tape input, punched card conversion, and printing, without slowing down the main processing.

The Census Bureau received the first unit in March 1951, after a battery of rigorous tests. These trials, jointly developed by the Bureau of Standards and the Census Bureau, evaluated both computing capabilities and input-output system performance. The computer notably had to prove its ability to execute 2,500 operations in 1.26 seconds and correctly manipulate 1.4 million characters on magnetic tape.

The arrival of UNIVAC I transformed organizational computerization. The Census Bureau used it for monthly economic reports and business censuses. General Electric, in Louisville, employed it for payroll, materials tracking, and sales analysis. Franklin Life Insurance Company adopted it to manage premium billing, calculate commissions, and maintain dividend accounting.

The U.S. Air Force demonstrated the machine's potential well. Its "Air Materiel Command" managed a massive inventory of 1.2 million items, approximately six times that of a department store. Thanks to UNIVAC, establishing the forecasted budget for aircraft spare parts took only one day, compared to several weeks previously with punched card systems.

Maintaining this technological marvel required strict discipline. The Census Bureau ran its system 24 hours a day, seven days a week, except for four eight-hour windows dedicated to preventive maintenance. The latter consumed approximately 21% of available time. Vacuum tubes caused the most frequent failures: on average, two tubes failed each week and had to be replaced.

Operator training proved crucial for maximizing system performance. Remington Rand quickly realized that a UNIVAC I required a supervisor of the caliber of one managing a large punched card installation, but with a smaller team of highly qualified operators. This human expertise enabled both sound operational decisions and early detection of malfunctions.

Experience gained with the first UNIVAC I generated continuous improvements. Engineers progressively identified and corrected various prob-

lems: parasitic oscillations in certain circuits, relay bouncing in the Uniservo units (magnetic tape drives), and other mechanical issues. These corrections naturally benefited subsequent models.

By October 1954, Remington Rand had delivered twelve UNIVAC I systems and received orders for four additional systems. This commercial success demonstrated the viability of the business computer market. UNIVAC I was the first computer in a computer industry oriented toward commercial applications, distinct from the purely scientific calculators that had previously dominated. Its success validated Eckert and Mauchly's vision: that of a universal computer capable of serving diverse applications, from scientific computation to business management.

A-0 Compiler

In the spring of 1952, American mathematician Grace Murray Hopper created the A-0 (Arithmetic Language 0), the first compiler ever designed. This creation occurred at a time when computing was attempting to break free from military laboratories and open up to other horizons.

The path toward this innovation began during World War II. In 1943, this Yale graduate joined the U.S. Naval Reserve before being assigned to Harvard. There she worked on the Mark I, then the most imposing digital calculator, under the supervision of Howard Aiken. She was among the first three programmers in history to operate this ancestor of modern computers.

On the Mark I, programming was performed via punched tape: a hole symbolized a 1, the absence of a perforation a 0. Hopper wrote the machine's operating manual, a 500-page *Manual of Operation for the Automatic Sequence Controlled Calculator*. This deep dive into the machine's inner workings gave her a unique perspective on the difficulties of programming at its most elementary level.

In 1949, she left Harvard for the Eckert-Mauchly Computer Corporation. She participated in the design of UNIVAC I, the first commercial electronic computer. It was there that she noticed a practical problem: programmers were wasting hours copying existing code to adapt it to new projects. This tedious task generated numerous errors, primarily when manually adjusting memory addresses.

This observation led her in 1951 to create a system to automate the process. She developed the A-0, capable of assembling pre-existing sub-

routines into a coherent program. Her system used alphabetic characters to indicate the nature of operations (a for arithmetic, x for exponential). Arguments and results appeared in a numbered list, referenced by their position.

The magic of the A-0 lay in its ability to translate between symbolic language and machine code. To calculate the square of a number, a programmer simply specified the operation and references to variables, without writing all the corresponding binary instructions. This abstraction constituted a breakthrough because, for the first time, humans programmed in a language more natural to them.

The gains proved considerable, with weeks of programming reduced to a few hours. Errors decreased, reliability increased. The A-0 introduced a layer of abstraction between humans and machines that would become the cornerstone of modern computing.

The system evolved with versions A-1 and then A-2 in 1953. This latter version introduced short mnemonics for operations, making the code more readable. For example, `ADD 00X 00Y 00Z` adds X and Y, storing the result in Z. A-2 was the first compiler widely adopted by UNIVAC users, thus validating the concept.

In 1954, Hopper took charge of automatic programming for the UNIVAC division. This position gave her the opportunity to realize her vision of a more accessible language. She understood that most users had little appreciation for mathematical symbols and preferred something more natural: English. This reflection led to B-0 (Business Language 0) in 1955, renamed FLOW-MATIC, which allowed programs to be written with syntax close to everyday English.

The A-0 reached its peak in the development of COBOL starting in 1959. At the Conference on Data Systems Languages (CODASYL), where Hopper served as technical advisor, the foundations established with A-0 and FLOW-MATIC directly inspired this language. The structure of code in distinct sections and the use of syntax close to English primarily characterized COBOL.

The A-0 compiler established several foundational principles for computing: code reusability, abstraction of machine operations toward high-level instructions, and accessibility of programming. These concepts remain essential, to the point that it's difficult to imagine contemporary computing without them.

Hopper's contribution earned her numerous distinctions. In 1969, she received the *Computer Sciences Man-of-the-Year Award* from the Data Processing Management Association, a title whose irony doesn't escape anyone with our contemporary perspective. She continued her career in

the U.S. Navy until 1986, reaching the rank of Rear Admiral, while advocating for computer innovation and the democratization of programming.

The A-0 compiler transformed a field once reserved for mathematicians and scientists into a more broadly accessible tool. Ease of use on one hand, and abstraction of technical complexities on the other, remain today at the heart of programming language designers' concerns, a direct legacy of Grace Hopper's work 73 years ago.

IBM 701

In the early 1950s, Thomas J. Watson Sr. approved James Birkenstock's proposal to create an innovative computing machine. This machine would use cathode ray tube memory of approximately 20,000 digits and feature a clock cycle capable of multiplying two numbers in one millisecond. Initially named the "Defense Calculator", the machine came into being during the Korean War at the request of the U.S. Department of Defense. A team led by J. A. Haddad and Nathan Rochester was assigned to its development.

The world had only about twenty functional digital computers. Birkenstock believed that approximately thirty of these machines would find buyers in the defense industry. The creation of the Defense Calculator was relatively rapid thanks to IBM's experience with its earlier products. The team adopted a minimalist philosophy: simple equipment, streamlined logic design, and the absence of superfluous devices.

The question of word length sparked considerable debate at IBM. As insufficient length would require using multiple words for a single number, Werner Buchholz emphasized the importance of this technical choice. Conversely, excessive length would slow operations and increase costs. After studying existing machines, IBM determined that a length between 10 and 12 digits (35 to 41 bits) would be ideal. Magnetic tape technology suggested that 6 parallel channels represented an optimum for storage. IBM therefore set the word size at 36 bits for its 701.

On April 29, 1952, Thomas J. Watson officially announced the IBM 701 Electronic Data Processing System at an annual meeting. The "Data Processing" designation emphasized its ability to handle different types of data in various contexts. The first model arrived at IBM headquarters in New York in December 1952. Installed in the former SSEC room, it became an attraction for passersby. This machine served technical consulting op-

erations for various clients, with a team of IBM scientists dedicated to solving their problems.

The second machine (serial number 1) was delivered to Los Alamos Scientific Laboratory on April 1st, 1953. Three days were sufficient to make it operational. It functioned until autumn 1956, before a second machine was installed in 1954. Its electrostatic memory, despite its lack of reliability, contained 2048 words of 36 bits. The system comprised the central unit, a card reader, a printer, a magnetic drum storage unit, two magnetic tape drives, and a card punch.

The 701 was the first machine produced in sufficient quantity to meet the needs of an expanding clientele. Its modular design, with detachable components, marked the beginning of a new philosophy at IBM: building expandable machines rather than monolithic systems requiring complete replacement. The standard configuration consisted of eleven elements: two magnetic tape units (each with two drives), a magnetic drum memory unit, a cathode ray tube electrostatic storage unit, an L-shaped arithmetic and control unit with operator panel, a card reader, a printer, a card punch, and three power supply units.

The IBM 701 displayed remarkable compactness. Its logic was housed in 64-pin modules, each carrying a row of 8 vacuum tubes on the front. Logical operations were executed via germanium diodes located at the base of the modules. These modules integrated into a backplane designed to allow their replacement without shutting down the system. IBM was among the first to use backplanes as a means of accommodating different types of functional modules.

The Analytical Control Unit (ACU) housed the arithmetic components, input-output control circuits, and stored program control circuits in a large cabinet. A separate cabinet contained the operator console. The ACU operated with a pulse repetition rate of 1 megahertz. A 12-stage ring generated twelve 1-microsecond pulses forming the basic machine cycle. This speed primarily met the regeneration requirements of the electrostatic storage unit. The latter contained 72 cathode ray tubes storing 10,240 digits, or 2,048 words of 36 bits, with an access time of approximately 12 microseconds. Data could be accessed randomly. The cost per bit approached one dollar, much less than vacuum tube storage. The monthly rental for this unit amounted to approximately $2,600.

The 701's input-output system performed five functions: selecting a unit for reading, writing, or control; interlocking the operation with program execution; copying data between the electrostatic storage unit and I/O units; synchronizing signals between the selected unit and the CPU; disconnecting the I/O unit once its work was completed.

In 1956, the IBM 737 Magnetic Core Storage Unit replaced the electrostatic storage unit. With a capacity of 4,096 words of 36 bits, it reduced the addition time, including access, from 60 to 36 microseconds. Its monthly rental reached approximately $6,100.

As a binary single-address scientific computer, the IBM 701 had a word size of 36 bits and an instruction set of 33 instructions. The 701's CPU remained 25% empty to facilitate error correction or the addition of features. Its monthly rental amounted to approximately $16,000. By the end of 1954, 18 machines were installed and fully operational.

Its successor, the 704, filled this space with additional functions. This was followed by a line of increasingly powerful and faster machines, maintaining backward compatibility and incorporating new features. This evolution culminated with the 7094 II and its simplified versions, the 7040 and 7044.

IBM 702

After the 701, IBM created the 702, among the first computers designed for business applications, alongside the LEO and UNIVAC. Introduced in 1954, this machine marked a break with the era of punched-card tabulating equipment that dominated administrative processing at the time. General Electric was among its first purchasers, seeking to modernize its management methods in a tense post-war social context. For GE, the automation of administrative tasks represented a means to reduce its dependence on unionized personnel, amid conflict with the United Electrical Workers organization.

The IBM 702 stood out with its electrostatic memory based on cathode ray tubes, storing up to 10,000 characters. This capacity corresponded to approximately 125 punched cards, a considerable leap compared to earlier machines that handled only one or two cards simultaneously. Additional magnetic drums could extend this capacity, each equivalent to 750 cards. The machine innovated with its variable word-length architecture. This approach differed from contemporary scientific computers using fixed-size words. The IBM 702 thus processed data ranging from one to several hundred characters, a flexibility ideal for commercial applications handling information such as names, addresses, and amounts. Numerical processing was performed in decimal rather than binary, a choice revealing IBM's intention to simplify programming for commercial uses. The computer manipulated both letters and numbers, an advantage for business data.

Each character used seven-bit encoding, including a parity bit for error detection.

Its architecture adopted serial operation, processing characters sequentially. This method, while theoretically slower than the parallel processing of scientific computers, offered great agility for manipulating and reorganizing data, common operations in the business world.

The IBM 702 excelled in input-output speed. While punched-card equipment processed approximately 100 cards per minute, its magnetic tapes reached the equivalent of 3,500 cards per minute, making possible the processing of previously inaccessible data volumes.

Programming was executed in machine language. Each instruction comprised a one-character operation code followed by a four-character address. Programs resided in the same memory as data, following the von Neumann architecture principle. Its instruction set included arithmetic, logical, and transfer operations designed for commercial applications. Memory addresses were absolute, requiring reprogramming whenever the data organization was modified. The programmer had to account for electrostatic memory access delays to optimize performance.

The installation of an IBM 702 at General Electric in 1954 illustrates business computerization during this period. The first targeted application was a payroll system, a project that mobilized a substantial team bringing together GE personnel, Arthur Andersen consultants, and IBM engineers. In parallel, a manufacturing control application was developed for the "Dishwashers and Disposals" department. Notably, the latter became operational before the payroll system despite more limited resources, foreshadowing the observations Fred Brooks would later formulate in his 1975 essay "The Mythical Man-Month" on software project management: adding personnel to a late project delays it further, small cohesive teams often outperform large fragmented structures, and communication complexity grows exponentially with team size.

Reliability was a paramount criterion. For a machine dedicated to financial operations, the IBM 702 incorporated numerous control mechanisms, notably the parity bit to detect errors in data transmission and storage. These automatic checks complemented traditional accounting procedures, such as the use of control totals.

The IBM 702 inaugurated a new chapter in IBM's history and commercial computing. Its success validated the value of electronic computers for business applications, creating a market that would become dominant. The concepts introduced, such as variable-length data processing and orientation toward commercial applications, had a lasting influence on the evolution of enterprise systems.

The machine had several successors, notably the IBM 705, which adopted its fundamental principles while bringing technical improvements. Nevertheless, its true legacy is found in the System/360 family, launched in 1964, which unified IBM's scientific and commercial lines in a common architecture. The System/360 retained certain characteristics of the IBM 702, such as the ability to efficiently process both numerical and alphabetical data.

The evolution of computers transformed scientific calculation tools into versatile processing machines. This transformation accompanied the modernization of American businesses during the 1950s, contributing to the emergence of business computing as we know it in the 21st century.

Assembly Language

In the 1940s and 1950s, programmers were constrained to code in pure binary. This tedious work generated numerous errors. Assembly language then emerged as an ingenious solution: replacing binary sequences with mnemonics that were easier to remember. This transition marked the first step toward abstraction in the art of programming.

Assembly language evolved in 1978 with the appearance of the Intel 8086 microprocessor. This 16-bit component brought an architecture that would durably influence the world of low-level programming. The saga continued with the 80186 and 80286 in 1982, which maintained valuable backward compatibility for the continuity of existing applications.

The leap to 32 bits occurred with the 80386 in 1985, followed by the 80486 four years later. This lineage, named x86, established a standard that still spans decades. The instruction set continuously enriched itself while preserving its historical compatibility, guaranteeing longevity for software developments.

The 1990s saw Intel's monopoly crumble. AMD launched its own processors compatible with the K5, K6, and K7 series. In 2003, the company revolutionized the market with the x86-64 architecture, a 64-bit extension of the classic instruction set. Other manufacturers like VIA and Transmeta joined the venture from 2000 onward. The x86 instruction set constantly expanded. Intel deployed various extensions: MMX, then different versions of SSE, while AMD countered with 3D Now!. By 2012, this instruction set reached over 900 instructions, testifying to the growing sophistication of this architecture.

Modern tools such as `MASM`, `NASM`, or `GNU Assembler` have transformed the programmer's work with their advanced features. Macros, debugging and optimization tools make assembly programming far less tedious today than in its early days. Certain domains still value assembly language. Embedded systems, where every byte counts, find it an ideal ally. Operating system creators use it for critical sections requiring complete hardware control. Video game developers employ it to extract maximum performance from computationally intensive sections.

Segmentation represents a notable specificity of the x86 architecture. This approach divides memory space into distinct zones dedicated to code, data, and stack. Originally created to overcome the hardware limitations of early processors, it ultimately proved an asset for securing systems by isolating access to different memory regions. The interface between assembly language and operating system has been standardized over time. System calls, particularly under DOS via interrupt 21h, gave assembly programs structured access to system resources. This standardization significantly improved portability and simplified application development.

Floating-point number processing illustrates the evolution of x86 assembly. The first processors in the line required an external mathematical coprocessor (x87). Integration of this unit directly into the processor core, starting with the 486, simplified assembly code writing while multiplying performance tenfold. Contemporary processors internally transform complex x86 instructions into more elementary micro-operations. This approach cleverly combines the benefits of CISC architecture for programming with the efficiency of RISC architecture for execution, thus preserving the richness of the instruction set while maximizing performance.

Although often overlooked, teaching assembly language remains essential in the training of 21st-century computer scientists. It offers an understanding of how computers work and provides a solid foundation for grasping low-level concepts. This mastery proves essential for developing efficient programs, particularly in sectors where performance is paramount, and perhaps also for reducing resource and energy consumption.

Far from being static, assembly language continues to evolve. The emergence of vector instructions and expanded registers now facilitates exploiting the capabilities of modern processors for parallel computing and multimedia processing. These advances prove the remarkable adaptability of assembly language to the ever-renewed demands of computing.

IBM 650

IBM's model 650, launched on the market in 1954, was the first computer sold on a large scale—over a thousand units—that opened the doors to a new digital era. Nicknamed the "magnetic drum calculator", the IBM 650 featured a remarkable architecture for its time. Its heart beat to the rhythm of a nickel-cobalt rotating drum 10 centimeters in diameter and 40 centimeters in length. This cylinder rotated at the dizzying speed of 12,500 revolutions per minute and stored 2,000 "words" of ten digits. The particular arrangement of data in 40 bands of 50 positions forced programmers into constant mental gymnastics to minimize access times related to drum rotation.

The physical architecture of the system was divided into three distinct blocks: the central console housing the famous drum and arithmetic circuits, the read-punch unit (model 533) for data input and output, and the power supply unit (model 655) that transformed the language of punched cards into electronic signals comprehensible by the machine.

A technical peculiarity of the IBM 650 lay in its "one plus one address" instruction set. Each instruction contained the operation to be performed as well as the location of the next instruction. This approach, sometimes criticized by modern theorists, proved brilliant in the context of magnetic drum constraints. With only 44 operation codes available, the machine forced economy and elegance in program writing. Donald Knuth, who began programming on this machine during his studies, retained throughout his life a particular fondness for this simplicity that stimulated inventiveness.

The machine evolved and the 653 system enriched the initial configuration with index registers, a floating-point calculation unit, and 60 words of fast memory. Additional peripherals such as magnetic tapes and printers gradually completed the 650 ecosystem, extending its capabilities while maintaining compatibility with existing programs.

The software aspect underwent equally rich development. The SOAP (Symbolic Optimal Assembly Program) assembler created by Stan Poley freed programmers from direct machine code programming. The Bell Labs interpreter, the work of Wolontis, considerably facilitated scientific calculations through floating-point arithmetic. An extensive library of over 200 programs emerged, covering domains as varied as pure mathematics, statistics, management, or solving differential equations.

At Carnegie Tech (now Carnegie Mellon), Herbert Simon and Allen Newell developed on the IBM 650 their IPL-V (Information Processing Language), a precursor in the field of artificial intelligence. This list-

processing language introduced determining concepts such as top-down programming and closed subroutines—principles that later became fundamentals of structured programming.

The robustness and relative accessibility of the IBM 650 transformed it into an unparalleled educational tool. Unlike its contemporaries that required highly specialized personnel, this machine could be used directly by students and researchers. Thousands of programmers cut their teeth on its consoles, forming the first massive generation of computing professionals.

The technical limitations of the machine—restricted memory and variable drum access times—proved paradoxically fertile. They forced programmers to develop sophisticated optimization techniques and to think about the structure of their algorithms. These practices durably influenced the teaching and design of computer programs up to the present day.

Beyond technical specifications, the IBM 650 represents the moment when computing left the laboratories to enter offices and universities. It was no longer an experimental curiosity but a practical, reliable, and relatively accessible tool.

IBM 704

In 1954, IBM launched a computer that would be a game-changer: the IBM 704. The first mass-produced computer equipped with a floating-point arithmetic unit, this machine marked a radical departure from the IBM 701 and 702 that preceded it.

What did this machine bring to the table? First, ferrite core memory replacing the Williams tubes used until then. The addition of three index registers also transformed the way of programming. To take advantage of these technical assets, the designers revised the instruction format, now using the complete 36-bit word. This instruction set would become the DNA of the entire IBM 700/7000 scientific line in the following years.

Under the hood of the IBM 704 was a 38-bit accumulator, a 36-bit multiplier-quotient register, and three 15-bit decrement registers. These latter had the peculiarity of subtracting their content from the base address instead of adding to it. The machine allowed simultaneous use of all three registers in one instruction, via a 3-bit tag field serving as a binary map to indicate the participating registers. A technical trick worth

highlighting.

The architecture of the IBM 704 was based on two instruction formats: "Type A" and "Type B". The first combined a three-bit prefix (instruction code), a decrement field (15 bits), a tag field (3 bits), and an address field (15 bits). The second combined a 12-bit instruction code, a 2-bit flag field, four unused bits, a 3-bit tag field, and a 15-bit address field.

Between 1955 and 1960, IBM sold 123 units of this machine. Its performance was approximately 40,000 instructions per second, a number that impressed at the time. Its substantial price reserved it for major scientific laboratories and industrial groups.

But the IBM 704 didn't just shine through its hardware; it also carried two important software creations. FORTRAN, first, created specifically for this machine. This language made scientific programming more natural, with code writing close to mathematical equations. An automatic translator converted this code into machine language, thus reducing errors and programming time.

LISP, another prodigy born on the IBM 704, still retains today in its `car` and `cdr` primitives the trace of this lineage. These names, abbreviations of "Contents of Address part of Register number" and "Contents of Decrement part of Register number", directly reflect the machine's architecture, where these fields served to store respectively the head and tail of linked lists.

Computer-assisted music took its first steps on this machine. Max Mathews developed MUSIC there, opening virgin territory to artistic exploration. In 1962, at Bell Labs, physicist John Larry Kelly Jr made the IBM 704 sing. His `vocoder` synthesizer reproduced "Daisy Bell" with musical accompaniment by Max Mathews. Arthur C. Clarke, an awestruck witness to this demonstration, immortalized it in the final scene of his novel "2001: A Space Odyssey", where the HAL 9000 computer intones this melody.

The IBM 704 distinguished itself in various scientific fields. Ed Thorp used it to model his equations on blackjack probabilities using FORTRAN. The machine participated in the space race: in the fall of 1957, it became the official computer of the Smithsonian Astrophysical Observatory's Operation Moonwatch. IBM assigned four scientists—Giampiero Rossoni, John Greenstadt, Thomas Apple, and Richard Hatch—to help Smithsonian researchers calculate the trajectories of the first artificial satellites.

The FORTRAN system developed for the IBM 704 testified to growing software maturity. Its reference manual, published in October 1956, described a complete environment with input-output management, automatic conversions between decimal and binary formats, multidimensional

array manipulation, and generated code optimization. The system also handled integration of external routines and allocation of available memory.

The technical documentation emphasized the efficiency of code produced by FORTRAN, comparable to that written by the best programmers in machine language. This performance, coupled with programming simplicity, explains the rapid adoption of the system in scientific circles.

The IBM 704 symbolizes that period when the computer, initially a calculating machine, transformed into a versatile creative tool. Its influence on computer languages, sound synthesis, and scientific computing marked the beginnings of modern 20^{th}-century computing.

Dartmouth Conference

A document written on August 31, 1955 by four scientists, John McCarthy, Marvin Minsky, Nathaniel Rochester and Claude Shannon, proposed organizing a summer conference at Dartmouth College. Their text, *A Proposal for the Dartmouth Summer Research Project on Artificial Intelligence*, envisioned a two-month gathering to explore the creation of machines endowed with intelligence.

These researchers advocated a bold idea: every aspect of learning and intelligence could be described with sufficient precision to be reproduced by a machine. They wanted to discover how these machines would use language, form abstract concepts, solve complex problems, and improve themselves.

The proposal detailed seven research directions. First, automatic computers: in their view, the limitation was not so much hardware as software – we did not yet know how to write programs that fully exploited available resources. Then came language programming, based on the principle that human thought essentially manipulates words according to certain rules. Neural networks constituted the third axis, seeking to understand concept formation in a neural network. The fourth aspect concerned a theory of calculations and their size, measuring the efficiency of computing devices. Self-improvement represented the fifth component, suggesting that a truly intelligent machine should be able to enhance its own capabilities. The sixth theme addressed abstractions and their classification. The last explored the role of chance and creativity, suggesting that a controlled dose of randomness distinguished creative thought from mere technical skill.

Each of the four organizers brought unique expertise. Claude Shannon, a mathematician working at Bell Labs, had developed the statistical theory of information and applied propositional calculus to switching circuits. Marvin Minsky, a young researcher at Harvard specializing in mathematics and neurology, had built a machine simulating neural learning. Nathaniel Rochester headed information research at IBM and had participated in the design of the IBM 701, one of the first widely adopted commercial computers. John McCarthy, assistant professor of mathematics at Dartmouth, studied the mathematical nature of the thought process.

The Rockefeller Foundation funded the event with 13,500 dollars, covering salaries (1,200 dollars per academic), travel, accommodation, and administrative expenses. Participants from companies like Bell Labs or IBM were supported by their organizations.

The proposal's appendices reveal the organizers' personal visions. Shannon wanted to apply his information theory to calculating machines and brain models, particularly to study reliable computation with unreliable components. Minsky wanted to create systems capable of developing sensory abstractions to model their environment. Rochester was interested in originality in machine performance, seeking how to introduce a controlled form of randomness to foster creativity. McCarthy, for his part, explored the relationship between language and intelligence, aiming to create an artificial language giving machines the capacity to formulate hypotheses and self-reference.

The conference generated keen interest. A list of researchers passionate about what was called the "artificial intelligence problem" was established. It included renowned scientists such as John Nash, Herbert Simon, and Warren McCulloch. This disciplinary diversity testified to the subject's appeal: mathematics, electrical engineering, neurology, psychology, and computer science converged there.

The expression "artificial intelligence" makes its first official appearance in this document. This terminological choice clearly asserted the ambition to create machines capable of genuine intelligence, far beyond the simple automation of specific tasks.

This summer 1956 conference established artificial intelligence as a distinct field, with its own objectives, methods, and vision. The questions it raised – machine learning, natural language processing, knowledge representation, automated reasoning – structure AI research today in the 21st century.

Its influence, beyond the technical framework, created a vibrant scientific community. Participants returned to their institutions to found AI laboratories, train students, and launch research programs that trans-

formed computing over the following decades.

The enthusiasm that shines through the Dartmouth proposal characterizes that era when computing seemed destined for a limitless future. Some hopes encountered more complex obstacles than anticipated, but the conceptual foundations laid during that meeting made possible the applications transforming our daily lives, from voice recognition to automatic translation, from expert systems to deep learning.

IBM 350 RAMAC

On May 6, 1955, IBM quietly unveiled a technical breakthrough that would forever change our relationship with data. In a laboratory in San Jose, a team of engineers had just created the first hard disk drive in history. This device with a name hardly evocative to the general public, the IBM 350 RAMAC (Random Access Method of Accounting and Control), marked the beginning of a new era in digital storage.

It all began three years earlier. In January 1952, W. Wallace McDowell contacted Reynold B. Johnson, a former teacher who had joined IBM. Management wanted to establish a small laboratory on the American West Coast, far from the habits and structures of the East. Johnson was given carte blanche to build a team limited to 50 people. His mission was to explore different technical approaches, particularly impact-free printing and data reduction.

Johnson wasted no time. Within weeks, he rented an empty building in San Jose, launched recruitment, and began operations. Louis D. Stevens, temporarily sent to assist him, would ultimately become his technical right hand. By July 1952, the laboratory was operational with about thirty collaborators. The atmosphere was distinctive, shaped by three rules established by Johnson: understand your machine and its use, know all ongoing projects in the laboratory, and always give priority to helping colleagues.

The project that would lead to RAMAC was initially called “Source Recording”. Its objective was simply to convert alphanumeric data into machine-readable format, while storage options could be counted on one hand: punched cards, magnetic tape, and magnetic drums (more marginally). These media all had limitations, often leaving processors idle during information retrieval.

In the fall of 1952, the team became interested in the “bin files” of

punched cards widely used in businesses. These large rectangular cabinets contained cards sorted by customer or item number. Employees manually searched through them to produce invoices and delivery notes, a slow process prone to errors.

After exploring various approaches, Johnson made a bold decision in January 1953 that met with disbelief: concentrate all efforts on magnetic disks. One engineer predicted a crushing failure, and the project earned the mocking nickname "salami slicer".

The technical problems seemed insurmountable. Maintaining a microscopic gap between the read head and the rotating surface, creating perfectly flat surfaces, developing a reliable magnetic coating... The list seemed endless. The team tested different materials: aluminum, brass, glass, plastics, magnesium. Aluminum disks deformed at high speed. The solution finally came from compressed and heated aluminum laminates beyond the annealing point. The magnetic coating posed another puzzle. Jake Hagopian found a makeshift method: pouring the coating onto the inner face of a rapidly rotating disk. Bill Crooks improved it by filtering the solution through nylon and using... paper cups for measurements. This rudimentary technique would be used for a year before being automated. Norman Vogel solved the read-write head problem. He designed a system that retracted the head during movement between disks, with three mini pneumatic pistons to control position.

On February 10, 1954, the first successful information transfer between cards and disks was achieved. The prototype looked like a bizarre contraption, but the team persevered. A complete redesign resulted in Model II, with the innovation of switching to a vertical axis for the disks, facilitating their replacement and leaving more space for access mechanisms.

On January 10, 1955, less than three years after the laboratory's creation, the first Model II of the RAMAC 350 was successfully presented. The system was commercialized in September 1956, integrated into the IBM 305 RAMAC system which included processor, card reader, and printer.

The RAMAC 350's specifications were unprecedented. Fifty 61 cm disks rotating at 1,200 rpm, a throughput of 100,000 bits per second, a total capacity of 5 million characters. The density reached 2,000 bits per square inch, with an average access time of 600 milliseconds. To measure the progress made, the IBM 3380 of the 1980s used nine 35.5 cm disks rotating at 3,600 rpm, with a throughput of 24 million bits per second and a capacity of 1.25 billion bytes. The monthly rental cost per megabyte dropped from 130 dollars for the RAMAC 350 to about one dollar for the 3380.

The arrival of the RAMAC 350 revolutionized our relationship with data. For the first time, information became directly accessible without prior physical manipulation. This technological breakthrough made interactive computer applications possible: airline reservations, inventory management, automated banking services, space flight control, word processing...

This innovation, born in a modest California laboratory, launched a storage industry generating billions of dollars, and stimulated the development of fast processors and computing services. In 1984, the American Society of Mechanical Engineers officially recognized the RAMAC 350 as an international historical milestone, acknowledging its role in the evolution of mechanical engineering and its influence on 20^{th} century society.

The RAMAC marks the true birth of the digital era in which we live today, where instant access to information has become so natural that we sometimes forget the technological marvel it represents.

IPL

Computers, machines initially created to solve mathematical equations, would soon tackle a different domain: that of symbolic processing and complex problem solving. It was in this ferment that Information Processing Language emerged, a family of languages that blazed a trail in the history of artificial intelligence.

The IPL adventure began with the bold dream of building a program capable of proving logical theorems. IPL-I, sketched out in the early 1950s, never existed anywhere but on paper. This draft nevertheless constituted a first attempt to formalize the symbolic manipulation mechanisms essential to automated theorem proving.

IPL-II and IPL-III were subsequently developed at the RAND Corporation, deployed on the JOHNNIAC computer. These versions brought to life the Logic Theorist program, a pioneering work in the field of artificial intelligence. This program demonstrated theorems using heuristic methods modeled on human thought, breaking away from the purely computational approach of computers.

The migration of IPL to the IBM 650 by a team from the Carnegie Institute of Technology gave birth to IPL-V, the accomplished and influential version of this lineage. Meanwhile, IPL-IV ran on the JOHNNIAC for innovative applications: a chess program and a system for balancing

industrial production lines.

IPL-V marked a breakthrough in the design of computer languages. It introduced concepts such as list manipulation and dynamic data structures. The language made it possible to create and dynamically modify data during program execution, a rare characteristic at a time when memory generally remained fixed.

IPL's great strength lay in its ability to model tasks that humans accomplish without being able to describe the process. Playing chess, making business decisions, or proving mathematical theorems are all activities where human expertise relies on intuition rather than defined algorithms. IPL provided a framework for transcribing these adaptive behaviors typical of natural intelligence.

IPL's architecture was built around a few fundamental principles. The system employed cells to store IPL words, formed by two prefixes (`P` and `Q`) and two symbols (`SYMB` and `LINK`). The symbols, regional or local, offered flexible data representation. The language popularized the concept of the "list", a fundamental structure for organizing information hierarchically.

A remarkable creation of IPL was its "description list" system. This mechanism associated descriptive information with a list, information that could be modified during execution. This approach foreshadowed the current concepts of metadata and object attributes found in modern 21st century programming.

IPL also integrated advanced memory management functionalities, notably with its "available space list" which automated the allocation and release of resources. This method freed developers from manual management constraints, leaving them more energy for algorithmic logic.

The language offered approximately 150 fundamental processes covering a broad spectrum of operations: list manipulation, arithmetic calculations, input-output management, and processing of complex structures. An elaborate system of "generators" made it possible to execute repetitive operations with elegance.

IPL served as a foundation for various applications in the nascent field of artificial intelligence. Programs for discrimination learning, binary choices, and theorem proving emerged thanks to this language. These creations proved IPL's ability to support sophisticated symbolic processing applications.

IPL-VI, the final version of this family with an ambitious vision, proposed a hardware architecture designed specifically to directly execute this language, without going through a software interpretation layer. This approach aimed to overcome the performance limitations inherent in the interpreted nature of earlier versions on conventional machines.

IPL inspired the creation of other systems such as LISP, list processing in FORTRAN, or COMIT. These languages extended the exploration of the paths opened by IPL in the field of symbolic processing and artificial intelligence. Despite its weaknesses in terms of speed, particularly for traditional numerical computations, it demonstrated that the benefits of a high-level system far outweighed its resource costs, especially for complex applications requiring fine-grained management of data and processes.

TX-0

The TX-0 (Transistorized Computer Zero) came to life at MIT's Lincoln Laboratory in 1956. The history of this exceptional machine began with the ambition to create the first fully transistorized general-purpose computer. Two goals guided this endeavor: to experiment with using transistors as logic elements in a high-speed digital calculator, and to test a ferrite core memory with an extraordinary capacity of 65,536 words.

The team led by Kenneth Olsen chose a minimalist architecture: an 18-bit word, with 16 bits for addressing and only 2 bits for the instruction code. This minimal instruction set offered just four commands. The first stored the accumulator, the second added a value to it, the third handled conditional transfers on negative values. The fourth, a true tour de force, enabled microprogramming with multiple register transfers within a single machine cycle. This clever design made the TX-0 remarkably flexible despite its streamlined conception.

The TX-0's technological contribution lay in its components. The Philco L-5122 transistors, later marketed under the reference 2N240, underwent rigorous selection before installation. The results exceeded all expectations: after 10,000 hours of operation, these semiconductors showed only a minute decrease in gain. Even more astonishing, in 1974, after 49,000 hours of operation, barely a dozen had failed.

The TX-0's electronic architecture relied on RC circuits, a combination of resistor and capacitor, coupled in negative logic, with -3V representing 1 and 0V representing 0. The true achievement lay in the exclusive use of transistors to perform logic functions, without resorting to the usual diodes. The `AND` and `OR` operations were obtained through subtle voltage manipulation.

The TX-0's first memory, dubbed "S Memory", constituted a technological monument: 1.25 million tiny ferrite cores, handcrafted at Lincoln

Laboratory. This matrix stored 65,536 19-bit words with a cycle time of 5 microseconds. In 1958, this memory was transferred to the TX-2 project, giving way to a new transistorized memory of 4,096 words.

In 1958, the TX-0 left its birthplace to join MIT's Cambridge campus. This move transformed its role from research object to experimental tool, accessible day and night to researchers and students. This open-access policy sparked a wealth of innovative applications.

Hardware improvements followed one after another. Memory expanded to 8,192 words in 1959, while the instruction set gradually grew. The addition of an index register and the extension of the operation code to 5 bits opened new possibilities. The input-output system was enriched with analog-to-digital and digital-to-analog converters, paving the way for real-time applications.

Software followed this transformation. Jack Gilmore's initial utility gave way to sophisticated assemblers. The Macro assembler, created by J.B. Dennis in 1959, brought advanced features such as macro-instructions and automated constant management. Its successor, Midas, enabled manipulation of longer symbols and introduced recursive macro-programming.

The TX-0's applications illustrate the extent of its influence. The communications biophysics laboratory adopted it to analyze electrophysiological data from the auditory cortex. The speech research team used it to create the first voice recognition techniques. Other researchers explored image processing and character recognition, as well as a robotic hand equipped with sensors thanks to this computer.

Kenneth Olsen, drawing on his TX-0 experience, founded DEC (Digital Equipment Corporation). The concepts developed on the TX-0 directly inspired the PDP-1, DEC's first commercial computer, which adopted many characteristics of its academic ancestor.

The TX-0 demonstrated the relevance of an interactive and personal approach to computing. Its direct-access operating mode challenged the dominant batch processing model. Its longevity testified to the excellence of its design. After more than 49,000 hours of activity, the machine still required very little maintenance. In 1975, the TX-0 found refuge at DEC's Computer Museum in Marlborough, Massachusetts, preserving for posterity a foundational milestone of transistorized computing.

FORTRAN

IBM launched the FORTRAN project during the summer of 1954, an acronym for FORmula TRANslating, under the leadership of John Backus. At that time, programming IBM's new computer, the 704, was an uphill battle. Nearly two-thirds of scientific project budgets were consumed in the mere preparation of programs. Worse still, planning, writing, and debugging swallowed up more than 90% of the total project time.

Faced with this reality, Backus's team nurtured the clear ambition to create a language that would resemble mathematical notation while automatically generating efficient machine code. The bold objective was to reduce the time spent on coding and debugging by at least 80%.

The system's development stretched over two and a half years. A team of 18 people worked tirelessly on it. Their efforts culminated in 1957 with a complete system comprising two fundamental elements: the FORTRAN language and its translator, called the *executive routine*.

FORTRAN disrupted conventions with its innovations. Arithmetic expressions were now written naturally, close to classical mathematical notation. The asterisk (`*`) symbolized multiplication while the double asterisk (`**`) represented exponentiation. The system generated optimized machine code, avoiding redundant calculations and selecting the best instructions for each operation.

The language was packed with novel features. The `DO` and `IF` statements controlled execution flow. Multidimensional arrays could be easily manipulated through subscripted variables. The `READ`, `PRINT`, and `FORMAT` statements handled input-output flexibly. Programmers could define their own functions, promoting code reuse and better program structure.

The technical prowess of the FORTRAN translator lay in its ability to produce machine code nearly as efficient as that manually written by seasoned programmers. This performance resulted from fine-grained analysis of the source code and innovative optimization techniques. The translator's architecture was divided into six distinct sections, each fulfilling a precise function in the translation chain.

A concrete example illustrates the system's efficiency. After a day of training and a few consultations of the manual, a programmer completed an application in just four hours, using 47 FORTRAN instructions. Compilation on the 704 took six minutes and generated approximately 1000 machine instructions. The programmer spotted and corrected an error in the code without resorting to complex debugging techniques. This program would have required three days of manual coding, not counting debugging time.

Success came quickly. FORTRAN established itself as the reference in scientific programming. After its first standardization in 1966, the language was widely adopted, easy to teach, and machine-independent. It demonstrated the advantages of subroutines and separate compilation.

The 1970s saw new languages emerge that revealed certain limitations of FORTRAN. A new version, Fortran 77, was standardized in 1978. Despite welcome improvements, this update did not meet all the expectations of a community hungry for more profound changes. Other languages like Pascal, Ada, Modula 2, C, and C++ began to gain ground in scientific and academic circles.

It wasn't until 1991 that Fortran 90 emerged, a major evolution. This version introduced a plethora of modern features: free-form source allowing longer variable names, revised control structures, precise specification of numerical precision, treatment of arrays as entities in their own right, dynamic memory allocation with pointers and recursion, user-defined data types, modules with operator overloading and generic procedures.

Fortran 95, finalized in 1996, brought more modest but significant improvements such as the `FORALL` statement for parallel processing, user-defined pure and elemental subroutines, as well as various optimizations concerning pointers and derived types.

A leap forward occurred with Fortran 2003, which integrated object-oriented programming. This version included inheritance of derived types, polymorphism allowing variables to change type during execution, type-bound procedures, and interoperability with the C language. Input-output capabilities were enhanced, and support for IEEE arithmetic exceptions appeared.

The latest evolution, Fortran 2008, further enriched the language with support for parallel programming via `coarray` arrays, an increase in the maximum array rank to fifteen dimensions, new features to improve performance such as the `DO CONCURRENT` construct, and the addition of numerous intrinsic mathematical functions for scientific computing.

From a pioneering language designed to simplify the programming of scientific computers, FORTRAN has transformed into a modern tool that integrates contemporary programming paradigms while remaining faithful to its primary vocation: offering scientists and engineers an efficient means of expressing their calculations. Its longevity stems from its ability to evolve while maintaining backward compatibility, thereby preserving the massive investments made in existing code. Fortran 77 code remains a valid subset of recent versions, making it possible to gradually modernize organizational applications without rebuilding everything from scratch.

This exceptional robustness finds a striking illustration in the Voyager

space probes. More than 45 years after their launch, 24 billion kilometers from Earth, they still operate with their original FORTRAN code. In 2023, NASA teams achieved a technical feat by successfully deploying patches to this code, testifying to the extraordinary longevity of the language under extreme conditions.

Integrated Circuit

At the end of World War II, the complex electronics of B-29 bombers faced a significant technical challenge: each aircraft carried nearly a thousand vacuum tubes and tens of thousands of passive components. This multiplication of elements led to an exponential increase in manufacturing costs and a worrying decline in reliability. American military forces, confronted with this technical reality, feared that this complexity would drastically hinder the evolution of electronic systems.

The National Bureau of Standards launched the first research efforts to simplify manufacturing. Centralab division of Globe-Union proposed an innovative technique based on ceramic substrates supporting metallic interconnections and capacitors, onto which miniaturized vacuum tubes were mounted. This method, resulting from the work of S.J. Rubenstein, Mr Ehlers, Gertrude B. Sherwood, and Howard J. White, marked the first attempt to create components *in situ*.

After 1945, work continued under the direction of R.L. Wolff and A.S. Khouri, and Centralab developed high-speed screen printing processes. The company manufactured amplifiers for hearing aids incorporating dozens of passive components and several tube sockets. The use of a substrate with a high dielectric constant enabled the economical production of RC networks intended for radios and televisions.

Everything changed with the arrival of the transistor in 1948. More compact, less power-hungry, and more reliable than the vacuum tube, it also generated less heat. However, its integration into existing circuits required a lengthy technological adaptation. Jack Kilby, at Texas Instruments, and Robert Noyce, at Fairchild Semiconductor, independently conceived the integrated circuit in 1958.

Kilby imagined that all components of a circuit could be manufactured in a block of semiconductor material. In September 1958, his demonstration of a phase-shift oscillator using transistors, resistors, and capacitors in germanium proved the validity of the concept. This pioneering ap-

proach used gold wire bonded interconnections that proved incompatible with industrial production, which was a significant drawback.

Noyce developed a different solution at Fairchild, based on Jean Hoerni's planar technology. His design used silicon as a substrate, covered with an insulating oxide layer on which aluminum traces were deposited for interconnections. This method, better suited to industrialization, became the foundation of modern integrated circuit technology.

These inventions radically transformed electronics. Manufacturing complete circuits on a single silicon chip drastically reduced costs while spectacularly improving reliability. In 1961, Fairchild commercialized the Micrologic family, the first set of compatible logic circuits with diffused resistors and interconnections. That same year, Texas Instruments delivered to the US Air Force a small computer incorporating several hundred bits of semiconductor memory.

The integrated circuit industry followed an exponential trajectory, described by Gordon Moore in 1965: the number of transistors per circuit doubles approximately every two years. This prediction held true for over fifty years. From the first circuits containing a few thousand transistors, we have progressed to chips containing several billion components.

Technological advances have constantly reduced the size of transistors: from several microns in the 1970s to a few nanometers today. This miniaturization is accompanied by increased performance and reduced power consumption. Current technology is, however, approaching its physical limits, with transistors measuring only a few dozen atoms wide.

Integrated circuits made possible the personal computer, the mobile phone, the Internet, and artificial intelligence. The rise of the semiconductor industry has transformed the global economy, creating new sectors and changing our ways of life.

Faced with the physical constraints of miniaturization, researchers are now exploring three-dimensional integration or quantum computing. The power requirements for artificial intelligence and *machine learning* in the 21st century continue to drive innovation in this field.

ALGOL

The birth of ALGOL (ALGOrithmic Language) dates back to the late 1950s. This language resulted from a shared vision among European and American researchers: to create a universal tool for expressing algorithms

independently of machines. At a symposium in Darmstadt in 1955, this idea took root in discussions among automatic computation specialists.

The technological context of the era fully justified this approach. In the United States, computing was transforming into an industrial sector with IBM and other manufacturers marketing their first computers. A multitude of specific languages flourished according to needs and teams. Europe, meanwhile, saw the birth of its first large machines dedicated primarily to scientific calculations.

The GAMM (Gesellschaft für Angewandte Mathematik und Mechanik) and the ACM (Association for Computing Machinery) decided in 1957 to form working groups on this common language. The Zurich meeting in 1958 resulted in the merger of European and American proposals under the name IAL (International Algebraic Language), later renamed ALGOL 58.

In 1959, John Backus made a breakthrough by inventing a formal notation for describing programming language syntax. This Backus-Naur Form (BNF) replaced imprecise natural language descriptions with a rigorous definition. Peter Naur perfected this notation and integrated it into the definition of ALGOL 60, thus laying the groundwork for a now-classic method.

ALGOL 60 distinguished itself through several innovations: code blocks with their own lexical scope, recursive procedures, parameters passed by name or by value. Its mathematical rigor was evident in the requirement to declare data types. Control structures such as the `for` loop achieved an unprecedented level of elegance and clarity.

This new approach stimulated research on compilation techniques. Friedrich Bauer and Klaus Samelson published their method for sequential translation of formulas in 1960, while Edsger Dijkstra achieved the feat of developing an ALGOL 60 compiler in just ten months.

Computer science theory was enriched through ALGOL. Seymour Ginsburg and Gordon Rice established a connection in 1962 between languages defined by BNF and Chomsky's linguistic work on context-free grammars. This convergence between computer science and linguistics fostered the emergence of theoretical computer science.

ALGOL's influence reached the academic world. Scientific journals adopted it for algorithm publication, notably in the "Algorithms" section of *Communications of the ACM* from 1960 onward. European universities made it their preferred teaching language.

Despite these successes, ALGOL struggled to establish itself against FORTRAN in the commercial world. This paradox can be explained by the absence of a major industrial player to support it, the lack of standard libraries for input-output operations, and the inertia related to existing

FORTRAN programs. Europe, less dependent on this legacy, adopted it more widely, particularly in Germany where the government integrated it into university systems.

From 1962 onward, responsibility for ALGOL's development fell to Working Group 2.1 of IFIP (International Federation for Information Processing). Work on a successor resulted in ALGOL 68, a more complete but also more complex language. Its definition sparked heated controversies within the group, to the point that in 1969, nearly half the members published a "Minority Report" expressing their disagreement.

ALGOL inspired Pascal, Simula (precursor to object-oriented programming), and Ada. Its fundamental concepts became essential references: structured blocks, lexical scope, recursion. BNF notation remains today the standard method for defining programming language syntax.

ALGOL testifies to the maturation of computer science as a scientific discipline. It transformed programming from a craft practice into a formalized activity. Its mathematical vision shaped 20^{th}-century computational thinking. While its commercial success was limited, its intellectual influence has shaped generations of languages and programmers.

Lisp

In 1956, the Lisp project emerged as part of the Dartmouth summer research on artificial intelligence. John McCarthy was then seeking an algebraic language to process lists on the IBM 704. He took an interest in the work of Newell, Shaw, and Simon on IPL 2, a language designed for the RAND Corporation's JOHNNIAC, but refused to reproduce its form, which was too dependent on its original environment. The algebraic approach of Fortran appealed to him more.

Two circumstances favored this development: the creation of an IBM computing center at MIT, accessible from Dartmouth, and the launch by IBM of a program for demonstrating geometric theorems based on an idea by Marvin Minsky, for which McCarthy worked as a consultant. IBM seemed to be committing to emerging AI research.

In 1958, McCarthy proposed the concept of the Advice Taker, a system designed to represent information through sentences in a formal language and make decisions through logical deductions. The representation of sentences by list structures naturally emerged, as did the use of a list-processing language to program deductive operations.

The first technical challenge concerned the implementation of list structures on the IBM 704. This computer used 36-bit words, with two 15-bit sections named address and decrement. The addressing limited to 15 bits made it necessary to use pointers of identical size for list structures. McCarthy initially established functions including `cwr` (Contents of Word in Register) and four extraction functions. He noted that extracting a sub-expression required composing the address extraction with `cwr`, while progressing through the list required composing the decrement extraction with `cwr`. These observations gave birth to the `car` (Contents of Address Register) and `cdr` functions, which would become emblematic of the language.

During 1957 and 1958, McCarthy designed conditional expressions while programming chess routines in Fortran. The limitations of Fortran's `IF` statement led him to create an `XIF(M,N1,N2)` function returning N1 or N2 depending on whether M was zero. This solution simplified programs but suffered from the systematic evaluation of all three arguments. He invented the true conditional expression, evaluating only the necessary expressions.

In the summer of 1958, invited by Nathaniel Rochester at IBM, McCarthy worked on differentiating algebraic expressions. This work introduced several advances: writing recursive functions using conditional expressions, the `maplist` function forming a list of applications of a functional argument to the elements of a list, and the use of Church's λ notation to name functions. His recursive definition of differentiation did not provide for erasing abandoned structures, as he refused to compromise the elegance of his definition.

In the fall of 1958, McCarthy joined MIT as an assistant professor in communication sciences and launched the MIT artificial intelligence project with Minsky. Lisp development began with the initial goal of creating a compiler. The team started by manually compiling various functions into assembly language and writing Lisp environment subroutines to read and write list structures.

Programs were first written in an informal notation designed to resemble Fortran. This notation, called M-expressions, used brackets for function arguments, reserving parentheses for list structure constants. The M notation would never be fully formalized, as the representation of Lisp functions by lists emerged as the dominant approach after the interpreter appeared.

The `READ` and `PRINT` functions established a standard external notation for symbolic information. For example, the expression $x + 3y + z$ became `PLUS X (TIMES 3 Y) Z`. This notation would be named Cambridge Pol-

ish, in reference to Lukasiewicz's prefix notation and its earlier use by Quine.

Memory management raised a particular problem. Explicit erasure, used in IPL, lacked elegance. Two options presented themselves: erasing the old contents of a variable when updating it, which required reference counters impossible to integrate into the six available bits, or using garbage collection, which abandoned storage until the free list was exhausted, marked storage accessible from variables and the stack, then transformed unmarked storage into a new free list.

Subsequent versions of Lisp brought various functionalities: property lists associated with atoms, insertion and deletion of elements in lists (via `rplaca` and `rplacd`), efficient application of numbers as atoms, and the *program feature* for writing sequential programs with assignment statements and jumps.

Two reasons explain Lisp's longevity. On one hand, its core occupies a form of local optimum in the space of programming languages, with the recursive use of conditional expressions, the representation of symbolic information by lists, and the uniform representation of programs and data. On the other hand, Lisp offers unique operational characteristics that make it a privileged vehicle for symbolic computation and artificial intelligence systems: its runtime system providing access to host machine functionalities, its internal representation facilitating compilation from higher-level languages, and the availability of its interpreter as a command language.

DEC PDP-1

In 1960 came the PDP-1 (Programmed Data Processor), the first commercial computer from Digital Equipment Corporation. This machine broke with the computing behemoths of the era through its radically different philosophy of wanting to make the computer accessible and interactive.

Kenneth Olsen and Harlan Anderson, founders of DEC in 1957, designed this computer by drawing inspiration from the MIT Lincoln Laboratory's TX-2, where they had previously worked. The TX-2 served as their model, but they simplified its architecture while retaining its interactive dimension.

One of the PDP-1's distinctive features lay in its advanced technical performance. Its ferrite core memory contained 4,096 18-bit words, with

a 5-microsecond cycle time. It processed 100,000 additions per second, surpassing contemporary mainframe computers by a factor of 2.5 thanks to its parallel architecture and 5 MHz circuits. The replacement of vacuum tubes with Micro-Alloy and Micro-Alloy-Diffused transistors reduced its footprint to just 17 square meters.

Its originality came above all from the direct interaction it offered with the user. The 16-inch cathode ray display, the Friden Flexowriter keyboard, and the paper tape reader reading 300 lines per second created an immediate user experience, while batch mode largely dominated computing practices.

From a technical standpoint, the PDP-1 brought several lasting innovations. Its instruction set comprised 32 opcodes with arithmetic, logical, and control operations. Its sequence break interrupt system managed multiple peripherals simultaneously. Multiple indirect addressing brought unprecedented flexibility in data processing. These characteristics would mark future generations of computers. This model used one's complement for its arithmetic, a choice abandoned in later versions in favor of two's complement, illustrating its experimental nature.

Sold for $120,000, the PDP-1's commercialization demonstrated DEC's audacity and challenged IBM computers. This machine required neither air conditioning nor a raised floor, operated on a simple 110-volt power supply, and incorporated self-diagnostic circuits for its maintenance. DEC thus sought to "democratize" access to computing.

The software environment included an assembler with macro support, a text editor called Expensive Typewriter, one of the first Lisp interpreters, and various utilities. The system accepted several additional peripherals such as a punched card reader, a high-speed printer, IBM magnetic tapes, or analog-to-digital converters.

In 1962, MIT students created *Spacewar!* on this machine. This program, considered the first computer video game, illustrated the PDP-1's graphical and interactive capabilities while inventing a new form of entertainment.

Despite the production of around fifty units between 1960 and 1969, the PDP-1's influence established DEC as a major player in interactive computing. Its architecture directly inspired future models in the series: PDP-4, PDP-7, PDP-9, and PDP-15.

Today, the Computer History Museum in Mountain View preserves three PDP-1s, one of which was still operational in 1995. These machines bear witness to a period when computing began to leave calculation centers to move closer to users. Unlike IBM, which favored centralized systems, DEC bet on human-machine interaction and accessibility. The PDP-1's

success validated DEC's strategy based on less expensive and more accessible machines. The PDP-1 thus marked the first steps toward computing where miniaturization and interactivity were paramount.

IBM 1401

The IBM 1401, announced in 1959, was the first computer widely adopted by businesses around the world. It marked the transition between the era of mechanical punched card machines and that of modern computing. At that time, business data processing—inventory, billing, customer accounts, payroll—was performed by passing stacks of punched cards through electromechanical machines that sorted, calculated, collated, and totaled. These operations were determined by manually wired control panels. Mainframe computers with vacuum tubes certainly offered flexible stored-program capability and magnetic tape storage, but their cost made them inaccessible to most businesses. By the mid-1960s, half of all computers in service worldwide were IBM 1401s or members of its family.

The story of the 1401 began in 1957, when IBM France and IBM Germany developed a prototype transistorized "universal accounting machine" (WWAM) in response to a European competitor. Despite using a control panel, the entry-level configuration proved too expensive. In Endicott, New York, a bold system architect, Fran Underwood, discovered that he could economically replace the control panel with stored-program control. Retaining the WWAM's data paths, he designed an instruction set suited to business applications and named the new computer SPACE (Stored-Program Accounting & Calculating Equipment).

SPACE development was led by Charles Branscomb, who expanded the team to 60 engineers working around the clock. He integrated a printer under development, discouraged superfluous features, and met cost targets for the entry-level system. By mid-1959, with a working prototype and sales training underway, the machine, officially renamed IBM 1401, was poised to revolutionize the business market.

On October 5, 1959, IBM announced the 1401 via closed-circuit television to 50,000 participants in 104 different locations. In just five weeks, 5,200 systems were ordered, exceeding projections for the product's entire lifetime and surpassing the total number of commercial computers then in service worldwide. A year later, the first system was delivered to Time-Life, which transferred 60 million punched cards to a few hundred

magnetic tapes. The tape-equipped 1401 systems thus freed users from a seven-decade-old practice: storing data on punched cards.

In 1962, 1401 revenues exceeded those of traditional accounting machines. In 1965, the 1401 and its family (models 1410, 1440, 1460, and 7010) represented half of all computers worldwide, reaching a peak of 16,000 systems in 1967. The majority of 1401s were leased. An entry-level system leased for $2,500 per month, equivalent to three accounting machines and a calculator. A typical system leased for $6,500 per month, or could be purchased for $500,000—approximately one-tenth the cost of a mainframe computer.

Considered a "small system" computer, a 1401 weighed between 2 and 3 tons, consumed up to 14,000 watts, and contained approximately half a million parts. Maintained primarily by IBM customer engineers, these systems were renowned for their reliability and often operated around the clock.

Technically, the 1401 relied on serial processing of digits and characters. During its 11.5-microsecond clock cycle, an instruction could add two decimal digits or process a character for input, output, or formatting for printing. Variable-length numbers and strings were stored at one digit or character per memory position. Memory size ranged from 1,400 to 16,000 positions, each 8 bits wide—6 bits for the character, 1 bit to mark the end of a number or string, 1 parity bit.

The 1401's processor operated at 87,000 cycles per second. For comparison, a current 3-gigahertz PC can add two 20-digit numbers approximately one million times faster than a 1401. The machine used inexpensive germanium alloy-junction transistors and point-contact diodes, mounted on approximately 4,000 SMS (Standard Modular System) cards. The ferrite core memory, consisting of a stack of grids of tiny magnetic rings threaded with fine wires, cost approximately 60 cents per bit—400 million times more expensive than current RAM.

The 1401's success was largely due to its peripherals: the 1403 chain printer, whose speed (600 lines/minute, rapid skipping) and exceptional print quality (horizontally rotating interchangeable chain) made it an industry standard; the 1402 card reader-punch (800 and 250 cards/minute); and the mainframe-type 729 magnetic tape drives (15 million characters per reel). The RAMAC 1405 disk drive (20 million characters) and an optical bank check reader were available from 1961.

On the software side, IBM offered assemblers for machine language programming (Autocoder, SPS), compilers for Fortran, COBOL, and RPG (Report Program Generator) languages, as well as sorting and input-output utilities. IBM did not develop an operating system for the 1401,

but created one for the 1410.

In late 1964, Honeywell challenged IBM's market dominance by announcing the H200, faster and more powerful, accompanied by its *Liberator* software capable of running 1401 programs without modification. In 1961, IBM began planning the unification of its many computer lines. The System/360, announced in April 1964, consolidated software, peripherals, and support into a family of compatible computers. A key technology was the flexible *control store* that allowed its entry-level model, the Model 30, to competitively execute 1401 programs, thus countering the H200. The 1401 emulation also allowed IBM to deliver Model 30s when S/360 system software fell behind schedule.

While development of the 1401 family slowed after the S/360 announcement, its members remained more numerous than S/360s until 1968. IBM continued to offer the 1401 until 1971, and PC emulators are available today.

The 1401 also leaves an unexpected legacy in the arts. The machine could produce music by melodically activating the 1403's print hammers or by modulating its radio wave emissions picked up by an AM receiver. Inspired by a recording of radio music made during a 1401 decommissioning ceremony in 1971, an Icelandic composer wrote a 60-movement orchestration in 2001 entitled *IBM 1401: A User's Manual* (available on Spotify, Apple Music, etc.). Finally, in the 1964 film "Dr. Strangelove", actor Peter Sellers makes his first appearance in front of a 1403.

The IBM 1401 continues to fascinate computing enthusiasts. Volunteers at the Computer History Museum, including a dozen former IBM customer engineers, have devoted more than 40,000 hours to restoring two 1401s, one acquired in 2003 from a German garage and the other in 2008 from a Connecticut home (where it had operated until 1995).

COBOL

In May 1959, the Pentagon hosted a meeting in Washington that would transform business computing for decades to come. Charles A. Phillips, head of the Data Systems Research Staff, gathered about fifty people: military personnel, civil servants, consultants, and engineers from the largest computer manufacturers. Each manufacturer at the time had its own language and its own machines. A program written for an IBM wouldn't run on a UNIVAC. How could efforts be pooled in a sector where administrative applications were beginning to proliferate?

This meeting gave birth to CODASYL, a committee tasked with examining computer languages. A smaller group, the "short-term committee", received the mission of analyzing the strengths and weaknesses of existing compilers, primarily AIMACO (its military variant), FLOW-MATIC (Grace Hopper's creation at Remington-Rand), and COMTRAN (a highly theoretical IBM project).

No one expected this committee to produce a new language. Yet in November 1959, six people locked themselves away for two weeks and emerged with the specifications for a language they named COBOL: Common Business Oriented Language. This document, finalized in April 1960, went far beyond a simple synthesis of existing languages.

COBOL's technical design was based on its structure of four divisions: `IDENTIFICATION` for program information, `ENVIRONMENT` to describe the execution environment, `DATA` to structure data, and `PROCEDURE` for instructions. The syntax resembled English sentences, making the code readable even for non-specialists. Variable names could contain up to 30 characters, an innovation that allowed the use of clear, understandable names rather than illegible abbreviations.

The moment of truth came on December 6 and 7, 1960, during a public demonstration: the same COBOL program executed correctly on two radically different machines, an RCA 501 and a UNIVAC II. For the first time, code portability became reality.

COBOL's early years were marked by several evolutions: COBOL-61, COBOL-61 Extended, then COBOL-65. ANSI standardization arrived in 1968, followed by other standards in 1974 and 1985. Backward compatibility, ensuring that old programs would continue to function, was a remarkable characteristic that was maintained.

The U.S. government required that any computer sold or leased to government agencies must have a COBOL compiler, and this was one of COBOL's success factors. This decision forced manufacturers to develop their own compilers, propelling COBOL to the status of an essential standard in business computing.

One might question the extraordinary longevity of this language born as a temporary solution, which several factors explain. First, its relative simplicity made it accessible to non-academic programmers who formed the bulk of corporate IT teams. Its readability facilitated application maintenance. Its orientation toward administrative data processing corresponded perfectly to the needs of the service sector.

Facing the Y2K bug, the world discovered with astonishment the omnipresence of COBOL in the workings of the economy. Staggering estimates showed that of the 300 billion lines of code in production worldwide

in 1997, approximately 240 billion were written in COBOL. More than 95% of financial and insurance data passed through these programs.

Against all apparent logic, COBOL crossed the Y2K threshold without losing its importance. In 1999, more than 50% of new critical applications were still being developed in this language, despite the emergence of Java. Projections for 2004-2005 estimated that 15% of new applications (5 billion lines) would be in COBOL, while 80% of deployed applications would constitute extensions to existing COBOL programs.

This unexpected resilience stems from the language's technical characteristics: no pointers, elementary data types, detailed description of files and print outputs. These limitations, which might seem handicapping for scientific applications, are actually assets for business applications by reducing the risk of catastrophic errors.

In our world obsessed with perpetual innovation, sometimes simplicity and stability prevail over sophistication. A language deemed obsolete by computer scientists since the 1970s continues to run the world's financial systems in the 21st century, demonstrating that theoretical elegance is not always a guarantee of practical relevance.

Machine Learning

Checkers as a laboratory for intelligence. This is how Arthur Samuel conceived his approach. This pioneer, active from the 1940s through the 1960s, transformed computing by infusing it with the ability to learn from experience, a capacity until then reserved for humans.

The choice of checkers was by no means random. In the landscape of artificial intelligence, games played a role comparable to that of Drosophila for geneticists—simple organisms, easy to manipulate, perfect for experimentation. Checkers, less complex than chess, offered an ideal testing ground where Samuel could observe learning mechanisms without getting bogged down in overly sophisticated rules.

In 1952, the first program came to life on the IBM 701. Three years later, the version equipped with autonomous learning captivated the public during a televised demonstration. Without reaching the level of champions, this program innovated through two methods: rote learning and learning by generalization.

The first technique, rote learning, resembled our own memory more than one might imagine. Samuel programmed his computer to remem-

ber each configuration encountered during its games. Concretely, when the computer analyzed a position, it calculated its value using an algorithm called minimax that simulates the best possible moves for each player. The machine then stored this position with its evaluation. In subsequent games, faced with a situation already seen, no need for lengthy calculations—the answer sprang instantly from its memory. Samuel integrated an ingenious mechanism he called "depreciation". To understand this, let us visualize a tree where each branch represents a possible move. The deeper we go into this tree, the further we advance into the hypothetical unfolding of the game. Samuel arranged for positions distant in this tree to be worth slightly less than those close to the present moment, even at equal game value. This subtlety encouraged the program to favor short paths to victory. Without this trick, the computer could have become mired in endless strategies, preferring to win in 50 moves rather than 10, simply because both scenarios mathematically led to victory.

His second method, learning by generalization, foreshadowed modern temporal difference learning techniques. Samuel had his program play against itself thousands of times. At each move, through a brilliant and intuitive idea, the computer subtly modified its way of evaluating the board. If a game position leads to other advantageous positions, then this initial position must be favorable. Imagine a move that seems mundane but systematically leads to winning configurations three moves later—the program gradually learns to value this apparently unremarkable move. The computer did not wait for the end of the game to adjust its evaluations; it learned continuously, after each movement. Without being explicitly told "this is good" or "that is bad", the program discovered effective strategies on its own. Its compass remained nonetheless material advantage—the number of pieces owned relative to the opponent—with a marked preference for kings, those crowned pieces with superior movement capabilities. This approach, remarkably modern, constitutes one of the first manifestations of autonomous learning in 20$^{\text{th}}$-century computing.

Beyond the framework of theoretical research, Samuel's work on non-numerical computation shaped the architecture of early IBM computers. The logical instructions he proposed became an industry standard, their utility proving essential for all non-mathematical processing. Before revolutionizing artificial intelligence, he had distinguished himself as an electrical engineer. A graduate of Emporia College in 1923, then holder of a master's degree from MIT in 1926, he taught there briefly before joining Bell Labs. His research on electron tubes, particularly on space charge between parallel electrodes, marked the era. During the war, his work on devices protecting radar receivers proved crucial.

In 1946, having become a professor at the University of Illinois, he

participated in the creation of one of the first electronic computers. It was there that he dreamed of a checkers program capable of defeating a world champion. This project remained unfinished until his arrival at IBM in 1949, where he worked on the 701, the firm's first stored-program computer. Samuel considerably improved its Williams tube memory, quadrupling its storage capacity and stabilizing its operation. The first public demonstration of his program so impressed Thomas J. Watson Sr., founder of IBM, that he predicted—correctly—a 15-point rise in the stock price.

Samuel contributed to IBM's international expansion, shaping the direction of European laboratories, notably in Vienna (computing) and Zurich (physics). His discretion probably explains why the importance of his work was not fully recognized until after his departure from IBM in 1966.

At Stanford, where he became a research professor, he continued his work on checkers until other programs surpassed his in the 1970s. He became interested in speech recognition and supervised numerous doctoral students. His talents extended to technical writing. He excelled at understanding confusing documentation and producing clear manuals.

Samuel continued programming to a remarkable age. His last contribution, at 85, was to adapt multi-font printing programs for Stanford's computer science department. Only Parkinson's disease brought this exceptional activity to an end. His last login was on February 2, 1990, which probably makes him the oldest active programmer of his time.

Backus-Naur Form

The first high-level languages emerged in the 1950s, bringing with them the challenge of clearly and unambiguously describing their syntax. John Backus provided an answer to this question that would transform how we design computer languages.

Previously, describing languages was almost an art of divination. Computer scientists struggled to explain syntactic rules in prose, producing often confusing texts. Fortran, Backus's creation at IBM, was no exception to this difficulty. Its documentation was full of convoluted explanations such as "A real number is any sequence of decimal digits with a decimal point preceding or intervening between two digits or following a sequence of digits, the whole potentially preceded by a plus or minus sign". Not exactly crystal clear.

In 1959, at the IFIP congress in Zurich, Backus presented a formal notation for describing ALGOL's syntax. His proposal, which he had developed without any particular theoretical pretensions (he later admitted his relative lack of knowledge in mathematical logic), could have gone unnoticed. But it was Peter Naur who immediately grasped the significance of this idea. He refined the notation and applied it to the definition of ALGOL 60. This collaboration gave birth to the name *Backus-Naur Form* (BNF), a designation proposed by Donald Knuth to replace the original term "Backus Normal Form" which he considered mathematically inadequate.

The brilliance of BNF lies in its ability to express recursion with simplicity. To define an integer, for example, writing `<integer> ::= <digit> | <integer><digit>` indicates that an integer is either a single digit or an integer followed by a digit. In a single line, we can describe numbers of any length.

Backus made another decisive choice by breaking free from hardware constraints. Unlike Fortran, which limited numbers for memory reasons to the interval $[10^{-38}, 10^{38}]$, BNF focuses on pure syntax without worrying about the physical limitations of computers. This conceptual break gave languages a theoretical dimension independent of their technical implementation.

BNF's success came quickly. Its conciseness was immediately appealing: a few lines replaced entire pages of explanations. Its precision swept away the ambiguities of natural language. Its operational aspect sealed the deal: BNF rules allowed for the direct construction of automatic syntax analyzers.

ALGOL 60 thus became the first language entirely defined by a formal notation. This standardization gave developers on different machines the ability to create compatible compilers, realizing ALGOL's universal ambition. The way computer languages were conceived was thereby transformed. BNF encouraged computer scientists to adopt a more rigorous and systematic approach. Students could grasp the structure of programming languages more clearly thanks to this pedagogical formalism.

Over time, extensions enriched the original notation. EBNF (*Extended BNF*) integrated operators to express repetition or the optional nature of elements more simply. Graphical representations emerged, but the fundamental principles established by Backus and Naur remained intact.

BNF illustrates the contribution of formal notations to computer science. It transformed a craft-based approach into a scientific method, driven by practitioners confronting concrete problems. The separation it establishes between syntax and implementation has become a cardinal

principle in language design.

More than sixty years after its introduction, in the 21st century, this notation remains fundamental. It serves today to define the syntax of programming languages, data formats, communication protocols, and various formal notations. BNF is thus one of those discreet yet powerful tools that transform an entire discipline without making headlines.

V

1960

The Dawn of a New Technological Era

The 1960s transformed our relationship with machines. Between two blocs glaring at each other in an undeclared war, computing carved out a path that exceeded mere military concerns. The shock caused by Sputnik in 1957 still echoed through the corridors of Washington. The creation of DARPA in 1958 was America's response: a think tank that would revolutionize our relationship with technology.

America then invested colossal sums in research. MIT, Stanford, Berkeley—these names became synonymous with innovation. In their laboratories, young men in white shirts and narrow ties imagined the future. The American government, worried about the rising Soviet power, spared no expense to maintain its technological edge.

Companies began equipping themselves with these enormous machines at dizzying prices. Banks, insurance companies, and industries sought to tame the growing flood of data overwhelming their accounting departments. These first computers, true electronic cathedrals, transformed work organization. Punched cards gradually replaced handwritten ledgers.

AT&T modernized its network with electronic switching. This technical change, hardly spectacular for the general public, constituted the first building block of future computer networks, a colossal edifice. Bell Labs engineers conducted the first digital transmission experiments.

The youth of the 1960s viewed the world differently. University campuses bubbled with new ideas. For these students, the computer was not merely a military or accounting tool. They saw it as a means of expression, a tool for freedom. This alternative vision of computing germinated in the minds of future microcomputer pioneers.

The space race stimulated innovation. How could a man be sent to the Moon without powerful computers? NASA demanded reliable computers capable of withstanding extreme conditions. Embedded computing was born from this constraint. Later, these advances were adapted to civilian needs. The Apollo program thus gave birth to innovations that subsequently extended beyond the space domain.

Europe, scarred by war, tried to keep pace. In France, Charles de Gaulle launched the Plan Calcul in 1966, dreaming of technological independence. The Compagnie Internationale pour l'Informatique was to embody this French ambition. The United Kingdom capitalized on the legacy of Alan Turing and the Bletchley Park computers. Germany, focused on its industrial reconstruction, automated its factories.

On the other side of the world, Japan awakened. Fujitsu and NEC, names unknown in the West, developed their first computers. The Land of the Rising Sun initiated its future dominance in electronics.

In 1964, IBM changed the rules of the game with its System/360. This family of mutually compatible computers created a de facto standard. Gone were the days when each machine spoke a different language. This standardization accelerated the adoption of computing in medium-sized companies. The software market found fertile ground to develop.

MIT bubbled with ideas. The Compatible Time-Sharing System allowed multiple people to use the same computer simultaneously, a concept that heralded interactive computing. At Xerox PARC, researchers invented the graphical interfaces that would later make computers accessible to everyone.

These technological transformations worried unions, who feared massive job losses. These questions were already being asked, so current with artificial intelligence. Society discovered that a technological revolution always brings social upheavals.

The computer fascinated the general public. Television showed these mysterious machines with blinking lights. In Stanley Kubrick's *2001: A Space Odyssey*, the computer HAL 9000, embodied both the promise and threat of artificial intelligence. Computing entered the collective imagination.

The late 1960s saw the birth of a new concern: personal data protection. The computerization of administrations raised unprecedented ques-

tions about privacy. The first legal texts framing these issues began to appear.

The 1960s thus saw computers leave laboratories to enter the real world. This decade defined the contours of our relationship with machines. The technical, economic, and cultural choices of this period continue to influence our digital present. In American universities of the 1960s, the history of the 21st century was being written. Computers weighed several tons and cost the equivalent of 4,000,000 euros today. Nowadays, everyone carries in their pocket a machine infinitely more powerful.

Quicksort

In 1961, Charles Antony Richard Hoare had no idea he had just designed one of computing's most enduring algorithms. His Quicksort emerged in a particular context where computers were processing ever-growing volumes of data and traditional sorting methods were showing their limits. As a young British researcher, he proposed a radically different approach.

The idea seems simple at first glance. You choose an element from the array, the "pivot", then reorganize the other elements around it: smaller ones on one side, larger ones on the other. You then repeat the operation on each half until you obtain a completely sorted array. This "divide and conquer" method revolutionized the approach to sorting in computing.

His 1962 publication in the *Computer Journal* presented the first formal results. He demonstrated that his algorithm clearly outperformed existing methods. But Quicksort's story was only beginning.

As early as 1965, R.S. Scowen spotted a weakness in the random choice of pivot. He developed Quickersort, which instead selected the median element of the array. This modification changed the game for partially sorted arrays, which had posed problems for Hoare's original version.

R.C. Singleton took a new step in 1969 with his *median of three* method. Instead of settling for a single element, he examined three values: the first element, the middle one, and the last. He then chose the median of these three as the pivot. This trick improved average performance by about 5%, a substantial gain for the time.

Robert Sedgewick truly transformed understanding of the algorithm. Defended at Stanford in 1975, his doctoral thesis mathematically dissected Quicksort in all its aspects. His work, subsequently published in 1977 in *Acta Informatica*, established precise formulas for calculating execution

time on real machines. He didn't stop at theory and introduced *loop unwrapping*, a technique that reduced the overhead of managing internal loops.

The following year, in 1978, Sedgewick proposed a new partitioning method that would become the reference. Two indices traversed the array in opposite directions, gradually approaching each other. This approach minimized the number of element swaps, speeding up execution.

The 1980s saw the birth of more sophisticated variants. Roger L. Wainwright explored a hybrid approach in 1985 with Bsort, which mixed Quicksort and bubble sort techniques. Two years later, he developed Qsorte, capable of detecting already sorted subsequences and thus avoiding unnecessary partitioning.

In 1993, Bentley and McIlroy's work aimed to create an optimized version for the C language standard library. They designed an adaptive algorithm that changed strategy according to array size. Their most remarkable innovation remained the three-part *fat* partitioning, specifically designed to efficiently handle arrays containing many identical elements.

At the end of the 20th century and the beginning of the 21st, researchers adapted Quicksort to character strings, parallel architectures, and large databases. The algorithm became embedded in virtually all standard libraries of modern programming languages.

Beyond its performance, Quicksort captured minds through its elegant structure. It served as a textbook case for teaching divide-and-conquer techniques and probabilistic analysis of algorithms. Computer science students worldwide dissected its operation, learned to calculate its complexity, and understood why it worked so well on average.

Quicksort's evolution reflects that of algorithmics as a whole. Early work focused on theoretical complexity, seeking to prove mathematically the efficiency of methods. Gradually, attention shifted toward practical performance.

More than sixty years after its birth, Quicksort remains fully relevant. Its average speed, low memory consumption, and reliability make it the default choice for many computing systems. Modern processors, new architectures, and ever-growing data volumes continue to inspire researchers who still adapt this venerable algorithm.

Bull Gamma 60

In 1960, Compagnie des Machines Bull, a flagship of French industry, unveiled an exceptional creation: the Gamma 60. This machine, designed to handle both commercial and scientific tasks, made its mark through bold technical innovations.

The Gamma 60 stood out through its radical approach to information processing. Its distinctive feature was its ability to manage functions or problems simultaneously without specific programming of this parallelism. A technical achievement that enabled maximum utilization of hardware resources.

The internal organization followed a forward-thinking modular logic. Its core operated around a main unit equipped with magnetic core memory, coupled with a superior control unit. The latter comprised two strategic elements: a program distributor and a data distributor. Various specialized units revolved around these: arithmetic, logic, comparison, translation, magnetic drum storage, not to mention input-output devices such as card and tape readers, punches, and printers.

The main memory utilized saturated magnetic core technology, with 32,768 "catenæ"—blocks of 24 bits representing approximately 786,432 bits. This capacity allowed storage of 196,608 decimal digits or 131,072 alphabetic characters, with each catena accessible in just 11 microseconds. The Latin term "catena" meaning "chain" was deliberately chosen by the engineers to distance themselves from Anglo-Saxon computing vocabulary. This evidently did not catch on.

The Gamma 60's coding system demonstrated remarkable adaptability. Numerical data used a 4-bit decimal code, while alphanumeric data employed 6 bits. Instructions appeared in pure binary. This variety of formats aimed to save space, encode efficiently, simplify conversion between media, and maximize transfer rates.

The arithmetic unit performed operations on fixed-point and floating-point numbers. The numerical representation utilized two catenæ, structured with a sign bit, 40 bits to encode 10 decimal digits, and 7 bits dedicated to the decimal exponent, covering a range from 0 to 79. Depending on their complexity, instructions required 2 to 4 catenæ. For example, a three-address floating-point addition took 150 microseconds, compared to 88 microseconds for a single-address version.

The system's peripherals displayed solid performance. The magnetic tape units used half-inch media with 8 information channels. With a density of 200 bits per inch and a running speed of 50 inches per second, they achieved throughputs of 20,000 decimal digits or 13,333 alphabetic char-

acters per second. The readers processed 300 punched cards per minute, while flywheel printers produced 300 lines per minute, each capable of containing 120 characters from a palette of 60.

The Gamma 60's true breakthrough lay in its control organization. Each functional element had its own program and its own control unit. Once configured, these modules operated in complete autonomy. Only the main memory and transfer buses were shared among the different units. The program distributor orchestrated the distribution of instructions and established the necessary locks to prevent conflicts between concurrent programs. In parallel, the data distributor managed access priorities to the transfer buses.

This sophisticated architecture enabled the side-by-side execution of highly heterogeneous tasks: payroll calculation, matrix inversion, multiple conversions from tape to printer, from cards to magnetic tape, or from punched ribbons to magnetic tape. And this with virtually no performance loss: the time lost in this simultaneous execution almost never exceeded a few percent compared to sequential processing.

Emerging from the French computer industry of the 20^{th} century, the Gamma 60 was marketed starting in 1960, but its trajectory was unfortunately cut short by Bull's financial difficulties, which led to the company's acquisition by General Electric in 1964. This acquisition ended the independent development of the machine, but its concepts regarding parallel processing and modularity were innovative.

APL

At Harvard, Kenneth E. Iverson worked on his doctoral thesis under the supervision of Howard Aiken and Wassily Leontief. His research focused on the automatic solution of linear differential equations within an economic model. This initial foray into programming raised a question for him: how could algorithms be better represented and data manipulated?

The answer began to take shape in 1956 when Iverson taught in the brand-new Automatic Data Processing program that Aiken had just established at Harvard. He started developing a formal notation to describe and analyze data processing. This notation, which did not yet have a name, initially served as a pedagogical tool. Together with Frederick P. Brooks Jr., then a doctoral student, he wrote a book entitled "Automatic Data Processing" that formalized these early ideas.

In 1957, Iverson worked as a consultant at McKinsey & Company. His notation found concrete application there, acting as a communication language between designers and programmers working on complex systems. The teams discovered the value of this approach for clarifying architectures that were otherwise difficult to grasp.

Career circumstances brought Iverson to IBM in 1960. There he met Adin Falkoff, who immediately took interest in this notation for his work on associative memory systems. This collaboration would transform an academic idea into a new computing paradigm.

In 1962, Iverson published "A Programming Language". The work formally presented his notation, but it remained confined to blackboards and paper. To use it on a computer, it had to be manually translated into Fortran or another existing language. A frustrating but necessary intermediate step.

Everything changed in 1964. Iverson and Falkoff decided to transform this theoretical notation into an operational programming language. Lawrence M. Breed joined them in 1965 to design a practical version of APL intended for interactive time-sharing systems. The first constraint to overcome concerned characters. The IBM 1050 terminal could only display 88 symbols. This limitation, initially perceived as a handicap, pushed the team to rationalize and generalize several aspects of the language. Ultimately, this constraint proved beneficial as it forced a salutary conceptual clarification.

By 1970, APL won over its first users. Thousands of people adopted it, including IBM's research and design teams, staff at NASA's Goddard Space Flight Center, and financial planners at major corporations. The language attracted users through its elegance and practicality, despite criticism from the established computing community who judged it too esoteric.

At Johns Hopkins University's Applied Physics Laboratory, the F.T. McClure Computing Center pushed innovation further. The team developed special APL system software for the IBM 360/91 computer. These developments exploited the machine's vector processing capabilities and radically transformed execution performance. In 1978, the center took another step forward with an APL system capable of handling very large applications through virtual memory technology.

APL's true innovation lay in its treatment of arrays as primitive objects. This approach revolutionized the way algorithms were conceived. Programmers naturally thought in terms of vector and matrix processing rather than loops and conditions. The language distinguished itself through its remarkable syntactic simplicity with only three possible types

of statements. Name assignment, branching, or neither. The semantic rules remained few and the definitions of primitive functions were independent of data representations.

The 1970s saw APL progress despite limited support from major computer vendors. The introduction of the shared variable system in 1973 with APLSV was an innovation that solved several APL/360 system problems, particularly communication between concurrent programs and with external devices. The possibilities expanded considerably.

The following decade confirmed this expansion. APL was installed on virtually every computer on the market, including personal computers such as the IBM PC and the Tandy TRS-80. Many commercial timesharing companies offered APL as a service. Some specialized in international financial databases updated in real time, exploiting the language's ability to efficiently process large volumes of numerical data.

Unlike other major languages of the era that moved away from traditional mathematical notations, APL continued their development. It retained established symbols when possible and employed consistent mathematical terminology. This fidelity to mathematical conventions did not prevent occasional departures dictated by principles of simplicity and uniformity, such as adopting a single form for dyadic and monadic functions.

APL's development illustrates the importance of clearly stated principles. First, simplicity, pursued through uniformity, generality, and conciseness. Then pragmatism, manifested by attention to real applications and hardware limitations. The fact that the language was developed for eight years without a machine installation provided a freedom to modify that would have been impossible for languages constrained by an existing user base.

Simula

In 1961 at the Norwegian Computing Center, Kristen Nygaard initiated the development of a new programming language intended for discrete event simulation. This project, which would become Simula, marked the beginning of a close collaboration with Ole-Johan Dahl, who joined him in 1962. The two researchers shared experience at the Norwegian Defence Research Establishment, but their backgrounds differed: Nygaard had focused on operations research and Monte Carlo simulations, while Dahl had developed system software and programming languages.

The development of Simula responded to the concrete need for appropriate tools to simulate complex systems such as neural networks, communication systems, or traffic flows. However, programming such simulations in machine language or Fortran proved difficult. What was missing was a coherent set of concepts for understanding and describing the structure and interactions in these complex systems.

The first version of the language, Simula I, was developed between 1962 and 1964 under contract with UNIVAC. The designers chose to base their language on ALGOL 60, particularly for its block structure and programming safety. The major innovation of Simula I lay in the introduction of the concept of processes, which represented entities capable of executing actions in quasi-parallel while carrying their own data. These processes could interact and be programmed to simulate the behavior of real-world objects.

The development of Simula I required a new memory management system based on linked lists, developed by Dahl, freeing itself from the "last in, first out" (LIFO) constraint of ALGOL 60 and allowing processes to exist independently of the block call structure. The compiler was operational by the end of 1964, and the language achieved some success in the simulation community. It was notably used on UNIVAC 1100, Burroughs B5500, and URAL 16 computers. However, Dahl and Nygaard realized that the concepts developed for Simula I could be useful beyond simulation, for general programming and system design.

This reflection led to the development of Simula 67, a complete redesign of the language integrating the concepts of class and inheritance. The inspiration came partly from Tony Hoare's work on record manipulation, presented at a NATO summer school in 1966. The conceptual breakthrough occurred in January 1967 with the idea of "class prefixing", meaning that one class could prefix another, thus creating an inheritance relationship where the second inherits the properties of the first.

Simula 67 introduced fundamental concepts that would have a lasting influence on programming: objects as data structures with their associated operators, classes as templates for creating these objects, inheritance for specializing classes, and virtual procedures allowing the modification of object behavior in subclasses. The language distinguished the internal view of an object (its variables and constants) from its external view (the procedures accessible remotely).

The standardization of Simula 67 took place at the Simula Common Base Conference in June 1967, which established the Simula Standards Group to manage the language's evolution. The first installations appeared as early as 1969 on Control Data computers, followed by versions

for UNIVAC and IBM. However, the language's dissemination was hindered by the high prices demanded by the Norwegian Computing Center, under pressure from the Norwegian Research Council, which considered that a programming language had only a three-to-five-year lifespan.

Alan Kay and his team at Xerox PARC drew inspiration from Simula to create Smalltalk, which popularized object-oriented programming in the 1970s. Simula's concepts influenced the development of graphical interfaces, first on the Macintosh and then in Windows. The language inspired numerous successors, such as C++ by Bjarne Stroustrup, which widely disseminated object-oriented programming, Eiffel by Bertrand Meyer, and more recently Java.

The success of Simula's ideas is partly explained by their origin in simulation modeling, where design in terms of cooperating objects is natural. The generality of the approach allowed its application to numerous aspects of software development. The concepts introduced by Dahl and Nygaard have become so ubiquitous.

SNOBOL

The year 1962 witnessed the birth of a project in the corridors of Bell Labs in Whippany, New Jersey, that would change the way string processing was conceived. David J. Farber, Ralph E. Griswold, and Ivan P. Polonsky were working at the Programming Research Studies department, a unit that would soon relocate to Holmdel. Under Chester Lee's direction, this laboratory explored diverse territories ranging from automata theory to graph analysis and associative processors. Their daily work confronted them with the frustrating limitations of the SCL language, created by Chester Lee himself. Disappointing performance, restricted data space, and convoluted syntax that turned simple operations into veritable obstacle courses finally convinced them that an alternative was necessary.

After some attempts with COMIT, a language dedicated to natural language processing, the trio decided to take matters into their own hands. The development of what would become SNOBOL began in utmost secrecy, unbeknownst to Chester Lee, who was precisely refining a new version of his SCL. This somewhat comical situation took an embarrassing turn in November 1962 when both projects came to light. The department could not afford to fund two competing developments, but the SNOBOL team had a considerable head start. Chester Lee, being a good sport, finally accepted the inevitable.

The naming of the new language was rather improvised. First called SCL7, then renamed SEXI for String EXpression Interpreter, it ultimately inherited the name SNOBOL: StriNg Oriented symBOlic Language, a deliberately whimsical pseudo-acronym. This joke would later embarrass its creators during official presentations, never imagining the impact their creation would have.

In early 1963, the first operational version ran on the IBM 7090. The team made the bold choice of pure interpretation, immediately dismissing the idea of a traditional compiler, which they deemed unsuitable for the intended specificities. This decision proved rewarding, as three weeks were sufficient to obtain a usable version, an achievement in which the team took legitimate pride and which proved that a programming language could come to life quickly without consuming considerable budgets.

SNOBOL disrupted established practices by treating character strings as unified entities rather than simple arrays. This approach was accompanied by sophisticated pattern recognition mechanisms with backtracking, opening unprecedented perspectives for text analysis. The success achieved at Bell Labs naturally led to enhanced versions. SNOBOL2 arrived as early as 1964 with built-in functions for manipulating strings and numbers. SNOBOL3 followed by introducing user-definable functions.

In 1966, SNOBOL4 revolutionized the approach. Patterns became data objects in their own right, multiple types appeared including arrays and tables, dynamic compilation made its entrance with unevaluated expressions. The use of an abstract machine language called SIL guaranteed remarkable portability.

SNOBOL's distribution philosophy broke from the practices of the time. Unlike established customs, the language spread freely, without any restrictions, source code included. This unusual generosity, accompanied by free technical support from Bell Labs, facilitated its massive adoption in universities and contributed greatly to its influence in the computing community.

Applications exceeded initial intentions. Designed to manipulate formulas and analyze graphs, SNOBOL found its way into document formatting and humanities computing. Compilation, rapid prototyping, and non-numeric data processing further expanded its scope. Its concepts of string manipulation and pattern recognition influenced numerous subsequent creations, while SNOBOL extensions enriched Fortran, PL/I, and many others.

Starting from a practical need of a few researchers, SNOBOL ended up transforming the approach to string processing and durably influencing the evolution of programming languages. It also demonstrates that freedom of

experimentation and generosity in distribution, well before free software, constitute powerful drivers for technological innovation.

Spacewar!

In 1962, MIT had just received its new PDP-1 from Digital Equipment Corporation, a machine that stood apart from the usual computing behemoths. This computer featured a cathode ray tube display capable of showing images in real time, a technical feat that would prove game-changing.

Steve Russell, a young programmer passionate about science fiction, discovered this machine with wonder. An avid reader of Edward Elmer Smith's space adventures and his *Lensman* series, he immediately envisioned recreating on this screen the cosmic battles that had so captivated his imagination. With a few accomplices, he embarked on creating what would become *Spacewar!*, unaware he was giving birth to the first interactive video game in history.

The principle they developed seemed deceptively simple: two spaceships battling in space, each controlled by a player. Rotation, acceleration, torpedo firing—the basic controls were enough to create fierce duels. But Russell and his team placed a star at the center of the screen whose gravitational pull disrupted trajectories and turned each match into a tactical challenge.

The PDP-1 wasn't designed for this kind of exercise. Lacking multiplication and division instructions, it forced the programmers to exercise considerable ingenuity in calculating rotations and trajectories. This technical constraint, far from discouraging the team, stimulated their creativity. Alan Kotok and Bob Saunders also built custom control boxes, abandoning the computer's crude panel switches for controls worthy of a space cockpit.

Peter Samson pushed the refinement even further. He programmed a star field of astronomical precision, faithfully reproducing the celestial vault. This addition, ironically dubbed *Expensive Planetarium*, transformed the screen into a window on the cosmos. The team soon added a "hyperspace" function, a panic button that teleported the distressed ship to a random point on the screen, at increasing risk of self-destruction with each use.

The distribution of *Spacewar!* followed a different rhythm than the

legends tell. The 55 units that came off the production lines, about twenty of which equipped with displays, made the PDP-1 a rare machine. The game first circulated within the closed circle of universities and research centers that possessed these exceptional computers. It took until the late 1960s and the arrival of new display models from IBM, DEC, HP, and Control Data Corporation for *Spacewar!* to truly begin spreading.

Each installation developed its own variants. At the University of Minnesota, Albert Kuhfeld devised the "Minnesota Panic Button" in 1967, a camouflage device that made ships invisible. At Cambridge, M.S. Peterson and John C. Viner created DUEL, a version where space transformed into a "viscous" environment that gradually slowed combatants. At the University of Illinois, on the PLATO system, Richard Blomme adapted the game with ASCII character graphics and invented a player matchmaking system of striking modernity.

These programming nights at MIT never took place during official hours. Students waited for the machines to free up from their research tasks before indulging in their experiments. This nocturnal clandestinity forged an unprecedented computing culture, where the pleasure of programming trumped institutional constraints. *Spacewar!* was an unexpected training ground where students who programmed it absorbed computing principles faster than with traditional academic methods.

The game soon attracted attention beyond the academic world. In 1971, Nolan Bushnell and Ted Dabney, after discovering *Spacewar!* at Stanford, created *Computer Space*, the first attempt at commercializing video games in arcades. Bill Pitts and Hugh Tuck followed an identical path with *Galaxy Game*, an adaptation installed at Stanford. These initiatives marked the early stirrings of an industry that would revolutionize entertainment.

Russell and his team's approach broke with the utilitarian vision of computing. Their machines, initially designed for scientific calculation and administrative management, suddenly revealed their creative and recreational potential. *Spacewar!* demonstrated the importance of interactivity, graphical interface, and user-centered programming—concepts that would influence the development of personal computing.

The source code freely circulating among universities represented the spirit of sharing that would later characterize free software. This openness encouraged programmers to modify and enrich the game, creating a multitude of versions. DEC adopted *Spacewar!* as a test program for its computers: if the game ran correctly, the machine was declared operational.

In 1972, Stewart Brand organized the "Spacewar Olympics" at the

Stanford Artificial Intelligence Laboratory, a tournament that thrust the game into the spotlight. *Rolling Stone* magazine covered the event, introducing the general public to this new form of electronic competition. Ralph Gorin had developed for the occasion a sophisticated version supporting up to five simultaneous players, enhanced with space mines and other tactical innovations.

ASCII

The ASCII code was born in the 1960s, when every computer manufacturer was inventing its own rules. At that time, exchanging data between machines was an ordeal. A character on an IBM computer was unreadable on a Control Data machine, and vice versa. This technological cacophony hindered any serious development of computing.

In 1961, the American Standards Association decided to end this chaos. Its X3 committee received the mission to create a universal standard. The project landed in the hands of subcommittee X3.2, then working group X3.2.4. John L. Little, an engineer at the National Bureau of Standards, took charge of this delicate undertaking. He had to convince manufacturers who were already using their own systems, like Control Data with its 6-bit FIELDATA or IBM with its 8-bit EBCDIC. But each one held firm to their positions.

After months of debate, the committee decided on a 7-bit code. This compromise made it possible to represent 128 different characters, sufficient to cover the Latin alphabet, numbers, and punctuation. A 6-bit code would have been too restrictive, while 8 bits would have increased transmission costs. The 7 bits also left the possibility of adding an eighth bit for error detection, a valuable function when telephone lines crackled.

ASCII was born in 1963, underwent a revision in 1967 before its final version in 1986. The designers reserved the first 32 positions for control characters, inherited from teleprinters, such as carriage return, line feed, and tabulation. The following 95 positions accommodated letters, numbers, and symbols. Nothing was left to chance in this organization. The numbers aligned to facilitate conversions, the letters followed alphabetical order, and punctuation found its logical place.

In November 1968, the National Bureau of Standards adopted ASCII as the first federal information processing standard. President Lyndon Johnson signed this decision, making ASCII mandatory for all US govern-

ment computer purchases. Since the government was the sector's largest customer, this measure pushed the entire industry toward ASCII.

In reality, 7-bit ASCII is generally integrated into an 8-bit code: either the additional bit is set to zero, or it serves for parity checking. This approach works well with computers that process information in groups of 8 bits. The parity bit detects transmission errors: in an even parity system, it adjusts to maintain an even number of 1 bits. If an error occurs during transfer, the count changes and signals the problem.

ASCII's success inspired national variants. Canada developed its version for French characters, India created ISCII, Vietnam VISCII, Yugoslavia YUSCII. Each country adapted the code to its linguistic needs. In 1972, ASCII was integrated into the international standard ISO/IEC 646, which authorized certain national rearrangements. But ASCII had established itself as the de facto standard, sometimes creating confusion when other countries redefined certain positions.

The arrival of 16- and 32-bit computers opened the way for 8-bit extensions. These versions kept the original 128 ASCII characters and added 128 new positions. The IBM PC introduced code page 437, which replaced control characters with graphic symbols. Digital Equipment Corporation designed the multinational character set for its VT220 terminal, one of the first extensions designed for international use rather than graphics. The ISO/IEC 8859 standard eventually prevailed, derived from Digital's system. Microsoft then developed Windows-1252, which added typographic symbols necessary for quality printing. These three systems—7-bit ASCII, ISO-8859-1, and Windows-1252—dominated computing until the late 2000s.

Unicode has since taken over from ASCII, capable of handling tens of thousands of characters. But Unicode's designers had the wisdom to preserve compatibility: the 128 ASCII characters retain their original codes. ASCII is valid in UTF-8, the currently dominant encoding, making it a subset of Unicode.

More than sixty years after its birth, ASCII continues to structure our digital exchanges. When everything collapses, when sophisticated formats fail, ASCII endures. The Internet rests on it, communication protocols use it, and our smartphones still decode its characters. This exceptional longevity testifies to a remarkable design, the fruit of a rare industrial consensus.

Bell 103 Modem

In 1963, AT&T commercialized the Bell 103, a modem model that would transform how computers communicate with each other. This device can be described as a simple and groundbreaking technology. It enabled bidirectional transmission of digital data over ordinary telephone lines at the respectable speed of 300 bits per second.

That year, computer networks were few and far between. Computers, those costly behemoths, were just beginning to spread to a few large corporations. Each machine remained isolated in its data center, unable to communicate with its peers. How could digital information travel over a network designed for the human voice? Yet the switched telephone network represented the only truly extensive and immediately available communication infrastructure.

AT&T found the solution in a technique called frequency-shift keying, hence its name: MOdulation/DEModulation. The principle rests on the clever idea of transforming each bit into a musical tone. The Bell 103 translated 0s and 1s into distinct sound frequencies that telephone lines could carry. For transmission, it used 1070 Hz to represent one bit and 1270 Hz for the other. On reception, it listened on different frequencies: 2025 Hz and 2225 Hz. This ingenious distribution allowed simultaneous communication in both directions, known as *full duplex*.

The connection between two Bell 103 modems resembled a musical ritual. The calling device first intoned a continuous note at 1270 Hz. Its correspondent answered with a tone at 2225 Hz. This acoustic handshake synchronized the two machines and confirmed they spoke the same language. Once this sequence was completed, data could flow freely.

The Bell 103 displayed disarming simplicity. No mechanism verified the integrity of transmitted data. Information traveled bare, without protection against interference or transmission errors. This minimalist approach reflected the technical constraints of that period, but also a certain confidence in the quality of the telephone network.

AT&T simultaneously launched its DATA-PHONE service. This opened the doors of the public telephone network to corporate computers. Machines previously imprisoned in their climate-controlled rooms suddenly discovered the existence of their counterparts. Files traveled from site to site, databases became remotely accessible, distributed processing took its first steps.

The speed of 300 bits per second makes us smile today, but it revolutionized the practices of the time. Thirty characters transmitted each second were more than sufficient for many professional applications. The

computer interface respected the RS-232 standard, that famous 9-pin serial port equipping all computer systems. This standardization accelerated the adoption of the Bell 103 by guaranteeing its compatibility with different computer brands.

The success of AT&T's modem created a de facto standard in the telecommunications industry. Its modulation frequencies and transmission method set the example. Competitors aligned with these specifications to ensure interoperability. This technical convergence culminated in 1981 when the International Telecommunication Union formalized these parameters in its V.21 recommendation, eighteen years after the birth of the Bell 103.

This exceptional longevity was explained by the robustness of the chosen technical solution. Frequency-shift keying, though rudimentary, also worked on failing telephone lines. It tolerated distortions, withstood interference, and maintained communication when other more sophisticated techniques failed.

Using the Bell 103 sometimes required picturesque manipulations. To establish a connection, the operator dialed the phone number, waited for the answer tone, then delicately placed the handset on an acoustic coupler. This device, resembling a plastic cradle, ensured the conversion between sound and electrical signals. Later versions fortunately integrated automatic dialing, making the procedure less acrobatic.

Against all odds, the Bell 103 survived until the 1990s, valiantly resisting the arrival of faster modems. This remarkable persistence stemmed from different factors. Its technical simplicity made it equipment of unshakeable reliability. When line quality degraded, it continued to function where its successors gave up. Its modest speed was then an asset, providing an acceptable fallback solution. Its ubiquity in businesses created a powerful network effect.

After the Bell 103, modem manufacturers refined modulation techniques to increase throughput. They added error correction protocols to secure transmissions. Automatic dialing and answering simplified daily use. These successive improvements led to the high-speed modems of the 1990s, capable of reaching 56,000 bits per second on the same analog telephone lines as their ancestors.

Music V

The 1960s saw the birth of the wild idea of making computers sing. At Bell Labs, Max Mathews observed oscilloscopes tracing the curves of the human voice and thought that after all, if speech could be converted into numbers, why couldn't the reverse be done? In 1963, he published an article in *Science* that would raise eyebrows: "The Digital Computer as a Musical Instrument". The idea seemed absurd. Computers calculate, they don't make music.

Yet the principle Mathews developed could be explained in a few lines. The computer generates a sequence of numbers which, transformed into electrical pulses and smoothed out, produce sounds. Nothing simpler in theory. In practice, it's another story. To obtain something decent, you need at least 30,000 samples per second if you want to cover audible frequencies up to 15,000 Hz. The catch is that Bell Labs' IBM 7090 only calculates 5,000 numbers per second for even moderately complex sounds. Mathews then found a workaround by first storing everything on magnetic tape, then playing it back at the right speed. This workaround worked perfectly, but it was only a beginning, because the real innovation came with MUSIC, and MUSIC V, a programming language entirely dedicated to sound creation.

The power of the innovation lay in the separation between instrument construction and their use. Mathews invented "unit generators", small program blocks that each did one specific thing like oscillating, generating noise, adding signals. By combining them, you create "instrument units", true virtual instruments. This modularity predated by several years the Moog and Buchla synthesizers that wouldn't see the light of day until after 1964.

The composer therefore worked in two stages. First, they built their instruments by assembling these generators like mounting an electronic circuit. Then they wrote their score as lists of numbers: start time, duration, pitch, amplitude... Each note was a line of figures. This numerical approach had to win over musicians first, but it offered unprecedented precision on every sound parameter.

Jean-Claude Risset, who joined Mathews' team at Bell Labs between 1964 and 1969, pushed the system to its limits. He analyzed trumpet sounds under an acoustic microscope and discovered that their distinctive timbre came from subtle correlations: the louder the note, the more high harmonics it contained. Armed with this observation, he programmed virtual trumpets that reproduced this behavior. The result fooled listeners. But he didn't stop at imitation and explored the unprecedented

possibilities offered by digital synthesis. He created sounds impossible in the physical world, like his famous "infinite glissando" that seemed to rise eternally while returning on itself, thus exploring the limits of auditory perception through the computer.

The research conducted with MUSIC V transformed the understanding of sound. It showed the importance of attack transients in timbre recognition, the role of random micro-variations in perceived richness. The computer was an acoustics laboratory where each parameter could be isolated, modified, and its effect on perception observed. However, MUSIC V suffered from crippling limitations. Real-time playing was impossible, you had to wait for the computer to finish its calculations before hearing the result. Programming with lists of numbers put off musicians accustomed to musical staves, and sound quality depended on a fine understanding of acoustics that few possessed. These obstacles didn't prevent certain composers from adopting the tool between 1964 and 1969, using the catalog of synthesized sounds published by Jean-Claude Risset as a reference. MUSIC V demonstrated that a computer could become a full-fledged artistic creation instrument, not just a calculating machine.

MUSIC V's modular architecture directly inspired analog and digital synthesizers. Its separation between instrument definition and musical control foreshadowed the MIDI standard of the 1980s. It established programming principles – modularity, abstraction, separation of concerns – that went far beyond the sound domain.

MUSIC V transformed the conception of computing by showing that the machine was also a means of expression rather than a simple automaton. Mathews' intuition finds its culmination today in every modern audio application that carries within it a bit of MUSIC V's DNA, that first digital cry uttered in the carpeted corridors of Bell Labs.

Sketchpad

In 1963, Ivan Sutherland worked in the laboratories at MIT on the problem of interaction between humans and machines beyond simply typing lines of code on a keyboard. His answer was called Sketchpad, and it would be a game-changer. For the first time, drawing directly on a computer screen with a light pen became possible.

The idea was straight out of science fiction. Engineers who wanted to draw an electrical circuit or a mechanism first had to describe it in writing,

line by line, instruction by instruction. Needless to say, designing even the simplest geometric shape was an ordeal. Ivan Sutherland understood that this logic needed to be broken.

He installed his system on the Lincoln Laboratory's TX-2 computer, a machine equipped with a 9-inch cathode ray tube display. The arsenal consisted of a light pen for drawing and selecting, push buttons to launch commands, switches to activate various functions. A few rotary potentiometers made it possible to rotate or enlarge what appeared on screen. But the real magic happened elsewhere. Unlike a traditional drawing fixed on paper, a Sketchpad drawing kept in memory the relationships between its various elements. The system knew that one line remained parallel to another, that a circle stayed tangent to a straight line, that an angle preserved its measurement. When you moved part of the drawing, the entire assembly automatically adjusted to respect these relationships. This technical feat relied on the ring data structure, a bold software architecture. Each stroke, each curve transformed into a block of information that stored its shape, position, and connections with the other elements composing the drawing. The elements communicated with each other, coordinated, maintained their geometric agreements.

Ivan Sutherland took the concept further with his *master drawings*. You drew a symbol once—an electrical resistor or a gear—and could then reproduce it at will in the drawing, with each copy linked to its original model. If you modified the base form, all its instances transformed instantly. Industrial draftsmen, accustomed to constantly redrawing the same components, discovered an incredible time-saver.

Applications multiplied quickly. Printed circuit board designers found their footing in drawing repetitive patterns. Mechanics simulated the operation of joints and connecting rods. Engineers calculated the distribution of forces in a structure like a bridge and immediately visualized the results. The interface established rules that would span decades. The light pen functioned in two modes: it captured coordinates when positioning a new element, and it designated existing objects when you wanted to modify them. The system often guessed the user's intentions through its "pseudo-position": the cursor automatically locked onto nearby points and lines, facilitating drawing precision. Sketchpad transformed the screen into an intelligent drawing board.

The limitations of the era nevertheless constrained certain ambitions. Constraint calculations sometimes slowed the system down on complex drawings. Screen resolution and storage capacity imposed their law. But as early as 1963, Timothy Johnson extended Sketchpad into the third dimension with Sketchpad III. Lawrence Roberts built upon these data

structures for his work on object recognition in photographs, thus initiating computer vision.

Ivan Sutherland conceived of the computer not as a sophisticated calculator, but as an amplifier of human intelligence. This vision inspired the entire personal computing movement that followed. By showing that a natural conversation between humans and machines involved images as much as text, Sketchpad redrew the contours of computing. Sixty years later, our touchscreens and drawing software still bear the trace of this pioneering intuition, but so too do generative artificial intelligences with their vision capabilities.

GECOS

Bull was going through a turbulent period when General Electric acquired the company in 1964. This acquisition triggered the creation of GECOS – General Electric Comprehensive Operating System – designed to equip the GE-635 system. The name bears the mark of the industrial upheavals that characterized computing at that time. Six years later, Honeywell absorbed Bull General Electric and once again redrew the company's boundaries.

These successive ownership changes were not mere financial maneuvering. They directly shaped the technical evolution of operating systems and determined future strategic directions. When CII and Honeywell Bull merged in 1976, the legacy became even more complex. The commercial offering presented a bewildering peculiarity where three distinct operating systems coexisted under the same GCOS banner.

This tripartition reflected the fragmented geographic and technical origins of the system. GCOS6 originated from Honeywell's research laboratories in Boston, GCOS7 emerged from Bull's engineering offices in Paris, while GCOS8 inherited the initial developments from General Electric in Phoenix. Three cities, three teams, three technical approaches that gave birth to systems bearing the same name but incapable of communicating with each other.

The mutual incompatibility of these three lines constituted a troubling commercial paradox. Despite their common designation, no application could migrate from one to another. The only possible exchanges occurred through telecommunications, as if these systems belonged to distinct technological universes. This situation perfectly illustrated the integration difficulties that arose when multiple technical legacies collided abruptly.

Bull's architecture in the 1980s also included CTOS, a multi-station system developed by Convergent Technology, which joined the family after CIT Transac acquired the technology. DNS completed this heterogeneous ensemble by managing the lower layers of telecommunications: X21 and X25 protocols, Ethernet, transport, and various terminals. This system ran on a Datanet front-end shared between GCOS7 and GCOS8, while integrating directly under GCOS6. This diversity reflected Bull's strategic dilemmas in the face of changes in the computing market. As early as 1982, Jacques Stern formulated the clear-eyed assessment that Bull would never reach the critical mass to compete globally with its proprietary systems alone, regardless of their technical quality. This realization triggered reflection on the balance between maintaining the existing base and preparing for the future.

Bull's engineers then devised technical solutions to facilitate this delicate transition. A UNIX coprocessor was created for GCOS7 and GCOS8, functioning as a front-end tightly coupled to the proprietary processor. This hybrid architecture allowed the addition of modern functionalities to legacy applications such as relational databases, web interfaces, or contemporary development tools. For GCOS6, the approach differed with the creation of an emulator under AIX, IBM's operating system, designed to recover the code from business applications that had survived the transition to the year 2000. The maintenance of these three distinct lines was organized according to a geography inherited from Bull's turbulent history. Boston continued to develop GCOS6, Paris handled GCOS7, Phoenix steered the evolution of GCOS8. This international distribution testified to the company's ability to preserve distinct centers of expertise despite economic turbulence and ownership changes.

The alliance with IBM in 1992 stemmed from a realistic analysis: Bull could no longer keep pace alone with the frenetic evolution of the UNIX market. The adoption of AIX was accompanied by Bull's specific technical contributions, particularly in telecommunications and multiprocessor architectures. Facing American dominance of the sector, this cooperation illustrated a form of European specialization. The European Commission actually encouraged this approach by supporting the development of a strong European computing industry. The creation of the "Round Table of 12" brought together the continent's major companies, with Bull at the forefront, to coordinate efforts against American giants. This political initiative reflected awareness of the strategic stakes linked to technological independence. In 2025, we can see that the results are mixed, and the same questions remain.

The persistence of GCOS in the contemporary computing landscape is surprising for its longevity. Despite Bull's repeated transformations and

accelerated technological evolution, these systems continue to run critical applications in numerous private and governmental organizations. This resilience is explained by the intrinsic technical quality of the systems, customer loyalty, and Bull's ability to evolve its platforms while preserving compatibility.

The history of GECOS reveals the contradictions and limits of the European computing industry. Between technical legacy and innovation, between local specificities and international standards, between independence and cooperation, Bull navigated a complex environment while attempting to preserve its technological distinctiveness. This experience illuminates contemporary issues of digital sovereignty and technological diversity in an increasingly uniform computing world.

IBM System/360

In 1964, IBM bet $5 billion on a groundbreaking idea, equivalent to $50 billion today. Never before had a private company invested such sums in a technological project. At that time, IBM dominated the world of electromechanical computing machines with its famous 80-column punched card. But the company was facing a paradoxical crisis. Its product line included eight computers that were completely incompatible with each other. Each migration to a more powerful system forced customers to repurchase all their peripherals and rewrite their programs. This situation was untenable.

Thomas Watson Jr., who chaired IBM, entrusted this thorny issue to Thomas Vincent Learson. Learson appointed Bob O. Evans, head of systems development in Poughkeepsie. Evans quickly reached a radical and bold conclusion: all ongoing projects had to be abandoned and everything started from scratch. The decision sparked heated debates, particularly with Frederick P. Brooks Jr., who was leading the development of the 8000 series.

Evans' team devised an audacious vision. Code written for the smallest model in the family had to run without modification on the most powerful processors. Printers, communication devices, and storage units had to work with any computer in the range. This universal compatibility required a complete rethinking of computing. Operating system software had to be created for a family of compatible processors, unprecedented peripherals had to be designed, and an integrated circuit technology called

Solid Logic Technology had to be developed. IBM mobilized its worldwide resources. Teams of engineers worked simultaneously on six new processors and dozens of peripherals, both in America and Europe. Hundreds of programmers began writing millions of lines of code.

On April 7, 1964, IBM unveiled the System/360 with military precision. The company organized press conferences in 165 American cities and 14 countries. The presentation revealed six new processors and 44 peripherals. The reception exceeded all expectations: more than a thousand machines were ordered in four weeks. But the euphoria of orders gave way to two years of hell. The production of SLT modules posed the first problem. IBM built a factory of over 9,200 square meters in East Fishkill. Production soared from 500,000 modules in 1963 to 12 million in 1964, then 28 million the following year. Quality defects piled up, delaying deliveries by two to four months. In 1966, the factory produced 90 million modules, more than all other semiconductor manufacturers combined.

Logistics turned into a nightmare. Orders flooded in while delays accumulated in component manufacturing and software development. Tension mounted between sales teams and production departments. An amusing episode actually illustrates this period: the company ran out of copper circuit breakers, temporarily paralyzing all production.

Software development represented the third pitfall. Initially budgeted between $30 and $40 million, the final cost reached $500 million. The team had to create an operating system that worked across different processors and peripherals, a feat never before accomplished. The code grew from 100,000 lines for the IBM 1401 to over one million for the System/360.

By 1966, seven to eight thousand systems equipped companies, generating over $4 billion in new revenue. IBM hired 25,000 employees that year and built over 270,000 square meters of additional production space. Between 1964 and 1970, revenue doubled, rising from $3.2 to $7.5 billion.

The System/360 standardized the 8-bit byte and introduced numerous innovations in storage and memory. Its architecture immediately inspired competitors. RCA launched the Spectra 70 series, compatible with the System/360. The Soviet Union developed the Ryad, an almost exact copy. Gene Amdahl, who participated in the system's design, founded Amdahl Corporation to produce compatible processors. An entire industry emerged around compatible peripherals with companies like Telex, Memorex, and Storage Technology.

Bank of America, NASA, and other major organizations chose the System/360 for applications ranging from banking transaction processing to scientific calculations. Thanks to its unified architecture, companies could now compare price and performance across an entire product range,

simplifying their purchasing decisions and IT planning.

In 1985, President Ronald Reagan presented the National Medal of Technology to Erich Bloch, Fred Brooks, and Bob Evans for their contribution to the System/360. This official recognition underscored the project's historical importance in the development of modern computing. Its architecture influenced all subsequent generations of IBM computers, from the System/370 to the System/390, up to today's IBM zSeries servers. The principles of compatibility and standardization it established remain pillars of the contemporary computer industry.

BASIC

In 1964, in the corridors of Dartmouth College, New Hampshire, John G. Kemeny and Thomas E. Kurtz completed BASIC, an acronym for Beginners All-Purpose Symbolic Instruction Code. This creation was born from legitimate frustration with the state of computing at the time. Giant, expensive computers required skills possessed by only a few initiates. To program, one first had to go to a punch card center where instructions were laboriously translated into punched cards, then wait in line at the computing center. A single error in the code meant starting all over again.

A former collaborator on the Manhattan Project and assistant to Albert Einstein, Kemeny had joined Kurtz at Dartmouth in 1956. Their first meeting had given birth to a collaboration that would last for decades. Before BASIC, they had attempted to create simplified languages such as Darsimco and DOPE, but these experiments had never really taken off. The idea that drove them remained to make computing accessible to the greatest number of people.

Having become director of the Kiewit Computing Center, Kurtz envisioned an innovative time-sharing system. This system required a language that was both simple and powerful. The two men set ambitious goals for a versatile language that was easy to learn, extensible, interactive, and above all hardware-independent. Drawing inspiration from FORTRAN and ALGOL, they designed a structure that favored common English words. Program lines were numbered, an innovation that greatly simplified editing: to delete a line, one simply indicated its number, and to modify it, one rewrote it with its number.

On May 1st, 1964, at four o'clock in the morning, two BASIC programs ran simultaneously on Dartmouth College's General Electric 225. This

date marked the official birth of the language. Kemeny and Kurtz then made a decision that would influence the entire history of computing: they declined to patent their invention. This generosity would transform BASIC into a worldwide phenomenon.

General Electric, which had provided the computer to Dartmouth, was among the first users of the new language. Around 1970, the company marketed machines equipped with the fifth version of BASIC, without waiting for the next version in preparation. This rush marked the beginning of the language's fragmentation. Kemeny and Kurtz published their *BASIC the Sixth* in 1971, but other developers were already creating their own dialects.

Gordon Eubanks, the future CEO of Symantec, developed BASIC-E in 1970, using an innovative approach in which instructions were first transformed into intermediate code before being converted to machine code. This technique foreshadowed the one that Java would use decades later with *byte code*. Unlike its creators, Eubanks protected his next version, CBASIC, which he marketed through his company Compiler Systems.

The proliferation of BASIC versions concerned ANSI, which recognized in 1974 the need for standardization. A committee formed to define two standards: minimal BASIC and standard BASIC. The process stretched over years, long after the language had conquered the planet. ANSI finally published its specifications in 1978 for minimal BASIC and in 1987 for standard BASIC. ISO adopted these standards in 1984 and 1991 respectively.

The year 1975 saw the birth of Bob Albrecht and Dennis Allison's TinyBASIC, a marvel of miniaturization that functioned with only 2 KB of RAM. William H. Gates III and Paul Allen created their version for the MITS Altair. Constrained by memory limitations, they chose an interpreter rather than a compiler. This solution, which fit in 4 KB, presented the unexpected advantage of greatly facilitated debugging through the interactivity it provided. Kemeny and Kurtz, initially skeptical of this approach, would eventually acknowledge its role in spreading their language.

The explosion of personal computers in the late 1970s transformed the BASIC landscape. Each manufacturer developed its own version: Radio Shack with its Level 1 BASIC for the TRS 80, Apple with the Integer BASIC of the Apple II in 1977, Commodore with PET BASIC, Atari with its BASIC in 1978. The list grew with Timex Sinclair 1000 BASIC, Texas Instruments' TI-BASIC, and the Commodore BASIC of the VIC 20 and C64.

In the early 1980s, IBM disrupted the market by integrating an interpreted BASIC into its PC's ROM. Extensible through loading BASICA,

this BASIC accompanied each PC-DOS diskette. Its MS-DOS equivalent, GW-BASIC, quickly dominated the compatible PC market. Microsoft took a new step in 1984 with the commercialization of BASCOM, its first BASIC compiler.

Evolution accelerated with QuickBASIC in 1985, which reached version 4.5 in 1988. The language was enriched with structured syntax, subfunctions, customizable data types, and multi-file management. In 1990, Microsoft BASIC Professional Development System 7.1 crossed the symbolic barrier of 64 KB of memory.

The arrival of Windows once again transformed BASIC with Visual Basic, specially designed for graphical applications. This version, whose object-oriented character was debated among programmers, achieved phenomenal success. By the late 1990s, estimates suggested that 90% of Windows software was developed with Visual Basic.

The history of BASIC ultimately tells the story of technological democratization. This language born in a university laboratory accompanied the transformation of computing from a science reserved for experts to a tool accessible to all. It trained entire generations of programmers.

PL/I

Computing in the early 1960s resembled an archipelago of isolated communities. On one side, scientists operated their IBM 7090s, coded in FORTRAN, and gathered at SHARE meetings—a user group for IBM *mainframes*. On the other, the business world worked on IBM 7080s, swore by COBOL, and organized its own circles within G.U.I.D.E—another IBM computer user group. A few specialized machines like the IBM 7750 had their dedicated languages, such as JOVIAL (Jules' Own Version of the International Algorithmic Language).

This geographical partition of computing territory was beginning to show its limits. Researchers no longer wanted to wait hours at their terminals to retrieve a simple numerical value; they demanded structured, readable reports worthy of their work. Their data files were swelling and now rivaled the volumes handled by businesses. On the commercial side, marketing departments were discovering the joys of statistical analysis and demanding floating-point calculations on their sales data. IT managers found themselves juggling two operating systems, two teams of programmers who literally didn't speak the same language, and budgets that were spiraling out of control.

IBM had already laid the groundwork for a solution with the System/360 and OS/360, this family of machines and operating systems designed to unify everyone's needs. What remained was to create a programming language that would follow the same philosophy. In October 1963, IBM and SHARE created the Advanced Language Development committee. Its mission was to define a universal language that would end this computing Tower of Babel.

The assembled team blended backgrounds and skills. Hans Berg from Lockheed brought his training expertise, Jim Cox from Union Carbide knew FORTRAN inside and out, Bruce Rosenblatt from Standard Oil chaired the discussions. On the IBM side, C. W. Medlock was the expert in optimized FORTRAN compilers, Bernice Weitzenhoffer moved effortlessly between FORTRAN and COBOL, George Radin led the project with his scientific programming experience. Meetings followed one after another every two weeks, three or four days at a stretch, mainly between New York and Los Angeles. Members outside IBM kept their full-time positions, with this new language remaining a side mission. Originally planned for December 1963, the specifications freeze ultimately slipped to February 1964. Deadlines were tight.

The idea of extending FORTRAN was abandoned. Its syntax didn't fit modern terminals, its declarations organized by type rather than by identifier clashed with conventions, its column-major array storage puzzled business application programmers. The necessary extensions would have so transformed FORTRAN that maintaining any compatibility lost its meaning.

The language was initially called NPL (New Programming Language) in the grand tradition of descriptive acronyms. In 1965, a conflict with the British National Physical Laboratory forced IBM to rename it PL/I (sometimes written PL/1). The first official presentation took place in March 1964 in San Francisco, before the SHARE assembly. Reactions were mixed. Some praised its comprehensiveness and attention to programmers of all levels. Others pointed to its complexity and redundancies. One SHARE member compared it to "a Swiss Army knife with a hundred blades", a metaphor that spoke volumes about the project's ambitions.

Compiler development landed in IBM's Hursley laboratories in England. John Fairclough's team took charge of stabilizing the language and making it practically usable. Without the Hursley programmers, PL/I would have remained a beautiful theory. They built its compilers, participated in its standardization, and transformed an ambitious specification into an everyday working tool.

PL/I shook up established conventions. It introduced asynchronous

tasks, defined `ON` conditions to specify code blocks to execute during particular events, authorized recursive procedures with static and automatic storage. Its palette of data types was impressive: arithmetic or character strings, decimal or binary, fixed or floating, real or complex. Arrays accepted arbitrary dimensions with freely chosen bounds. Input-output received particular care, representing more than 10% of the language definition. PL/I married FORTRAN's formatting capabilities with certain COBOL functionalities for record processing, while exploiting OS/360's new possibilities like asynchronous input-output.

One of PL/I's most controversial peculiarities resided in its declaration handling. The language didn't require all attributes to be explicitly declared. In case of omission, implicit types were assigned according to variable usage in the program. Missing attributes received default values depending on those explicitly mentioned. This flexibility responded to the constraints of an era when each program submission involved waiting and delays.

PL/I's evolution testifies to a constant formalization effort. The first 1964 documents left numerous gray areas and ambiguities. Ray Larner and John Nicholls developed an initial formal semantic definition called Universal Language Definition II. The Vienna laboratory then took over with an even more rigorous definition. PL/I thus joined the select circle of the best-defined languages of its time.

Success was forthcoming. In the 1970s, IBM sold more PL/I compiler licenses than FORTRAN for System/370. COBOL certainly kept first place for business applications, but PL/I had found its niche. The language and its variants were used to program entire operating systems like MULTICS and OS/VS2 Release 2, as well as numerous compilers. Educational subsets facilitated its teaching in universities.

PL/I favored object code efficiency and ease of use over theoretical elegance. Its development relied on programmers' concrete experience rather than abstract mathematical considerations. This philosophy contrasts with current concerns like extensibility or formal program verification.

IBM CP-40

The history of virtual machines begins in the laboratories of MIT, where Fernando Corbató and his team work on the *Compatible Time-Sharing System*. This system transforms computer usage by introducing

time-sharing, but its implementation requires delicate hardware modifications. IBM, the supplier of MIT's machines, maintains an on-site liaison office to support these technical adaptations.

In 1964, IBM unveils its new System/360 line, intended to unify enterprise computing. However, MIT customers discover with disappointment the absence of address translation capabilities in these new machines. This function seems essential to them for developing their time-sharing systems. Faced with this disappointment, IBM responds by creating the Cambridge Scientific Center under the leadership of Norm Rasmussen. The objective was to catch up in the field of time-sharing.

Robert Creasy, a veteran of the CTSS project, takes charge of the new CP-40 project. With Les Comeau, he conceives at the end of 1964 a radically different approach: simulating several independent System/360 machines on a single physical computer. This idea appeals through its conceptual simplicity. Each user has their own virtual machine, completely isolated from others. Gone are the risks of interference between programs.

Realizing this concept requires technical prowess. The CSC team modifies an IBM System/360 model 40 by grafting onto it a device called the Cambridge Address Translator. Bruce Lindquist and Rex Seeber design this mechanism around a 64-word associative memory. The challenge consists of translating addresses without slowing program execution. Mission accomplished: the translation occurs without noticeable performance loss.

The CP-40 system architecture rests on a clear separation between two components. On one side, the Control Program manages the creation and administration of virtual machines. On the other, the Cambridge Monitor System provides a simple operating environment for these virtual machines. This division of labor constitutes a remarkable architectural innovation. John Harmon leads the development of CMS with his team including Lyndalee Korn and Ron Brennan. They draw heavily on the CTSS interface to create a user-friendly system. Paradoxically, CMS is single-user and all the complexity of resource sharing falls to the Control Program.

January 1967 sees the production deployment of CP-40 and CMS. The system immediately demonstrates its ability to run OS/360 in a virtual machine. This backward compatibility would prove decisive for the technology's future adoption. The team also discovers the pitfalls of virtual memory. The *thrashing* phenomenon surprises them: when paging is excessive, performance collapses abruptly.

The success of CP-40 paves the way for CP-67, adapted to the System/360 model 67 which natively integrates address translation. Dick Bayles, Dick Meyer, and Harit Nanavati lead this evolution. CP-67 brings

substantial improvements such as dynamic memory management and flexibility in virtual machine configuration.

May 1968 marks the first distribution of CP-67 to eight pilot sites. In June, the system becomes available as a Type III program, a status reserved for contributions from IBM employees. The success exceeds all expectations. Former employees create two commercial companies that sell time-sharing services based on the CP/CMS platform: National CSS and Interactive Data Corporation.

The versatility of the virtual machine concept appeals to different communities. Operating system developers find in it a secure test environment. End users appreciate the coexistence of different operating systems on the same machine. Computer centers exploit this capability to facilitate their technological migrations.

August 1972 sees the birth of VM/370 as part of the System/370 Advanced Functions. The Burlington team, enriched by Dick Newson, Carl Young, and Dave Tuttle, pushes innovation further. VM/370 introduces the ability to run VM under itself, a considerable simplification for development and testing.

The separation between hardware management and system services still structures our contemporary hypervisors. The hardware interface as an isolation boundary is a central paradigm. This transparent simulation of hardware resources flourishes today in *cloud computing*.

Electronic Mail

Sixty years ago, the business world operated at the pace of paper. Executives dictated their letters to their secretaries, who mastered the delicate art of shorthand before transcribing on typewriters, with that characteristic sound of keys striking the ink ribbon. Each message followed the rigid protocol of the official letterhead, codified courtesy formulas, and carbon copies for the archives. The mail then departed into the maze of the postal system.

As early as 1965, at MIT, Tom Van Vleck and Noel Morris tinkered with something unprecedented. On their laboratory's time-sharing computer, they created a command called `mail` that allowed users to send messages stored in virtual mailboxes. The idea seemed trivial, but it already disrupted habits: no more need to physically move to transmit information, no more waiting, no more paper lying around.

Six years later, Ray Tomlinson accomplished the gesture that would change the history of human communication. This engineer at Bolt Beranek and Newman was working on ARPANET when he decided to adapt the `sndmsg` program to enable distant computers to communicate. He needed a way to distinguish the user from their machine, and it was almost by chance that he chose the `@` symbol on his keyboard. This at sign, which accountants used to note "so many units at such a price", bridged the gap between digital identity and the computer's physical address. First message sent: "QWERTYUIOP" or something close. Tomlinson forgot the exact content, which shows he didn't grasp the magnitude of his invention.

The contagion effect was immediate on ARPANET. Researchers and engineers discovered they could exchange without time or geographic constraints, and instantaneously. No more interrupted phone calls, no more letters taking a week to cross the country. Larry Roberts, who directed ARPA's IPTO office, quickly understood the value of this technology. In 1972, he developed `rd`, a program that sorted and classified received messages. Barry Wessler improved it with `nrd`, and John Vittal took a decisive step in 1974 with MSG, which introduced "reply" and "forward" functions. Electronic mail ceased being a simple transmission system to become a true conversation tool.

Technology followed usage. Early message formats resembled makeshift solutions, with each system having its own conventions. RFC 561 of 1973 attempted to impose order by defining the `From:`, `Subject:`, and `Date:` fields. This standardization work continued laboriously until RFC 822 of 1982, which defined the foundations of the format still used today.

Jon Postel, a legendary figure of the Internet, tackled the problem of message transport. Early systems used FTP, but this protocol proved unsuitable for the growing volume of exchanges. He designed Simple Mail Transfer Protocol in 1982, a protocol of elegant simplicity that would span decades without aging. This exceptional longevity in the computing universe testifies to the quality of its design.

The 1980s saw a proliferation of incompatible networks. UUCP connected UNIX machines, CSnet served universities, Bitnet linked computing centers. Each network had its addressing rules, conventions, and limitations. System administrators juggled with bewildering addresses like `ihnp4!ucbvax!user@host`, veritable computer roadmaps that specified the exact path of the message. This complexity disappeared with the arrival of the domain name system in 1984 and `MX` records two years later. The `user@domain.com` address then established itself as the universal standard.

In 1981, Eric Allman developed `sendmail`, and Dave Crocker created

MMDF in 1982. These sophisticated programs handled message routing between heterogeneous systems, but ran into a limitation: electronic mail could only transport ASCII text. Impossible to send a photo, formatted document, or presentation. Users worked around the problem by encoding binary files, but the manipulation was tricky. The MIME standard of 1992 finally freed electronic mail from these constraints by allowing multimedia content. This evolution transformed messaging into a true professional exchange platform.

From a few hundred pioneer users on ARPANET, electronic mail now brings together more than 4.37 billion people. This massive adoption is explained by remarkably judicious technical choices: simple and robust protocols, open standards, and decentralized architecture. The system adapted to evolving uses without ever breaking its fundamental compatibility. The emergence of instant messaging and social networks has not succeeded in completely dethroning this sixty-year-old technology, which continues to route more than 350 billion messages across the planet each day in 2024.

Multics

In 1962, Joseph Carl Robnett Licklider, then director of DARPA, launched the idea for the Multics operating system. The contract was awarded in August 1964. Multics extended the work of the Compatible Time-Sharing System, developed at MIT under Fernando Corbató. Computers still operated as batch processing machines, with programs executing one after another, and users sometimes waiting entire days for their results. The first time-sharing systems struggled to manage more than a few simultaneous programs, forced to offload inactive ones onto magnetic tape.

The project brought together an unusual alliance consisting of MIT for research, General Electric for hardware, and Bell Labs for software development. GE secured the contract after IBM rejected the proposed paging and segmentation concepts. This convergence of academic and industrial talent assembled some of the greatest minds in computing of that era.

The technical ambition exceeded anything previously attempted. Multics integrated virtual memory, segmentation-based protection, hierarchical file system, shared-memory multiprocessing, security, and online reconfiguration. The system was among the first to be developed primarily in a high-level language and to support multiple programming languages.

MacLisp and `troff` found their origins in Multics, as did much of the modern UNIX command line.

Implementation proved challenging. The GE 645 hardware arrived late, and Multics only achieved self-hosting in 1968. In 1969, DARPA threatened to pull the plug, pushing Bell Labs toward the exit. This withdrawal would give birth to UNIX, created by former Multics developers: Kenneth Lane Thompson, Dennis MacAlistair Ritchie, Douglas McIlroy, and Joseph Francis Ossanna.

Honeywell, which acquired General Electric's computer business in 1970, commercialized Multics from 1973 until 1985 at an initial price of $7 million. The customer base consisted of the US Air Force, universities such as the University of Southwestern Louisiana and the French university system, and major corporations like General Motors and Ford. In total, approximately 80 licenses were sold. The last Multics system, operated by the Canadian Department of National Defence, was shut down in 2000.

The system architecture rested on two pillars: processes and segments. Processes provided execution contexts, while segments stored code, data, and I/O devices. These segments were organized in a directory hierarchy, ancestors of current file systems. A process's protection domain delimited accessible segments and authorized operations.

Regarding security, Multics introduced protection rings, a hierarchical structure ranging from ring 0, the most privileged, to higher, less powerful rings. The GE 645 provided for 64 rings, but only 8 were actually implemented. The Multics supervisor, confined to rings 0 and 1, could only be modified by code executing in these privileged rings. This approach still influences modern processors, which typically use two levels: supervisor (*kernel*) and user.

In 1974, Paul Karger and Roger Schell of the US Air Force conducted a vulnerability analysis that revealed several flaws. A hardware vulnerability bypassed access checks in certain indirect addressing cases. Software flaws appeared, particularly related to the master mode of execution in user rings, a modification introduced to improve performance. These discoveries illustrated the difficulty of maintaining security in a complex system under performance optimization pressures.

Several factors explain Multics's relative commercial failure. Integrated circuits evolved at an exponential rate, transforming the computing landscape. When Multics reached the market in 1973, it perfectly addressed the problems of 1964, but the world had changed. The system cost too much, depended too heavily on proprietary architecture, and was too focused on centralized shared computing while decentralized

departmental computing was taking off. Meanwhile, Digital Equipment Corporation was marketing entry-level versions of the PDP-11, and the 8080 microprocessor was running the first versions of CP/M as early as 1974.

DEC PDP-8

In 1965, Digital Equipment Corporation surprised the computing world by introducing the PDP-8, a computer that fit in half a cabinet. The era of steel monsters occupying entire rooms seemed over. This machine inaugurated the minicomputer era and radically transformed our vision of what a computer system should be.

The story began three years earlier with the PDP-5. DEC experimented with the input-output bus, an innovative concept. Gone were the complex radial connections that required anticipating each piece of equipment and its wiring to a central point. The bus allowed the progressive addition of peripherals, an unprecedented flexibility. But the PDP-5 was a sketch, and DEC's engineers saw further ahead.

The PDP-8 was born from this ambition. Logic technologies were evolving and promised spectacular speed gains. Ferrite core memories improved storage performance by reducing cycle time from 6 to 1.6 microseconds. Made possible by falling logic component costs, the program counter, previously stored in memory, migrated to hardware, drastically reducing execution times.

The PDP-8 architecture stood out for its simplicity. This 12-bit single-address computer did not seek complexity to impress. It embraced its limitations of reduced arithmetic calculations and modest primary memory. These constraints were its strengths in certain domains, particularly industrial process control and laboratory applications that found in it an ideal partner. Controlling a pulse height analyzer or a spectrum analyzer did not require the power of a giant; precision and reliability were sufficient.

The market responded enthusiastically. Approximately 50,000 units sold until 1979, not counting the CMOS versions that followed. For a computer not produced by IBM, this figure was remarkable. The PDP-8's size partly explained this success. Placed on a laboratory bench or integrated into larger equipment, it adapted to space constraints. This flexibility actually gave birth to the OEM market, where manufacturers purchased computers to incorporate into their systems.

The PDP-8's evolution tells the story of successive technologies. In 1966, the PDP-8/S attempted to conquer the low-end market with a serial architecture where data was processed bit by bit sequentially. The savings came at a steep price as performance collapsed and sales disappointed. DEC learned its lesson. In 1968, the PDP-8/I exploited medium-scale integrated circuits to surpass the original model while costing a third less. The PDP-8/L targeted customers content with simple configurations. Then came the PDP-8/E and its Omnibus architecture. This unified 144-pin bus revolutionized modularity. The processor, which occupied 100 cards in the PDP-5, now required only three 8 × 10-inch cards. Miniaturization accelerated and redefined possibilities. The PDP-8/M further compacted the concept to attract cramped installations. In 1975, the PDP-8/A leveraged hexagonal cards and semiconductor memories. Its microprogrammed processor fit on a single card, a remarkable technical achievement.

The culmination came in 1976 when Intersil etched the first PDP-8 processor on a single chip, using CMOS technology. This feat came with minimal power consumption. This miniaturization enabled systems of unthinkable compactness just a few years earlier, such as the VT78 video terminal.

The PDP-8's input-output bus became a reference and its architectural simplicity inspired generations of engineers. This machine proved that a computer could reconcile simplicity, power, and economy. It opened a new market between costly systems and specialized calculators.

CDC 6600

In 1957, William C. Norris left his position at Sperry Rand with a handful of colleagues to set up shop in a disused Minneapolis warehouse. Their plan was to establish Control Data Corporation to build the world's fastest computers. This audacious ambition took shape when Seymour Cray joined the team in 1958. Six years later, their CDC 6600 revolutionized computing by becoming the first true supercomputer.

IBM dominated the market with its 7094, the benchmark machine competing against CDC's 1604. These computers represented the pinnacle of available computing power, but their legacy architectures constrained any major evolution. Maintaining backward compatibility imposed technical compromises that limited performance gains. Transistors were reaching

their physical limits and could no longer drive machines to new heights on their own.

Seymour Cray and his team recognized that a breakthrough was needed. James Thornton, one of the engineers, attended a seminar at UCLA on high-performance machines where one thing became clear: they all exploited some form of parallelism. The team discovered that a quarter of computation time was lost in signal transmission between logic gates. This observation would shape their approach with the arrival of silicon transistors from Fairchild Semiconductor. These new components enabled unprecedented density and shortened critical paths. Circuit boards could now be mounted back-to-back, with all components housed inside. This organization solved at a stroke the cooling problems posed by increased circuit concentration.

The CDC 6600 broke the mold with its ten specialized functional units working in parallel under the coordination of a central control unit. With this architecture, different parts of a program executed simultaneously, an unprecedented technical feat. The central processor focused on scientific calculations while ten peripheral processors handled input-output and system tasks.

The approach remarkably anticipated future RISC processors. Memory was organized into pages protected by an advanced system. A sophisticated hardware *scoreboard* managed dependencies between instructions and optimized their parallel execution. These innovations propelled the machine toward one million instructions per second, a performance never before achieved.

IBM responded to the 6600 announcement by promising its own supercomputer. This counterattack temporarily stalled orders, as customers preferred to wait for the giant's response. Problem: IBM's machine existed only on paper. CDC filed an *antitrust* lawsuit and promptly launched its 7600, openly defying IBM, which this time remained silent. The 7600 was one of the most powerful computers ever built. Five years later, CDC won its lawsuit.

Scientific laboratories, universities, and military research centers massively adopted the CDC 6600. Its unmatched power advanced meteorology, nuclear physics, and aerodynamic design. SCOPE, the operating system developed specifically for the machine, provided advanced resource management and multiprogramming capabilities. Its modular design, innovative use of parallelism, and sophisticated resource management inspired future developments. Modern processors adopted its principles in instruction-level parallelism and the organization of specialized functional units. Control Data established itself as the leader in scientific computing

and developed the CYBER series as a continuation of the 6600.

In 1972, Seymour Cray left CDC to found Cray Research. The company invested $300,000 in this new venture, maintaining close ties with its former architect. The emergence of mini-supercomputers and microprocessors in the 1980s gradually eroded CDC's dominant position. Competition from DEC, IBM, and Cray Research intensified. CDC accumulated losses, pushing William Norris out in 1986. Despite several restructurings, the company split its computing operations in 1992, marking the end of an era.

The CDC 6600 remains a major technical innovation that established the foundations of high-performance computing. Its elegant design demonstrated that a carefully conceived architecture could transcend the apparent limitations of components. This machine ushered in the age of supercomputers and set performance standards that durably guided the evolution of scientific computing.

ELIZA

In 1966, Joseph Weizenbaum published an article in the "Communications of the ACM". In it, he described ELIZA, a program capable of conversing in natural language with a human being. At MIT, where the researcher worked, this achievement created a sensation in artificial intelligence circles.

The name chosen for this program was not random. Eliza Doolittle, the heroine of George Bernard Shaw's *Pygmalion*, embodies the transformation of a simple flower seller into a distinguished lady through the lessons of Professor Higgins. Joseph Weizenbaum saw in this metaphor a machine that learns to "speak" under a teacher's supervision while retaining its limitations—a perfect illustration of his program.

On MIT's MAC time-sharing system, ELIZA ran using the MAD-SLIP language on an IBM 7094. Users connected from remote terminals and began what resembled genuine conversation. They typed their sentences, the program responded. The illusion worked. Yet the mechanisms animating ELIZA were disconcertingly simple. The program scrutinized each sentence searching for keywords. Once a term was identified, it applied predefined transformation rules. If no keyword was detected, it drew from a repertoire of generic responses or reused a previous transformation.

This approach revealed its full subtlety in the hierarchization of key-

words. When a sentence contained "I" and "everyone", the program favored the latter term, because excessive generalizations often deserve attention. The transformation rules broke sentences down according to preset patterns to reconstitute them into coherent responses.

ELIZA's modular architecture constituted its true innovation. The conversational rules lived in scripts separate from the program's core. This design allowed behavioral modifications without touching the source code. Scripts thus emerged in German and Welsh, proof of the system's flexibility.

The script that would make ELIZA famous simulated a Rogerian psychotherapist. Carl Rogers had developed a person-centered therapy where the therapist reformulates the patient's statements without claiming to understand everything. This approach perfectly matched the limited capabilities of Joseph Weizenbaum's program. Faced with "I am sad", ELIZA responded "I am sorry to hear that you are sad" or "Can you explain what makes you sad?" The program maintained the illusion through reformulation, open-ended questions, and requests for clarification.

Joseph Weizenbaum did not anticipate the impact of his creation. Users attributed genuine understanding to ELIZA. Some requested private sessions with the program. This reaction troubled the researcher, who realized that humans projected their own interpretations onto the machine's mechanical responses. They themselves constructed the meaning of the exchange. This spontaneous anthropomorphization still resonates in our relationships with voice assistants and other chatbots. ELIZA revealed how elementary mechanisms generate the illusion of intelligence, raising lasting questions about the nature of understanding.

The program's technical weaknesses are glaringly obvious: no memory of past exchanges, no model of the interlocutor, no knowledge of the real world. Responses emerged from syntactic transformations blind to meaning. But these limitations do not erase ELIZA's historical importance, which established the foundations of automatic language processing: keyword recognition, identification of minimal contexts, selection of appropriate transformations, generation of default responses.

Beyond the technical aspects, ELIZA crystallized our reflection on artificial intelligence. Joseph Weizenbaum observed this paradox where, once the mechanisms had been explained, the "magic" evaporated and the program revealed itself as a collection of trivial procedures. Yet the effect on users persisted. ELIZA teaches us that the perception of intelligence in our exchanges with machines depends as much on our projections as on the actual capabilities of the systems.

The questions raised by Joseph Weizenbaum's program about the na-

ture of understanding, the boundaries of simulation, and the ethical stakes of artificial intelligence have lost none of their relevance. ELIZA remains that foundational moment when computing began to explore the ambiguous territories between mechanical language processing and genuine intelligence.

Information Management System

In 1966, in the engineering offices of Rockwell's Space Division, engineers working on the Apollo program juggled thousands of technical drawings, with each modification to a component triggering a cascade of verifications across the entire space capsule. There was no room for error when it came to sending men to the Moon.

Uri Berman, an IBM consultant seconded to Rockwell, observed this organized chaos. Together with Pete Nordyke, they envisioned a bold solution that completely separated data management from application code. Their prototype, which they named DATE (Disk Applications in a Teleprocessing Environment), radically transformed how engineers accessed technical information. Gone were the endless trips to paper archives—a terminal connected to an IBM 7010 now sufficed to check the status of a drawing and its ongoing modifications. This separation between data and processing formed the foundation of Data Language/One (DL/I), an interface that freed programmers from the physical constraints of disk storage. The idea gained ground that an application should not reinvent the wheel of file management. DL/I standardized these operations, improved data integrity, and simplified program maintenance.

The project took on new scope in 1966 with the arrival of IBM System/360 computers. Dr. Robert R. Brown assumed technical leadership, supported by consultant Bob Patrick. The team set itself the goal of creating software that met IBM Type 2 standards without modifying the OS/360 operating system. This seemingly limiting constraint would actually ensure the longevity of their creation.

The collaboration expanded beyond Rockwell, so Caterpillar Tractor joined the venture, bringing its specific industrial management needs. Near twenty programmers worked on what became IMS/360, integrating DL/I into a more ambitious architecture. On August 14, 1968, the first `READY` message from IMS appeared on a Rockwell terminal. The Apollo mission could now count on a new computer companion.

The IMS architecture rested on three pillars: the database manager (IMS DB), the transaction manager (IMS TM), and common services. This modularity allowed à la carte usage. Some clients exploited only the database component (`DBCTL`), while others focused on transaction processing (`DCCTL`). Everyone found what they needed according to their specific requirements.

An organizational innovation accompanied the technical deployment. Dan Gilbert became Rockwell's first *Database Manager* in history, foreshadowing the database administrator profession. This position centralized responsibility for data organization and integrity, breaking with the previous dispersion of these tasks across different departments. IMS's hierarchical approach contrasted with the relational world that would later dominate. This structure, born from the needs of managing space bills of materials, excelled at predefined access patterns. Relationships between data were established in advance, creating optimized paths. This philosophy differed radically from relational databases, where relationships are constructed dynamically during query execution.

In 1988, twenty years after its inception, IMS equipped 7,000 installations worldwide. In the 2000s, the system processed more than 50 billion transactions daily and managed over 15 million gigabytes of data for more than 200 million users. These volumes testified to its robustness and capacity for evolution.

Technological evolution did not spare IMS, which adapted without renouncing its principles. The introduction of OTMA (Open Transaction Manager Access) opened the system to TCP/IP applications, breaking the isolation of IBM's proprietary network architectures. Java, XML, and web services gradually found their place, inscribing IMS into the service-oriented architecture era. Integration with Microsoft .NET or SAP further expanded its ecosystem.

IBM's Parallel Sysplex gave IMS a second youth. Sharing data and messages between multiple instances optimized resource utilization while strengthening availability. These sophisticated disaster recovery mechanisms made it a preferred choice for applications where downtime is not an option.

Today, IMS remains a pillar of transaction processing in the financial, manufacturing, and government sectors. This longevity in a field marked by frantic innovation cycles validates the initial architectural choices. Separation of concerns, data integrity, and processing performance remain central priorities, more than fifty years after the first technical drawings of the Apollo capsule.

LOGO

In 1966, four researchers at BBN (Bolt, Beranek and Newman) in Cambridge, Massachusetts, embarked on a project that would transform learning: Seymour Papert, Wallace Feurzeig, Daniel Bobrow, and Cynthia Solomon created LOGO.

Seymour Papert's background explains much. A trained mathematician, he spent five years in Geneva with Swiss psychologist Jean Piaget, absorbing his theories on cognitive development. Piaget rejected the idea that learning consists of filling an empty brain. For him, children actively construct their knowledge. In 1964, Papert joined the MIT Artificial Intelligence Laboratory, directed by Marvin Minsky. The two men shared a common obsession with understanding human thought by observing how machines learn.

The name LOGO comes from the Greek *logos*, meaning "word". Wallace Feurzeig proposed this name because the language initially focused on manipulating words and sentences. The creators started from the simple observation that children naturally play with words, while mathematics often terrifies them. Technically, LOGO descends directly from LISP, the artificial intelligence language invented by John McCarthy. It inherited interactive evaluation, recursion, and list manipulation, while keeping its syntax accessible to young people.

The first full-scale experiment took place in 1969 at Muzzey Junior High School. Twelve terminals connected to a PDP-1 computer hosted 12-year-old students programming in LOGO. They created sentence generators, coded games like tic-tac-toe, and developed educational programs. Also in 1969, Seymour Papert and Cynthia Solomon left BBN to found the LOGO group at MIT's AI laboratory.

The following year brought the innovation that would make LOGO famous: the "turtle". This small robot moved across the floor, drawing a line behind it. Then a graphical version appeared on screen. Turtle geometry transformed mathematics learning: children "played turtle" with their bodies, establishing a direct bridge between physical movement and mathematical abstraction.

The 1970s saw the birth of the "microworlds" concept, these learning environments designed around specific domains. Language and geometry were joined by new territories of exploration. Andrea diSessa developed the "dynatortue" to explore Newtonian physics. The *sprites* on the TI-99/4 enabled animation. Radia Perlman pushed innovation further by

creating interfaces for very young children: *button boxes* and the *slot machine* made programming accessible without knowing how to read or write.

In 1980, with the advent of personal computers, Seymour Papert published *Mindstorms*, a book that presented his pedagogical vision, where the computer transformed into an instrument that helps children think about their own thinking. Success was immediate. Commercial integrations multiplied: Terrapin and LCSI released their versions for the Apple II in 1981, soon followed by adaptations for practically all microcomputers of this generation.

The language evolved and diversified over the following decades. Object orientation entered LOGO with Object LOGO and TLC LOGO. LCSI's LogoWriter combined programming and word processing. StarLogo introduced parallel programming and complex system simulation. Variants proliferated and over 300 dialects emerged.

LOGO pedagogy rests on principles that disrupted traditional teaching. *Debugging* ceased being the shameful correction of errors to become a natural process of learning through trial and error. Developing procedures taught children to break down problems into named sub-problems. Students chose their projects, teachers guided them.

These pedagogical innovations sparked fierce academic debates. Some studies, notably those by Pea and Kurland, failed to demonstrate the announced benefits on general problem-solving abilities. These results sometimes cooled institutional enthusiasm for LOGO, but these studies faced severe criticism. Their methodology and restrictive vision of learning limited their scope.

LOGO traversed the decades. Scratch, born at the MIT Media Lab, adopted the construction-based philosophy in a colorful block programming environment. NetLogo continued the multi-agent simulation tradition. App Inventor transposed LOGO's principles to mobile programming. LOGO proposed a radical vision of education. Computing was not an additional subject to teach but a tool for rethinking learning. By giving children control of technology, it offered an alternative to lecture-based teaching. LOGO is a programming language and a philosophy that places the child at the heart of their learning.

Read Only Memory

In the world of electronic components, ROM stands apart. While most memories hasten to forget their data the moment power is cut, ROM carefully preserves its information, like a digital safe. This characteristic earns it its name: Read Only Memory.

The story begins in the 1950s, when engineers discovered they needed to store data permanently. The first ROMs emerged from this necessity, entirely hand-wired in factories. Each bit was literally etched into the material, frozen when the silicon took shape. These mask ROMs, as their creators named them, suited mass production perfectly. Manufacturing a hundred thousand identical copies of a program became economically viable. But woe to anyone who discovered an error after fabrication: no turning back.

This rigidity eventually became burdensome. The 1960s saw the birth of PROM, a ROM that users could program themselves. The idea of purchasing a blank chip filled with binary ones, then customizing it with specialized equipment, proved appealing. The operation remained irreversible, which earned these components the unflattering nickname of "one-time programmable." One typo, and into the trash it went.

The following decade brought a breath of freedom with EPROM. For the first time, developers could correct their mistakes. The secret lay in a small quartz window positioned atop the chip. Exposing the component to intense ultraviolet light completely erased its contents, like an electronic eraser. True, the operation took time and cost more, but what a revolution for those who needed to test and retest their code!

EEPROM arrived in the 1980s with the promise of eliminating UV sessions. This time, erasure was electrical, with precision that modified each byte individually. Designers appreciated this information surgery, never mind if write times dragged on and cost discouraged intensive use.

Flash memory, appearing toward the end of the 1980s, finally combined all the sought-after advantages: respectable density, affordable cost, decent read speed, and electrical reprogramming. Erasure only worked on entire blocks, typically from a few hundred bytes to several kilobytes, which was its sole constraint. A technical detail that didn't prevent its commercial triumph.

This evolution transformed our relationship with machines. ROM populates our electronic devices. It houses computer BIOS, those primitive instructions that wake the machine and verify its proper functioning. Calculators, printers, mobile phones—all these familiar objects conceal ROM that preserves their basic programs. Its nature naturally protects against

unauthorized modifications, forming a silent but effective bulwark against intrusions.

Industry found in this technology a balance between permanence and flexibility. This trajectory illustrates a trend in computing: the constant pursuit of adaptability. Laboratories continue exploring new avenues such as resistive memories (ReRAM), phase-change technologies, or quantum approaches. This research sketches the contours of future components that could marry ROM's traditional robustness with unprecedented performance. Some researchers also mention intrinsic protection against ransomware. The ROM adventure seems far from reaching its epilogue.

NLS

On December 9, 1968, Douglas Engelbart took the stage in a San Francisco auditorium packed with 2,000 spectators. For an hour and a half, this quiet man would revolutionize everyone's vision of computing. His NLS (oN-Line System) unveiled what personal computers would become in the following decades. The mouse, hypertext, videoconferencing, real-time collaborative editing... so many inventions that seemed to appear out of nowhere but which, in reality, had been brewing in his mind for more than twenty years.

In the mid-1940s, Engelbart, an electronics technician in the Navy, came across an article by Vannevar Bush entitled "As We May Think." Bush described the Memex, an imaginary machine capable of organizing and linking all human knowledge. This reading struck Engelbart like a revelation. Back in civilian life, he pondered this idea. Faced with the growing complexity of the world and increasingly difficult problems, he was convinced that tools needed to be created to multiply our capacities for thinking and collaboration.

In 1957, he joined the Stanford Research Institute as a consultant. There, he matured his vision of computing that augments human intelligence rather than replacing it. Unlike proponents of artificial intelligence who dreamed of thinking machines, he bet on human-machine symbiosis. In 1962, he put his ideas down in a paper that would make history: "Augmenting Human Intellect: A Conceptual Framework." This manifesto earned him support from ARPA, NASA, and other organizations. The following year, he founded the Augmentation Research Center within SRI. The team he assembled would give birth to the NLS system. Bill English joined the adventure and co-invented with Engelbart that small wooden

object that would revolutionize the human-machine interface: the mouse. Two orthogonal wheels in a housing, which advantageously replaced the light pen of the era. Other attempts emerged: devices controlled by the knee, by the head, but nothing matched the simplicity of the mouse. The team also developed a chord keyboard that transformed the user into a computing pianist, composing commands with one hand.

NLS transformed how people worked with information. Gone were programs that ran in batch mode; in their place came direct interaction. Documents became structured, linked to each other, modified in real time by multiple users simultaneously. It became possible to view them from different angles, search them, annotate them. Messaging integrated naturally into the system, as did videoconferencing. The screen was divided into windows, objects manipulated directly.

The 1968 demonstration orchestrated all of this with remarkable showmanship. Engelbart sat at his console, connected to a computer 50 kilometers away. His screen and image were projected onto a 6-by-5-meter wall. Behind the scenes, Bill English coordinated a technical team juggling video, sound, and remote connections. The technology of the time made the feat all the more impressive. On stage, he edited text as if it were natural, created hierarchical lists, inserted graphics. Fifty kilometers away, Bill Paxton collaborated with him live. Their two cursors danced together on the giant screen. The video conversation with the Menlo Park team astonished the audience. These actions seemed like magic at the time.

This "Mother of All Demos" planted the seeds of our modern computing. In the audience, Alan Kay and Charles Irby watched attentively. They would later join Xerox PARC, where the first commercial graphical interfaces would be born. Yet immediate success eluded its creator. Renamed Augment in 1978, NLS remained a tool for specialists. The system required expensive hardware and expert users. For Engelbart, power took precedence over ease of use. He compared learning his system to learning a musical instrument. This philosophy clashed with the emergence of personal computers that prioritized simplicity. NLS's centralized architecture aged poorly in the face of this change. The team dispersed, its members leaving to develop more accessible interfaces elsewhere.

NLS was only a first step toward transforming human collaboration. Engelbart envisioned a process of *bootstrapping* where each generation of tools spawned the next, in a virtuous spiral between technical progress and the evolution of practices. This legendary demonstration embodied a certain vision of computing as an amplifier of collective intelligence. Our current multimedia practices stem from this performance, blending

television, telecommunications, and computing. Above all, it reminds us that the potential for augmenting our cognitive capacities remains largely untapped.

Engelbart would continue until his death in 2013 to explore these questions, notably at the Bootstrap Institute he created in 1989. The Turing Award he received in 1997 crowned a visionary work.

ARPANET

In American laboratories of the early 1960s, computers lived in autarky. These expensive machines operated in isolation, each jealously guarding its data and programs. But at MIT, Joseph Carl Robnett Licklider saw further. In 1962, this visionary man sketched in his memoranda the outline of a "galactic network" that would radically transform our relationship with machines. He imagined a world where anyone could instantly tap into computing resources scattered across the country. His dream finally found concrete ground when he took charge of ARPA's computer research program, this Pentagon agency specialized in cutting-edge projects.

Leonard Kleinrock published the first paper on packet switching at MIT in July 1961, an idea he would develop three years later in a complete book. Meanwhile, Paul Baran was working at RAND Corporation on a communication network without a central point, where redundancy would guarantee the survival of exchanges in case of partial destruction. Across the Atlantic, Donald Davies at the British National Physical Laboratory was exploring parallel paths.

In the summer of 1965, Lawrence Roberts and Thomas Merrill managed to connect two computers separated by an entire continent, one in Massachusetts, the other in California. A simple telephone line was sufficient. The experiment worked, certainly, but it also revealed the glaring limitations of the traditional telephone network for this type of use.

Roberts joined ARPA the following year and gave substance to the ARPANET project. His detailed plan, presented in 1967, initially aroused distrust among the researchers involved. The proposed technical architecture seemed too complex to them. Wesley Clark saved the situation by suggesting a solution of genius simplicity: create computers dedicated to routing data, called Interface Message Processors. These famous IMPs immediately won approval. In December 1968, Bolt Beranek and Newman

secured the construction contract. The network awakened in September 1969. The first node took up residence at UCLA, where Leonard Kleinrock directed the Network Measurement Center. The second was installed at Douglas Engelbart's Stanford Research Institute, which housed the Network Information Center. The University of California at Santa Barbara and the University of Utah completed this founding quartet.

October 29, 1969 will remain in history: on that day, the first bits traveled between UCLA and Stanford. Contrary to a widespread idea, ARPANET was not born from a military obsession related to the Cold War. The much more pragmatic objective was to make colossal investments profitable by sharing these prohibitively expensive computers among research centers. ARPA was funding often incompatible machines that sat idle much of the time. Connecting them was simply common economic sense.

Steve Crocker headed the Network Working Group which delivered the Network Control Protocol in December 1970, the first communication protocol between ARPANET host computers. Adoption occurred gradually between 1971 and 1972, allowing the first real applications to emerge. Bob Kahn orchestrated a public demonstration in October 1972 that caused a sensation at the International Computer Communication Conference in Washington. From the basements of the Hilton Hotel, visitors discovered this mysterious network in real time. The success exceeded all expectations. That year also saw the birth of Ray Tomlinson's electronic mail, the first application that truly won over users.

Growth became exponential. Fifteen nodes by the end of 1971. In 1973, first international connections to University College London and the Norwegian Royal Radar Establishment. ARPANET had just invented the first international packet-switching network.

Vinton Cerf and Bob Kahn achieved a decisive breakthrough in 1973 by designing TCP/IP. They wanted to interconnect networks of different types, such as future radio and satellite networks. Their open architecture allowed each network to retain its specificities while communicating with others via TCP/IP. On January 1, 1983, ARPANET switched to TCP/IP. This technical migration required modifying all the network's computers simultaneously. To force the transition, the designers progressively deactivated the old NCP protocol in 1982, one day after another, before the final shutdown. The operation succeeded perfectly, demonstrating the maturity of the new architecture.

ARPANET also invented a development method that survives today. Steve Crocker created the Request for Comments in 1969, these informal working documents where researchers shared their thoughts. RFCs be-

came the reference tool for documenting and standardizing the network. This tradition of openness spans decades still. The initial Network Working Group evolved into the Internet Working Group and various structures such as the Internet Activities Board. Researchers and engineers developed protocols together, far from commercial or political pressures.

ARPANET transformed scientific collaboration. It gave birth to an international community united by a common culture of openness and cooperation. The technical and organizational choices of the 1970s influenced the architecture and governance of the contemporary Internet. In 1990, ARPANET quietly faded away, replaced by NSFNET and the first commercial networks that used TCP/IP.

B

In 1969, in the corridors of Bell Labs, Ken Thompson and Dennis Ritchie were working on UNIX, that operating system promising to revolutionize computing. But how could they program efficiently on these resource-limited minicomputers without getting lost in the maze of assembly language?

Thompson was well-acquainted with BCPL, the language developed by Martin Richards at Cambridge. Powerful, certainly, but too heavy for the PDP-7 and PDP-11 machines available to the researchers. The idea then emerged to create a simplified, streamlined version that would preserve BCPL's spirit while adapting to hardware constraints. Thus B was born, a name chosen by simple alphabetical contraction that would prove prescient with the later arrival of the C language.

B's philosophy lay in its radical simplicity. Gone were the complex types and syntactic subtleties that weighed down development. In B, everything is a machine word. Variables are automatically sized according to the architecture, typically 36 bits. This approach, which might seem limiting, actually concealed formidable efficiency for systems programming. Every byte counted when resources were measured in kilobytes.

The syntax inherited from BCPL but Thompson stripped it of its embellishments. Classic control structures remained: `if`, `while`, `for`. Arithmetic and logical operators kept their familiar notation. One innovation left a lasting mark on the computing landscape: the use of braces to delimit code blocks. This seemingly trivial convention would span decades to appear in countless modern languages.

B supported pointers and arrays, concepts essential for manipulating memory at the lowest level. This capability proved essential for UNIX, where every processor cycle had to be optimized. Thompson and Ritchie developed a remarkably compact compiler, fitting in just a few thousand lines. Later, they added an interpreter to facilitate interactive development. This compilation-interpretation duality offered the best of both worlds: performance for production, flexibility for testing.

Entire portions of the operating system, initially coded in assembly, were rewritten in B, making UNIX its privileged testing ground. The experience revealed B's strengths and weaknesses. On one hand, development accelerated considerably and facilitated code maintenance. On the other, certain limitations emerged, notably in string manipulation where the absence of distinction between characters and integers complicated operations.

Portability guided design choices. In a computing world fragmented among incompatible architectures, B promised to free developers from these hardware constraints. Theoretically, an application written in B could migrate from one machine to another through simple recompilation. Though reality would prove more nuanced, this forward-thinking vision would influence future languages.

The evolution of architectures toward 8 and 16 bit words made B's model less optimal. UNIX's growing needs progressively exceeded its capabilities. Dennis Ritchie, drawing on the experience gained, undertook to design a successor. In 1972, C was born, preserving B's spirit while resolving its main limitations through a more sophisticated type system.

This evolution perfectly illustrates the iterative nature of computing progress. B lived only a few years, but its influence exceeded this brief existence. Every programmer who writes a `for` loop in Java or JavaScript uses syntax inherited from B. The braces that structure code, the familiar operators, the procedural approach—so many elements that B bequeathed to posterity.

At Bell Labs and elsewhere, B served as training for an entire generation of programmers. Its relative simplicity made it an excellent pedagogical vehicle for understanding systems programming. Many who would later contribute to the rise of UNIX and the Internet cut their teeth on this language. Brian Kernighan wrote a tutorial that established enduring standards of pedagogical quality, demonstrating that a technical language could be taught with clarity and method.

B's philosophy—favoring simplicity over complexity, efficiency over exhaustiveness—transcends eras. This minimalist approach, where each element must justify its presence through practical utility, still characterizes

languages like Go, developed by Bell Labs veterans. An entire development culture is transmitted.

This reminds us that computing innovation rarely proceeds through abrupt ruptures. It advances through accumulation, each generation building on the achievements of the previous one while seeking to surpass its limits. B drew inspiration from BCPL, spawned C, which would give birth to countless descendants, with each link in the chain counting.

The First RFC

In 1969, a handful of students and researchers tackled a project that would change the world: ARPANET. This initiative by the Pentagon's Advanced Research Projects Agency aimed to connect four universities on the West Coast: UCLA, Stanford Research Institute, UC Santa Barbara, and the University of Utah. Each site had different machines, with their own systems and technical peculiarities. This diversity, which might seem an asset, proved to be more of a headache.

Steve Crocker, a student at UCLA, struggled to sleep one night in April. Technical discussions between teams were piling up, but nothing was being documented anywhere. How could these vastly different machines communicate, and how could the solutions found be documented? In the darkness of his room, he picked up a sheet of paper and began drafting what would become RFC 1, simply titled "Host Software".

This document was born within the Network Working Group, an informal collective of young researchers who met physically to discuss the very network they were building to avoid travel. The irony of the situation escaped no one. These meetings took place without any particular hierarchy, without official mandate, in a relaxed atmosphere that contrasted sharply with the usual administrative burdens.

Crocker deliberately chose the term "Request for Comments" rather than a more authoritative formulation. He feared that too academic a tone might offend more experienced experts who were perhaps, he imagined, working on the same questions from their offices on the East Coast. This initial caution shaped the spirit of RFCs: documenting without imposing, proposing without constraining.

On April 7, 1969, this first RFC addressed questions about message structuring between computers that seem rudimentary today. Crocker proposed 16-bit headers with destination addresses of only 5 bits, which

limited the network to 32 machines. He discussed *links*, those fixed connections between machines, far removed from today's dynamic connections. The network latency, estimated at one second, seems enormous compared to today's microseconds.

A visionary proposal emerged from this document: DEL, or Decode-Encode Language, a terminal description language intended to enable communication between heterogeneous systems. This idea gave rise to the communication standards that would later structure the internet.

The publication of this RFC established new principles. First, complete free access: the documents could be consulted by anyone without paying a cent. Second, openness: anyone could write an RFC, whether a tenured professor or a simple student. These democratic rules challenged academic and industrial conventions. Distribution was initially by postal mail, before evolving to FTP and then email. The first RFCs were sometimes handwritten or typewritten, witnesses to an era when computing was finding its way. Some documents still bear the marks of pen corrections, touching remnants of this artisanal period.

Jon Postel became the first official RFC editor in 1971. Under his benevolent authority, the process became more structured without becoming rigid. The documents gained uniformity while preserving their original spontaneity. He established quality criteria that raised the overall standard without discouraging contributions. Fifty years later, over 8,500 RFCs have been published. They cover all aspects of the internet, from fundamental protocols to the most sophisticated applications. This collection constitutes the technical memory of the internet, a documentary heritage unique in its scope and coherence.

The RFCs prove that it is possible to maintain rigorous standards within an open and collaborative framework. This adventure also reminds us that the foundations of the internet were laid by young people driven by the desire to share rather than to dominate.

RAM

In American laboratories of the 1960s, a glaring contradiction tormented engineers. On one hand, they were miniaturizing logic components down to the size of a small pea. On the other, they retained magnetic-core memory systems as large as barrels. These devices inherited from the 1950s worked marvelously, but their bulk was grotesque compared to the progress in integration.

Bob Norman of Fairchild Semiconductor sought a way out as early as 1961. He envisioned integrating several *flip-flop* circuits on a silicon chip, with each element storing one bit of information. The idea looked good on paper, but the technology refused to cooperate. Fairchild's management shelved the project, deeming it unfeasible. However, Texas Instruments delivered a small computer to the US Air Force equipped with a few hundred bits of semiconductor memory. This demonstration proved the concept had legs, at small or large scale. IBM sensed the potential and published an ambitious report in 1965 titled *Potential for Monolithic Megabit Memories*. The company pushed experimentation by developing a 16-bit chip for the System/360 Model 95, a machine specifically designed for NASA's needs in 1966.

Tom Longo turned the tables the following year. With his team at Transitron, he developed the first commercial multi-cell memory chip. These 16 bits exploited TTL technology and immediately found buyers at Honeywell. The success prompted Fairchild and Texas Instruments to develop their own versions.

From IBM's laboratories, Robert Dennard revolutionized the approach in 1966. He invented the single-transistor DRAM cell, a technical feat that radically simplified memory architecture. Existing SRAM cells required multiple transistors per stored bit. This innovation paved the way for modern DRAM memories.

Fairchild revived its ambitions in 1967 with the SAM (Semiconductor Advanced Memory) program. The goal was to produce a 64-bit chip. The following year, manufacturers offered TTL memories of this capacity, organized into 16 words of 4 bits each, with access times reaching 60 nanoseconds.

Radiation Inc., future Harris Semiconductor, introduced PROM (Programmable Read-Only Memory) in 1969. These user-programmable components appealed to computer developers. They allowed microcode modification during debugging phases, an unprecedented flexibility that accelerated system development.

In 1970 with the Intel 1103, William Regitz and Joel Karp, supported by Ted Hoff and Ted Rowe, developed the first commercially viable DRAM. This 1,024-bit chip still suffered from limitations in speed and ease of use, but it definitively proved that semiconductor memories could equip computers.

Dov Frohman made a strong impact the following year with the Intel 1702, the first EPROM memory. This invention exploited a floating-gate technique conceived at Bell Labs. The principle: electrically program the memory and erase it by exposure to ultraviolet light through a small

quartz window built into the package.

Bob Proebsting of Mostek changed the rules in 1973. His MK 4096 offered 4 Kbit in a package of only 16 pins thanks to a clever multiplexed addressing technique. This approach influenced all subsequent generations of DRAM, like an industry standard.

The capacity race accelerated. The 16 Kbit chips arrived in 1976, exploiting dual-layer polysilicon technology that optimized the arrangement of memory cells. The 64 Kbit followed in 1979, then the 256 Kbit in 1982. Each generation brought its share of technological innovations and pushed the limits of miniaturization.

In 1983, Intel struck hard with the first 1 Mbit CMOS DRAM. This technological transition to CMOS outlined the sector's future. Paradoxically, Intel soon abandoned the DRAM market, leaving the field open to its Japanese and Korean competitors who rushed into the breach.

Innovation continued with the 1 Mbit chips in 1986. These components inaugurated the use of non-planar memory cells, stacked or trenched to save space. Capacities climbed relentlessly: 4 Mbit in 1988, 16 Mbit in 1991, 64 Mbit in 1994. Each technological leap came with increasingly formidable manufacturing constraints.

The year 1998 saw the birth of 256 Mbit DRAMs that introduced high-permittivity dielectrics. This material innovation allowed the continuation of miniaturization while maintaining the electrical properties necessary for proper memory cell operation.

This frantic race for performance followed Moore's Law with striking regularity. The doubling of capacities approximately every two years was accompanied by a vertiginous drop in cost per stored bit. This economic democratization propelled random-access memories into all electronic devices, from personal computers to mobile phones.

UNIX

Bell Labs was buzzing with activity in the late 1960s. A team of researchers was working tirelessly on Multics, a titanic project carried out with MIT and General Electric. This colossal ambition was to create an operating system capable of managing multiple users and tasks simultaneously. Among these visionaries were Ken Thompson and Dennis Ritchie, two men who would soon shake up computing.

Multics was struggling. AT&T, weary of astronomical costs and endlessly stretching deadlines, pulled out in 1969. Thompson and Ritchie found themselves with solid experience but no machine to continue their research. Management refused their request to acquire a new computer. Thompson had the idea of recovering an old PDP-7 gathering dust in the laboratories. On this obsolete machine, he began building something unprecedented. He developed a basic kernel, an assembler, a shell, and some fundamental tools like `rm`, `cat`, and `cp`. The name UNIX emerged from a play on words with Multics, an ironic nod to simplicity in the face of its predecessor's complexity.

The year 1970 was a game-changer. The team secured a PDP-11. This machine had no disk and only 24 KB of memory, but it enabled the emergence of the first official edition of UNIX. Development accelerated. Thompson created the B language, a streamlined version of BCPL, while Ritchie worked on what would become the C language between 1971 and 1973.

This invention of C revolutionized everything. With UNIX's fourth edition in 1973, Thompson and Ritchie dared to completely rewrite it in this new language. This gamble, audacious at a time when operating systems were written in assembly, made UNIX portable from one architecture to another. The first six editions came in quick succession between 1971 and 1975.

The sixth edition, nicknamed V6, marked UNIX's first major distribution outside Bell Labs. AT&T distributed the system to universities with its complete source code for next to nothing, a decision that shook the computing world.

The University of California at Berkeley entered the UNIX saga in 1975 when Bob Fabry installed the fourth edition there. Ken Thompson spent a sabbatical year there that energized two brilliant students: Bill Joy and Chuck Haley. They threw themselves into enhancing the system, developing the `ex` and `vi` editors, the latter remaining a cornerstone of text editing today, inspiring projects like neovim. In 1977, Bill Joy assembled the Berkeley Software Distribution (BSD), printed in only about thirty copies but destined to spawn many followers.

The 1980s saw a myriad of UNIX versions flourish. AT&T commercialized System III and then various releases of System V through its UNIX System Group. Berkeley meanwhile continued BSD development, bringing remarkable innovations like the Fast File System and especially the TCP/IP network stack. The latter, integrated into version 4.2BSD in 1983, propelled BSD to glory.

Bill Joy left Berkeley in 1982 to co-found Sun Microsystems, a company

that would weigh heavily in UNIX's evolution. Sun developed the Network File System (NFS) and the VFS/vnode architecture, making file sharing over networks invisible to users. In 1987, an unexpected alliance formed between Sun and AT&T to create System V Release 4, merging the best features of SunOS and System V Release 3.2.

This proliferation of UNIX versions created a crying need for standardization. The `/usr/group`, an association for promoting UNIX established in 1980, tackled the standardization of user interfaces. These efforts led in 1985 to the birth of the POSIX standard (Portable Operating System Interface for Computing Environments), now an essential reference for compatibility between UNIX systems. The `X/Open` group, created in 1984, developed the X/Open Portability Guide (XPG), integrating POSIX and other standards.

The early 1990s saw an unexpected rival emerge: Linux. In 1991, Linus Torvalds, a student at the University of Helsinki, developed a UNIX-compatible kernel capable of using GNU tools from the Free Software Foundation. Linux captured UNIX's spirit while adapting it to modern architectures. Its free license accelerated its distribution and massive adoption.

UNIX revolutionized computing through its innovative concepts. The hierarchical file system, where everything is a file, radically simplified resource management. Pipes, suggested by Doug McIlroy in 1972, connected one program's output to another's input, creating processing chains of unmatched power. The shell offered a complete programming environment from the command line.

UNIX's portability, made possible by its implementation in C, facilitated its adaptation to a multitude of architectures. This characteristic, combined with its modular design and exemplary documentation, explains its remarkable longevity. UNIX manuals, present from the first edition, established a technical documentation standard that remains a reference today. Its design philosophy still permeates software development: create simple programs that excel in their domain, use text as a universal interface, compose elementary tools to solve complex problems.

VI

1970

When the Digital World Transformed Everything

While bombs rain down on Vietnam, three engineers at Intel complete the design of the first commercial microprocessor. Their invention, almost unnoticed in a world obsessed with the Cold War, will change everything.

The opening decade unfolds in a peculiar climate. America bleeds in Vietnam. Young people take to the streets. The USSR and the United States eye each other warily. Oil becomes a weapon. And amidst this global chaos, a handful of men and women invent the digital future.

American campuses are boiling over. Social protest rejects established authority, including that of large centralized computers symbolizing institutional power. At Berkeley as at MIT, students dream of a different kind of computing—more personal, more accessible. In university laboratories, people work on machines that bear no resemblance to the air-conditioned colossi of corporations. There, away from the paths marked out by IBM, another vision of the computer is born.

In 1973, the oil shock strikes. Lines grow at gas stations. Inflation soars. Faced with this crisis, companies seek to reduce their costs. Computerization appears as a solution. Banks are the first to embrace it, as they must manage an increasing number of transactions in an unstable economic context. Their needs propel the development of ever more so-

phisticated databases. Oracle emerges in this context, based on an idea by Larry Ellison inspired by a theoretical IBM article.

In the East, behind the Iron Curtain, the USSR attempts to keep pace. It launches its own computing plan, sometimes copying Western architectures. The SALT agreements change nothing—technology remains a power issue. Soviet computers equip factories and administrations but suffer from a growing lag. Computing visibly marks the difference between the two blocs.

Japan plays its own tune. Without the military constraints weighing on American industry, Japanese companies focus on consumer electronics. They excel in miniaturization. Sony, Panasonic, and Sharp establish themselves worldwide. Their components soon equip the first personal computers. The country, defeated in 1945, takes its technological revenge.

Europe advances in scattered order. France relies on its Calcul Plan and the International Company for Computing to create a national champion. Germany depends on Siemens. The United Kingdom develops ICL. But these national initiatives struggle against American giants. European universities, however, shine in theoretical research. The Pascal language, created by Niklaus Wirth at the Swiss Federal Institute of Technology in Zurich, marks an advance in programming language design.

Television then reigns supreme over Western households. Images of the first step on the Moon, viewed by hundreds of millions of spectators, demonstrated the power of communication networks. Telecommunications engineers reflect on digital data transmission, particularly with the first videotex experiments. In France, the PTT launches studies that will lead to Minitel, the first mass-market network before the Internet era.

In schools and universities, computer science takes its first steps as a teaching discipline. The BASIC and Logo languages are designed with an educational objective. Seymour Papert, at MIT, theorizes learning through programming. These reflections open a new field: computing is not only a calculation tool but also a means of learning differently.

The working world transforms—cathode ray tube screens gradually replace cardboard files. In offices, secretaries discover word processing. The first daisy wheel printers reproduce typewriter quality but with electronic flexibility, changing administrative professions. Unions worry and negotiate agreements on screen working conditions, but the march toward all-digital is launched.

Science fiction seizes upon the computer theme. “2001: A Space Odyssey” popularized the image of HAL 9000, an artificial intelligence gone mad. In 1973, the film “Westworld” imagines uncontrollable robots. These works express the hopes and fears aroused by computerization. Simulta-

neously, the first commercial video games like Pong attract a new audience to these machines once reserved for scientists.

Healthcare doesn't escape this wave. Hospitals equip themselves with computer systems for patient management. The first medical scanners, appearing at the beginning of the decade, produce digital images that only computers can process. Pharmaceutical research increasingly relies on molecular modeling. Medical computing emerges as a new discipline.

Large cities become laboratories where urban computing applications are tested. Traffic light management, optimization of water or electricity networks benefit from real-time control systems. These applications require computers capable of reacting instantly to their environment, stimulating the development of embedded systems.

International trade metamorphoses with the adoption of electronic exchange standards. Barcodes appear in American supermarkets. Banking transactions are automated with the SWIFT network. Supply chain computerization begins. These developments accelerate the globalization of subsequent decades, where information flows will circulate as quickly as goods.

Steve Jobs and Steve Wozniak found Apple in 1976. Their Apple I symbolizes the spirit of this new computing: artisanal, personal, almost rebellious. At the same time, Bill Gates and Paul Allen create Microsoft. These young entrepreneurs bear no resemblance to the leaders of traditional large computing companies. They inaugurate a new era where the computer is both a consumer product and an object of passion.

Fundamental research advances rapidly. Alan Kay's work at Xerox PARC on graphical interfaces enables more intuitive computing. Dennis Ritchie and Ken Thompson's creations at Bell Labs—the C language and the UNIX system—launch modern computing. These innovations, initially confidential, will transform the industry's face over subsequent decades.

At the end of the 1970s, the first amateur computer clubs gather enthusiasts around rudimentary machines like the MITS Altair 8800. These communities develop a culture of sharing that contrasts with the closed world of professional computing. Specialized magazines publish programs that readers copy line by line. No one yet imagines that these obscure practices represent 21st-century computing.

The computer, once a tool reserved for large organizations, begins its transformation into a daily companion during the 1970s. The ideas emerging during this period will shape our digital world for the following 50 years, designing a society where information processing is as essential as the production of material goods.

ALOHAnet

In September 1968 at the University of Hawaii, Norman Abramson faced a problem that seemed almost insurmountable. How could he connect computers scattered across multiple islands, separated by hundreds of kilometers of ocean? Cables cost a fortune, telephone links were slow and unreliable. Yet the university needed to connect its main campus at Manoa, near Honolulu, to research centers scattered across Oahu, Kauai, Maui, and the Big Island of Hawaii.

The idea germinating in his mind seemed somewhat crazy: use radio waves to enable computer communication. In 1968, no one had yet attempted such an endeavor. Computer communications were confined to direct cable connections, point-to-point, following patterns inherited from telegraph and telephone systems. But Hawaii imposed its own rules. The volcanic geography of the archipelago transformed what elsewhere would be a constraint into a laboratory of innovation.

The system Abramson envisioned relied on a deceptively simple architecture. At the heart of the setup stood an IBM 360/65 equipped with 750 KB of memory, installed on the main campus. This machine communicated with a more modest computer, an HP 2115A christened MENEHUNE, named after the small beings of Hawaiian mythology reputed for accomplishing "impossible" tasks in a single night. The MENEHUNE managed radio transmissions through two 100 kHz UHF channels: one broadcast at 407.350 MHz to remote terminals, the other collected their responses at 413.475 MHz. Each channel operated at 24,000 baud.

But the project's true boldness lay elsewhere. In the downlink direction, from MENEHUNE to the terminals, everything was conventional: a single source transmitting, recipients listening, scheduling following well-established priority rules. It was in the other direction that innovation emerged. How to manage communications when dozens of terminals all want to speak simultaneously on the same channel? Traditional multiplexing techniques would impose on each terminal a fixed time or frequency slice, whether it used it or not. An unacceptable waste for computer traffic consisting of short bursts separated by long pauses.

The solution proposed by Abramson's team broke all established codes. Each terminal transmitted whenever it pleased, without asking anyone's permission. Data traveled in standardized packets: 80 eight-bit characters for payload, plus 64 bits for identification, control, and error detection. When MENEHUNE received an intact packet, it sent an acknowledg-

ment. Otherwise the sender waited a random delay and tried again. This approach accepted the unacceptable: packet collisions. Two terminals transmitting simultaneously would see their signals mix, creating gibberish incomprehensible to the receiver. Rather than seeking to avoid these collisions at all costs, ALOHA considered them a necessary evil, the price to pay for unmatched operational simplicity.

Abramson's mathematical calculations revealed the limits of this philosophy. The system exploited at best only 18.4% of the channel's theoretical capacity, a performance reached when offered load represented exactly half of that capacity. Beyond that, collisions multiplied, retransmissions accumulated, and the network collapsed into chaos of interfering signals.

ALOHA proved that a system could function without a conductor, without central coordination, without pre-established scheduling. Terminals managed among themselves, accommodated conflicts, found their equilibrium in what resembled organized anarchy.

In 1973, a young Xerox engineer named Robert Metcalfe visited the installation. He understood that ALOHA's principles could adapt to media other than radio waves. A few years later, Ethernet would be born, a wired transposition of Abramson's ideas that would revolutionize local area networks.

Meanwhile, the Hawaii researchers refined their creation. They invented "slotted ALOHA", a variant where transmissions could only begin at predefined instants, like beats of an invisible metronome. This synchronization doubled theoretical efficiency, raising maximum throughput to 37% of channel capacity. The improvement paved the way for an entire family of derived protocols.

The project's success illustrates a truth about technological innovation. Often, the most severe constraints generate the most creative solutions. Hawaii could not simply copy continental recipes. Geographic isolation forced researchers to fundamentally rethink computer communications. In doing so, they discovered universal principles that transcended their particular situation.

Wi-Fi networks, satellite communications, cellular systems—all incorporate mechanisms inspired by the Hawaiian protocol. The idea that a network can function without strict central control, managing conflicts rather than avoiding them, has become a paradigm of distributed computing. Forty years after its creation, ALOHA retains its inspirational force. Not so much through its raw performance, far surpassed by current standards, but through the conceptual audacity it exemplifies. Sometimes, you just need to let the machines fend for themselves.

FORTH

Charles "Chuck" Moore worked at MIT, then at the Stanford Linear Accelerator Center. The computing landscape of the time frustrated him because FORTRAN and ALGOL monopolized attention, but these languages forced him to spend an inordinate amount of time between writing code, compiling it, and executing it. He was looking for something more direct, more immediate.

His solution took the form of a text interpreter that he wrote in ALGOL. The principle was that everything you gave it was either a number or a word representing a command. Gone were the separate phases of editing, compilation, and execution. Everything happened interactively, in real time. He invented special words like ":" to tell the system it needed to start compiling the following code, and ";" to tell it to return to normal interpretation mode. When the compiler encountered such a block, it stored this "word"—the equivalent of our current functions—in a dictionary for later reuse.

FORTH's architecture relies on two stacks working together. The first stores temporary values and parameters, the second manages stack frames. This approach produces compact and fast code, but above all, code that is infinitely extensible. In 1970, Moore was working on a third-generation IBM computer that limited names to a maximum of five characters. He therefore named his language "FORTH", an allusion in English to a fourth generation of programming language.

The National Radio Astronomy Observatory in Arizona welcomed the first public version of FORTH in 1971. The system was used to collect and analyze data while controlling the radio telescope in real time. The astronomy community adopted this technology. Moore then created several versions: MiniFORTH for minicomputers, MicroFORTH for microcontrollers. Each variant adapted to the specific constraints of its environment.

FORTH uses postfix notation, also called reverse Polish notation. Operators come after their operands. The expression `1 2 +` means that 1 and 2 are added. However, the operation is interactive: when the user types this expression, the interpreter first pushes 1 onto the stack, then 2, before executing + which retrieves these two values, adds them, and puts the result back on the stack.

FORTH's real strength comes from its ability to grow. Programmers create new commands, "words", which become an integral part of the language. This extensibility adapts FORTH to the precise needs of each

application. The language combines low-level calculations with a powerful mechanism for declaring functions. Once compiled, FORTH programs run almost as fast as hand-written assembly. With a compiler that optimizes aggressively, FORTH is highly efficient in numerical processing. For embedded systems, prototype hardware, or boot loaders, a minimal compiler kernel is all that's needed. Once this kernel is available, you can build a complete FORTH compiler using nothing but FORTH.

FORTH's minimalist philosophy is reflected in memory management. The language only offers storage of simple or double values, typically integers. To build an array, you must allocate a variable and increment an internal pointer in the dictionary to reserve more space. An array in FORTH is just a region of memory without any particular documentation. The language doesn't offer a native heap either; programmers write their own memory management routines.

Concrete applications of FORTH are multiplying. Sun Microsystems integrates it into the OpenBoot firmware of its workstations. WearLogic has developed an electronic wallet based on an AVR microcontroller. Advanced Energy Monitoring Systems (AEMS) uses FORTH for its Yatesmeter, a sophisticated instrument that monitors pumps in real time. These successes show that FORTH remains relevant for embedded systems and hardware control.

Technological evolution has not diminished interest in FORTH. Its interactive approach and ability to build abstractions step by step make it a valuable educational tool. Projects like pbFORTH for LEGO Mindstorms robots reveal its potential for teaching programming. Its particular syntax and mental model, which differs from conventional languages, offer students another perspective on programming concepts.

The FORTH community is active. User groups still exist in several countries, conferences like euroFORTH are held regularly. Recent developments include GNU Gforth, a portable version that complies with standards, and VFX Forth, a modern optimizing compiler that speeds up FORTH programs by a factor of three to five compared to traditional compilers.

Rather than imposing an abstraction between man and machine, this language creates a direct link to the hardware while allowing the construction of useful abstractions.

Pascal

Niklaus Wirth works at the Swiss Federal Institute of Technology in Zurich. A frustration seizes him when faced with existing programming languages. These tools seem incoherent to him, their constructs defy all logical explanation. For a mind accustomed to systematic, methodical, and orderly reasoning, this situation is unbearable.

He designs Pascal with two distinct ambitions. He first wants to create a language truly suited to teaching, where each concept is naturally reflected in the syntax. But he refuses to sacrifice efficiency for the sake of pedagogy: his implementations must run reliably on the machines of the era.

The legacy of Algol 60 weighs heavily in this approach. This language meets pedagogical requirements better than any other at the time. Wirth adopts its structuring principles and the form of its expressions. Yet he resists the temptation to make Pascal a simple subset of Algol 60, because certain declaration mechanisms would prevent him from integrating the new features he has in mind. These features primarily concern data structures, whose absence in Algol 60 largely explains why that language is confined to a restricted application domain. By introducing records and files, Wirth hopes that Pascal can tackle commercial management problems, or at least serve to demonstrate them effectively in a programming course.

The first version appears in 1970, accompanied by an implementation on CDC 6000 computers. Three years of intensive use, teaching, and experimentation give birth to a revised version in 1972. The changes reflect lessons learned in the field: constant parameters give way to value parameters, the class structure disappears, file handling is rethought.

Standardization arrives in 1983 with the ISO 7185 standard, which defines unextended Pascal. An update occurs in 1990, the year extended Pascal standard ISO 10206 is born. This distinction addresses the tension between maintaining the simplicity of the original language while meeting the growing needs of modern programming.

Extended Pascal brings a separation between modularity and compilation. Each module allows the export of interfaces containing values, types, schemas, variables, procedures, and functions. The system finely controls visibility, and character strings receive unified treatment. Fixed-length strings, character values, and variable-length strings are compatible with each other. The concatenation operator combines them freely, while the programmer specifies the maximum lengths of variable strings. Variable binding constitutes a remarkable innovation. A variable declared as

a binding can connect to file storage, the real-time clock, or command lines. This restriction limits external connections to only those variables explicitly intended for this purpose.

The emergence of object-oriented programming pushes certain Pascal compilers toward this new paradigm. In 1993, the Pascal standards committee publishes a technical report on "Object-Oriented Extensions to Pascal". The members of this committee represent a surprising range of organizations: from Pace University to the US Air Force, from Apple Computer to Microsoft and Digital Equipment Corporation.

Pascal marks the history of programming through its clear syntax and strong typing philosophy. An entire generation of programmers discovers through it the principles of structured programming and rigorous design. Its massive use in teaching during the 1970s and 1980s forges disciplined programming habits that will long outlive it. Commercial success accompanies pedagogical success, notably thanks to Borland's Turbo Pascal. These commercial versions often enrich the standard language with practical extensions, never betraying the clarity and rigor that characterize the original.

The Pascal language embodies the idea that a program must be a precise mathematical construct, whose correctness can be formally proven. This vision contrasts with current trends that favor development speed and flexibility, sometimes at the expense of formal rigor, and therefore sometimes of security. Wirth may have been right: in a world where software governs our lives, the discipline he advocated has never been more necessary.

DEC PDP-11

In 1970, Digital Equipment Corporation revolutionized the minicomputer world with the PDP-11. This machine marked a decisive step in computer evolution, not through raw power, but through an architectural vision: the UNIBUS. This unified bus connected processor, memory, and peripherals on a common interface, where previous machines multiplied specialized connections. The innovation appeared technical, yet it radically transformed computer system design.

The PDP-11's history began around 1968 in DEC's laboratories. Gordon Bell, Harold McFarland, and Roger Cady led a team confronted with the limitations of minicomputers of the era. Cramped address space, in-

sufficient registers, absence of hardware stack, approximate interrupt management... These defects handicapped the development of sophisticated applications, so the engineers explored different approaches. A prototype called PDP-X emerged, followed by a DCM (Desk Calculator Machine) version. These experiments ultimately converged toward the PDP-11's definitive architecture.

The result exceeded expectations. The PDP-11 offered an orthogonal instruction set, the same addressing modes worked with all instructions. Its eight general-purpose registers provided unprecedented flexibility, one serving as a stack pointer, another as a program counter. Direct byte addressing, quite rare, facilitated character processing and variable-length data. These technical innovations translated into more natural and efficient programming.

The market responded enthusiastically. Between 1970 and 1975, DEC sold more than 20,000 PDP-11 units. This success relied on the architecture's modularity which allowed an extended range from small systems costing a few thousand dollars to configurations reaching several hundred thousand dollars. Technological advances accompanied this rise in power. The first models relied on magnetic core memory, soon replaced by MOS semiconductor memory, then by fast bipolar cache memory.

The family expanded. The initial PDP-11/20 model gave way to the 11/45 in 1972, a more robust machine that gained in performance. The 11/70 arrived in 1975 with its cache memory and extended addressing capacity of 4 megabytes. At the opposite end, the 1975 LSI-11 exploited large-scale integrated circuits to miniaturize and reduce costs. From this diversity, the PDP-11 would conquer varied territories such as scientific computing, business management, industrial process control, and communications.

DEC understood that a computer without an operating system is an empty shell. The company developed several software environments adapted to specific needs: RT-11 for simple real-time applications, RSX-11M for complex real-time multitasking, RSTS/E for interactive time-sharing. These mature systems, accompanied by quality development tools, facilitated the PDP-11's adoption by a diverse clientele.

The introduction of DECnet in 1975 interconnected PDP-11s with each other and with other DEC machines according to a distributed approach that contrasted with IBM's hierarchical philosophy. DECnet deployed a complete network architecture, from the physical layer to applications. Its successive evolutions enriched its capabilities: Phase II in 1978 established gateways to VAX systems, Phase III introduced adaptive routing.

The PDP-11's architecture directly inspired future processors like the

Motorola 68000. Competitors adopted UNIBUS as a de facto standard. The general-purpose register organization and sophisticated addressing modes of the PDP-11 permeated modern processor design. Mechanisms like program counter-relative addressing or auto-increment/decrement became widespread throughout the industry.

The PDP-11 also found its place in UNIX history. Ken Thompson and Dennis Ritchie chose this machine at Bell Labs to develop their operating system. The clarity and consistency of the PDP-11's architecture made it an excellent pedagogical support. Numerous university textbooks used it as a reference, thus training an entire generation of computer scientists in its concepts.

But time gradually revealed this architecture's weaknesses. The 16-bit address space limited memory to 64 kilobytes per program, a constraint that weighed increasingly heavily with applications' growing sophistication. DEC added memory management mechanisms to circumvent this limitation, at the cost of increased complexity. The proliferation of instruction set variants, notably the different floating-point options, complicated software portability.

The decline began in the 1980s. 16-bit microcomputers gained power while remaining much less expensive. Above all, DEC launched the VAX which offered 32-bit address space and relegated the PDP-11 to second place. Production nevertheless continued until the 1990s for specialized applications that did not require the VAX's capabilities.

The PDP-11 left its mark on the era. It established DEC as a major player in the computer industry, and demonstrated that a well-designed architecture can transform a market and influence technological evolution.

B-tree

When discussing modern computing, certain innovations remain in the shadows despite their ubiquity. The history of the B-tree begins in the late 1960s. Computers were already juggling considerable volumes of data, but accessing this information stored on magnetic disks posed a real challenge. Each read required time-consuming mechanical movements, and programmers desperately sought ways to organize their indexes to limit these accesses. At Sperry Univac, Howard Chiat and Meyer Schwartz tackled this challenge in collaboration with Case Western Reserve University.

They were not alone: Bruce Cole, Stewart Radcliffe, and Michael Kaufman conducted parallel research at Control Data Corporation, supported by Stanford University.

These scattered efforts found their theoretical culmination through Rudolf Bayer and Edward McCreight. At Boeing Scientific Research Labs, these two researchers formalized the emerging concepts and published their paper *Organization and maintenance of large ordered indices* in 1972. This text laid the mathematical foundations of the B-tree and revolutionized index management.

The genius of the B-tree lies in its conceptual simplicity. Where classic binary trees limit each node to two children, the B-tree allows wider nodes containing multiple keys. This architectural freedom radically transforms operational efficiency. The tree automatically maintains its balance during insertions and deletions, guaranteeing a logarithmic height that minimizes disk accesses. Imagine an intelligent phone directory that constantly reorganizes itself so that each search requires an identical number of steps, regardless of the directory's size.

Other variants emerged in the following years. Donald Knuth proposed the B+-tree, a clever modification that concentrates all data in the tree's leaves. Internal nodes serve only for indexing, considerably simplifying sequential data traversal. This variant became the reference standard in database management systems. In 1977, Bayer partnered with K. Unterauer to create the Prefix B+-tree. Their innovation consisted of storing only the prefixes necessary to distinguish entries, saving space and increasing the branching factor.

Industrial adoption followed naturally. IBM integrated B-trees into VSAM (Virtual Storage Access Method), demonstrating their commercial viability. The system exploited a variant of the B+-tree and introduced optimizations such as replicating sequence nodes on disk cylinders. This concrete implementation proved that the theory could indeed improve production performance.

Multi-user requirements emerged, along with the need for multiple programs to simultaneously modify the same structure without corrupting it. Bayer and Mario Schkolnick solved this equation in 1977 with sophisticated locking protocols. Their system allowed multiple simultaneous readers while properly managing concurrent modifications, a technical feat from which modern databases inherit.

Theoretical research accompanied these practical developments. Raymond A. Miller studied optimal tree construction in 1977, while Andrew Chi-Chih Yao analyzed the probabilistic behavior of nodes in 1978. His calculations revealed an average occupancy rate of 69%, a figure that

helped designers size their systems.

The influence of B-trees extends beyond their origins. MySQL exploits them in its InnoDB engine, PostgreSQL uses them for its indexes, and numerous file systems such as NTFS, HFS+, or Ext4 organize their data according to these principles. This ubiquity testifies to their conceptual robustness.

The 1990s and 2000s brought new problems. Multicore architectures and complex memory hierarchies required adaptations. Researchers developed variants optimized for processor caches and created the Bε-tree specifically for SSD disks. These evolutions showed how an idea can adapt to emerging technological constraints.

Big Data and real-time processing pushed the limits even further. New versions improved parallelism and scalability, while current research explores optimization for non-volatile memories and distributed architectures.

More than fifty years after its formalization, the B-tree continues to structure our daily data. From smartphones to data centers, this discreet yet essential invention silently organizes digital information.

Floppy Disk

In 1967 at IBM, David Noble was tasked with solving an apparently trivial problem: finding a way to load programs into the new System/370 computers at startup. The semiconductor memories equipping these machines lose their content as soon as they're powered off, unlike the older magnetic core memories. An external, reliable medium was therefore needed.

Noble's team, which included Alan F. Shugart, Herbert Thompson, Ralph Flores, and Warren L. Dalziel, began by exploring magnetic tape cartridges. But Noble had another idea: why not use a flexible disk? The first experiments bordered on amateurism. The team bought a 45 RPM vinyl record player from a store and jury-rigged a magnetic read head mounted on the arm. The flexible disk rested on a foam-lined platter.

These experiments quickly revealed a nightmare, as the slightest speck of dust or stray hair caused data loss. Worse, these impurities literally "crawled" across the disk, pushed by the read head. The brilliant idea emerged from an evening work session between Herbert Thompson and Ralph Flores. Frustrated with the turntable approach, they abandoned

this path and rushed to a grocery store to buy pink absorbent paper, the kind used for cleaning.

By cutting a file folder to the right size and gluing this pink paper to it, they created the first protective envelope. They cut openings for the read head and the central mounting system. The dust problems disappeared instantly. This pink paper would later give way to a specialized white material, but the principle remained unchanged for decades.

The first IBM floppy disk drive, dubbed "Minnow", hit the market in 1971. It was read-only and equipped the IBM 3330 hard disk controllers as well as certain System/370 models. The following year, Memorex made a major breakthrough with the 651, the first OEM drive capable of both reading and writing. Wang Laboratories and DEC eagerly adopted it to replace their aging punched tape readers.

The real turning point came in 1973 when IBM launched its 3740 data entry system equipped with the 33FD drive. A single floppy disk could now store the equivalent of 3,000 punch cards, drastically reducing paper consumption. Its 80-kilobyte storage capacity revolutionized working habits. The industry adopted this format as a de facto standard. Shugart Associates, the company founded by Alan Shugart, took the lead in the compatible drive market.

In 1975, the market was organized around well-defined segments. Data entry accounted for 12,500 units sold, intelligent terminals 10,200, minicomputers 3,100, word processing 800 units, not to mention other emerging applications. A drive cost about $380, a considerable sum.

The year 1976 marked the arrival of the 5.25-inch format, born from an informal conversation at Shugart Associates. Customers complained about the bulkiness of the 8-inch drives, which weighed 17 kg each. The team developed a prototype the size of a cocktail napkin in six weeks. This innovation preserved the technical performance of the 8-inch format in a case adapted to the emerging desktop computers.

IBM responded in 1977 with the 53FD model, the first double-sided drive that doubled capacity while maintaining compatibility with single-sided disks. Adapting this technology proved to be an uphill battle. Shugart Associates nearly went bankrupt with its double-sided read heads inspired by Winchester hard drives. It was ultimately Tandon that found the viable solution with one fixed head and one mobile head.

This year 1977 also saw the birth of the Apple II, the first mass-produced personal computer equipped with 5.25-inch floppy disk drives. Personal computers adopted this configuration as a standard that helped standardize storage media. The software industry could finally take off.

The genesis of the 5.25-inch format reveals entertaining many anec-

dotes. Wang Laboratories dreamed of a desktop computer with a $100 drive. Steve Jobs, unknown to the general public, regularly showed up at Shugart Associates seeking an affordable drive to replace the audio cassettes Apple used for data storage. Don Massaro and Jimmy Adkisson's team deliberately chose the 5.25-inch dimension because it was the smallest size that didn't allow the diskette to slip into a pocket, thus avoiding damage from carelessness.

The 8-inch format experienced its golden age in the early 1980s with 1.5 million units produced annually. The industry comprised 25 American manufacturers, 11 Japanese, and 8 European. Margins eroded in the face of fierce competition, dropping from 55% to 40%. Japanese manufacturers, obsessed with production quality, gradually dominated the global market.

The decline of the 8-inch format began with the success of the 5.25-inch, but production continued until the late 1990s. IBM maintained contracts with YE Data in Japan to deliver 5,000 to 10,000 drives per year, exclusively for maintaining old systems still under contract.

The 1980s saw most personal computers accommodate both a 5.25-inch drive and a 3.5-inch drive. The latter format eventually became the new standard. The 1990s marked the peak of the 3.5-inch floppy with over five billion units sold each year. But the arrival of rewritable CDs, DVDs, and then USB flash drives spelled the end for this technology. In 2011, Sony, the last global manufacturer, closed the book on floppy disk production.

The floppy disk saga implicitly tells the story of the complete transformation of computer storage media. From a simple program loading tool, it evolved into a versatile peripheral that heralded modern removable storage. Without it, neither the software industry nor the democratization of personal computing would have reached the scale we know today.

ed

Some tools mark their era through radical simplicity rather than sophistication. The `ed` editor belongs to this category of innovations that, born at Bell Labs in the early 1970s, shaped how we conceive text editing. Its development responded to the hardware constraints of teletype terminals that could only print one line at a time on paper, but its design reveals a philosophy that transcends these technical limitations.

The genius of `ed` lies in its ability to transform text editing into a di-

alogue between user and machine. Unlike modern editors that expose the entire document, `ed` requires thinking of text as a sequence of numbered lines, stored in a memory space called a *buffer*. Far from being a weakness, this abstraction is a strength that forces users to develop a precise mental representation of their document's structure.

The syntax of `ed` stands out through its conciseness. The commands `a`, `i`, `d`, `s`, and `p` suffice to cover most editing operations. Behind this apparent simplicity lies an addressing system of remarkable richness. Users can designate the current line with a simple dot, the last line with a dollar sign, or specify entire ranges using regular expressions. This approach revolutionizes text manipulation by introducing a formal grammar to describe textual patterns.

Regular expressions constitute perhaps the most enduring legacy of `ed`. These patterns, which describe text structures rather than literal strings, transform search and modification into logical operations. The dot representing any character, the asterisk quantifying repetitions, the brackets defining sets—all concepts that would later inspire programming languages and text analysis tools. The substitution command `s` illustrates this power since it can transform entire documents with a single instruction, provided the user masters the language of patterns.

This editing philosophy finds its purest expression in `ed`'s minimalist interface. When faced with an error, the editor simply displays a laconic `?`. This parsimony, which disconcerts beginners, reflects a conception where the tool must disappear behind the task. The creators of `ed` operated on the principle that efficiency arises from mastery, and that verbosity harms concentration.

Technical limitations become apparent in certain constraints, such as the 512-character maximum per line, or the 256-character limit for global commands. These restrictions, which seem trivial today, testify to the ingenuity required to create powerful tools with limited resources. Programmers had to consider every byte, optimize every algorithm.

The influence of `ed` extends beyond the realm of text editing. The `grep` utility takes its name from the command `g/re/p` (global search for regular expression and print), transforming an `ed` feature into a standalone tool. The `sed` editor extends `ed`'s logic to stream processing, while `vi` preserves its fundamental concepts while adapting them to visual editing. This genealogy reveals how innovations propagate and evolve within the UNIX ecosystem.

`ed` persists in all UNIX and Linux systems. Its presence is no longer anecdotal—in constrained environments, during limited network connections, or in administration scripts, its original qualities regain their rele-

vance. The editor that seemed destined for computing museums proves sometimes to be the most suitable tool for certain situations.

The original `ed` documentation, written by Brian W. Kernighan, sets the standards for UNIX technical documentation. It combines theoretical rigor and pragmatism, explaining what the tool does and why it does it that way. This pedagogical approach, where each concept builds on concrete examples, still influences how we document software.

File Transfer Protocol

In 1971, the Internet resembled a technological landscape where every simple operation required mastering cryptic commands. The idea of exchanging files between remote machines was a technical challenge. This is when the File Transfer Protocol, better known by the acronym FTP, was born with the publication of RFC 114 by the Internet Engineering Task Force. This document describes the first version of a protocol that would span decades.

Initially, RFC 114 from April 1971 proposed a simple architecture with a single full-duplex connection to carry both commands and data. This straightforward approach reflected the state of computer networks at the time. But the protocol's designers would come to understand the limitations of this approach. Through successive revisions in the 1970s, a major innovation emerged with the complete separation of the control channel and the data channel. This architecture, formalized in RFC 765 from June 1980 and then perfected in RFC 959 from October 1985, establishes two distinct TCP connections. The first, on port 21, serves solely for command exchanges between client and server. The second, traditionally on port 20, is dedicated to the actual file transfer. This evolution, seemingly minor, conceals remarkable intelligence. If the transfer crashes, the control connection remains active and error messages continue to flow.

The protocol spread throughout American universities. Students discovered they could submit their assignments to remote servers, developers distributed their software, and system administrators synchronized their data between machines. FTP joined the pillars of the Internet alongside email and remote access.

But FTP conceals subtleties. The protocol offers two operating modes that reflect the evolution of network architectures. Active mode, implemented first, has the server initiate the data connection to the client.

This approach worked perfectly in the early days of the Internet, when machines were directly connected to the network. The arrival of firewalls and network address translation changed everything. How can a server establish a connection to a client hidden behind a NAT? Passive mode reverses the logic: it's the client that establishes both connections. This evolution perfectly illustrates how protocols adapt to emerging technical constraints.

The distinction between ASCII mode and binary mode betrays the legacy of early computer systems. UNIX terminated its lines with one character, Windows with two, and Mac with yet another. FTP's ASCII mode automatically converts these markers according to the destination system. Binary mode, on the other hand, copies the file bit by bit without any transformation. Though seemingly archaic today, this differentiation testifies to the diversity of systems that FTP had to connect.

Anonymous authentication constitutes one of FTP's most democratic innovations. Rather than forcing each user to create an account, the protocol accepts the username "anonymous" with an email address as a password. This functionality revolutionized the distribution of free software and public documents. At the same time, FTP supports traditional authentication by username and password for restricted access.

Security gradually became a critical issue. FTP, in its original version, transmits everything in clear text: credentials, commands, and data. An eavesdropper placed on the network could capture passwords or the contents of exchanged files. This weakness stimulated the development of secure extensions. FTPS encapsulates FTP in SSL or TLS, either implicitly on port 990 or explicitly by negotiating encryption on a standard connection. SFTP, despite its name, is actually a distinct protocol based on SSH that offers a modern and secure alternative.

The integration of FTP into operating systems reveals its status as a fundamental protocol. UNIX natively integrates command-line clients and servers. Windows incorporates FTP support into its file explorer, democratizing access to the protocol. Software like FileZilla enriches the user experience with intuitive graphical interfaces while scrupulously respecting standards.

Resume capability after interruption transforms the user experience: no longer necessary to completely restart the download of a multi-gigabyte file after a network interruption. IPv6 support and extensions for large files modernize the protocol. Each evolution preserves backward compatibility, a cardinal principle of Internet protocol design.

FTP's security flaws raised awareness in the technical community about the importance of encryption. The complications of active mode with fire-

walls fueled reflection on network protocol architecture. Managing two simultaneous connections enriched understanding of the challenges in securing communications.

The emergence of *cloud* storage and modern web protocols gradually eroded FTP's market share. HTTPS became the standard for many consumer uses thanks to its simplicity and integrated security. Yet FTP maintains its relevance in certain professional contexts, particularly for web server administration and website maintenance.

This longevity is explained by certain qualities, including the protocol's simplicity, its proven robustness, and its ability to evolve without breaking compatibility. The separation between control channel and data channel, a 1970s innovation, proved invaluable for error management and transfer resumption. Standardized numeric response codes facilitate automation and programmatic management. These evolutions prove that a fifty-year-old protocol can adapt to modern requirements. FTP perfectly illustrates that in computing, the value of a technology is not measured solely by its novelty, but by its ability to evolve and the strength of its foundations.

Intel 4004

On November 15, 1971, a small advertisement appeared in *Electronic News*. Intel Corporation presented the 4004, which it described as a "microprogrammable computer on a chip." Behind this wording lay an invention that would change the world: the first commercial microprocessor in history.

This innovation was born from an unlikely encounter between an American startup specializing in memory and a Japanese calculator company. In 1969, Busicom was looking for a partner to develop integrated circuits for its future calculators. Intel, founded the previous year by Bob Noyce and Gordon Moore, accepted this contract to balance its finances while waiting for its main business to take off.

Busicom's initial specifications called for twelve different integrated circuits, each dedicated to a specific function. Three Japanese engineers, including Masatoshi Shima, arrived at Intel in June 1969 to supervise the project. Ted Hoff, the company's twelfth employee and head of applications, had to liaise with this team from Asia. He examined the specifications and frowned. The proposed system required too many connections between chips, which would make the final product prohibitively expensive. He proposed abandoning the idea of specialized circuits to create a

universal programmable processor, accompanied by a few memory chips. The architecture would be simpler, cheaper, and above all infinitely more flexible.

Stanley Mazor joined the venture in September 1969. His experience at Fairchild in computer design brought the necessary technical credibility to the project. Together, Hoff and Mazor refined their vision and managed to convince Busicom's skeptics. They developed a 20-byte interpreter to execute macro-instructions, thus demonstrating the viability of their approach.

In October 1969, at an official meeting, Busicom gave Intel's new architecture the green light. Shima extended his stay until December to write the calculator's basic programs. The signed contract granted the Japanese company exclusivity over this system. All that remained was to transform this great idea into physical reality. However, neither Hoff nor Mazor mastered integrated circuit design. Intel recruited Federico Faggin in April 1970. This Italian had developed the silicon-gate MOS technology at Fairchild, essential for achieving the desired performance and integration density.

The team finalized the MCS-4 system architecture. Four circuits made up the whole: the 4004 processor, the 4001 ROM memory, the 4002 RAM, and the 4003 input-output register. The heart of the system, the 4004, contained approximately 2,300 transistors on a 3.0 × 4.0 mm chip. It executed 8,000 instructions per second and addressed up to 4 KB of ROM and 640 bytes of RAM.

Manufacturing of the first prototypes began in October 1970. The peripheral chips worked fairly quickly. The 4004 proved more resistant but required several revisions before reaching maturity in March 1971. A month later, Busicom validated the entire system in its calculator. Intel pulled off a masterstroke in negotiations. In May 1971, the company recovered the marketing rights for the MCS-4 for all applications other than desktop calculators, in exchange for a price reduction for Busicom. Ed Gelbach, the new vice president of marketing, orchestrated the official launch of the 4004 in November 1971.

The success exceeded expectations. Intel followed up with the 8008, the first 8-bit microprocessor, announced in April 1972. These two processors opened two distinct paths that still structure the semiconductor industry today: on one side, integrated microcontrollers for embedded systems, heirs to the 4004; on the other, processors for programmable computers, descendants of the 8008.

The 4004's legacy can be measured in staggering figures. In 1995, twenty-four years after its birth, 4-bit microcontrollers represented half

of the units produced worldwide. The 8008 architecture evolved into the 8080, then into the x86 processors that equip the majority of today's personal computers.

In 2024, nearly 70% of semiconductors manufactured globally are microprocessors, microcontrollers, or associated components. This market is worth over 100 billion dollars. From automobiles to phones, household appliances to connected devices, these chips descended from the 4004's DNA populate our daily lives.

This technology has changed our relationship with machines and transformed the way we live. It perfectly illustrates how an elegant technical solution to a specific industrial problem sometimes generates unpredictable societal upheavals. The 4004 was initially just a component for Japanese calculators. It became the foundation of our digital civilization.

Intel 8008

In April 1972, Intel launched the Intel 8008, the first 8-bit microprocessor in history. This innovation did not come out of nowhere, however. Three years earlier, Computer Terminal Corporation (CTC) was looking for a solution to modernize its Datapoint 2200 terminal. The company contacted Intel, a company founded just one year earlier by Gordon Moore and Robert Noyce, which was limited to memory manufacturing.

CTC had a concrete problem: its terminal used MSI (Medium Scale Integration) components scattered across multiple boards. The idea of concentrating all this logic on a single chip appealed to the company, which named this project with the codename 1201. Stan Mazor and Ted Hoff took charge of the functional specifications before handing over to Federico Faggin and Hal Feeney for the technical implementation.

The architecture faithfully replicated that of the existing CTC processor. An 8-bit accumulator worked with six 8-bit general-purpose registers (B, C, D, E, H, L). The processor handled 14-bit addressing, providing access to 16 KB of memory. Its instruction set remained basic but covered common arithmetic and logical operations, with rudimentary interrupt handling. Technically, approximately 3,500 PMOS transistors etched at 10 microns were packed into an 18-pin package. The clock frequency reached 200 kHz, enabling execution of 60,000 instructions per second.

This 18-pin constraint nearly compromised the project. Intel wanted to control its production costs and refused to increase the number of con-

nections. The design team found itself at a technical impasse, wondering how to pass all the necessary information with so few available connections. The solution came from multiplexing the address and data buses, a technique that complicated usage but made manufacturing economically viable.

The first prototypes came off the production lines in late 1971. Unfortunately, they revealed operational defects. The engineers had to correct the etching masks and restart production. March 1972 finally saw the birth of the first fully functional chips, but CTC had changed its mind in the meantime. The company now considered the 8008 too slow and preferred to develop its own solution in TTL logic.

Intel found itself stuck with a product on its hands but decided to commercialize it. The gamble proved successful. The 8008 found its place in computer terminals, industrial controllers, and scientific instruments. Its launch price—$120 for the standard version, $180 for the fast variant—remained affordable for many designers.

The creation of this processor disrupted the habits of draftsmen who drew the masks by hand on enormous sheets exceeding 1.5 meters in width, at a scale 500 times larger than the actual circuit size. Logic simulations were laboriously performed on the few available mainframe computers in time-sharing mode. The team invented techniques such as bootstrap loads to improve the circuit's electrical performance.

Intel did not leave its customers to fend for themselves with this unprecedented technology. The company published detailed programming manuals, complete reference documentation, and offered evaluation boards like the SIM-8. These resources enabled engineers to master the microprocessor's specificities.

Kodak integrated the 8008 into its photocopiers, Toledo Scale into its industrial scales. McDonald's used it for its first point-of-sale terminals. Some applications continued to use this processor more than ten years after its release. The lack of binary compatibility with subsequent generations like the 8080 explains this loyalty. Rather than reprogramming everything, many preferred to keep their existing developments.

This small component demonstrated that a single chip could house a complete calculator, mass-produced at reasonable cost. Electronic designers began to see things differently. The 8008's architecture, with its general-purpose registers and bus multiplexing, had a lasting influence on subsequent generations of microprocessors.

Sales remained modest, a few hundred thousand units over its entire commercial career. These figures paled compared to those of the 4004 or the future 8080, but Intel's management hesitated. It feared becoming a

competitor to its own computer manufacturer customers by entering this market too boldly. Texas Instruments attempted to create a rival to the 8008 without succeeding. Ironically, this company filed in 1971 the first patent covering the microprocessor concept, while Intel had only protected certain specific technical aspects of its implementation. A legal battle that illustrated the radical novelty of these technologies.

The lessons from the 8008 directly fed into the design of the 8080. This time, the engineers adopted a 40-pin package that finally gave them the necessary flexibility. They kept the general architecture but improved it considerably in every aspect.

The entire x86 family that dominates the personal computer market retains traces of the 8008. Architectural choices such as byte order (little-endian) or register organization survived fifty years and are found in the most modern processors. As the first 8-bit microprocessor ever commercialized, the 8008 established technical and commercial directions that shaped the semiconductor industry for subsequent decades. Its creation required innovations in methodology and tooling. Despite its technical limitations, it revealed the tremendous potential of microprocessors.

C

At Bell Labs in the early 1970s, the mood was far from optimistic. The company had just walked away from Multics, that monumental project undertaken with MIT and General Electric that had consumed considerable budgets without delivering on its promises. Disappointed, Ken Thompson decided to take matters into his own hands to create his own computing environment, according to his personal vision.

He salvaged an old DEC PDP-7, a machine already outdated with its meager 8 KB of memory. On it, he built the first versions of UNIX, initially in assembly language. But programming directly in assembly was a daily nightmare. Thompson had tested Fortran, without success. He created B, a stripped-down version of BCPL that fit within the PDP-7's memory constraints.

Dennis Ritchie joined the team and found that B had its limitations. Between 1969 and 1973, he undertook to evolve it into something more ambitious. 1972 was the decisive year. The move to the PDP-11 revealed B's weaknesses: its character handling was problematic, and its pointers generated absurd conversions between word indices and addresses. B com-

pletely ignored floating-point numbers, even though the PDP-11 would soon be able to handle them.

Ritchie first introduced types with "New B", adding `int` and `char` with their arrays and pointers. But it was with C that the idea truly took shape. The type system became sophisticated, allowing the composition of complex structures from basic types. Ritchie had a brilliant insight for declaration syntax: make it mirror usage. If you write `int *p`, it's because `*p` gives you an `int`.

Array handling sparked debates. In C, an array name automatically converts to a pointer to its first element in expressions. This approach elegantly solved the string problem while preserving existing B code. The preprocessor arrived in 1972-1973, modest at first with just file inclusion and a few macros.

By 1973, C was mature. Ritchie and Thompson tackled a complete rewrite of the UNIX kernel. It was a bold gamble: no one had yet proven that a high-level language could produce code fast enough for an operating system. The gamble paid off. Between 1973 and 1980, C continued to grow: `unsigned` and `long` types, `union`, enumerations, structures becoming first-class citizens.

Brian Kernighan and Dennis Ritchie published *The C Programming Language* in 1978. This small white book, which everyone called "K&R", became the language's bible. Its remarkable clarity contributed greatly to popularizing C. But the language lacked an official standard, and each compiler brought its own small variations.

Standardization became necessary in the early 1980s. C was overflowing from UNIX and colonizing other systems. The ANSI X3J11 committee formed in 1983 at the initiative of M. D. McIlroy. By the end of 1989, the ANSI C standard was ready, adopted as international standard ISO/IEC 9899-1990. The committee remained cautious, introducing only one major change: integrating argument types into function signatures. This idea, borrowed from C++, strengthened type checking without breaking existing code.

C's expansion was meteoric. In the 1980s, compilers appeared on virtually every architecture and system. Commercial software developers for microcomputers adopted it massively. This success stemmed from the language's balance: simple enough to master, close enough to the machine to produce efficient code, abstract enough for portability.

C spawned a prolific lineage. Bjarne Stroustrup created C++, and others gave birth to Objective C and Concurrent C. Subsequently, the modern languages that borrowed its syntax and concepts are countless. Pointers, array handling, and type declaration syntax remain lasting contributions,

even if they spark debate.

The language has its flaws. Modularizing large systems remains problematic with its two-level naming structure. Manual dynamic memory management regularly traps programmers. These weaknesses have not, however, prevented the development of complete operating systems in C.

The history of C shows how a tool born from immediate practical necessities becomes, through judicious design choices, a standard that spans eras. Its longevity in a constantly shifting technological landscape testifies to the soundness of its creators' insights. More than fifty years after its birth, C still ranks among the most widely used languages in systems and embedded programming. Go, of which Ken Thompson is co-creator, is trying to catch up with it.

Prolog

In 1970, Alain Colmerauer was not trying to invent a new programming language. This assistant professor of computer science at the University of Montreal was leading the TAUM project (Traduction Automatique de l'Université de Montréal) and simply wanted to advance his research on computational language processing. Two young 25-year-old instructors from the newly established Faculty of Science at Luminy, Robert Pasero and Philippe Roussel, joined him for two months. They discovered Q-systems, a language that Colmerauer had developed for his translation project.

Jean Trudel, a Canadian doctoral student under Colmerauer's supervision, was working on automated theorem proving. He built upon Alan Robinson's seminal paper on the resolution principle, published in 1965. He had the advantage of having taken a logic course with Martin Davis in New York and already mastered unification using a modern approach where all computations reduced to pointer manipulation.

In early 1971, the team reconvened in Marseille. Alain Colmerauer obtained a position as associate professor of computer science, and Jean Trudel accompanied him thanks to a two-year scholarship from Hydro-Québec. Their goal was to perform deductions from French texts. The work was divided such that Jean Trudel and Philippe Roussel tackled deductions, while Robert Pasero and Alain Colmerauer focused on natural language. Their working environment was luxurious by French standards of the time. Their department's IBM 360-44 offered nearly 900 KB of

internal memory, without virtual memory. Jean Trudel developed an interactive monitor, and they monopolized the machine at night.

In June 1971, the team invited Robert Kowalski, one of the inventors of *SL-resolution*. This meeting changed everything. For the first time, they engaged with a specialist capable of explaining the resolution principle, its variants, and its refinements. Kowalski discovered researchers passionate about his work and determined to apply it to natural language processing.

In February 1972, the IRIA (Institut de Recherche d'Informatique et d'Automatique) granted them 122,000 FF for 18 months. This funding allowed them to invest in a teletype terminal, a dedicated 300-baud connection to the IBM 360-67 in Grenoble, and the recruitment of Henry Kanoui.

The fall of 1972 saw the birth of the first Prolog system. Philippe Roussel developed it in Algol-W while Alain Colmerauer and Robert Pasero created in parallel the French-language human-machine communication system. The interaction between Philippe, who coded Prolog, and Alain and Robert, who programmed in a language under development, became permanent. It was Philippe's wife who suggested the definitive name: Prolog, an abbreviation of "PROgrammation en LOGique" (Programming in Logic).

The first major program written in Prolog was the human-machine communication system. Its 610 clauses were distributed as follows: Alain wrote 334 for analysis, Robert Pasero 162 for the deductive part, and Henry Kanoui developed French morphology in 104 clauses. This morphology handled the relationships between singular and plural for all common nouns and all verbs, both regular and irregular, in the third person singular present tense.

Official recognition came in 1973. The CNRS established an "associated research team" named "Human-machine dialogue in natural language" and allocated 39,000 FF for the first year. Users of the preliminary version of Prolog had programmed enough for their experience to inform a second version, oriented toward a true programming language rather than merely an automated deductive system.

Between June and December 1973, three DEA students—Gérard Battani, Henry Méloni, and René Bazzoli—wrote the interpreter in FORTRAN and its supervisor in Prolog. This version introduced all the basic features of current Prologs. It was also during this period that the occur check disappeared, deemed too costly.

Dissemination truly began in 1974-1975. David Warren stayed in Marseille from January to March 1974 and used Prolog to write Warplan, his plan generation system. The interactive version of Prolog running in

Grenoble via teletype generated numerous requests. Gérard Battani and Henry Méloni were overwhelmed with distributions. They shipped Prolog to Budapest, Warsaw, Toronto, Waterloo, and traveled to Edinburgh to help David Warren install it on a PDP 10. Hélène Le Gloan, a former student, installed it at the University of Montreal. Michel Van Caneghem did the same at IRIA in Paris before joining the team. Maurice Bruynooghe took Prolog to Louvain after three months in Marseille.

In 1975, the entire team participated in porting to a 16-bit minicomputer, Télémécanique's T1600. The machine had only 64 KB and required a specific virtual memory management system. Pierre Basso tackled this task and won the competition for the shortest instruction sequence that achieves 32-bit addressing while testing for page faults. Each lab member then translated two pages of FORTRAN into machine language. The reassembled fragments worked. After five years, they finally had their own machine, and their Prolog ran—slowly, but it ran.

The story of Prolog's birth ends in late 1975, but its influence endures. Alan Robinson's January 1965 article, "A machine-oriented logic based on the resolution principle", contained the seeds of the Prolog language. While this article gave rise to a significant body of work on automated theorem proving, the specific contribution of the Marseille team was transforming this theorem prover into a programming language. To achieve this, they did not hesitate to introduce purely computational mechanisms and restrictions that constituted heresies for the existing theoretical model. These modifications, often criticized, ensured Prolog's viability and thus its success.

Robert Kowalski isolated the concept of "Horn clause", legitimizing their main heresy: a linear demonstration strategy with backtracking and unifications only at clause heads. The Marseille team had ideal conditions for this creation. Colmerauer belonged to the first generation of French computer science PhDs, and his specialty was language theory. He had gained valuable experience creating Q-systems within the University of Montreal's machine translation project. The meeting with Philippe Roussel and the particular circumstances in Marseille did the rest. The team benefited from freedom of action in a newly created scientific center and, without external pressures, could devote itself fully to its project.

Smalltalk

In 1969, at the heart of Xerox PARC in Palo Alto, Alan Kay began dreaming of a different kind of computer, amid the excitement of the first interactive machines, emerging graphical displays, and the mice that Douglas Engelbart had just invented. Kay envisioned a "personal medium" capable of amplifying human intelligence. From this radical vision would emerge Smalltalk, a language that would revolutionize our approach to programming.

The idea grew from three major influences. Simula had introduced objects, LISP mastered symbolic manipulation, and Sutherland's Sketchpad demonstrated what graphical interaction could achieve. Alan Kay mixed these ingredients with his characteristic boldness and set out to define "the most powerful programming language in the world" in a single page of code. He succeeded with Smalltalk-71, a first version created in just a few days that demonstrated an entire system could be based on message passing between objects. Gone were the languages where objects coexisted with primitive types in an awkward hierarchy. Here, everything is an object: numbers as well as control structures, windows as well as characters. This conceptual radicalism was no intellectual whim. It forged a system of rare coherence, where every element obeyed the same fundamental rules.

The concept matured through successive versions. Between 1972 and 1976, the PARC team refined this world of communicating objects. Dan Ingalls made a decisive contribution with the invention of overlapping windows. This graphical interface would later become an industry standard, particularly when Apple drew inspiration from it for the Macintosh.

With Smalltalk-76, Ingalls consolidated the gains from previous experiments by proposing a complete system: just-in-time compilation, automatic memory management, integrated graphical interface. The development environment allowed code modification while the program was running, something unprecedented. This flexibility transformed programming into continuous exploration.

But it was with Smalltalk-80, released publicly in 1980, that the system reached maturity. Ten years of research culminated in this version that standardized the language and its environment. The fundamental concepts crystallized: everything is an object, objects communicate through messages, classes are themselves objects, inheritance enables code reuse. These principles would lastingly influence the subsequent evolution of computing.

Smalltalk's strength lies in its conceptual purity. Where other languages like C++ or Java awkwardly graft object-oriented features onto a

procedural foundation, Smalltalk applies its paradigm throughout. This theoretical coherence is accompanied by remarkable practical innovations: the integrated development environment, the garbage collector that frees the programmer from memory management, the modern interface with its windows, menus, and mouse.

Smalltalk's expansion coincided with the emergence of personal computers powerful enough to host it. Steve Jobs discovered the system during his visit to PARC in 1979. The impact was such that it directly shaped the development of the Macintosh interface. The influence then spread throughout the industry: Objective-C, Ruby, Python, Java each borrowed elements from Smalltalk's object model.

Alan Kay had initially designed Smalltalk for teaching programming to children. Objects intuitively model real-world entities, the graphical interface facilitates experimentation. This educational dimension survives in projects like Squeak and Etoys, which perpetuate the idea of computing accessible to all.

Smalltalk's development illustrates the importance of institutional context in innovation. Xerox PARC offered its researchers exceptional freedom, without immediate commercial pressure. This autonomy made the emergence of an innovative system possible. Paradoxically, Xerox never managed to capitalize commercially on this major innovation.

The 1990s and 2000s saw numerous implementations flourish: VisualWorks for professionals, Squeak for education, GNU Smalltalk for the free software world. This diversity testifies to the persistent vitality of the model and its adaptability.

Smalltalk is no longer a dominant language today, but the principles it introduced—objects, messages, inheritance, interactive environment—have become industry standards. The evolution toward touch interfaces and block-based programming extends its vision of intuitive and accessible computing. This story reveals the importance of long-term vision in technological progress. Smalltalk combines an elegant theory of object-oriented programming with practical innovations in user interface. Its journey shows that ideas need time to find their place, and that the true impact of an innovation is often measured by subsequent generations.

IBM VM/370

In 1972, IBM launched VM/370, standing for Virtual Machine Facility/370. Behind this technical name lay the ability to run multiple operating systems in parallel on a single physical machine. The idea was not entirely new, as it extended work done on CP-67/CMS for the System/360 Model 67, but its systematic implementation scaled to enterprise computing.

It took boldness to envision this concept at the time. VM/370 created "virtual machines", complete but simulated computing environments, each believing it had its own computer. This illusion rested on four components: the CP control program that orchestrated physical resources, the CMS conversational system for interacting with users, the RSCS subsystem that managed file exchanges, and IPCS which handled incidents. The architecture masked complexity in favor of efficiency, with each virtual machine operating in its bubble, completely unaware of its neighbors. This isolation enabled different IBM systems—DOS, OS/VS1, OS/VS2—to coexist without interfering with one another. The technical feat lay in this ability to partition a single computer into multiple sealed environments.

On the memory side, VM/370 introduced dynamic address translation that converted virtual addresses into real physical addresses. The system divided memory into 64 KB segments and 4 KB pages, creating multiple levels of abstraction. This intellectual gymnastics allowed running virtual machines whose combined memory exceeded available physical memory. The innovation extended to peripherals with "spooling", this technique that managed input-output operations for all virtual machines. A disk or printer could serve multiple environments without creating conflicts. VM/370 juggled resources like a magician with his props.

Each user received their identifier and password, and resource access followed strict security rules. Isolation between virtual machines constituted a natural barrier sufficient to ensure that a crash in one would not affect the others. This robustness immediately appealed to developers who could test their programs without fear of bringing down the production system.

The impact on the industry exceeded IBM's expectations. Companies discovered they could exploit their expensive computers much more intensively. Instead of leaving resources idle, they shared them among various applications. Gone were the endless waits for computer scientists to test their programs—everyone had their own virtual environment.

VM/370 gradually adapted to new IBM hardware, from the modest Model 135 to the powerful Model 168. The system integrated support for

3330, 3340, and 3350 disks, gained network functionalities, and improved in performance. This constant evolution testified to the vitality of the concept.

The fundamental principles of VM/370—isolation, resource sharing, dynamic memory management—still shape modern virtualization technologies. VMware, Xen, KVM, and other contemporary solutions draw from this conceptual heritage forged in the 1970s. More than a technical feat, VM/370 changed the way enterprise computing was conceived. The system proved it was possible to partition computing resources without sacrificing performance or security. Future architectures would therefore be more flexible and economical.

The system's influence on computer science education was considerable. Schools and universities adopted VM/370 to teach operating systems. Students could experiment on their own virtual machines without monopolizing an entire computer. This democratization of access to computing resources trained a generation of computer scientists familiar with virtualization. VM/370 illustrates how a technical innovation can transform an entire sector. By demonstrating the viability of virtualization, IBM was redefining computer usage. This transformation continues to resonate in modern data centers, where virtualization now constitutes the norm rather than the exception.

Ethernet

In 1972, at Xerox's laboratories in Palo Alto, Bob Metcalfe and David Boggs worked on connecting computers to laser printers within a building. Their solution, which they named Ethernet, went far beyond this initial requirement by achieving 2.94 megabits per second.

The basic idea was brilliantly simple. All computers share the same cable, like people talking around a table. Before speaking, each listens to ensure nobody else is talking. If two start at the same time, they stop, wait a random moment, then try again. This method, a technique called CSMA/CD, works remarkably well despite its apparent chaos.

The first Ethernet used a 75-ohm coaxial cable. Machines connected through special boxes that detected collisions and electrically isolated each station. It was possible to connect or disconnect a machine without paralyzing the entire network, which represented a considerable advantage over centralized systems. Four years later, Metcalfe and Boggs described

their invention in a paper in the "Communications of the ACM". Ethernet left the laboratory to reach the industrial world. The first installations demonstrated the ability to connect hundreds of machines over more than a kilometer.

Xerox then understood the commercial potential of its creation. In 1980, the company partnered with Intel and Digital Equipment Corporation to transform Ethernet into an open standard. This strategic decision proved successful: freeing the technology from its creator accelerated its adoption. The new specification pushed throughput to 10 megabits per second and adopted 50-ohm coaxial cable. Machine addresses expanded from 8 to 48 bits, ensuring their worldwide uniqueness.

The IEEE took over with its 802.3 committee in the early 1980s. This standardization inscribed Ethernet into the OSI model and made it completely vendor-independent. The CSMA/CD protocol became the MAC sublayer of the data link layer, while physical aspects fell under 802.3 standards.

In 1990, the 802.3i standard abandoned the coaxial bus in favor of twisted pairs and star topology. Gone were the cables snaking through false ceilings; each machine now had its own link to a central hub, making installation and maintenance a breeze. Fast Ethernet arrived in 1995 with its 100 megabits per second, followed three years later by Gigabit Ethernet.

Metcalfe and Boggs made technical choices that endured. Ethernet handles neither priorities nor quality of service, leaving these questions to higher-level protocols. This apparent limitation proved to be a strength because the simplicity of the basic mechanism allowed for all subsequent adaptations. The system's core remained remarkably stable. The controller monitors the channel and applies CSMA/CD rules. The transceiver interfaces with the physical medium. Frames carry source and destination addresses, a type field to identify the encapsulated protocol, data, and a CRC checksum. This streamlined architecture crossed decades without major modifications.

During the 1990s, switches radically transformed Ethernet's nature. Instead of a shared channel where everyone listens, the network evolved into a set of point-to-point links. Different conversations could occur simultaneously. Full-duplex mode gradually replaced CSMA/CD, which became less useful with switching.

Why did Ethernet triumph where others failed? First through its low cost, a direct consequence of its simplicity. Then through its ability to evolve: throughput increases, media change, but fundamental principles remain. Finally through its openness, with no manufacturer able to mo-

nopolize the network.

The designers validated their choices through experimentation. The experimental Ethernet provided precise measurements showing medium utilization approaching 98% under certain conditions. The binary exponential backoff (BEB) mechanism, an algorithm to limit network load when a collision occurs between two messages, ensures stability under any load. These concrete data guided the development of the industrial standard.

Today, Ethernet carries Voice over IP, connects storage arrays, and powers data centers. Throughput reaches 400 gigabits per second over fiber optics. The technology born to connect a few computers to printers now structures the global internet. The most enduring solutions are not always the most sophisticated.

TCP/IP

In the late 1960s, in the particular context of the Cold War, the United States sought to establish a military communication network capable of withstanding a Soviet nuclear attack. The U.S. Department of Defense, through its DARPA (Defense Advanced Research Projects Agency), launched the ARPAnet project in 1969, which aimed to interconnect the country's main computing centers. Military constraints shaped the network's design from the outset. There could be no Achilles' heel, no central point whose destruction would compromise the entire system. Messages had to be able to take alternative routes dynamically if certain portions failed. More complex still, the network had to connect computers running different operating systems.

ARPAnet started with the NCP protocol, but its shortcomings became apparent in short order. In 1974, two researchers changed the game: Vinton Cerf and Robert Kahn published "A Protocol for Packet Network Interconnection", a foundational article that introduced TCP. This publication outlined what would become TCP/IP, standardizing communication between machines. The genius of TCP/IP lies in its layered structure. IP handles routing data packets from one point to another on the network, while TCP ensures transmission control and reassembles complete messages upon arrival. This division of labor makes the system more robust and easier to evolve, an engineering principle that would prove its worth.

In 1983, TCP/IP officially replaced NCP on ARPAnet. Although this point is widely debated and debatable, the concept of the Internet was

truly born at this moment, in the literal sense of the term. TCP/IP's success rests on three decisive advantages. It works on any hardware, computer, or physical network. It maintains communication when part of the network fails through dynamic routing. It breaks messages into packets that can take different paths before being reassembled. First-generation IP addressing uses 32-bit addresses that uniquely identify each machine. The system classifies these addresses into categories A, B, and C to simplify management. But the explosive growth of the network quickly revealed the limitations of this approach. Computer scientists then developed workarounds such as subnetting, while awaiting IPv6 and its 128-bit addresses.

The TCP/IP architecture is organized in four layers: Network Access for physical connection, Internet (IP) for routing, Transport (TCP/UDP) for transmission, and Application for user programs. This hierarchy facilitates adding new functionality without disrupting the existing system, a tremendous advantage for the system's evolution. Several auxiliary protocols complement TCP/IP. ARP translates logical IP addresses into physical network card addresses. ICMP carries control and error messages. These mechanisms contribute to the overall system's reliability and efficiency.

Berkeley UNIX contributed to TCP/IP's dissemination during the 80s. Native integration of the protocol into BSD facilitated its adoption by universities and research centers that extensively used this operating system. Technical documentation was public and accessible to all in the form of RFCs, documents open to community contributions. This decentralized approach stimulated innovation and continuous improvement, an approach that would durably influence technological development.

The explosion of the World Wide Web in the 1990s spectacularly validated the choice of TCP/IP. The protocol absorbed without flinching an exponential growth in the number of users and the arrival of resource-intensive applications. Its flexibility allowed the addition of advanced features such as *multicast* for broadcasting to multiple recipients, and quality of service to prioritize certain flows. The history of TCP/IP continues to be written. The anticipated exhaustion of IPv4 addresses and new security and performance needs drove the development of IPv6 in the 1990s. This version offers a virtually infinite address space—3.4×10^{38} addresses—and substantial improvements. Migration to IPv6 is progressing very slowly, complicated by the need to maintain compatibility with existing systems.

From military project to universal standard, TCP/IP has spread to all contemporary domains. Computers of course, but also smartphones, connected devices, industrial equipment—the list grows longer every day.

This ubiquity stems from its remarkable ability to adapt to technological developments without breaking with its legacy. The standardization of TCP/IP made possible the emergence of a global network where billions of machines communicate, regardless of their origin or geographic location.

Telnet

In the computing world of the early 1970s, ARPANET had just been born, but its promises of universal interconnection were running up against technical reality. Each manufacturer developed its own protocols and standards, creating so many isolated technological islands. A terminal operating with the EBCDIC character set simply could not communicate with a PDP-10 computer configured for ASCII. Machines from different manufacturers needed to communicate with each other.

The Telnet protocol emerged to address this problem. Its designers understood that they needed to invent a common computer language that would allow any terminal to connect to any remote computer. This insight gave birth to the Network Virtual Terminal (NVT), a concept of remarkable elegance in its simplicity. It functions as a universal translator. Rather than forcing each machine to understand all the possible dialects of its peers, the system imposes a single standardized format. Each computer needs to master only one translation: that between its native language and this virtual *lingua franca*. An EBCDIC terminal translates to the NVT, which then retranslates to the ASCII expected by the destination machine. The efficiency of this approach immediately struck ARPANET engineers.

But Telnet's history contains a delightful episode. In 1972, Bernard Cosell and David Walden, both BBN employees working on the Terminal Interface Message Processor, found themselves facing a recurring puzzle. Telnet specifications were constantly evolving, and each modification required simultaneously updating all existing instances across the network. A monumental task that considerably slowed development. It was on a plane en route to UCLA that Cosell had his epiphany. Why not let the two ends of a connection negotiate between themselves the functionalities they wished to use? This idea, which he developed with Walden during the flight, revolutionized the approach to network protocols. Back on the ground, they designed a mechanism based on four terse commands: `DO`, `DON'T`, `WILL`, and `WON'T`. These four magic words transformed Telnet into a living protocol, capable of evolving without breaking what already existed.

When a machine proposed a new functionality via `WILL`, its counterpart could accept it with `DO` or politely decline with `DON'T`. If it did not understand the proposal, a simple refusal sufficed, and communication resumed on the common ground of the NVT. This ongoing negotiation allowed the protocol to gradually enrich itself, like a language that naturally integrates new terms without losing its coherence. Ray Tomlinson, already famous for inventing email, was then working on the Telnet implementation for the TENEX system. As soon as he discovered the mechanism imagined by Cosell and Walden, he immediately grasped its significance. David Walden formalized the idea in RFC 435, published in January 1973, laying the groundwork for what would become a lasting standard.

Final standardization came ten years later, with RFC 854 in 1983. Telnet gained its modern form: bidirectional TCP connection on port 23, 7-bit ASCII encoding, and sophisticated data flow management. Engineers added an ingenious synchronization mechanism, designed for emergency situations where a user must regain control of a runaway remote program. The "Synch" signal uses TCP's urgent capabilities to bypass normal flow control and guarantee immediate transmission of critical commands. This elegant architecture made Telnet far more than a simple terminal emulator. The protocol became the backbone of remote access to computer systems. In the 1980s and 1990s, it fueled the explosion of the first online services and BBSes, those electronic bulletin boards that preceded today's forums. Its simplicity appealed to developers and users alike.

Designed in the 1970s, when computing existed in a world of relative trust, Telnet encrypted nothing. Passwords, sensitive data—everything traveled in clear text across networks. This weakness, trivial at first, became prohibitive with the democratization of the Internet and the emergence of security concerns. The arrival of graphical interfaces dealt an additional blow to the venerable protocol. Users gradually abandoned text-mode terminals for colored windows and clickable icons. SSH, which appeared in the 1990s, inherited Telnet's strengths while adding the encryption it so sorely lacked.

Though banished by security experts, Telnet has not disappeared. In server rooms, it remains the diagnostic tool par excellence. Network administrators have long used it—perhaps still do—to test connectivity on a given port or configure recalcitrant old equipment. This persistence testifies to the robustness of its original design.

The virtual terminal concept spread to countless subsequent protocols. The option negotiation mechanism, born on that plane to UCLA, inspired engineers facing the challenge of extensibility. How do you evolve a protocol without breaking what exists? Cosell and Walden had found

a disarmingly simple answer that continues to serve as a model. Faced with the growing complexity of systems, the temptation was strong to develop sophisticated solutions with abundant features. Telnet chose to favor interoperability and flexibility over sophistication.

The SQL Language

In 1969, an IBM researcher named Edgar Frank Codd published a seemingly innocuous paper: "Derivability, Redundancy, and Consistency of Relations Stored in Large Data Banks". No one suspected that this text would revolutionize the way we store and manipulate computer data. The following year, Codd followed up with a more ambitious paper in Communications of the ACM: "A Relational Model of Data for Large Shared Data Banks". This time, the industry took notice.

Codd's relational model broke with the hierarchical and network approaches dominant at the time. He proposed organizing data in simple tables, linked to each other through mathematical relationships. The idea seems almost trivial today, but it revolutionized computer science thinking. Academic and industrial communities embraced these theoretical concepts.

At IBM, Donald Chamberlin and his team wasted no time. In 1974, they developed System R, the first truly worthy relational database prototype. The associated query language was named SEQUEL, for Structured English QUEry Language. This choice revealed an ambition: to create a language close to natural English, accessible to non-specialist users. Gone were cryptic commands, replaced by instructions comprehensible to the masses.

The project progressed incrementally. SEQUEL-XRM emerged between 1974 and 1975, followed by a complete rewrite in 1976-1977. This new version, dubbed SEQUEL/2, introduced essential functionalities like multi-table queries, multi-user support, advanced data management. But an unexpected obstacle arose: the British company Hawker Siddeley Aircraft Company already owned the SEQUEL trademark. IBM had to rename its language with the acronym SQL.

The story took an unexpected turn in 1979. While IBM was perfecting its prototypes, a small California company named Relational Software commercialized the first relational database management system. In a stroke of marketing genius, this company numbered its product "Version

2", thus avoiding the usual reluctance toward first versions. Relational Software would become Oracle Corporation, beating the IBM giant at its own game.

IBM caught up with SQL/DS in 1981, and DB2 in 1983. Meanwhile, customer tests conducted as early as 1978 had validated the approach: SQL won over users with its ease of use and power, allowing them to describe what they wanted to obtain without specifying how to obtain it. This declarative approach contrasted with traditional procedural languages and democratized database access.

Standardization naturally became necessary. In 1986, the American National Standards Institute certified SQL as an official standard. The International Organization for Standardization followed in 1987. These first standards, approximately 150 pages long, standardized the SQL dialect of IBM DB2. However, they suffered from gaps and imprecision, consequences of the divergent commercial interests of software vendors.

The SQL2 standard of 1992 corrected these flaws. This 500-page document eliminated some of the weaknesses of the previous version and standardized advanced conceptual functionalities, sometimes beyond acceptable technical capabilities. Three levels of compliance emerged: *Entry* for minor modifications, *Intermediate* for at least 50% of the standard, and *Full* for total compliance.

SQL99, or SQL3, marked a new evolution at the end of the 1990s. This 2000-page standard integrated object-relational models, calling interfaces, and advanced integrity management. It replaced the old compliance levels with two categories: Core SQL99 and Enhanced SQL99. The document was now structured into five specialized parts, testifying to the growing complexity of the language.

The 21st century saw the birth of SQL:2003, SQL:2008, and SQL:2011, each version enriching the language with new capabilities. But beyond this technical chronology, SQL owes its success to deeper qualities. Its syntax inspired by natural English makes it accessible to non-specialists. Its declarative nature frees users from procedural constraints. The underlying relational model structures data in a clear and coherent manner.

The concepts of primary and foreign keys, normalization, and referential integrity guarantee the consistency of stored data. This mathematical rigor, combined with flexibility of use, explains SQL's massive adoption by the industry. Oracle, MySQL, PostgreSQL, Microsoft SQL Server, and so many other systems perpetuate this legacy, extending the language's capabilities while preserving compatibility.

System R

During 1974, a team of researchers at IBM's San José laboratory sought to demonstrate that the relational model theorized by Edgar F. Codd four years earlier could work in reality, with real data and real users. This project, named System R, was not merely a technical experiment. It was about building a bridge between the mathematical elegance of the relational model and the harsh constraints of the industrial world.

The team led by W. F. King began with what they called "Phase Zero", an exploratory period that lasted from 1974 to 1975. The first prototype, limited to a single user, already implemented the foundations of what would become SQL. This initial version relied on the XRM relational access manager developed by Raymond Lorie at IBM's Cambridge Scientific Center. The researchers discovered that transforming an elegant theory into functional code raised unexpected questions. How to store data efficiently? How to guarantee transaction integrity? How to ensure queries execute within reasonable timeframes?

The real revolution occurred during "Phase One", between 1976 and 1977. The team designed a two-tier architecture that clearly separated responsibilities. The RSS (Research Storage System) handled everything related to physical storage, locks, and modification logging. Above it, the RDS (Relational Data System) managed permissions and query optimization. This separation was not just engineering elegance: it enabled system maintenance and evolution.

In early 1976, Raymond Lorie proposed the idea of compiling SQL queries instead of interpreting them. Traditionally, database systems analyzed and executed queries on the fly, which limited their performance. System R's approach transformed each query into optimized machine code. For applications that repeated the same operations thousands of times, the gain was spectacular. For one-off queries, the compilation overhead was largely offset by execution speed.

The automatic query optimizer constituted the system's brain. It analyzed each request, consulted statistics stored in the system catalog, and chose the most efficient data access strategy. Its algorithm calculated input-output costs and processor time for different approaches. Indexes, in the form of B-trees, guaranteed fast access to large data volumes.

Hierarchical locking addressed concurrent access issues with variable granularity. A user could lock a single record for a specific modification, or an entire table for a maintenance operation. Three isolation levels

allowed developers to choose between strict consistency and performance. The most stringent level guaranteed that reading a record multiple times within a transaction always yielded identical results.

Recovery after failure relied on an ingenious mechanism of "shadow pages". During a modification, the system created a new version of the page at a different disk location, preserving the old one until transaction validation. Periodic checkpoints purged obsolete versions. While this complicated the physical data organization, this approach considerably simplified crash management.

"Phase Two", from 1978 to 1979, took System R out of the laboratories and installed it with real users. IBM pilot sites and three selected customers tested the system under real conditions. These experiments confirmed the initial intuition: developers gained productivity through the uniformity of the SQL language. Whether writing application code, launching ad-hoc queries, or defining data structures, they used the same syntax.

Testing also revealed the system's limitations. For simple transactions manipulating little data, using hash tables or direct pointers would have yielded better performance. The shadow page mechanism, despite its advantages for recovery, prevented efficient grouping of related data on disk.

An unexpected phenomenon emerged during load testing. The team called it a "convoy": when a process holding a highly demanded lock was put into long wait by the scheduler, all other processes became blocked in a chain behind it. The solution required rethinking the lock release protocol to avoid these bottlenecks.

System R transformed the industry permanently. Its architectural innovations—query compilation, automatic optimization based on statistics, hierarchical locking—were adopted by practically all relational systems that followed. In 1981, IBM commercialized SQL/DS, its first relational database product, followed two years later by DB2 for mainframes. Oracle, Ingres, and other vendors developed their own systems, democratizing this technology.

The impact went far beyond the technical. System R proved that a computer system could combine ease of use with internal sophistication. Users manipulated intuitive concepts—tables, rows, columns—without worrying about the complex mechanisms that guaranteed consistency and performance. This separation between logical and physical views became a principle of business computing.

The project validated Edgar F. Codd's vision on the importance of abstraction in data management. It also established the importance of

a rigorous experimental approach in technological development. Rather than settling for elegant theoretical demonstrations, the San José team built a complete system and tested it under real conditions.

Faced with the emergence of NoSQL systems that challenge certain aspects of the relational model, System R's architectural principles retain their relevance. Declarative query optimization, transaction management, and the separation of logical and physical levels remain benchmarks.

Intel 8080

In the technological ferment of the early 1970s, Intel was attempting to establish itself in the microprocessor market. Robert Noyce and Gordon Moore's company had just launched the 4004 in 1971, followed by the 8008 the next year, its first 8-bit processor. However, these initial achievements, despite their innovative nature, revealed their limitations in the face of growing market expectations.

The arrival of Italian physicist Federico Faggin in Intel's ranks had provided decisive momentum to these developments. His expertise in logic design and silicon-gate MOS technologies enabled the integration of all circuits onto a single chip, where other manufacturers struggled to master this complexity. Yet by 1972, criticism was mounting against the 8008. Plessey and Nixdorf, two European computer companies, pointed to a primitive interrupt structure, address and data multiplexing that penalized performance, and a sluggish operating speed of 0.5 MHz.

These criticisms resonated all the more strongly as they came from potential customers with real needs. The Sac State 8008 project, designed to manage medical records, illustrated the possibilities of 8-bit processing well, but also revealed the processor's shortcomings for professional applications. Intel understood that a complete overhaul was necessary.

Development of the 8080 began in November 1972 under Masatoshi Shima's direction, supervised by Federico Faggin. Shima brought to the project his experience gained at Ricoh in designing interfaces between peripherals and minicomputers. His working method contrasted with usual practices: he created detailed tables to validate each instruction, applying a systematic approach to logic verification that foreshadowed modern design methods.

Sixteen months elapsed between the first sketches and commercialization in January 1974. The time breakdown reveals the meticulousness of

the process: one month to define the product, eight months to design the chip, and five months to manufacture masks, produce wafers, and debug the whole system. This rigor paid off: the 8080 integrated approximately 5,000 transistors in a coherent and efficient architecture.

The NMOS technology adopted for the 8080 came directly from Intel's developments on 4K dynamic memories. This technical lineage explains the three supply voltages required: +5V, +12V, and -5V, a constraint that would disappear with subsequent generations. The 40-pin package, more than twice the 8008's 18 pins, testified to the project's ambition and finally freed the processor from connectivity limitations.

The internal architecture revolutionized this approach. Shima abandoned the 8008's limited internal stack in favor of a stack pointer in main memory, providing hitherto unknown flexibility. The registers were organized according to a thoughtful hierarchy: three pairs (BC, DE, HL) with differentiated capabilities, HL being the most powerful, BC the most basic. This asymmetry, sometimes criticized by programmers accustomed to uniform architectures, resulted from a deliberate choice to optimize hardware resources.

The instruction set was enriched while preserving assembly-level compatibility with the 8008. A remarkable innovation, the 8080 introduced 16-bit operations: increment, decrement, addition, and subtraction. Shima favored this generic approach over adding a specialized index register, a debated choice that nonetheless offered appreciable programming flexibility.

Initial performance tests revealed the magnitude of the technological leap. Two benchmark programs – a 256-byte memory transfer and 16-digit decimal additions – demonstrated gains of 10 to 20 times compared to the 8008 at identical frequency. Running at 2 MHz, the 8080 widened the gap with its predecessor.

Intel accompanied the launch with a bold commercial strategy. The initial price of $360 per unit might have seemed prohibitive, but the company was betting on its processor's added value. This gamble proved successful: manufacturers accepted this premium to benefit from the enhanced capabilities. The ecosystem developed around the 8080 reinforced this position: specialized circuits such as the 8224 (clock generator) and 8228 (bus controller), Intellec development systems, all contributed to simplifying processor adoption.

The success of MITS's Altair 8800, based on the 8080, confirmed Intel's intuitions. This personal microcomputer, sold as a kit, paved the way for a new generation of machines accessible to hobbyist enthusiasts. Personal computing found its first true technical foundation there.

A design flaw nearly tarnished this success. The main ground line, too narrow, limited use to low-power TTL circuits. Intel responded by offering the corrected 8080A version, which became the market reference. This responsiveness to technical problems illustrated the company's newfound maturity.

The 8080's influence extended far beyond Intel's borders. Federico Faggin, who left to found Zilog, designed the Z80 drawing directly from his work on the 8080. Motorola responded with the 6800, a competing architecture but with similar principles. This emulation stimulated innovation throughout the microprocessor sector.

Beyond its performance, the 8080 established lasting standards: 8-bit data bus separated from 16-bit address bus, 40-pin package, general-purpose register architecture. These characteristics were found in many subsequent processors, testimony to the soundness of Shima and Faggin's technical choices.

The 8080's legacy is also measured against the industry it helped create. By demonstrating the commercial viability of 8-bit microprocessors, it enabled personal computing to come into its own and confirmed Intel's dominant position in this strategic market. Its balanced architecture and performance established benchmarks that guided technological evolution for years.

CP/M

In 1972, Gary Kildall, then a consultant for Intel through his company Microcomputer Applications Associates, developed PL/M, a programming language designed for Intel microprocessors. This language, derived from the XPL language used for writing compilers, was the first high-level language specifically designed for microprocessor programming. The first substantial program written in PL/M was a text editor for the Intel 8008 microprocessor, which would later become CP/M's ED editor (Control Program for Microcomputers).

In 1974, Kildall recognized the need for an operating system to exploit Shugart Associates' new 8-inch floppy disk drives. That year, the dominant storage medium was punched paper tape, which cost approximately 100 times more than floppy disks for equivalent data capacity. Kildall proposed to Intel the development of a complete operating system including an editor, assembler, and loader, for $20,000. Intel declined the offer but provided Kildall with the necessary hardware to continue his development.

CP/M development took place in the workshop behind Kildall's house at 781 Bayview Avenue in Pacific Grove, California. Unable to create alone the complex electronic interface between the microprocessor and the floppy disk drive, he called upon John Torode, a university friend teaching at Berkeley, who designed the necessary controller. In fall 1974, Kildall successfully ran CP/M for the first time, loading the system from paper tape to floppy disk and then booting the computer from it.

One of CP/M's major innovations was its modular design separating hardware-dependent functions into a distinct component called BIOS (Basic Input/Output System). CP/M could thus easily adapt to different hardware configurations by modifying only the BIOS. The rest of the system, including the file manager and command interpreter, remained identical regardless of the platform. This portability proved decisive for CP/M's commercial success.

In 1976, Gary and Dorothy Kildall founded Digital Research Inc. (initially named Intergalactic Digital Research) to commercialize CP/M. Initial sales were conducted from a rented office at 716 Lighthouse Avenue in Pacific Grove. CP/M floppy disks were sold for $70 each, and the couple went to the post office daily to collect checks resulting from advertisements placed in magazines such as Byte and Dr. Dobb's Journal.

A licensing agreement with IMS Associates, Inc. in 1977 significantly increased CP/M's credibility in the industry, making the system a standard adopted by most personal computer manufacturers of the era, including Altair, Amstrad, Kaypro, and Osborne. In 1978, revenue reached $100,000 per month and Digital Research moved into a Victorian house at 801 Lighthouse Avenue. By 1980, the company employed over 20 people and generated $3.5 million in revenue, five times that of Microsoft at the time.

In 1980, Bill Gates, who knew Kildall from discussions about a possible merger between their companies, recommended Digital Research to IBM for the operating system of its PC under development. Negotiations failed over confidentiality and business model issues: IBM wanted to purchase CP/M without royalties, while Digital Research preferred a per-copy-sold model. Microsoft seized this opportunity by purchasing a CP/M clone from Seattle Computer Products, renamed PC-DOS for IBM and MS-DOS for other manufacturers.

Faced with this situation, Kildall threatened IBM with a lawsuit for illegal copying of CP/M. IBM agreed to offer CP/M-86 as an option on the PC, but at $240 versus $40 for PC-DOS. This price difference massively steered the market toward PC-DOS/MS-DOS. Digital Research nevertheless continued to thrive for several years, developing multitasking oper-

ating systems for the IBM PC-XT and introducing graphical interfaces before Apple and Microsoft.

In the mid-1980s, MS-DOS's dominance in the IBM-compatible PC market seriously affected Digital Research's business. Gary Kildall, who never particularly enjoyed managing a large company, sold Digital Research to Novell in 1991. The assets were subsequently resold to Caldera in 1996, which would use DRI's intellectual property in a lawsuit against Microsoft.

CP/M underwent major evolutions. MP/M (Multi-Programming Monitor) added multitasking and multi-user capabilities. CP/NET, introduced in 1980, connected CP/M computers to MP/M servers via various network protocols. A 16-bit version, CP/M-86, was developed for Intel 8086/8088 processors. However, the 8-bit version achieved the greatest success, with approximately 200,000 installations on over 3,000 different hardware configurations.

CP/M established fundamental concepts such as the separation between operating system and hardware through the BIOS, the use of standardized commands, and disk file management. These principles directly influenced MS-DOS's design and, by extension, the evolution of personal operating systems. In 1995, the Software and Information Industry Association awarded Gary Kildall a lifetime achievement award, citing eight major contributions to the microcomputer industry. In 2014, the IEEE installed a commemorative plaque in front of Digital Research's former headquarters, recognizing CP/M as an important milestone in personal computing history.

CP/M's success illustrates the importance of open standards and software compatibility in the computer industry's development. By enabling the development of a hardware-independent software ecosystem, CP/M helped transform microcomputers from specialized tools into general purpose platforms accessible to all. This standardization fostered the emergence of an independent software market, foreshadowing the dominant economic model of the personal computer industry.

Diffie-Hellman

In the 1970s, cryptography existed in a closed world. Military and government agencies held the keys to this secret science, leaving the rest of society to make do with poorly secured communications. The systems

of that era imposed a significant constraint: two people wishing to exchange secret messages first had to meet physically to share a common key. Imagine having to cross the Atlantic to hand-deliver a secret code before being able to send a simple encrypted telegram!

Whitfield Diffie and Martin Hellman, two researchers from Stanford, revolutionized this logic in 1976. Their paper *New Directions in Cryptography*, published in IEEE Transactions on Information Theory, proposed the unthinkable: creating a shared secret between two strangers without them ever having exchanged any confidential information. This idea seemed as absurd as asking two people to choose the same card from a shuffled deck without consulting each other.

The feat relied on modular arithmetic and the fascinating properties of one-way functions. Diffie and Hellman exploited the fact that calculating $g^x \bmod p$ is easy but retrieving x when knowing only the result is extraordinarily difficult. Each participant generates their secret number, calculates a public version which they transmit openly, then combines this information with their correspondent's public number. Through a kind of mathematical magic, they both obtain an identical secret result.

This discovery revolutionized traditional cryptography. AT&T quickly grasped the commercial interest of this innovation by developing the Common Channel Interoffice Signaling system to protect their telephone communications. But it was with the explosion of the Internet in the 1990s that the protocol found its true destiny. SSL, and its successor TLS, integrated Diffie-Hellman at the heart of their mechanisms, transforming each HTTPS connection into a living demonstration of this mathematical prowess.

The protocol's security relies entirely on the difficulty of calculating discrete logarithms. No classical computer can efficiently solve this mathematical problem, which explains why the protocol has held strong for nearly fifty years. However, quantum computing threatens this balance. Shor's algorithm could one day transform this reputedly unsolvable problem into a simple calculation exercise.

Mathematicians have been working tirelessly to adapt and improve the original protocol. Victor Miller and Neal Koblitz proposed a variant using elliptic curves in 1985, called ECDH. This version drastically reduces the size of keys needed while maintaining a constant level of protection. A real efficiency gain that has won over many developers concerned with optimizing their applications.

The Diffie-Hellman protocol gave rise to RSA, developed by Rivest, Shamir and Adleman in 1978, and inspired an entire generation of cryptographers. This innovation also democratized cryptography research,

breaking the monopoly of secret agencies and giving birth to a worldwide academic community.

Official recognition took time. Diffie and Hellman received the Turing Award in 2015 only, nearly forty years after their discovery. This wait was probably due to the Cold War climate surrounding any cryptographic innovation. Moreover, British researchers at GCHQ had developed similar ideas a few years earlier, but their work remained classified until 1997. A fine illustration of how state secrecy can sometimes delay scientific progress.

The anticipated arrival of quantum computing is currently mobilizing cryptographers worldwide. They are working tirelessly on post-quantum variants of the protocol, exploring new mathematical problems resistant to future quantum machines. This research extends the pioneering spirit of Diffie and Hellman, pursuing their vision of cryptography accessible to all. They proved that an open and academic approach to cryptography could rival the most secret government laboratories. This new intellectual paradigm unleashed researchers' creativity and gave birth to a thriving computer security industry.

Every time you connect to your online bank or buy something on the Internet, you benefit from the legacy of these two visionaries. Their protocol continues to protect billions of daily communications, a living testament to an era when two researchers were able to imagine the impossible and transform it into reality.

Altair 8800

In January 1975, when MITS (Micro Instrumentation and Telemetry Systems) launched the Altair 8800, no one suspected that this small metal box selling for less than $400 would revolutionize the computing world. Yet this first personal computer to achieve genuine commercial success had just been born in the facilities of a company on the brink of bankruptcy.

The story begins with H. Edward Roberts, an electrical engineering graduate from Oklahoma State University who served as an officer in the US Air Force. At the Albuquerque weapons laboratory, he met Forrest Mims III. Together, they worked on laser and model rocket projects. This collaboration gave birth to MITS in 1969, a company specializing in telemetry instruments for model aviation.

The company then turned to electronic calculators and enjoyed sev-

eral profitable years. But in 1974, competition from major manufacturers caused prices to collapse. Roberts saw his revenue plummet. He needed to find something else, fast. That's when he envisioned an affordable personal computer, an idea that seemed crazy when computing was the exclusive domain of large corporations and universities.

The Intel 8080 microprocessor had just arrived on the market. This 8-bit component far surpassed its predecessor, the Intel 8008. Roberts saw an opportunity to build his machine. The Altair's architecture leveraged the processor's capabilities: a 16-bit address bus accessing 65,000 words of memory, a basic instruction set of 78 instructions, and most importantly an expandable bus system. The basic configuration offered only 256 words of RAM, expandable by the user according to their needs and budget.

The Altair's front panel was striking in its apparent complexity. Rows of switches and LEDs enabled direct programming in machine language. A spartan approach that discouraged novices, but one that reflected the cost constraints of the era.

Roberts bet everything on an article in the January 1975 issue of Popular Electronics. With William Yates, he wrote a paper with a catchy title: "World's First Minicomputer Kit to Rival Commercial Models". The magazine came out in December 1974, and the response exceeded all expectations. MITS had hoped to sell a few hundred units; orders arrived by the thousands within the first few days.

This wave revealed the existence of a market that no one had truly imagined. The Altair attracted two young Harvard students, Paul Allen and Bill Gates, who developed a BASIC interpreter for the machine. This was the birth of Microsoft. In March 1975, the Homebrew Computer Club formed around the Altair in Silicon Valley. Steve Wozniak cut his teeth there and presented the work that would lead to the creation of Apple Computer.

The Altair's technical influence extended to making its system bus a standard known as S-100. Many manufacturers adopted this architecture for their own machines. The Altair used the CP/M operating system, which would dominate the microcomputer market until the arrival of the IBM PC and MS-DOS. The modular approach, with its expansion cards, inspired an entire generation of computers.

The machine's limitations reflected the state of technology in 1975. No screen, no keyboard in the standard version. Programming was done through the front panel switches, a laborious exercise that deterred beginners. Memory was scarce and users had to wait for expansions to have real storage capabilities. But these constraints stimulated creativity: a multitude of manufacturers developed compatible cards and peripherals,

creating a genuine ecosystem.

MITS discovered the joys and sorrows of unexpected success. The company struggled to fulfill orders and maintain an acceptable level of quality on hand-assembled kits. Technical support, provided notably through the Computer Notes bulletin, was not always sufficient in the face of questions from users sometimes overwhelmed by the complexity of their purchase. These growing pains did not prevent success, but they illustrated the challenges of creating a new market.

The Altair adventure ended in 1977 with the sale of MITS to Pertec Computer Corporation. Production ceased, but the legacy endured. The machine had proven that a personal computer could be commercially viable and technically credible. It gave birth to a community of enthusiasts that foreshadowed modern computing culture.

The Altair 8800 marks the boundary between two worlds: that of institutional computers, reserved for insiders, and that of personal computing, promised to all. Its open and modular architecture laid the foundation for principles that still govern the industry today. The enthusiasm it sparked among hobbyists transformed a technical curiosity into a cultural movement. More than an engineering feat, the Altair demonstrated that a different relationship with computing was possible.

X.25

In 1976, when the International Telegraph and Telephone Consultative Committee adopted X.25, no one suspected that this protocol would survive for more than thirty years. Yet this technical standard, born from heated debates between French and British engineers, would lastingly shape the architecture of data networks.

The first questions about the future of digital communications were emerging. ARPANET had taken its first steps across the Atlantic since 1969, but in Europe, the situation was radically different. Telecommunications administrations reigned supreme over the infrastructures. In France, the Direction générale des télécommunications held the reins, just as the British Post Office did in the United Kingdom. These public monopolies did not look favorably upon the relative anarchy that characterized American developments.

Packet switching sparked passionate technical debates. This method, which breaks messages into small portions to optimize their circulation,

divided the scientific community. On one side, Rémi Després from CNET staunchly defended virtual circuits: each communication follows a predetermined path through the network, which guarantees a certain order. On the other, Louis Pouzin and his team at IRIA advocated for datagrams: packets navigate freely, finding their route according to network availability.

This technical quarrel concealed much broader stakes. Virtual circuits appealed to traditional telephone operators through their predictability. They recalled the familiar world of classic telephone switching, where each call establishes a dedicated circuit. Datagrams, on the other hand, embodied a different philosophy, closer to the libertarian spirit blowing across American campuses.

The adoption process at the CCITT resembled a game of diplomatic chess. Larry Roberts, despite being the father of ARPANET, paradoxically supported virtual circuits from his private company. But without voting rights, his influence was limited. A similar situation existed for the Canadians at Bell Canada, who were nevertheless developing Datapac on this technical basis.

The turning point came with the unexpected rallying of the British Post Office to French positions. Halvor Bothner, the Norwegian rapporteur, then abandoned his initial position. He who had coined the term "datagram" ultimately sided with virtual circuits. This about-face sealed the fate of X.25, adopted in March 1976 and then solemnly confirmed in the fall.

Ironically, this French victory immediately benefited the international community. Spain and the Netherlands built their national networks on X.25. Bell Canada integrated it into Datapac, thus validating European technical choices. IBM, ever pragmatic, adapted its equipment to this standard as early as 1975.

In France, X.25 found its apotheosis with Transpac. This national network would support Minitel traffic for more than three decades, from 1980 to 2012. A remarkable performance for a technology that some had already deemed outdated in the 1990s. Banks, in particular, remained loyal to it: its inherent reliability and security corresponded perfectly to their strict requirements.

The emergence of the Internet could have sounded the death knell for X.25. TCP/IP, a direct heir to Pouzin's datagrams, gradually established itself as the de facto standard. But X.25 persisted in its specialized niches. Costa Rica was still using it in 1999, France until 2012. This longevity contrasts with the resounding failure of OSI, that pharaonic ISO project that claimed to standardize all computer communications.

X.25 perfectly illustrates the caprices of technical innovation. A standard can be born from a political compromise, flourish in unexpected uses, and survive far beyond its initial predictions. It also testifies to the ability of French engineers to carry weight in international technical debates, when technical quality meets diplomatic skill.

This story above all reveals the complexity of technological ecosystems. Contrary to popular belief, the "best" technical solution does not always prevail. X.25 and TCP/IP coexisted for decades, each meeting specific needs. This peaceful coexistence contrasts with the standards wars that regularly shake the computer world.

Cray-1

The year 1976 witnessed the birth of a machine that would redefine the boundaries of scientific computing. The Cray-1, the fruit of Seymour Cray's technical obsession, emerged from an already remarkable journey in the world of supercomputers. Cray had previously designed the Control Data 6600 and 7600, two benchmark systems that had established his reputation as a visionary architect. But this time, something else was at stake: building the fastest machine ever conceived.

In 1972, Cray left Control Data to create his own company, Cray Research. The stated objective could be summed up in a few words: build the world's most powerful computer. Four years of intensive development culminated in a spectacular achievement. The Cray-1 first struck observers with its appearance: a cylinder 2.75 meters in diameter and nearly 2 meters high, breaking with the usual rectangular aesthetics of computers. This circular form was not an aesthetic whim. It responded to an implacable technical constraint: minimizing connections between components to gain precious nanoseconds.

The figures were staggering. The machine reached 80 million floating-point operations per second in normal operation, with peaks at 250 million. A performance that shattered all established records. This raw power relied on a decisive innovation: vector architecture. The Cray-1 simultaneously processed entire data series through a "chaining" technique that orchestrated eight vector registers of 64 elements each.

Beneath its futuristic casing, the machine concealed a design philosophy of disarming simplicity. Seymour Cray used only four different types of integrated circuits: 5/4 NAND gates, memory chips, and registers, all

based on ECL technology. This minimalist approach nonetheless concealed a dizzying complexity: 1,662 modules distributed across 113 different types, with each module capable of containing up to 288 integrated circuits. The ensemble mobilized the equivalent of 2.5 million transistors.

The electronic density generated a formidable thermal challenge. Concentrating so many components in such a reduced space produced heat that traditional cooling systems could not evacuate. Cray then developed a freon-based solution of remarkable sophistication. Aluminum and stainless steel cooling bars wound through each column of the machine, maintaining a stable temperature of 21°C. This technical feat was essential to the reliable operation of the entire system.

Memory reached one million 64-bit words, organized into 16 independent banks. This architecture offered the ability to access data simultaneously from multiple processes, reducing wait times that penalized performance. A SECDED system protected calculation integrity by automatically correcting single errors and detecting double errors.

Commercial success exceeded all predictions. Between 1976 and 1982, Cray Research sold approximately 80 machines at a unit price of 19 million dollars. Early customers included American government research centers, but also European institutions such as the European Centre for Medium-Range Weather Forecasts. This international dimension confirmed the technological lead Cray had achieved.

Application domains revealed the machine's versatility. Meteorology, aeronautics, nuclear research: wherever numerical simulation required intensive floating-point calculations, the Cray-1 excelled. Its ability to rapidly solve complex systems of differential equations opened new perspectives for scientific modeling.

The technical influence went far beyond raw performance. The operating system and FORTRAN compiler, specially optimized to exploit vector architecture, established new standards. Programming proved more accessible than on competitors like the ILLIAC IV, favoring adoption by the scientific community.

Development involved considerable challenges. Perfecting the cooling system required eighteen months of intensive research. Manufacturing the five-layer cards, with their extremely precise interconnections, pushed the limits of existing techniques. These constraints stimulated innovations that benefited the entire industry.

In 1990, fourteen years after its commissioning, Lawrence Livermore National Laboratory retired its Cray-1, surpassed by new generations. The machine was sold at auction in 1993 for $10,000, or 0.05% of its initial price. This vertiginous drop illustrated the acceleration of technological

progress in supercomputers.

The Cray-1 crystallized an era when the pursuit of absolute performance drove innovation. Its influence persists in current design principles: optimization of inter-component communications, sophisticated thermal management, simplified architecture despite the complexity of technical challenges. The machine embodied Seymour Cray's vision, which prioritized elegance and coherence in the quest for technical excellence.

This achievement demonstrated that bold architecture, supported by meticulous engineering, could dramatically push the frontiers of scientific computing. The principles established by the Cray-1 confirm its place as a foundational milestone in the evolution of high-performance computing.

UUCP

In 1976, Mike Lesk was working on a practical problem at Bell Labs. ARPANET already existed, but access remained the privilege of a restricted circle: in January of that year, only 63 hosts were connected to it. For UNIX users scattered across the United States, exchanging files or executing remote commands was an uphill battle. Lesk designed UNIX-to-UNIX Copy Protocol (UUCP), a pragmatic solution that transformed ordinary telephone lines into digital arteries.

The first version of UUCP was integrated into UNIX in February 1978. The protocol operated on 300-baud connections, a speed that seems trivial but represented notable progress. A few months later, in October, the Seventh Edition of UNIX included a revised version of the protocol. This time Lesk partnered with Dave Nowitz, with help from Greg Chesson, to correct the initial system's teething problems.

In late 1979 at the University of North Carolina, Steve Bellovin, a student seeking to master the system's intricacies, developed a modest news distribution program. Meanwhile, Tom Truscott and Bellovin experimented with a UUCP connection between UNC and Duke University. From this experimentation emerged an idea that would transform UUCP usage: why not distribute information to other sites using Duke as a hub? Remote sites reimbursed Duke for telephone costs, a rudimentary but effective system.

In early 1980, the network had three participants: UNC, Duke University, and Duke Medical Center's physiology department. Jim Ellis presented the concept at a USENIX meeting in Boulder in January. The

reception was favorable. Steve Daniels produced an implementation of the software called A News, distributed on the summer 1980 USENIX tape in Newark. The network expanded to 15 sites.

Armando Stettner and Bill Shannon, working at Digital Equipment Corporation, proposed an attractive deal to the University of California at Berkeley: they would cover Berkeley's connection costs in exchange for network access via their decvax machine, located in New Hampshire. Stettner didn't stop there and financed the first international connections to Europe, Japan, and Australia.

The network was named Usenet, in homage to USENIX. Growth exceeded all predictions: within a year, more than 100 sites were exchanging approximately 25 articles daily. This expansion revealed the initial system's weaknesses. Mark Horton, a student at Berkeley, partnered with Matt Glickman, then a high school student, to completely rewrite A News. Their creation, B News, appeared in 1981. Horton continued to refine his work until 1984, before passing the torch to Rick Adams from the Center for Seismic Studies.

A geographic map of USENET revealed the phenomenon's scope in June 1984. Connections now extended to Australia, Hawaii, Canada, and Europe. Version 2.11 of B News, released in 1986, incorporated contributions from Rick Adams, Spencer Thomas, Ray Essick, and Rob Kolstad. Although new versions appeared until 1994, the system showed its limitations as early as 1989.

The NNTP (Network News Transfer Protocol), published in February 1986 by Brian Kantor and Phil Lapsley, heralded a new era. A year later, Geoff Collyer and Henry Spencer from the University of Toronto launched C News, a more efficient alternative that definitively buried the old systems.

The financial question obsessed network actors. Lauren Weinstein presented the Stargate project at the 1984 summer USENIX conference, a bold proposal to use unused portions of the television video signal to transport ASCII data at 65 kbps. Despite support from several companies and successful tests, the project bogged down and was eventually abandoned.

Facing the inadequacies of existing commercial networks, Rick Adams proposed to USENIX in 1985 the creation of a central site accessible via Tymnet. UUNET was born in 1987 with an initial budget of $35,000. The service met a real need: many people lost their email and Usenet access when changing employers. Success exceeded expectations. Installed on a 16-processor Sequent B21 computer, UUNET had more than 50 subscribers by June 1987 and several thousand five years later.

UUNET participated in the creation of the Commercial Internet Ex-

change Association in 1991. The following year, the company co-founded MAE-East with Metropolitan Fiber Systems, the world's largest Internet exchange point for a time. The IPO occurred in May 1995, consecrating the concept's commercial success.

UUCP's success stemmed from its innovative technical characteristics. The system operated asynchronously, with task queuing. Transmissions occurred in the background, accommodating the constraints of often-busy telephone lines. The protocol required no operating system modifications and adapted equally well to direct connections and modems.

Security aspects were not neglected. Each site configured files and commands accessible to remote systems. Authentication relied on passwords, and some sites used automatic callback to verify caller identity. Sequence counters could be activated to counter identity spoofing attempts.

UUCP persisted into the 1990s, particularly in situations where permanent Internet connection was inaccessible or too expensive. On slow serial lines, it proved more efficient than TCP/IP. Its batch architecture was perfectly suited to email and Usenet news transport.

UUCP's legacy spans decades. Email address syntax (user@host) owes its existence to it, as do certain email routing mechanisms. The protocol demonstrated that it was possible to build large-scale decentralized computer networks relying on existing telecommunications infrastructure. Its influence is found in the subsequent development of the Internet and asynchronous communication technologies.

Apple I

It was 1975, in the intellectual ferment of the Homebrew Computer Club in Silicon Valley. Steve Wozniak, a discreet but passionate figure at these electronics hobbyist meetings, had cobbled together something unusual: a computer he wanted to show the other club members. His friend Steve Jobs, whom he'd met four years earlier, immediately sensed something more than just an enthusiast's project.

The heart of this machine was based on a MOS Technology 6502 microprocessor running at 1.023 MHz. Wozniak had the idea of integrating a video interface directly into his system, capable of displaying 24 lines of 40 characters on any standard television set. This was far from obvious. The machine came with 4 KB of RAM, expandable to 8 KB, with dynamic shift registers managing the display.

Everything changed when Paul Terrell showed up. This owner of the Byte Shop, a computer store in Mountain View, ordered 50 machines outright. Jobs and Wozniak looked at each other: they had neither the money nor the infrastructure to produce anything at that scale. Wozniak sacrificed his precious HP-65 calculator, Jobs his Volkswagen van. With this money and a 30-day credit negotiated with suppliers, they launched into the venture.

Contrary to the legend of the family garage, the reality of production proved more prosaic. The boards were manufactured and wave-soldered in a real factory, probably Santa Clara Circuits. The famous Jobs family garage mainly served for final testing and assembly. Daniel Kottke, Jobs's friend, spent his evenings testing the machines while Wozniak intervened to troubleshoot the trickiest cases.

The Apple I hit the market in a form that surprised people. No case, no keyboard, no screen: just a bare motherboard that the buyer had to complete according to their means and desires. Paul Terrell was stunned when he received his order. He expected complete computers, not circuit boards. The price was set at $666.66, not out of provocation but because Wozniak loved repeating numbers.

On the software side, the Apple I remained spartan. A small monitor in ROM allowed direct entry of programs in machine language. Wozniak then developed his own BASIC interpreter, inspired by his experience at HP with calculators. But this BASIC had some flaws: impossible to manipulate decimals or numbers beyond 32,767, a limitation that would also plague the first version of the Apple II.

The audio cassette interface, added later, turned into a nightmare. Liza Loop, the first official buyer, struggled like many other users with her Apple I that refused to cooperate. Wozniak, brilliant with digital electronics, floundered a bit when tackling analog electronics. The Apple II would fortunately benefit from the support of other engineers to fix the problems.

Nevertheless, Wozniak incorporated remarkable technical innovations into his design. He cleverly diverted the video circuit counters to refresh the RAM, a technique he would refine on the Apple II. The video circuit kept its mysteries: certain parts remained obscure, even for Wozniak, who may have drawn inspiration from Don Lancaster's "TV Typewriter", published in *Radio-Electronics* in 1973.

In the end, fewer than 200 Apple Is came off the production lines. A modest score that in no way foreshadowed the tidal wave that would follow. Today, between 30 and 50 specimens survive in private collections or museums. Some come from Apple employees who recovered them in

the 1970s from unsold stock or customer returns.

The Apple I was never a commercial success, but it forged something infinitely more precious: experience. Jobs and Wozniak learned on the job the ins and outs of computer design, production, and sales. Every flaw in the Apple I was a lesson for the Apple II, which would break all records with more than 6 million units sold over sixteen years.

This first machine, beyond a technical anecdote, embodied the idea that a computer could address the general public, escape from laboratories and universities to land in the average person's home. The Apple I opened a breach through which millions of other machines would rush.

It also testified to that blessed era when a few electronics enthusiasts, armed only with their ingenuity and a few thousand dollars, could shake up an entire industry. The story of its creation perfectly captured this alchemy between Wozniak's technical genius and Jobs's commercial instinct, a recipe that would propel Apple to unimaginable heights.

vi

In 1976, at the University of California, Berkeley, Bill Joy faced a problem familiar to all programmers of that era: how to edit text decently on a UNIX system? The available tools, such as *ed*, were more akin to intellectual torture than programming assistance. Peter H. Salus did not hesitate to describe them as "the most user-hostile editors ever created". Imagine working line by line, never seeing your entire text!

Joy, a graduate student, refused to endure this situation. He was already working on a Pascal interpreter and collaborating with Chuck Haley to improve *ed*. The university had a PDP-11, a modest machine compared to the PDP-10s that MIT or Stanford could afford. This hardware constraint pushed them to innovate differently, seeking efficiency rather than raw power.

The summer of 1976 brought an unexpected breakthrough. Joy and Haley discovered *em* (Editor for Mortals), created by George Coulouris at Queen Mary College. Here was an editor that dared to display two error messages instead of one and offered a more visual approach to editing. Nothing groundbreaking in itself, but enough to spark inspiration. The two colleagues then developed *ex*, whose "visual" mode would give birth to the name "`vi`".

The genesis of `vi` owes as much to chance as to genius. Joy remembers

designing the interface one Saturday in 1976, with the radio in the background broadcasting Jimmy Carter's question-and-answer session shortly after his election. This anecdote reveals the relaxed atmosphere in which a tool destined to mark computing history was born. The editor integrated its innovations while adapting to existing terminals, notably the ADM-3A. The famous mode system—normal, insert, command—naturally arose from these technical constraints and the art of making the most of every keystroke.

Joy did not bother with sophisticated commercial channels to distribute his creation. He offered copies on 9-track magnetic tapes for $50, of which $40 went to fund development. This artisanal distribution met with unexpected success: hundreds of copies went to universities and laboratories. The technical superiority of `vi` over its competitors explained this enthusiasm, reinforced by its distribution with BSD, Berkeley's version of UNIX.

The arrival of VAX machines in 1977-1978 gave the project a second wind. Joy adapted `vi` to the capabilities of these new machines, particularly exploiting virtual memory. Development progressed well until a technical incident changed everything: a tape drive failure caused the source code to be lost. The planned advanced features, such as multi-window support, were abandoned. Joy preferred to stabilize what existed and produce proper documentation.

The story took a surprising turn when AT&T incorporated `vi` into UNIX System V. Joy learned this news after the fact, discovering his editor as a standard throughout the UNIX universe. This decision was explained by the massive adoption of `vi` within Bell Labs itself. Paradoxically, this popularity created problems: each team maintained its own copy, wasting machine memory.

`vi` established an editing philosophy that still divides opinion. Its modal system requires substantial initial learning but then offers unmatched text manipulation power. Some find it too complex, others swear by its efficiency once mastered. This duality has persisted through the decades without losing its relevance.

The 1990s saw the birth of `vim` (Vi IMproved), which extended the legacy of the original editor while adding features expected by modern users. `vi` is present on all UNIX systems and their descendants, including Linux and macOS. Its influence extends beyond these boundaries, and many modern editors offer a "`vi` mode" to attract its devotees.

This longevity is surprising in a field where obsolescence is rapid. A tool designed in the 1970s continues to serve millions of users daily. This longevity stems from its formidable efficiency for certain tasks, its univer-

sal presence, and its proven reliability. System administrators and programmers appreciate its guaranteed availability and rock-solid stability.

The `vi` adventure illustrates the importance of initial choices in software design. The constraints that guided Joy—memory economy, optimal keyboard exploitation, compatibility with limited terminals—produced a tool whose utility transcends technological eras. This success reminds us that good software is judged not only by its features but by its conceptual coherence and its response to users' fundamental needs. Fifty years after its creation, `vi` continues to prove this truth every day.

DEC VAX-11

In the mid-1970s, mainframe computers still dominated large organizations, while Digital Equipment Corporation's 16-bit minicomputers ruled an expanding market. But at DEC, a few visionary engineers understood they were reaching the limits of this architecture. Programs were growing, data multiplying, and 16 bits were showing their weaknesses.

In March 1975, a team quietly formed in the Maynard, Massachusetts offices. Their mission: to design an innovative 32-bit architecture. The project bore poetic code names: "Star" for the hardware, "Starlet" for the operating system. This unusual approach was immediately striking: unlike the customary practice where hardware and software evolved separately, the teams worked hand in hand from day one.

Three hundred man-years of intensive development elapsed before DEC unveiled its masterpiece. On October 25, 1977, at the annual shareholders' meeting, the presentation of the VAX-11/780 and its VMS system made a lasting impression. The chosen demonstration bordered on boldness: the computer faced a human in Scrabble. The word "sensibly" earned it 127 points and victory. This playful wink concealed an impressive technical reality: the VAX-11/780 combined the power of a mainframe, the interactivity of a minicomputer, and a finally affordable price.

The VAX-11/780 architecture challenged conventions. The 32-bit virtual memory freed programmers from the addressing constraints that had limited their ambitions. Error correction integrated into memory, a world first, transformed system reliability. DEC embraced a bold philosophy: designing a machine meant to last fifteen to twenty years, turning its back on the planned obsolescence already practiced by its competitors.

The market responded enthusiastically. By 1979, DEC's sales crossed

the $2 billion threshold. Universities massively adopted these machines for their research laboratories. Companies discovered a tool capable of handling their most demanding scientific applications. VMS, the operating system, impressed with its robustness and advanced features.

The VAX family gradually expanded. The VAX-11/750 arrived in 1980 with its semi-custom gate array technology. Two years later, the VAX-11/730 democratized access to this architecture. In 1984, the VAX 8600 set new performance records with quadruple the power of the original model.

A remarkable innovation emerged in 1983: VAXclusters. For the first time in the industry, multiple VAX computers could connect in a network and form a single system. This technical feat revolutionized high availability. Stock exchanges, industrial control systems, all critical environments embraced this technology that tolerated failures like no other.

In 1985, the MicroVAX concentrated the entire VAX architecture on a single chip, making more compact and less expensive systems possible. Success exceeded all expectations: 20,000 MicroVAX IIs found buyers during the first year of commercialization.

Technical evolution continued at a sustained pace. The 1987 CVAX adopted CMOS technology, more energy efficient. The Rigel processor in 1989, then the NVAX in 1991, continually pushed performance limits. The latter reached a speed thirty times that of the VAX-11/780 that had started it all fourteen years earlier.

VMS shaped the evolution of operating systems. Custom-designed for the VAX, it established unprecedented standards of reliability and functional sophistication. The native integration of DECnet anticipated the growing importance of machine-to-machine communications, foreshadowing the era of computer networks.

The early 1990s saw the emergence of 64-bit RISC processors. DEC prepared its transition to the Alpha architecture without abandoning its legacy. VMS, renamed OpenVMS, migrated to this new platform while maintaining support for older systems. The company continued its VAX production, releasing the 4000 model in 1996.

The VAX's success rested on a skillful blend of ingredients: computing power, legendary reliability, sophisticated operating system, and exceptional technical support from DEC. Anecdotes abound about this extraordinary robustness. That VAX-11/780 that continued functioning after falling from a forklift. Those systems that ran for years without interruption, defying the laws of electronic wear.

The VAX's 32-bit architecture influenced the design of many modern processors. VMS concepts, advanced virtual memory management or

clustering, are found in our current operating systems. This integrated hardware-software design philosophy now stands as self-evident throughout the industry.

The VAX testifies to a bygone era when computer innovation privileged long-term vision and technical excellence over the race for novelty. Its exceptional longevity—some VAX systems still function today—validates this approach. Beyond the technical object, the VAX embodies a foundational stage in computer history, that pivotal moment between the era of mainframes and that of distributed systems shaping our digital world.

RSA

In 1976, no one could have imagined that a research paper published by Whitfield Diffie and Martin Hellman would revolutionize the world of cryptography. Their publication, "New Directions in Cryptography", introduced an idea that seemed purely theoretical: asymmetric cryptography. Until that date, all cryptographic systems operated on a single principle: two people wishing to communicate securely first had to meet to exchange a secret key. This constraint paralyzed the development of digital communications.

The trio of researchers—Diffie, Hellman, and Ralph Merkle—proposed three concepts that would revolutionize the field: public-key encryption, digital signatures, and key exchange. Their paper, however, contained only one concrete implementation: the key exchange method that would bear their names. This Diffie-Hellman protocol constituted the first practical method for establishing a shared secret without prior meeting.

The following year, three researchers from MIT would take the decisive step. Ron Rivest, Adi Shamir, and Len Adleman formed a team with complementary talents. Rivest excelled at applying theoretical concepts to concrete problems. He devoured scientific literature and generated a constant stream of new ideas. Shamir possessed that rare ability to pierce through to the essence of a problem beyond its apparent complexity. Adleman, a rigorous mathematician, evaluated each proposition with the precision of a Swiss watchmaker.

For months, the trio explored various avenues without success. Then came that April evening in 1977. Rivest had spent the Passover evening at a student's home, where they had shared Manischewitz wine. Back home, unable to sleep, he settled on his couch with a mathematics textbook.

The question that had obsessed him for a year turned over in his mind: does there exist a mathematical function that is easy to compute in one direction but impossible to reverse without particular information? By sunrise, Rivest had drafted the entire paper describing the RSA system.

The genius of this discovery lies in its conceptual simplicity. The RSA system relies on a fundamental arithmetic property: multiplying two prime numbers is child's play, but finding those numbers from their product is a nightmare when they reach a respectable size. The system generates two mathematically linked keys: a public key that anyone can know, and a private key that only its owner possesses. What is encrypted with the first can only be decrypted with the second.

To demonstrate their confidence in this new system, the three inventors launched a bold challenge in the pages of *Scientific American*. They published a 129-digit number, the product of two secret prime numbers, accompanied by an encrypted message. One hundred dollars awaited whoever could decipher it before April 1, 1982. This wager testified to their conviction in the mathematical robustness of their creation.

RSA-129 would resist well beyond the fixed deadline. It would take until 1994 to see this numerical giant collapse under the coordinated assault of 600 people spread across 24 countries. Arjen Lenstra, Paul Leyland, Michael Graff, and Derek Atkins orchestrated this collaborative enterprise that mobilized 1,600 computers for seven months. The secret message revealed—"THE MAGIC WORDS ARE SQUEAMISH OSSIFRAGE"—rewarded this technical feat with poetry of rather relative quality.

This collective victory did not signal the death of RSA but illustrated the permanent evolution of the balance of power between cryptographers and cryptanalysts. Over the decades, numerous attacks emerged, often targeting implementation weaknesses rather than the mathematical foundations of the system. Michael Wiener demonstrated in 1990 that an excessively small private exponent compromises security. Paul Kocher revealed in 1996 that a smart card could betray its secret key through the simple time it takes to compute. Daniel Bleichenbacher discovered in 1998 that certain error messages divulge valuable information to a patient attacker.

These discoveries did not undermine the confidence placed in RSA, which became one of the pillars of the modern internet. Every time you connect to your online bank, make a purchase, or check your email, RSA works silently to protect your data. Its discreet yet omnipresent presence has made possible the explosion of e-commerce and the digitalization of financial services.

Yet, a sword of Damocles hangs over this cryptographic giant. In 1994,

Peter Shor published an algorithm that could, on a sufficiently powerful quantum computer, efficiently factor very large numbers. This theoretical threat stimulates research in post-quantum cryptography, a race against time to develop new systems before quantum computers become reality.

Forty-five years after its birth, RSA retains its relevance and reliability. Its journey demonstrates that an innovation can transform society far beyond its creators' initial intentions, as new cryptographic challenges emerge with the advent of the quantum era.

Apple II

With the release of the Apple II in 1977, Steve Wozniak and Steve Jobs were no longer content to tinker in a garage—they wanted to reach the mass market. Their first consumer machine bore no resemblance to the electronic kits cluttering computer clubs at the time. Gone were the bare motherboards and makeshift cases: the Apple II came in an elegant beige plastic housing with an integrated keyboard, ready to use right out of the box.

Wozniak designed the architecture around the 6502 microprocessor, cheaper than the Intel 8080 that powered its competitors. His technical genius shows through in the simplicity of the circuit board, where each component finds its place with remarkable economy of means. The machine started with 4 KB of RAM, expandable to 48 KB, and came with Integer BASIC built directly into ROM. But what immediately stands out is the native support for color graphics and sound. Wozniak had actually developed these features with a specific goal in mind: programming *Breakout* in BASIC.

Jobs focused on marketing and design. He categorically refused metal cases that made computers look like industrial equipment. Jerry Manock designed a molded plastic housing with rounded corners and that characteristic beveled shape that would become Apple's visual signature. The device had to fit into a living room like any television set.

Sales began in June 1977 at $1,298 for the base configuration. Success exceeded expectations. Individuals, schools, and small businesses competed for the machine. The open architecture with its eight expansion slots facilitated upgrades: additional memory, various interfaces, disk controllers. A complete ecosystem of third-party manufacturers developed.

With the arrival of the Disk II floppy drive in 1978, Wozniak hit a

home run: for $495, his creation offered 113 KB of storage on 5.25-inch diskettes. The DOS operating system, in its successive versions 3.1, 3.2, and then 3.3, simplified file management. This reliable and affordable solution definitively opened the doors of the professional market to the Apple II.

The year 1979 would be remembered. Apple released the Apple II Plus, which replaced Integer BASIC with the more sophisticated Applesoft BASIC in ROM, featuring floating-point numbers. Standard memory jumped to 48 KB. But most importantly, VisiCalc appeared. This first spreadsheet in history revolutionized computer use in business. Many bought an Apple II solely to run VisiCalc.

The Apple IIe of 1983, the true star of the lineup, had 64 KB of RAM expandable to 128 KB and an 80-column display, making it a credible office productivity tool. The full keyboard with lowercase letters facilitated text entry. Compatibility with existing software was total, ensuring longevity of software investments.

Apple took a gamble on compactness with the Apple IIc in 1984. This integrated model with a built-in floppy drive targeted homes and schools. Abandoning expansion slots in favor of standardized ports simplified usage but limited upgradability. Its polished design earned it several aesthetic awards, a first for Apple.

The swan song came with the Apple IIGS in 1986. Its 16-bit processor was compatible with the 6502, its graphics and sound capabilities were impressive, and its interface drew inspiration from the Macintosh. Alas, Apple was already betting everything on the Macintosh. The IIGS, despite its undeniable qualities, remained in the shadow of its bigger sibling.

Production of the Apple II ended definitively in 1993 after sixteen years of an exceptional career. Over six million units sold testify to a success that transcends computing. This machine democratized the personal computer, proving it could be both powerful and accessible.

Commodore P.E.T

In 1977, when Commodore International unveiled its Personal Electronic Transactor, home computing barely existed. Computers remained the domain of universities, businesses, and a few enthusiasts capable of assembling complex kits. Jack Tramiel, Commodore's boss, sensed an opportunity. Microprocessor prices were plummeting, and he envisioned a

different device: a complete, ready-to-use computer in a single housing.

The P.E.T broke with everything that existed at the time. Gone were components scattered across multiple boards, gone the jury-rigged television screen and tape recorder diverted from its original purpose. Commodore integrated everything: processor, memory, keyboard, 9-inch monochrome screen, and cassette drive. This approach revealed an astute commercial vision. Where other manufacturers targeted electronics hobbyists, Tramiel aimed directly at future everyday users.

The machine's heart beat to the rhythm of a MOS Technology 6502 processor running at 1 MHz. This technical choice, far from trivial, relied on the recognized qualities of this component: simple architecture, reduced cost, and honorable performance. The 4 KB RAM represented a reasonable compromise between functionality and cost. Moreover, Commodore offered enhanced versions with 8, 16, and 32 KB.

Microsoft BASIC resided in read-only memory, instantly transforming the P.E.T into a programmable machine upon power-up. This version of the language, adapted to the computer's specificities, integrated commands dedicated to peripherals and the screen's rudimentary graphics capabilities. Users could immediately type their first lines of code or load a program from cassette.

Storage posed a problem. Commodore's Datasette, derived from a standard tape recorder, proved exasperatingly slow. Loading a program of a few kilobytes required long minutes of waiting, punctuated by characteristic beeps and squeals. Yet this solution remained the only economically viable way to democratize access to computing. Floppy disk drives still cost too much to equip households en masse.

Early users discovered the limitations of the original keyboard. Commodore had adopted the pocket calculator principle, with flat, closely-spaced keys that made typing painful. Complaints poured in, and the American firm corrected course by offering a full keyboard on subsequent models. This responsiveness to criticism demonstrated attentive market listening.

The P.E.T's screen displayed 40 columns by 25 lines in green characters on a black background. No colors, no animated sprites like on future gaming machines. But this sobriety concealed unsuspected possibilities. Programmers ingeniously repurposed semi-graphic characters to draw shapes, create rudimentary interfaces, or design action games. The P.E.T also featured an optional high-resolution mode, MTU's Visible Memory, which allowed individual control of each screen pixel.

In schools, the P.E.T found its place. Its robustness, reliable operation, and affordable price made it the ideal tool for introducing students

to computing. Generations of students learned their first programming basics on these green and white machines. The IEEE-488 interface, inherited from the world of scientific instrumentation, enabled connection of professional measuring devices, extending the P.E.T's use to laboratories and research centers.

The user community organized spontaneously. The P.E.T User Group emerged in 1978 and brought together thousands of members worldwide. Specialized publications, cassette program exchanges, regional meetings: a vibrant ecosystem developed around the machine. This collective dynamic compensated for technical limitations through overflowing creativity.

Independent developers created remarkable applications. VisiCalc, the first consumer spreadsheet, found in the P.E.T an ideal platform for attracting small businesses and independent professionals. Sophisticated games, management software, and educational tools progressively enriched the available catalog. This software diversity transformed the P.E.T from a programming computer into a genuine productivity tool.

Commodore learned lessons from this pioneering experience. The Commodore P.E.T's successes and failures directly informed the design of the VIC-20 and then the legendary Commodore 64. The all-in-one approach validated, the importance of the price factor confirmed, the necessity of a robust software ecosystem understood: so many lessons that would guide the California company's future creations.

The P.E.T gradually faded in the early 1980s, supplanted by more powerful and cheaper machines. This machine proved that a personal computer could appeal beyond the circle of initiates. It outlined a new market where technology yielded to usage, where complexity disappeared behind simplicity of use. The P.E.T fascinates collectors and computer historians. Not for its performance, long since surpassed, but for what it represents: one of the first steps toward democratizing the personal computer.

Tandy TRS-80

In 1976, in the offices of Tandy Radio Shack in Fort Worth, Texas, Don French harbored a bold idea. This computer enthusiast and executive at a company with nearly 3,500 electronics stores across the United States wanted to convince his president, Lew Kornfield, to venture into selling computer kits. The idea seemed crazy: selling a $199 product in

stores accustomed to moving merchandise priced around $30. Management was skeptical. But times were changing. Sales of CB radios, those Citizen Band units that had made the company's fortune, were beginning to falter. The company needed a breath of fresh air, a product capable of maintaining its margins. French's project finally got the green light.

Spring 1976 saw French and his colleagues visit National Semiconductor. There, they met Steve Leininger, a chip designer and member of the famous Homebrew Computer Club. Initially solicited as a mere consultant, Leininger eventually accepted a position at TRS. The man would revolutionize the initial project. While kits dominated the market against much more expensive assembled systems, he pushed TRS toward the latter option. His bet: offer a fully assembled computer for $399.95. With the monitor, the package would reach $599.95. The price remained low enough to appeal to management.

Caution prevailed at Tandy. Don French confirmed it: only 3,500 units were initially produced, exactly matching the number of Radio Shack stores. The logic seemed obvious: if the venture went south, the machines would serve in the retail locations. On August 3, 1977, New York's Warwick Hotel hosted the official presentation of the TRS-80. The press conference went almost unnoticed. Two days later, fortune smiled on the company: the TRS-80's presence at the Personal Computer Faire in Boston earned a front-page Associated Press article.

Upon his return, French discovered that his brainchild had taken on unexpected proportions. The numbers speak for themselves: approximately 250,000 units sold for this first model. The attractive price compared to competition from Apple or Commodore doesn't explain everything. Radio Shack had an ace up its sleeve: its brand recognition and its roughly 5,000 stores and franchises in 1977. Anyone could now see and test a TRS-80 in a shopping mall or on a commercial street. The computer emerged from the circle of enthusiasts to reach the general public.

Businesses began taking interest in the TRS-80, but the first generation quickly showed its limitations. Compared to machines like the IBM 5110, it lacked power and storage space. Tandy struck back in June 1979 at the National Computer Conference in New York with the TRS-80 Model II. Mind you, this wasn't a successor to the Model I, which then took on that name. Radio Shack's advertising hammered the point: this was a different machine, considerably more expensive with a base price of $3,490 including a disk drive. The Z80 processor jumped from 1.77 to 4 MHz, memory climbed from 4K expandable to 16K toward 32K expandable to 48K, graphics gained resolution. Gone was the cassette drive, replaced by a specific TRSDOS and an enhanced BASIC.

July 1980 marked the arrival of the TRS-80 Model III, the true heir to the Model I. Electromagnetic interference from early personal computers prompted the FCC to tighten its directives. The Model I was discontinued in 1981. Like its big brother Model II, the Model III integrated everything into a single case: monitor, keyboard, disk drive. This design reduced radio interference and appealed to educational institutions, less exposed to theft than with separate components. The Z80 processor reached 2.0 MHz and featured a more powerful ROM BASIC 2. The architecture preserved broad compatibility with the Model I, but TRSDOS 1.3 was specific to the Model III. A complete model cost $2,495 in 1980, with the entry-level version priced at $699.

April 1983 saw the birth of the TRS-80 Model 4. Burned by compatibility problems between the Model I and Model III, developers bet on total compatibility with its predecessor. The computing world applauded this decision but lamented the absence of native software. The Model 4 broke new ground: 4 MHz clock, 64K of RAM with the possibility of adding 64K more, 80x24 screen with reverse video, TRSDOS 6 with Microsoft BASIC and CP/M compatibility. The basic configuration with a 16K cassette drive started at $999, the complete model with 64K and two floppy drives peaked at $1,999. Two limited versions would follow, but the original configurations would last until 1991.

The TRS-80's triumph can be explained in different ways. First, that immediate availability in the vast Radio Shack network. Then, a remarkable pedagogical approach with the Level I BASIC manual, exemplary in its clarity. The modular design also appealed: everyone could upgrade their machine according to their means and needs. Radio Shack provided after-sales service, a mark of confidence. The company developed an entire ecosystem: varied peripherals, software ranging from games to professional applications (accounting, inventory management). Over 100,000 systems found buyers, placing the TRS-80 among the first widely distributed personal computers.

The impact on the computerization of small businesses, schools, and American households proved considerable. The TRS-80 democratized access to computers far beyond the circles of specialists and professionals. It embodied this transformation of the microcomputer market between the late 1970s and early 1980s, this shift from a niche market toward the general public and the professional world.

Berkeley Software Distribution

When Ken Thompson, Dennis Ritchie, and Rudd Canaday created UNICS at Purdue University in 1969, they had no idea they were launching a lineage that would give birth to one of the most influential operating system families in computing history. UNICS was renamed UNIX two years later, and as early as 1973-1974, the system saw its first branches with PWB/UNIX and MERT. While PWB/UNIX would evolve into System III, MERT disappeared from the radar.

The true BSD adventure began on March 9, 1978, in the offices of the Computer Systems Research Group at the University of California, Berkeley. This first version of BSD was modest, but versions 2 and 3 marked a turning point. DEC's 32-bit VAX machines gradually replaced the aging PDP-11s, and BSD adapted to this technological transition. The developers integrated Thompson's reworked Pascal compiler, the *vi* editor that would become legendary, and the C shell that was a game-changer for system administrators.

The story might have ended there without DARPA's providential intervention. Bob Fabry secured an 18-month funding that transformed BSD 3 into an innovation laboratory. Virtual memory made its appearance, accompanied by demand paging and sophisticated page replacement algorithms. These improvements laid the groundwork for BSD 4, which introduced the ARPANET protocol—the ancestor of TCP/IP—and redesigned memory management. The `termcap` database advanced terminal management, while device support gained robustness.

Jim Kulp contributed to BSD 4.1 with job control, but it was especially the automatic hardware detection that transformed users' lives. Gone were the tedious kernel recompilations with every configuration change: the system now recognized and configured devices automatically at boot time. A kernel could now run on different machines without human intervention.

The year 1983 marked the pinnacle with BSD 4.2. Bill Joy and Kirk McKusick delivered their masterpiece with the Berkeley Fast File System, which far surpassed traditional UFS in terms of performance and disk space utilization. Signals revolutionized inter-process communication, IPC structured complex exchanges, and TCP/IP established itself as the networking standard. These innovations inspired a generation of systems: SunOS 1.2, Ultrix-II, Mach, and MIPS OS drew directly from this technological goldmine.

BSD 4.3 was released in 1986 with the mission of consolidating achievements and refining TCP/IP. The system reached remarkable technical

maturity, but the commercial aspect entered the picture. In 1990, BSDi launched BSD/386 based on BSD Net/2 code. This commercial version positioned itself as a credible alternative to AT&T's System V, with a compelling argument: a more affordable price. However, the sword of Damocles of AT&T's license still hung over BSD.

The bombshell came in 1992 when AT&T took BSDi to court for trademark violation and false advertising. After an initial failure, AT&T came back in 1993, this time against both BSDi and the University of California. The charge: unauthorized distribution of proprietary code. The situation became more complex when AT&T sold USL and UNIX rights to Novell. It would take until February 4, 1994, to see the birth of 4.4BSD Lite, purged of all elements under AT&T license.

Meanwhile, Bill Jolitz was working behind the scenes on 386BSD. His approach differed: rather than negotiating with lawyers, he developed alternatives to the problematic components. This approach resulted in a completely free version of BSD, freed from the legal constraints that had hindered its development. Distributions temporarily slowed, but this technical liberation constituted a strategic victory for free software.

The following years saw several branches emerge: NetBSD was born from BSD Net/2 code, FreeBSD extended the legacy of 386BSD, DragonFlyBSD later split from FreeBSD, and Apple discreetly integrated FreeBSD elements into macOS X's hybrid Mach/BSD kernel.

BSD's technical legacy spans decades. The FFS file system still inspires contemporary developers. Berkeley's TCP/IP stack remains a reference for understanding network protocols. Job control and automatic hardware detection are now part of the standard UNIX landscape. But beyond these technical contributions, BSD inaugurated an unprecedented development model: collaboration between university and industry, collective code improvement, sharing of innovations.

Its permissive license stands in contrast to more restrictive approaches. This openness favored the adoption of BSD code in commercial projects, creating an ecosystem where academic research and industrial development mutually enriched each other. macOS perfectly illustrates this symbiosis: a commercial system that integrates and values free components.

BSD's fragmentation into multiple variants (FreeBSD, NetBSD and OpenBSD) paradoxically reflects its vitality. Each branch explores specific territory: NetBSD focuses on universal portability, OpenBSD prioritizes absolute security, DragonFlyBSD rethinks multiprocessor architectures. This diversity enriches the ecosystem rather than weakening it.

In the grand history of computing, BSD embodies the transmission of technical knowledge across generations of developers. The system traveled

from university to industry, from research to critical applications, without losing its soul or intrinsic qualities.

Space Invaders

In 1978, at Taito Corporation's offices in Tokyo, Tomohiro Nishikado was working on a project that would soon revolutionize the nascent video game industry. The Japanese engineer had no idea he was about to create one of the most enduring cultural phenomena of his time. His *Space Invaders* emerged in a context where Japanese manufacturers were attempting to catch up with the Americans, who had dominated the sector since the resounding success of *PONG* six years earlier.

Where *PONG* relied on rudimentary TTL circuits, *Space Invaders* introduced an architecture coupling a microprocessor with bitmap video memory. This combination enabled the simultaneous display of dozens of animated graphic objects, an unmatched performance for the era. The game's visual ingenuity lay in its monochrome projection onto a background colored by filters—an economical yet remarkably effective solution.

Nishikado's genius extended beyond purely technical aspects. He forged the conventions of modern video gaming: scoring system, multiple lives, increasing difficulty, infinite loop. The concept appeared disarmingly simple: a laser cannon facing hordes of aliens descending inexorably. The better the player performed, the faster the invaders accelerated their deadly descent. This mechanic created tension that never relented.

The Japanese public's reception bordered on collective frenzy. Arcade halls transformed into temples dedicated to the new phenomenon. Some establishments abandoned all their other games to install exclusively *Space Invaders* machines. Legend has it that a shortage of 100-yen coins struck the archipelago—an invented story but symptomatic of the national craze. The numbers speak for themselves: hundreds of millions of dollars in revenue within a few months.

This lightning success redrew the industry's geography. Japanese manufacturers, galvanized by this unexpected triumph, poured capital into research and development. Namco, Sega, and Nintendo emerged from this creative effervescence that would dominate the following decade. The arcade game industry reached economic maturity.

The technical constraints of the original system—incapable of filling

more than a quarter of the screen in real time—paradoxically stimulated innovation. Namco responded in 1979 with *Galaxian* and its sprite technology, which managed moving objects more efficiently. This technological arms race never ceased: digital signal processors for 3D in the 1990s, then specialized graphics processors at the turn of the millennium.

Beyond entertainment, *Space Invaders* crystallized the geopolitical tensions of its time. 1970s America viewed Japan's economic ascent with apprehension. The pixelated invaders echoed fears of an "invasion" of Japanese products into the American market. Ironically, Atari acquired the game's rights for the United States, illustrating the complexity of exchanges between the two powers.

This cultural mutation came with professionalization of the sector. Gone were the days of solitary tinkerers: developing a game now required a structured team. Programmers, hardware engineers, designers, graphic artists, and producers formed the first proper studios. This organization marked the beginning of the sprawling industry we know today.

The game's social impact surprised its creators. Spectators gathered around arcade machines to admire virtuoso performances. These spontaneous gatherings held the seeds of video game spectacle culture that would explode with contemporary e-sports. *Space Invaders* revealed the collective dimension of a medium thought to be solitary.

Its influence endures in current creations, notably the defensive principle of protecting one's territory against waves of assailants. Score, progression, mounting tension: so many elements that have become inseparable from the video gaming experience. Nishikado's game demonstrates how creative intuition, supported by technical innovation, can transform an entire sector.

Space Invaders, this small program of a few kilobytes, redefined our relationship with digital entertainment. It marks the moment when video games emancipated themselves from their artisanal origins to establish themselves as an autonomous form of expression, capable of generating their own aesthetic and social conventions.

TEX

Donald Knuth once received the proofs for the second volume of *The Art of Computer Programming*. It was 1977, and the result disappointed him so much that he decided to stop everything. The typographic compo-

sition of his mathematical formulas appeared so mediocre that he chose to create his own system rather than continue writing his books. The printing industry was undergoing a technical upheaval: it was abandoning hot metal typesetting, where each character was cast in metal and assembled by hand, to adopt phototypesetting. This new method was cheaper and faster, certainly, but Knuth would not accept the decline in quality that came with it.

The Stanford computer scientist devoted his 1977-1978 sabbatical year to developing TEX. He did not simply write a page layout program: he designed a complete system with its document description language and its DVI output format, independent of the printer used. His first trials took place on a DEC PDP-10 computer, in the SAIL language. Frank Liang and Michael Plass, his students, tested successive prototypes and actively participated in the development.

TEX redefined the art of typesetting through its sophisticated algorithms. Frank Liang developed a hyphenation system that adapts to different languages, while Michael Plass helped Knuth design paragraph justification. The technical innovation lies in the concept of "boxes and glue": TEX manages the spaces between elements like virtual springs that compress or stretch according to needs. This approach produces justification of unmatched quality.

Knuth accompanied TEX with a second system: METAFONT. Rather than drawing letter outlines as PostScript and TrueType would later do, METAFONT programs characters by simulating pen movements. This parametric method generates entire font families by modifying a few variables. The Computer Modern fonts, created with this tool, have visually marked all TEX documents for forty years.

In 1982, the rewrite in Pascal gave birth to TEX82. This language change improved portability and served as the foundation for all subsequent versions. Knuth then adopted an original version numbering system: the numbers tend toward π for TEX and toward e for METAFONT. This mathematical approach reflects his desire to definitively stabilize the system. He committed to correcting only serious bugs, guaranteeing that a document would always produce an identical result, even decades later.

The American Mathematical Society placed its trust in TEX for its publications. This adoption by a reference institution contributed to its dissemination in universities around the world. Leslie Lamport understood that Knuth's system was too complex for many users. In 1984, he created LATEX, a layer of simplified commands that hides technical complexity behind a logical interface. The author could now focus on the structure of their document rather than its formatting, and its success

surpassed that of "plain" TEX.

Knuth also reinvented software documentation. He developed specifically for TEX the principles of literate programming, blending code and explanations in a single document. This approach produced exceptionally complete documentation that helps other programmers understand and modify the system. A user community formed around TEX. The TEX Users Group was founded in 1980 and published TUGboat, a technical journal that accompanied the system's evolution.

TEX remains the reference tool for scientists. Mathematicians, physicists, and computer scientists use it extensively to write their papers. Extensions like pdfTEX, XeTEX, and LuaTEX have added new functions without altering the original typographic quality. Paradoxically, Knuth's innovations have never been adopted by mainstream software. Microsoft Word or LibreOffice Writer offer significantly inferior typesetting despite the power of current computers.

The story of TEX shows how one man's demand for quality can change an entire discipline. It demonstrates the importance of open source code and exemplary documentation for a software's survival. In a world where technologies succeed one another at a frantic pace, TEX crosses decades while maintaining all its relevance. Knuth had planned to spend a few months on this personal project. Nearly half a century later, his work continues to typeset the most beautiful mathematical or literary texts on the planet.

WordStar

In May 1978, Seymour Rubinstein stormed out of IMSAI Manufacturing Corporation. A disagreement with Bill Millard had just shattered his career as marketing director. Two weeks earlier, Rob Barnaby, a talented programmer, had already jumped ship. Rubinstein tracked him down and made him a simple proposal: create software together for these microcomputers starting to emerge from California garages.

June 1978 saw the birth of MicroPro International Corporation. The name reflected Rubinstein's ambition: to develop "PROfessional" programs for "MICRO-computers". The first fruits arrived as early as September with WordMaster and SuperSort. WordMaster built on the ideas of a text editor that Barnaby had developed at IMSAI, completely rewritten to run on CP/M. SuperSort drew inspiration from IBM 360 sorting utilities.

These two programs brought in $10,700 in September 1978 alone, enough to face the future with confidence.

But resellers were demanding something else: a real word processor that handled printing directly. In early 1979, Barnaby embarked on a technical tour de force. In six months, he wrote over 137,000 lines of assembly code. IBM would have estimated this work at 42 man-years. The first version of WordStar was released in June 1979.

The software disrupted established practices. For the first time, the screen showed what would appear on paper. WordStar integrated context-sensitive help that adapted to the user's actions, guiding their steps without cluttering the interface. Typists found familiar ground: all commands remained accessible from the keyboard, and page breaks displayed on screen, an unheard-of luxury.

Success exceeded all expectations. MicroPro climbed from $500,000 in 1979 to $1.8 million in 1980, then $5.2 million in 1981. The arrival of venture capitalist Fred Adler in 1981 transformed the company. A real sales force was put in place. Revenue jumped to nearly $25 million, with a loss of one million for the fiscal year. In 1982, revenues reached $45 million.

Rubinstein thought big and aimed for international markets. He established European headquarters in Zug, Switzerland, to take advantage of favorable tax treatment. Offices opened in Germany, the United Kingdom, France, and Japan. WordStar was released in 42 languages. The entire world discovered this futuristic word processor.

WordStar inherited the philosophy of text editors rather than that of typewriters. Text was now a continuous flow, no longer a succession of pages. Formatting commands were inserted directly into the document via special characters, a method borrowed straight from programming. This approach appealed to academics and writers who handled lengthy manuscripts.

In January 1984, Rubinstein collapsed from a heart attack as the company prepared for its initial public offering. The company's lawyers persuaded him to sign documents that stripped him of control. H. Glen Haney took the reins and gradually sidelined the founder.

The new management accumulated missteps. WordStar 2000 was released in total incompatibility with the original version, sowing confusion among loyal users. Advertising for WordStar disappeared for three years. The network of 1,500 resellers was sacrificed in favor of a few large distributors. Sales plummeted from $72 million to $40 million.

WordPerfect arrived in 1980 by combining editing and formatting in a single display. Microsoft Word arrived in 1983 with the novel approach

that each character is an independent entity with its own attributes. This method, initially slower, would come into its own with the emergence of graphical interfaces.

The 1990s sounded the death knell for WordStar. The transition to Windows went poorly. The software passed from hand to hand, acquired by companies specializing in the distribution of low-cost programs, and finally by The Learning Company. Corel obtained a license but preferred WordPerfect, which it had just acquired. WordStar survived thanks to a few groups of nostalgic users.

Yet the legacy remains immense. WordStar proved that a microcomputer could rival professional typewriters. Its interface innovations inspired an entire generation of software. It helped structure the software industry by creating user license agreements and developing international distribution.

dBASE

At NASA's Jet Propulsion Laboratory in the 1970s, Wayne Ratliff probably had no idea he was about to write a chapter in the history of personal database systems. His JPLDIS system, designed to process information for the space laboratory, would become the ancestor of one of personal computing's most significant software products.

Ratliff transformed his initial work into a product he named Vulcan, which he sold through classified ads in specialized press. The commercial venture took a different turn when Ashton-Tate decided to distribute the software under a new name: dBASE II. This naming convention, which cleverly avoided version I, immediately suggested the maturity of an already proven product. The strategy worked perfectly. On 8-bit microcomputers running CP/M, dBASE II became the essential reference for data management.

What distinguished dBASE from other tools was its comprehensive approach. Where competitors offered isolated functions, dBASE provided a fully integrated work environment. Once launched, users accessed a complete universe: table creation, data manipulation, information retrieval, export to other formats. The software went further by offering a programming language to automate repetitive tasks or build custom interfaces.

This integration pushed the concept to include a text editor, file management tools, and the ability to interact directly with the operating sys-

tem. Everything happened within a single environment, without having to juggle between different programs. The command-line interface, recognizable by its famous dot prompt, became increasingly familiar to users who learned to interact directly with their data.

The technical architecture relied on logical information organization. Data tables, stored in .dbf files, structured information into columns and rows following the classic relational model. The system relied on different file types: indexes to speed up searches, memory for temporary calculations, report formats for presentation, screen formats for interaction, programs for automation.

dBASE III marked a major technical breakthrough. Maintenance and code evolution were facilitated following Ashton-Tate's abandonment of assembly language in favor of C. This version introduced date management and significantly increased processing capabilities. dBASE III Plus, released in 1986, reached a new level with pull-down menus inspired by Framework, doubled performance, and native multi-user support.

The numbers speak for themselves: unlimited files, up to one billion records per file, 128 fields maximum per record, 4 KB per record. Tests showed that sorting was twice as fast as the previous version, while indexing gained a factor of ten. Compatibility extended to PFS:FILE, Lotus 1-2-3, VisiCalc DIF, MultiPlan, and standard ASCII files.

Networking took on particular importance. dBASE III Plus directly integrated multi-user functionality at no extra cost. Automatic record and file locking prevented data corruption during simultaneous access. Eight levels of password protection secured access to sensitive information. Data encryption and decryption occurred automatically at each workstation level.

Yet by the late 1980s, the machine began to seize up. Wayne Ratliff's departure from Ashton-Tate in 1988 deprived the company of its original creator, while competition intensified with the arrival of products like FoxBASE and Clipper, often more efficient and less expensive. Legal problems mounted: Ashton-Tate lost several copyright infringement lawsuits, having never disclosed that dBASE descended directly from JPLDIS.

Unlike WordStar, which had successfully recalled its founders during similar difficulties, Ashton-Tate persisted in a strategy that proved losing. The market reorganized around new players: Borland absorbed Ashton-Tate, Microsoft acquired Fox (FoxBASE's publisher), Computer Associates purchased Nantucket (Clipper's creator).

After his departure, Wayne Ratliff developed Emerald Bay, a new product that bore troubling resemblances to Microsoft Access. In a 2007 interview with Visual Pro Magazine, Ratliff highlighted these similarities

without making explicit accusations, leaving doubt about potential "borrowing" of concepts.

dBase's integrated development environment concept, direct interaction with data through an intuitive user interface, data-oriented programming: all innovations that would leave a lasting mark on the industry. Microsoft Access, among others, directly inherited these pioneering concepts.

Jerry Pournelle perfectly captured the product's ambiguity in BYTE magazine's columns by describing it as "exasperating but excellent". This phrase sums up the mindset where the personal software industry still hesitated between pure technical innovation and ease of use. dBASE embodied this tension: innovative in its concepts, sometimes bewildering in daily use. Above all, dBASE demonstrated how database management systems became indispensable tools, both in personal and professional computing.

Ada

Picture for a moment the administrative nightmare that Pentagon IT projects represented in the mid-1970s. Over 400 programming languages and dialects were proliferating throughout the U.S. Department of Defense systems, generating three billion dollars in maintenance costs annually. This absurd situation often originated from the misguided initiative of a programmer who, convinced they were improving productivity, would cobble together an existing language for their particular application. Twenty years later, entire generations of developers still had to learn this dialect to maintain what had become a *legacy* program.

David Fisher was then leading the DoD's initiative to break out of this deadlock. The idea wasn't new: in the 1960s, the department had already mandated COBOL in its defense contracts. But this time, the approach would be different. For the first time in the history of programming languages, the requirements would be defined by a team completely separate from the one that would design the language.

The consultation process mobilized military, industrial, and academic experts from around the world. Between 1975 and 1977, several requirements documents emerged with evocative codenames: Strawman, Woodenman, Tinman, and finally Ironman. These specifications placed reliability, readability, and maintainability at the heart of concerns, far beyond the classic objectives of portability and efficiency.

Twenty teams responded to the call for proposals launched in 1977. Four were selected and designated by colors: Green, Red, Blue, and Yellow. After six months of hard work, only the Green and Red teams remained in the running. In May 1979, the Green team's proposal won. Its leader, Jean Ichbiah of CII Honeywell Bull, had an ace up his sleeve: his experience with LIS, the "System Implementation Language" developed since 1972 at his company.

LIS had been designed to improve operating system reliability and maintainability. This philosophy fit perfectly with the DoD's requirements. Ichbiah and his team drew on this experience to shape their proposal.

The next phase, called "Test and Evaluation", transformed the project into a true global laboratory. About a hundred teams spread across all continents tested the language by recoding existing applications. Their feedback fueled successive refinements that led, in 1980, to a proposed standard.

The standardization process took on a pharaonic dimension. Over a thousand people around the world participated in this endeavor. Ichbiah's team had to process approximately 7,000 comments on the proposed standard, relying on a computerized database to manage this flood of feedback. A technical feat.

In February 1983, Ada's ANSI standardization was finalized. The language stood out for remarkable innovations. Its package structure allowed a clear separation between the user interface and the implementation. The concept of linear reading transformed code comprehension: a programmer could read an Ada program line by line, their understanding at any given line depending only on the preceding lines.

Robert Dewar developed the first Ada compiler at New York University, actually an interpreter intended for teaching. The second compiler, created by Rolm and Data General for the Eclipse minicomputer, was validated in June 1983. Western Digital offered the third with their MicroEngine, marking Ada's entry into the microcomputer world.

In January 1984, the DoD struck a major blow: a directive mandated the use of Ada for all critical applications. This decision came after the validation of the Data General compiler, which proved the language's practical viability. But Ada did not remain confined to military applications. The civilian sector adopted it, particularly in civil aviation, rail systems, and real-time embedded systems where reliability is paramount.

The language evolved over the decades. Ada 95 introduced object-oriented programming. Ada 2005 and Ada 2012 brought substantial improvements in contract-based programming. Today, Ada powers critical

applications where the slightest failure can have potentially dramatic consequences.

Ada's design introduced practices that have now become standard: rigorous separation between specification and implementation, formal compiler validation, meticulous standardization process. These innovations influenced the design of other languages and transformed software development methods.

Jean Ichbiah had a clear vision: "Developing a large program may take less than two years, but its maintenance will extend over more than twenty years." This philosophy permeates every aspect of the language. Ada prioritizes code clarity and comprehension over ease of writing. An approach that seems counterintuitive at first but proves worthwhile in the long run.

The language's name pays tribute to Augusta Ada Lovelace, considered the first programmer in history. This choice was not insignificant: it reflected the project's ambition to make history in computing. Forty years later, Ada is regarded as a model of rigorous and methodical design. It demonstrates that investing in code quality generates substantial savings over a software's entire lifecycle. A lesson that many IT projects would do well to consider.

Usenet

The fall of 1979 at Duke University looked like any other: leaves were yellowing, students were returning to classes, and in the computer science department, Tom Truscott and Jim Ellis were wracking their brains over an apparently mundane problem. How could they efficiently connect the different computers on campus running UNIX? A technical question that would revolutionize the way people exchange information.

Computing still lived under the domination of large centralized systems. DEC's PDP-11s were just beginning to colonize university departments, while personal microcomputers remained technological curiosities. For remote communication, people used sluggish 300-baud modems that crackled over telephone lines. ARPANET existed, certainly, but its access remained jealously guarded by the military and a few privileged laboratories.

Truscott and Ellis found their solution by teaming up with a student at neighboring University of North Carolina: Steve Bellovin. The latter de-

veloped the first version of Usenet in just 150 lines of shell script, building on UUCP, a UNIX protocol that had just appeared and allowed copying files between remote machines. The idea was simple: create thematic forums called newsgroups where anyone could post messages, which would automatically propagate from one site to another.

The chosen architecture reflected the constraints of the era as much as its creators' ingenuity. Rather than centralizing data, each site locally stored all messages and transmitted them to its neighbors. Duke served as a central hub to limit telephone costs, a very real concern when each long-distance call counted in dollars. The A News protocol adopted a rudimentary but effective format: each message began with the letter A followed by a unique identifier, and headers indicated the path traveled to prevent messages from looping endlessly.

In January 1980, at a Usenix conference in Boulder, the team publicly presented their system. They described it as a "poor man's ARPANET", an economical alternative to the elite network. The source code was distributed freely on magnetic tape, in the spirit of sharing that characterized the UNIX community. This generosity wasn't purely philosophical: without financial means, the creators bet on network effects to make their invention prosper.

The beginnings were nevertheless modest. In April 1981, only fifteen sites comprised the network, orbiting around Duke, Berkeley, and San Diego. The takeoff came from Mary Ann Horton, who had the idea of creating a gateway between Usenet and popular ARPANET mailing lists. Suddenly, discussions from SF-LOVERS and other renowned groups became accessible to Usenet users. Horton also developed the uuencode and uudecode tools, which solved the thorny problem of transferring binary files over a network designed for text.

The B News version corrected the original system's growing pains. Gone were the days of having to read messages in arrival order: users could now freely navigate discussions, follow conversation threads, organize their reading. These improvements came at just the right time because the network was literally exploding. From 150 sites in 1981, Usenet jumped to 11,000 sites in 1988, processing 1,800 daily articles.

This growth reflected the broader evolution of computing. UNIX was spreading through universities and businesses, modems were getting faster, telephone lines more reliable. Above all, Usenet answered the need for technical mutual aid and knowledge sharing. Groups proliferated, covering every imaginable topic, from the most specialized to the most trivial.

But this expansion revealed the flaws of an architecture designed for a few dozen sites. Central nodes struggled to relay the growing flood of

messages. Governance posed questions: while creating an official group required a community vote, the "alt." hierarchy let everyone freely create their own forums. This organized anarchy fostered innovation but complicated controlling abuses.

The 1990s paradoxically marked both Usenet's peak and the beginning of its decline. The system served as a platform for announcements that would change the world: Linus Torvalds presented Linux there, Tim Berners-Lee described the World Wide Web. Ironically, it was precisely the Web that would supplant Usenet by offering a more accessible interface to the general public. The arrival of AOL, then social networks, gradually diverted users toward more user-friendly solutions.

Usenet didn't disappear entirely. The NNTP protocol replaced UUCP to adapt to the Internet, and data volume continued to grow, mainly for binary file sharing. Technical groups retain their usefulness for developers who appreciate the quality of discussions and absence of advertising.

Long before these concepts were fashionable, Usenet proved that decentralized and participatory social networks could be created. It enriched Internet vocabulary with words like "spam" or "troll", introduced the idea of discussion threads, experimented with collaborative moderation. Its technical choices have durably influenced our way of communicating online.

Technological revolutions sometimes arise from very concrete needs, and from solutions cobbled together by a few students in a university laboratory. Between total openness and necessary control, between freedom of expression and moderation of abuses, Usenet embodied the tensions that still run through our digital communities.

VisiCalc

In 1978, Dan Bricklin was attending classes at Harvard Business School when an ordinary situation sparked an idea that would transform personal computing. During a case study on Pepsi-Cola, he watched his professor fill in a financial spreadsheet on the blackboard. With each data modification, the instructor had to laboriously recalculate all the figures, erase, and rewrite. This scene reminded Bricklin of his own calculation sessions with his calculator: scribbled sheets of paper, errors creeping into transferred values, tedious restarts.

The student had solid technical experience from Digital Equipment

Corporation, where he worked on digital typesetting systems. He knew about computer systems designed to reduce the number of operator manipulations. This experience led him to envision what he would later call an "electronic blackboard and chalk." A system where calculations display, modify, and automatically propagate from one cell to another.

Back in his dorm room, Bricklin cobbled together a first prototype in BASIC on a borrowed Apple II. His program managed a small grid of twenty rows by five columns, but the principle worked. He reconnected with Bob Frankston, a friend from MIT with whom he had worked on Multics, the experimental operating system that familiarized them with sophisticated user interfaces. Frankston possessed the technical skills necessary to transform the idea into a commercial product.

Their meeting with Dan Fylstra was a game changer. Fylstra ran Personal Software and wrote for *Byte Magazine*. He immediately grasped the project's potential and steered the two men toward the Apple II, a machine that had the decisive advantage of a disk drive. Without external storage, a spreadsheet would remain unusable. Fylstra also suggested adapting the software to HP85 and HP87 programmable calculators, but this avenue led nowhere.

In January 1979, Bricklin and Frankston founded Software Arts Corporation. To develop their spreadsheet, they acquired a Prime 350 minicomputer, chosen for its connection to Multics and its support for the PL/I language. This cross-development approach, unusual for the time, gave them access to tools far more powerful than those available on microcomputers. They wrote and compiled code on the Prime, then transferred it to the Apple II for testing.

Hardware constraints dictated their technical choices. The Apple II provided 48 KB of RAM, part of which was used by the operating system. Every byte mattered. Rather than using faster binary arithmetic, they opted for decimal calculation. This decision sacrificed performance but avoided rounding errors that would have unsettled users accustomed to their desk calculators.

The interface they designed was striking in its apparent simplicity. A grid of cells identified by coordinates: A1, B2, C3. Each cell could contain a number, text, or a formula. Users pointed to other cells rather than manually entering their values. When data changed, all calculations updated instantly. This simulation feature—the famous "what if"—simplified exploring different scenarios with a single gesture.

This design stemmed from their experience with large computer systems and their desire to create a tool accessible to the general public. The coordinate notation actually resulted from deep reflection on ergonomics.

How to reference a cell intuitively? How to make calculation logic visible?

Personal Software released VisiCalc in May 1979 at a price of $100. Success didn't come immediately, but it came quickly. In July, Ben Rosen published an analysis in the *Morgan Stanley Electronics Letter* that would prove prophetic. VisiCalc was the first "killer app" of personal computing, the first software that let you use a computer without knowing how to program. Buyers purchased an Apple II solely to run VisiCalc.

The program found its place on other machines. The 1981 IBM PC version employed the same optimization techniques as the original, demonstrating the soundness of the initial choices. But success attracted competition. Microsoft launched Multiplan in 1982. The software struggled in the United States but found success elsewhere in the world.

The arrival of Lotus 1-2-3 in 1983 reshuffled the deck. Mitch Kapor, a former Personal Software employee, had developed a spreadsheet specifically optimized for the IBM PC. His program integrated graphics, managed databases, named cells, and automated tasks with macros. Lotus 1-2-3 dominated the professional world.

VisiCalc couldn't withstand this renewed competition. Relations between Software Arts and Personal Software deteriorated. Each company developed its own projects: VisiOn for Personal Software, TK!Solver for Software Arts. Conflicts escalated into cross-lawsuits in 1984, paralyzing product development. The advanced version of VisiCalc arrived too late in an already conquered market.

The legacy of the first electronic spreadsheet transcends its commercial success. VisiCalc proved that a microcomputer could serve as a serious professional tool. It accelerated business computerization and permanently influenced user interface design. Microsoft Excel, launched in 1985 on Macintosh and in 1987 on Windows, adopted its fundamental principles while adding macro programming capabilities.

Technically, VisiCalc established some practices that became standard: cross-development between development machine and target platform, code optimization for limited resources, the importance of an intuitive interface. But this ease of use sometimes conceals underlying complexity. Studies reveal that most spreadsheets contain errors, with approximately 3% of cells being incorrect on average. These mistakes can be costly in financial terms or strategic decisions.

The VisiCalc adventure also illustrates the evolution of the software industry. The collaboration model between technical developer and commercial publisher, initially perceived as an asset, strained relationships when the market evolved. The separation between creation and commercialization, which seemed natural, revealed its limitations in the face of

technological changes.

Forty years after its creation, VisiCalc continues to inspire. Its principles—polished user interface, resource optimization, balance between innovation and concrete needs—remain relevant. Its story reminds us that software succeeds as much through its technical design as through its ability to solve a real problem.

VII

1980

Computing Enters the Home

The 1980s opened on a fractured world. The Cold War had lost none of its vigor, and the East-West opposition continued to structure international relations. Yet another battlefield was taking shape: the mastery of information. The United States and the USSR pursued their space and military duel while Japan, an emerging economic power, conquered market after market through its electronics industry.

Information had become a strategic issue. The breakup of AT&T in the United States in January 1984 transformed the telecommunications sector. This break from the monopolistic model stimulated innovation. On the European side, people began questioning the grip of the PTTs on communications. Winds of change were sweeping through the economic landscape.

The service sector, expanding rapidly in Western economies, demanded appropriate tools for offices. Typists put away their typewriters, replaced by word processing systems. Accountants abandoned their calculators for electronic spreadsheets. Productivity was the watchword in the face of increasingly fierce international competition.

The purchasing power of Western households reached unprecedented levels. Color televisions now sat in most living rooms. This familiarity

with consumer electronics prepared the ground for the arrival of other devices, including game consoles, video recorders and, above all, microcomputers. Computing, previously confined to businesses, was inviting itself into private homes.

Schools did not escape this technological wave. Political leaders grasped the importance of training young people in these new tools. The French plan "Computers for All" launched in 1985 illustrated this awareness. Thousands of Thomson TO7 and MO5 computers invaded schools. Other countries followed similar paths, creating a substantial educational market.

In businesses, two worlds coexisted. Large centralized systems remained present, but saw their hegemony challenged by microcomputers. Users gained independence, much to the dismay of traditional IT departments. This decentralization raised unprecedented questions about compatibility between all these systems and data sharing. Standardization was the solution.

The financial world shifted into the electronic era. Stock exchanges abandoned manual quotation for computerized systems. Transactions accelerated, their volume exploded. However, this automation revealed its limits during "Black Monday" on October 19, 1987. Automated selling programs spiraled out of control, amplifying the ongoing crash. The incident revealed the fragility of overly autonomous systems.

Communication networks modernized at breakneck speed. Optical fiber progressively replaced copper. Telecommunications satellites multiplied in orbit. In companies, people no longer settled for isolated computers—they connected them. The first commercial electronic messaging systems appeared, harbingers of an interconnected world.

Popular culture seized upon the computing phenomenon. Hollywood portrayed hackers and artificial intelligences. *Tron* (1982) plunged its viewers inside a computer. *WarGames* (1983) evoked the risk of a nuclear war triggered by a computer error. Literature was not to be outdone with the emergence of cyberpunk, of which William Gibson and his *Neuromancer* (1984) were the leading figures.

Software publishing established itself as a major economic sector. The multiplication of platforms created a fragmented market where each machine had its own ecosystem. Publishers competed ingeniously to win over the general public. Ergonomics became a decisive selling point. In the shadow of this commercial effervescence, the first computer viruses made their appearance, heralds of future digital battles.

A new relationship with technology developed. Computing was no longer the preserve of specialists in white coats. Amateur clubs flourished,

magazines like *Hebdogiciel* and *Byte* disseminated know-how and programs to type in yourself. This democratization was accompanied by a particular philosophy when Richard Stallman launched the GNU project in 1983, the first stone in a free software movement that challenged the grip of commercial publishers.

Silicon Valley became the symbol of a new generation of entrepreneurs. Young people like Steve Jobs, Bill Gates, and Michael Dell disrupted established giants. The image of the computing genius working in a garage before conquering the world became embedded in the collective imagination. Investors flocked in, sensing the sector's growth potential.

States gradually became aware of the sovereignty issues linked to computing. Controlling digital technologies meant ensuring independence. Mastery of electronic components, operating systems, and networks was a strategic objective. Export restrictions on sensitive equipment to the Soviet bloc testified to this geopolitical dimension.

This decade initiated a transformation of society. The computer was a tool for scientific calculation or management, and a means of creation, communication, and entertainment. Its grip extended to all domains of human activity.

The end of the 1980s revealed a glimpse of a world where information would circulate without barriers. Technical innovations responded to the social aspirations of the era, between the desire for individual autonomy, the need for increased exchanges, and a thirst for modernity. Personal computing fit into this dynamic, translating into silicon the dreams of a generation.

Apple III

In May 1980, at the National Computer Conference in Anaheim, Apple unveiled its new machine with unusual fanfare. The company rented out Disneyland for five hours, spent $42,000, and chartered British double-decker buses to transport 7,000 conference participants. This grandiose staging announced the Apple III, Apple's first computer aimed at professionals.

Times seemed promising. By late 1978, the Apple II was selling beyond all expectations. Silicon Valley engineers competed for positions at Apple, some accepting pay cuts in exchange for stock options. Richard Jordan, who left Hewlett-Packard during the summer of 1978, recounts this period

of collective euphoria. With each halving of the stock price, teams felt invincible. Failure seemed impossible to them.

The project began in this exhilarating context. The Apple III featured a Synertek 6502A 8-bit microprocessor running at 2 MHz, twice the speed of the Apple II. Supporting up to 128 KB of RAM, it included a keyboard with integrated numeric keypad and a Shugart 5.25-inch floppy drive with 143 KB capacity. Four internal expansion slots remained compatible with Apple II cards, complemented by two serial ports. The machine could run in Apple II emulation mode, but revealed its true nature with its proprietary operating system SOS (*Sophisticated Operating System*), which insiders nicknamed "applesauce". An integrated real-time clock completed the equipment, while the display handled 24 lines of 80 columns in text mode and 560 by 192 pixels in monochrome graphics.

Steve Jobs led the project and imposed his vision. No fan: the aluminum chassis would suffice to dissipate heat. The size and shape of the case were decided without consulting the engineers, who then had to cram components into a space that was too small with inadequate ventilation. Feature creep worsened the situation: marketing, engineering, industrial design, manufacturing—each department added its requirements. The product swelled beyond its initial design.

Jerry Manock and Dean Hovey, the industrial designers, conceived the chassis before the motherboard was finished. Manock anticipated strict FCC standards on electromagnetic interference and chose a rather solid aluminum chassis. He contracted with Doher-Jarvis, an automotive parts manufacturer in Toledo. The external aesthetics—45-degree bevels, tilted keyboard, brown color—were meant to create a "house identity" for future Apple products.

Delays threatened the initial public offering scheduled for December 1980. Managers ignored engineers' warnings and launched production. By November, during the first deliveries, the nightmare began. In February 1981, Apple abandoned the real-time clock: the National Semiconductor chip failed to deliver on its promises. The price dropped to $4,190, with a $50 rebate for early customers.

March 1981 marked the beginning of volume deliveries and the full extent of the disaster. Twenty percent of machines arrived dead at customers' sites, with chips having dislodged during transport. Those that initially worked soon failed: heat caused components to expand and pop out of their sockets. Apple then recommended a solution that would become legendary: lift the front of the computer fifteen centimeters and let it drop to reseat the chips.

Other defects accumulated. Faulty connectors, case screws that pierced

internal cables, a motherboard so dense it caused short circuits. The rush also affected software: programmers discovered the machine only nine weeks before shipment. Manuals were proofread the day they were sent to the printer, allowing so many errors through that an addendum became necessary.

Apple attempted a rescue in November 1981 with a revised version at $3,495. The company maintained that problems stemmed from manufacturing and quality control, not design. Yet the new version incorporated different chip sockets, updated software, memory expandable to 256 KB, and offered an optional 5 MB hard drive. Of the 7,200 original Apple IIIs, 2,000 were replaced free of charge.

Sales remained disappointing. Analysts estimated that Apple sold 3,000 to 3,500 units monthly, one-tenth of Apple II sales. In December 1983, InfoCorp counted 75,000 installed units versus 1.3 million Apple IIs. The poor reputation and lack of software exploiting SOS deterred buyers.

A final stand came in December 1983 with the Apple III Plus at $2,995. This version included 256 KB of RAM as standard, a functional clock, a new motherboard, SOS version 1.3, improved ports with DB-25 connectors, and a case facilitating card installation. Too late: on April 24, 1984, Apple discontinued development. After $60 million in losses, the line disappeared definitively from the catalog in September 1985.

Ironically, this failure did not diminish Apple's success. On December 12, 1980, the initial public offering broke all records. Shares issued at $22 climbed to $29 and sold out within minutes. At the close of the first day, Apple was worth $1.2 billion, achieving the largest IPO since Ford in 1956. Jobs's 15% stake would soon exceed $250 million.

The Apple III experience shattered the myth of invincibility born from the Apple II's success. This setback forced a restructuring. In January 1981, president Mike Scott divided development into three groups: personal desktop systems (Apple II/III), peripherals (printers, modems), and professional systems (Lisa division). The R&D budget tripled to $21 million. These tensions cost Scott his position, replaced by Mike Markkula, while Jobs took the chairman role.

IBM's entry into the personal computer market in 1981 would paradoxically serve Apple. The giant legitimized this emerging sector. Apple acknowledged this with humor in a full-page advertisement in the *Wall Street Journal*: "Welcome IBM. Seriously".

Intel 8086/8088

When Intel decided to take the leap into 16-bit processors in the late 1970s, the company had no idea it would become a technological empire. The 8086, launched in 1978, entered a market dominated by Intel's 8-bit chips, the 8080 and 8085. This transition to 16 bits represented a considerable technical challenge, but Intel adopted a bold strategy: maintain compatibility with existing systems while multiplying performance.

The new processor pushed the boundaries of its time. Where its predecessors maxed out at 64 kilobytes of memory, the 8086 addressed up to 1 megabyte through its 20-bit address bus. Its 16-bit data bus processed double words in a single cycle. Intel designed a dual-unit architecture: the bus interface unit handled external exchanges while the execution unit focused on calculations. This division of labor enabled pipeline operation, where the processor could begin processing the next instruction before completing the previous one.

Manufactured in HMOS technology with its 29,000 transistors, the 8086 ran at 5 MHz in its base version. Intel later pushed the frequency to 8 and then 10 MHz. Its fourteen 16-bit registers were cleverly distributed among data, pointers, flags, and segments. The latter represented the most notable innovation: they divided memory into distinct zones and facilitated program organization.

But here's the catch. Migrating from 8-bit to 16-bit was expensive for manufacturers. They had to redesign motherboards, adapt all peripheral components, and revise machine architectures. Faced with market resistance, Intel released a stripped-down version in 1979: the 8088. This chip retained the 8086's internal 16-bit architecture but reduced its external data bus to 8 bits.

The compromise seemed awkward at first glance. The 8088's 16-bit transfers required two cycles instead of one, limiting performance. Yet this technical limitation became a commercial advantage because manufacturers could reuse their existing 8-bit components: controllers, memory, support circuits. Development costs dropped dramatically.

IBM seized the opportunity in 1981. For its first personal computer, the company sought a fast and economical solution. The 8088 was a natural choice. It worked perfectly with the 8288 bus controller, the 8259A interrupt manager, and the entire range of proven circuits. Its 6-byte cache partially compensated for the bottleneck of the reduced bus.

As the market standard, the IBM PC became a massive success within a few years. Microsoft developed MS-DOS specifically for this architecture, creating a software ecosystem that locked out the competition. IBM

clones proliferated, all equipped with the 8088. Intel's chip became the best-selling processor, far surpassing its more powerful sibling, the 8086.

This commercial success masked the underlying technical shift. The 8088/8086 architecture invented concepts that would span decades. Memory segmentation would survive in all subsequent x86 generations. The extended instruction set, with its string operations and hardware multiplication, would serve as the foundation for future extensions. The 80286's protected mode would build on these foundations.

A true trademark of the x86 family, Intel planned from the outset for both models to coexist. Same instruction set, same programming, only the external bus differed. Each new processor would execute software from its predecessors while adding its own innovations.

For sophisticated calculations, Intel offered the 8087 math coprocessor. This optional circuit connected directly to the main processor and handled floating-point operations. Its specialized instruction set would later be natively integrated into chips starting with the 80486.

The 8086/8088's modular architecture also enabled multiprocessing. In maximum mode, multiple chips shared the system bus via the 8288 controller. This possibility, ignored in consumer PCs, found its place in industry and servers.

Production spanned more than ten years. Superseded by more powerful processors, the 8086 and 8088 still equipped embedded systems and industrial controllers for a long time. Their simplicity and energy efficiency suited these discrete applications perfectly.

Forty years later, their influence endures. By establishing x86 as the de facto standard for personal computers, they charted a technological path that continues today. The need to preserve compatibility with their instruction set still weighs on contemporary processors.

This backward compatibility, initially perceived as a burden, became an ultimate weapon. It guaranteed developers and users that their programs would survive hardware evolution. This stability fueled the expansion of the IBM-compatible PC market.

The triumph of the 8088 in the IBM PC illustrates an eternal truth of the technology industry: the theoretically optimal solution doesn't always win. Economic pragmatism and practical constraints often dictate choices. Intel understood this and built its strategy on this lesson.

Smalltalk-80

By the late 1970s, Xerox PARC already housed several versions of Smalltalk, but these creations remained confined within the laboratory walls. They only ran on Xerox's specialized hardware: the Alto and Dorado, expensive machines inaccessible to ordinary computer scientists. This frustrating situation prompted Adele Goldberg to take charge of an ambitious project: bringing Smalltalk out of its gilded cage so it could flourish on other architectures.

The undertaking proved delicate. First, they needed to formalize with surgical precision the workings of a Smalltalk "virtual machine" that would serve as a universal foundation, much like Java's would many years later. This technical abstraction concealed a novel concept: everything is an object in this new universe. Numbers, characters, and classes behave as autonomous entities capable of receiving and sending messages. This radical uniformity disrupted the habits of programmers accustomed to carefully separating data and procedures.

In this unified world, inheritance weaves invisible links between classes. A hierarchy emerges naturally, where each descendant enriches or specializes the behaviors of its ancestors. Class methods orchestrate shared behaviors while instance methods give each object its own personality. This architecture fostered a form of code reuse previously unknown.

The development environment broke with the traditional cycle that required editing, compiling, then executing. Smalltalk-80 offered a continuous experience where programmers modified their code while the program was running. This fluidity transformed the act of programming into a direct conversation with the machine. The class browser and debugger, themselves written in Smalltalk, were living tools that users shaped according to their needs.

The graphical interface introduced innovative concepts for the time: windows overlapped, menus appeared with right-clicks, objects were manipulated directly on screen. These innovations established a visual language that still influences our daily interactions with computers. Steve Jobs grasped this immediately when he visited PARC in 1979; he understood the potential of these ideas for the general public.

Under the hood, automatic memory management finally freed programmers from a tedious technical chore. The garbage collector tracked abandoned objects to recycle their memory. Peter Deutsch and Allan Schiffman refined these mechanisms until they achieved acceptable performance on standard hardware, proving that conceptual elegance could coexist with practical efficiency.

Implementation across different platforms demanded engineering prowess. The PARC team designed a virtual machine with its own instruction set, leaving integrators to translate these abstract commands into native code. This strategy guaranteed portability while preserving the possibility of specific optimizations. Method caches accelerated searching through the class hierarchy: when an object received a message, the system memorized the association between the selector and the corresponding method for subsequent calls.

The management of execution contexts revealed another facet of the object approach. These contexts, objects manipulable by the program, would enable sophisticated mechanisms like exception handling and reflection to emerge. This conceptual uniformity pushed object logic into the most intimate workings of the system.

With the publication of Smalltalk-80 in a special issue of Byte magazine in 1980, the books by Adele Goldberg and David Robson disseminated these ideas well beyond academic circles. Objective-C, Ruby, Python, and many others drew directly from this source. The influence extended beyond the realm of languages: modern development environments retain traces of this vision where everything is accessible and modifiable in real time.

The Smalltalk-80 interface spread into the commercial world thanks to Jobs's insight. The Lisa and Macintosh popularized these concepts with the general public, creating a *de facto* standard that endures. The optimization techniques developed for Smalltalk-80 are now standard practices in implementing object-oriented languages.

The exhaustive documentation of the system, including virtual machine specifications and implementation details, constituted an invaluable gift to the computing community. Other teams would thus reinvent the system according to their needs, accelerating the propagation of these pioneering ideas.

UDP

In 1980, David P. Reed was working on a seemingly simple problem: how to transmit data between computers without all the complexity that TCP already imposed? The Internet architecture was taking shape around the TCP/IP suite, but certain applications called for something more direct, faster. Reed then conceived UDP, the User Datagram Protocol, which would become one of the most enduring protocols on the Internet.

Where TCP built solid connections, verified each packet, and guaranteed the order of data arrival, UDP deliberately chose the opposite. No persistent connection, no delivery guarantees, no packet ordering control. This minimalist approach is reflected in RFC 768, which defines the protocol: three short pages suffice where other specifications stretch across hundreds of pages.

This brevity was not negligence. Reed and his colleagues understood that the Internet would need two types of tools: sophisticated protocols like TCP for critical data, and lightweight mechanisms like UDP for other uses. The first service to truly exploit this philosophy was the Internet Name Service, the ancestor of DNS. When a computer requests the IP address of a website, it doesn't need to establish a complex connection; it sends its query and waits for the response. If the response is lost, it asks again. Simple, straightforward.

DNS still uses this approach, and for good reason—a server handles thousands of queries per second without maintaining state for each client. This resource economy proves invaluable when you consider that a root DNS server can receive millions of queries per hour.

The history of UDP took an unexpected turn with the emergence of multimedia communications. When Voice over IP took its first steps in the 1990s, developers discovered that UDP perfectly matched their needs. In a phone conversation, it's better to lose a few milliseconds of audio than to wait for a missing packet to be retransmitted. User experience takes precedence over technical perfection. This logic naturally extended to videoconferencing, then to online gaming. In an online shooter game, if the information that a player has fired arrives late, it's useless. Developers handle these temporal issues themselves with UDP, according to their application's specifics.

UDP's elegance lies in this flexibility. The protocol doesn't dictate how to handle errors or packet ordering; it leaves that responsibility to applications. Some games retransmit critical information multiple times to ensure reception. Others simply ignore lost packets and focus on the most recent data.

This approach has inspired new innovations. DCCP adds congestion control to UDP to avoid overloading the network, and DTLS brings cryptographic security. These protocols show that UDP serves as a stable foundation for building specialized solutions.

Performance measurements confirm Reed's initial intuitions. A study conducted between 2004 and 2014 on a transpacific link between Japan and the United States revealed variations in average IP packet size. UDP, generating less control traffic than TCP, contributes to more efficient

bandwidth usage.

The emergence of sensor networks has given UDP new life. These devices, often battery-powered, periodically transmit temperature, humidity, or position measurements. They don't need TCP's robustness; they prefer to conserve their energy by simply transmitting their data without establishing a complex connection.

Cloud computing and microservices have rediscovered UDP's virtues. When an application decomposed into dozens of services must communicate quickly, UDP offers minimal latency. Netflix thus uses UDP for certain internal communications, Spotify for synchronizing playlists between devices.

The evolution toward the UDP-Lite variant demonstrates this adaptability, tolerating certain errors in packet headers, useful in wireless communications where a few corrupted bits don't justify rejecting an entire packet.

Forty-three years after its creation, UDP continues to thrive on the Internet. Its longevity stems from a simple principle: rather than trying to solve all problems, it solves a few of them perfectly well. This specialization has allowed it to endure through the ages, from early university networks to modern data centers. This constancy makes it one of the discrete but indispensable pillars of the Internet.

Commodore VIC-20

In 1980 near London, Jack Tramiel called his teams together for a meeting that would change everything. The head of Commodore expressed a concern that haunted him: "The Japanese are coming, so we're going to become the Japanese". This pithy statement summed up the entire philosophy that would govern the birth of the VIC-20.

Tramiel saw far ahead. His Commodore empire certainly ruled in Europe, but in the United States, Apple and Radio Shack maintained an edge. More worryingly, Texas Instruments was eating away at its positions in the calculator market. The executive feared above all that Japanese manufacturers would arrive with subsidized machines that would crush the competition. His response? Create a color personal computer accessible to the general public, a machine that would complement the already established PET range.

This bold vision rested on a technical component that had been forgotten in MOS Technology's drawers for two years. In 1978, Alan Charpentier had designed the VIC (Video Interface Chip) to attract arcade cabinet manufacturers. The reception had been cold, and the chip had been dormant ever since. But now it finally found its calling in this mass-market computer project.

Mike Tomczyk arrived in April 1980 as Tramiel's assistant. In a few weeks, this energetic man embodied the soul of the VIC-20 project, to the point of earning the nickname "VIC Czar". By visiting Germany and Japan in a month, he laid the foundation for his program to restructure marketing.

Meanwhile, two teams competed in ingenuity. Robert Yannes, who would later create the famous SID sound chip, cobbled together at MOS Technology a first prototype with PET components and a calculator case. His vision leaned more toward a sophisticated game console. For their part, Bill Seiler and John Feagans assembled another prototype by mixing different elements. They added a 9-pin joystick port and a cartridge connector inspired by the Atari 2600, but above all insisted on integrating BASIC. Their conviction: a computer must allow programming, not just playing.

The final prototype married these two visions. The resulting machine delivered on its technical promises: a MOS 6502A processor running at about 1 MHz, 5 KB of RAM with 3.5 KB usable, cassette and disk connectors, a joystick port, a versatile user port, and four mono audio channels generated by the VIC chip. The whole thing fit into a compact case with clean lines.

Commodore first tested its mettle in Japan. In October 1980, the VIC-1001 was released at 69,800 yen. While fine-tuning production, this strategy of observing the reaction of Japanese manufacturers proved worthwhile. In May 1981, the VIC-20 arrived in the United States at $299.95, and in the United Kingdom in September at £199.99.

Tomczyk then deployed his marketing strategy. He trademarked the phrase "the friendly computer" and bypassed specialized retailers to invest in big-box stores. This approach deliberately placed the VIC-20 in competition with game consoles, highlighting its personal computer capabilities. The documentation reflected this philosophy: the manual favored simplicity and left technical subtleties to the Programmer's Reference Guide.

William Shatner, the unforgettable Captain Kirk of *Star Trek*, lent his charismatic face to the advertising campaign. His television appearances made an impression and gave the VIC-20 unexpected notoriety. The results followed. In 1982, Commodore factories produced up to 9,000 units

per day to meet exploding demand. Revenue reached $300 million.

The software ecosystem expanded at breakneck speed. Games naturally dominated, with notable creations like *Jelly Monsters*, an unauthorized but successful adaptation of *Pac-Man*, or *Sword of Fargoal* which cleverly exploited the machine's capabilities. Jeff Minter, future indie game guru, made his first works on the VIC-20 with *Matrix* and *Laser Zone*. The VICModem, the first modem sold under $100, opened the doors to the CompuServe network via the Commodore Information Network. Cartridges of 3, 8, or 16 KB extended memory up to 32 KB and multiplied possibilities.

The success exceeded all expectations. The VIC-20 was the first personal computer to break the one million units sold mark. This historic performance testified to the accuracy of Tramiel's vision: the home computer could appeal far beyond the circle of enthusiasts.

Yet this triumph carried the seeds of its own end. In August 1982, Commodore launched the Commodore 64, a more powerful machine that cannibalized sales of its little brother. VIC-20 production stopped at the end of 1984, and the last units disappeared from shelves in early 1985. About 2.5 million units would find buyers in four years of commercial existence, a remarkable performance but eclipsed by the longevity of the C64 or ZX Spectrum.

This premature disappearance in no way diminished the VIC-20's legacy. The machine financed the development of the Commodore 64, prepared the acquisition of the Amiga, and established the codes of home computing in the 1980s. Its design had a lasting influence on the industry: peripheral compatibility, importance of ergonomics, deliberate democratization. An entire generation of programmers cut their teeth on this machine, particularly in the United Kingdom where it catalyzed creativity.

A passionate community keeps the flame alive and new programs regularly see the light of day, such as *Astro Nell* or *Game Theory*, which reveal unsuspected capabilities despite hardware constraints. These recent creations testify to the soundness of the initial concept: a simple and accessible computer that liberates creativity rather than hindering it.

WordPerfect

In the summer of 1979, Alan Ashton, a computer science professor at Brigham Young University, decided to devote his vacation to a project close to his heart: creating software that would display text exactly as it would appear on paper. At the time, existing systems worked with unsightly formatting codes that cluttered the display. In his laboratory in Orem, Utah, Ashton laid the foundations for what would become one of the most resounding successes in personal computing.

As a former university orchestra conductor who had switched to computer science, Bruce Bastian brought his technical expertise to the project. Together, they developed SSI*WP, the acronym for Satellite Software International Word Processor. The name choice revealed their ambitions: they wanted to transcend the traditional geographical boundaries of software.

Marketing began in 1980 with a price that gave one pause: $5,500 for a license intended for Data General computers. The market reacted cautiously. Sales remained modest but sufficient for the small team to maintain control of its destiny. Don Owens joined the venture as sales manager and began building a network of resellers. Pete Peterson, a former accountant, took charge of financial matters in 1981. This founding team would split over a question: should they accept external investment to accelerate growth?

Owens argued for fundraising that would provide the means for rapid expansion. Ashton and Bastian preferred a more cautious approach, even if it meant progressing more slowly. These incompatible visions created tensions that erupted in 1982 with Owens' departure. In retrospect, this choice of financial independence would prove decisive for what followed.

The arrival of the IBM PC changed everything. This machine democratized personal computing and opened unprecedented possibilities. WordPerfect released its first PC version at the end of 1982. The software stood out from the competition through its remarkable stability and polished ergonomics. Crashes were rare, features numerous. The integrated spell checker caused a sensation, as did the native footnote management.

Version 4.2, launched in 1986, propelled WordPerfect to unexpected heights. More than half of word processing users now chose this solution. This success owed nothing to chance. The company, renamed WordPerfect Corporation, revolutionized industry standards by offering free telephone technical support. This commercial innovation upended established practices: until then, obtaining help was expensive and time-consuming.

WordPerfect's user interface cultivated its distinctiveness. Function

keys replaced the pull-down menus already featured in other software. A small cardboard template, placed on the keyboard, served as a reminder of available shortcuts. This approach was initially disconcerting but quickly revealed its advantages. Thanks to the "Reveal Codes" function, which displayed all hidden formatting codes in the document, the efficiency of experienced users doubled. Lawyers and notaries became enamored with this transparency that guaranteed them complete control over the formatting of their documents.

The company's growth defied comprehension. From 11 employees in 1981, it grew to more than 4,500 in 1992. Revenue reached $533 million in 1991. This meteoric expansion generated internal debates about organization. Peterson advocated for a flat structure with few hierarchical levels, while other executives called for more traditional management. These disagreements led to Peterson's departure in 1992, depriving the company of a coherent organizational vision.

Microsoft Windows 3.0 arrived in 1990. This modern graphical interface won users over with its user-friendliness. WordPerfect then made a major strategic mistake: the company was slow to adapt its software to this environment, remaining attached to MS-DOS text mode. Microsoft Word for Windows took advantage of this delay to establish itself. Integrated into the Office suite, it benefited from visual and functional consistency that WordPerfect struggled to match.

When WordPerfect for Windows finally came out in 1991, the damage was done. The software suffered from performance and usability problems. Users, seduced by the modernity of the graphical interface, switched en masse to Microsoft's solution. This transition marked the beginning of WordPerfect's decline.

Novell purchased WordPerfect Corporation in 1994 for $1.4 billion. This pharaonic acquisition turned into a disaster. The networking giant understood nothing about the word processing market. Sales collapsed in the face of Microsoft Office's growing hegemony. Two years later, Corel acquired WordPerfect for only $158 million, nearly ten times less than the initial purchase price.

This spectacular fall illustrates the brutality of reversals in the software industry. WordPerfect had dominated its market through technical excellence and impeccable customer service. But the company failed to anticipate the importance of the graphical interface. It underestimated Microsoft's firepower, which controlled both the operating system and could impose its office suite as the de facto standard.

WordPerfect retains a loyal user base, concentrated in the legal world. Legal professionals still appreciate its precision in formatting complex doc-

uments. The "Reveal Codes" function is unique and alone justifies the loyalty of certain users.

With WordPerfect, the company established free customer support as an industry standard. Its innovations in WYSIWYG display and spell checking inspired all its successors. In computing, technical leadership is not enough: one must also know how to adapt to technological disruptions and understand users' changing expectations. The software industry is merciless. A company can dominate a market for years and then collapse in a few months if it misses a technological turn.

ICMP

In 1981, engineers working on ARPANET faced a recurring problem: how to inform a sender that their packet had been lost in transit. The IP protocol, in its initial design, provided no mechanism for signaling errors. Packets would disappear into the maze of the network without a trace, leaving the sender in the dark about the fate of their data.

Jon Postel and Steve Crocker observed this glaring gap from UCLA as ARPANET expanded. The absence of feedback literally paralyzed fault diagnosis. An overloaded router would drop a packet, but the sender remained unaware. A destination was unreachable, but there was no way to know. This situation led to the creation of ICMP, the Internet Control Message Protocol.

RFC 792 formalized this solution by defining two major families of messages: those signaling errors and those querying network status. The first category informs the sender of problems encountered: destination unreachable, packet time-to-live expired, incorrect parameters. The second family serves to probe the network, verify that a host responds, or measure packet round-trip time.

ICMP's technical architecture is particularly surprising. Rather than positioning itself above IP as a higher-level protocol, ICMP integrates directly into it. Control messages travel in ordinary IP datagrams, taking the same paths as the data they concern. This approach ensures that error messages follow an identical route to the failed packets.

Mike Muuss revolutionized ICMP usage in 1983 with his `ping` utility. This small application exploits "echo request" and "echo reply" messages to create the Internet's most popular connectivity test. The principle appeals through its simplicity: send a message to a machine and wait for

its response. If it responds, the connection works. Otherwise, a problem exists somewhere along the path.

The success of `ping` inspired other tools. `Traceroute` cleverly hijacks "time exceeded" messages to map network routes. By sending packets with progressively increasing time-to-live values, the tool forces each intermediate router to return an error message, thereby revealing its identity and position along the path.

Path MTU discovery illustrates ICMP's adaptability to emerging needs. This technique uses "fragmentation needed" messages to determine the maximum size of packets traversing a network path without fragmentation. The process significantly improves performance by avoiding the division of packets into smaller fragments.

But ICMP also attracts the attention of hackers. "Ping of death" attacks exploit vulnerabilities in the handling of oversized ICMP packets, causing vulnerable systems to crash. Other attacks use ICMP to flood networks with control messages. These malicious exploits push many administrators to filter or block certain ICMP messages, sometimes creating undesirable side effects on normal network operations.

The arrival of IPv6 redefines ICMP's role. The new version, ICMPv6, absorbs functions previously handled by other protocols. Neighbor discovery, for example, replaces ARP and integrates directly into ICMPv6. This consolidation simplifies network architecture while strengthening ICMP's importance in modern infrastructures.

The ICMP protocol influences the design of sophisticated network monitoring tools that analyze control messages to detect anomalies and measure performance. Network administrators learn to interpret these signals to diagnose failures and anticipate problems.

This transformation of a simple control protocol into a cornerstone of Internet infrastructure testifies to its judicious design. Its flexibility has allowed it to adapt to the changing needs of computer networks for over forty years. Today, every `ping` launched on the Internet perpetuates the legacy of those early ARPANET engineers who simply wanted to know why their packets were getting lost.

Acorn BBC Micro

In 1980, the BBC was grappling with questions about the United Kingdom's digital future. Officials at the British public broadcaster watched

with concern as American companies dominated the microcomputer market. Available machines either cost a fortune or offered laughably limited capabilities. Faced with this situation, the BBC decided to take matters into its own hands and launched an ambitious project: to create a personal computer that would accompany its educational programming.

The specifications established by the BBC reflected this educational ambition. The machine would need to incorporate the BASIC language, feature a proper keyboard worthy of the name, display color graphics, and accept extensions such as teletext. Above all, its price would need to remain within reach of schools and motivated individuals. A considerable technical and commercial challenge.

The tender issued in late 1980 attracted several candidates. Among them was Acorn Computers, a small Cambridge company working on a prototype called Proton. This machine impressed the BBC evaluators, who favored it over the Sinclair ZX-81, still in gestation. Acorn won the contract and officially renamed its prototype the "BBC Microcomputer".

Between Acorn's engineers and the BBC's teams, the collaboration proved fruitful. The initial prototype underwent numerous improvements that far exceeded the original specifications. ICL and Cleartone handled manufacturing. The architecture was based on the 6502A processor clocked at 2 MHz, a reliable and proven chip. Two specialized integrated circuits completed the package: one handled graphics, the other serial interfaces.

Two versions were released simultaneously in early 1982. Model A sold for £235 with 16 KB of RAM, while Model B cost £335 for 32 KB. The latter featured more connectivity: serial and parallel interfaces, an 8-bit user port, analog inputs, and an expansion bus. One technical innovation deserves mention: the "Tube" interface, which allowed connection of a second processor.

The integrated BASIC was surprisingly rich. Similar in philosophy to Microsoft BASIC, it distinguished itself through numerous specific extensions. The operating system and BASIC occupied 32 KB of ROM, a generous allocation. The professional keyboard featured 64 keys arranged in the QWERTY standard, complemented by 10 programmable function keys.

Graphics performance constituted a major asset. Eight different modes allowed mixing of text and high-resolution graphics as needed. Mode 7, teletext-compatible, produced color graphics while consuming only one kilobyte of memory. A sound chip generated three simultaneous notes and various effects, with sophisticated software control of sound envelopes.

Extensibility set the BBC Micro apart from its competitors. Interfaces

for floppy disk drives and Econet networking were integrated directly on the motherboard. A speech synthesis option, developed with Richard Baker, the BBC's star presenter, could pronounce approximately 150 preprogrammed words. Prestel and teletext adapters, planned for spring 1982, opened the way for program downloads.

The expansion possibilities appealed to advanced users. A second 6502 processor, a Z-80 for running CP/M, or a 16-bit National 16032 processor capable of addressing 16 MB of memory could be connected via the Tube interface. Dedicated monitors completed the range: a monochrome model at £105 and a color model at £288.

The BBC commissioned professional applications: word processor, domestic database, spreadsheet, computer-aided design. Third-party publishers developed educational and entertainment software. A comprehensive user manual accompanied each machine, along with a cassette of 16 demonstration programs.

A few flaws marred the picture. The plastic case lacked durability. The 32 KB of user memory sometimes proved insufficient, with the operating system consuming up to 8 KB. In high-resolution graphics mode, only 2 KB remained available. Teletext mode sometimes displayed erroneous characters due to its specific character set. No cassette cable was supplied with the machine.

These minor defects did not detract from the product's overall excellence. Display quality was impressive: 80 characters per line remained perfectly legible on a standard black-and-white television thanks to a specially designed font. Teletext mode offered one of the best visual renderings of its generation.

The BBC Micro surpassed most Japanese and American productions of the era through its technical richness and modularity. Commercial success followed: 100,000 units were planned for the first year. This achievement resulted from the meeting of Acorn's technical expertise and the BBC's educational vision.

The machine became the standard educational tool in British schools during the 1980s. It introduced an entire generation to computing and fostered a national digital culture. The experience Acorn gained in processor design would later lead to the ARM architecture, present in virtually all modern mobile devices on the planet.

IBM PC-5150

When IBM announced its Personal Computer 5150 in August 1981, the company entered a market six years behind Apple, Atari, Commodore, and Radio Shack. This late arrival forced Big Blue to break with tradition: internal development gave way to external partnerships. A team of just twelve people would design this machine in one year, a record for IBM.

The approach was surprising. Intel supplied the processor, Microsoft delivered PC-DOS and BASIC. Distribution also broke with tradition: instead of its own channels, IBM chose Computerland and Sears. For advertising, the company abandoned its austere image and featured Charlie Chaplin's Tramp character in a series of offbeat television spots.

At $1,565 for the base model, without floppy drive or screen, the PC 5150 initially targeted individuals. They could connect their television as a monitor and use a tape recorder to save their data. Business users, however, paid $4,500 for a complete configuration with 64 KB of memory, two floppy drives, color display, and dot matrix printer. Adjusted for inflation and converted to current Swiss francs, these amounts would reach 6,000 and 14,000 CHF.

Technically, nothing spectacular. The Intel 8088 processor ran at 4.77 MHz, a thousand times less than our current chips. This economical derivative of the 8086 made do with an 8-bit data bus while addressing up to 1 MB of memory. The motherboard accommodated between 16 and 64 KB of RAM, two 5.25-inch drives of 160 KB each.

A few details nevertheless marked a break. The detached keyboard was exceptional at a time when most computers integrated it into the main unit. Its 83 keys weighed 2.7 kg and produced that characteristic click that would delight nostalgic users. The system unit with its drives weighed 12.7 kg, the monitor 7.7 kg. Total: 23 kg of home computing.

No standard hard drive, that would require the 5161 expansion. No mouse either, it wouldn't become widespread until later. Storage used audio cassettes via a specialized connector or floppy disks. Early versions used single-sided 160 KB media, then double-sided 360 KB for the Model B marketed as early as 1983.

IBM surprised with its technical transparency. The documentation provided with the PC 5150 revealed everything: electronic schematics, architecture diagrams, interface specifications, BIOS source code. This unusual openness boosted the creation of expansion cards and compatible software. But it also opened the door to cheaper clones that would proliferate.

Five internal slots accepted expansions via what IBM called the "I/O Channel", predecessor of the ISA standard. These 62-pin connectors accommodated floppy controllers, graphics adapters, parallel interfaces for printers, or RS-232 serial ports for modems. Contemporary PCs have abandoned these parallel buses for faster serial connections like PCI Express or USB.

PC-DOS managed files with the FAT system, without subdirectories in its first iteration. All files spread out in a flat structure: 64 maximum on single-sided floppy, 112 on double-sided. Computer viruses and piracy, though they existed, had not yet infected the planet. The manual addressed only physical security: storing diskettes safely, backing up regularly, controlling machine access. The first virus for IBM-compatible PCs, Brain, wouldn't appear until 1986.

Production of the 5150 stopped in 1987, replaced by the 5160 model. Internationalization began in 1983. Its architecture still influences our current machines, from connected objects to enterprise cloud servers. IBM's choices, like standardized components and comprehensive documentation, created an open ecosystem that would dominate the market.

MS-DOS

Until then focused on large systems, IBM set its sights on the personal computer market in 1980. The Armonk firm was looking for an operating system for its future machine. It turned to Microsoft, a small company that Bill Gates and Paul Allen had created a few years earlier. Problem: Microsoft didn't have a proper operating system.

The solution came from Seattle Computer Products. This company had developed QDOS – Quick and Dirty Operating System – under Tim Paterson's leadership. The story of QDOS deserves attention. In May 1979, Seattle Computer Products designed an 8086 processor board for the S-100 bus. The company was counting on Digital Research to deliver CP/M-86, but the promises dragged on. Tired of waiting, the engineers decided to create their own system. Paterson tackled the task in April 1980. Two months later, QDOS was born, and the system worked remarkably well for such rapid development.

Microsoft sensed a good opportunity. The company acquired a non-exclusive license for QDOS for $25,000 in December 1980. A few months later, it purchased all rights for an additional $50,000. Tim Paterson joined Microsoft's team to transform QDOS into MS-DOS.

The first IBM PC came out in 1981, equipped with MS-DOS 1.0. This rudimentary operating system operated entirely in text mode. Users typed commands: DIR to list files, COPY to duplicate them, ERASE to delete them. No directories, just 160 KB floppy disks. Austere, certainly, but functional.

Improvements arrived. MS-DOS 2.0 appeared in 1983 with support for 10 MB hard drives and a hierarchical file structure. This version borrowed some ideas from UNIX, notably a more sophisticated batch command processor. MS-DOS 3.0 followed in 1984, tailored for the IBM PC AT. The FAT16 file system made its appearance, managing partitions up to 32 MB.

The technical architecture relied on three main components. The hidden file MSDOS.SYS contained the system core and processed high-level program requests. IO.SYS handled hardware aspects, this abstraction layer between software and hardware. COMMAND.COM provided the user interface, that command prompt millions of users would come to know.

The FAT file system revolutionized disk space management. An allocation table tracked the usage of each unit, composed of several consecutive sectors. This approach avoided external fragmentation but generated internal fragmentation: the last unit of a file was never completely filled. A technical compromise that would prove its worth.

Microsoft played its commercial hand strategically. The agreement with IBM authorized the sale of MS-DOS to other manufacturers, strategy which established MS-DOS as the de facto standard for all IBM-compatible PCs. When Windows 3.0 arrived in 1990, the graphical interface relied on MS-DOS. Success followed.

MS-DOS 6.0 arrived in 1993 with its share of innovations: DoubleSpace for disk compression, an integrated antivirus, a backup system. But DoubleSpace posed problems. Data corruptions prompted Microsoft to release version 6.2, supplemented with ScanDisk to repair damaged disks. The year didn't end well: Stac Electronics sued Microsoft for patent infringement on the compression algorithm. Microsoft lost and had to remove DoubleSpace. Version 6.22 came out in 1994 with DriveSpace, using a new algorithm to avoid any dispute.

Windows 95 marked the end of MS-DOS's golden age. The system integrated MS-DOS 7.0 as a simple boot and compatibility component. Windows 98 and Me still kept MS-DOS, but its influence waned. Windows XP, in 2001, definitively closed this chapter, retaining only minimal compatibility to run stubborn old applications.

The MS-DOS system standardized the PC universe, accompanying

the emergence of an extraordinarily rich software ecosystem. From word processors to spreadsheets, from databases to games, an entire world of applications was born. Companies like Lotus, Borland, and WordPerfect built their fortunes on this platform.

MS-DOS democratized computing in its own way. Its command-line interface, austere and demanding, trained an entire generation of users. Learning MS-DOS meant understanding the machine's internal logic. This school of rigor forged skills that would prove valuable with the arrival of graphical interfaces, because the Windows command prompt still retains many commands inherited from MS-DOS: DIR, COPY, DEL still work. The concepts of file management, memory allocation, and program execution established at that time still form the foundations of current systems. DOSBox and other emulators preserve this history, allowing nostalgics to rediscover their old software.

MS-DOS is an important milestone in personal computing. This humble and efficient system made the PC explosion possible. Without it (and QDOS), Microsoft's history would have taken a different course, and consumer computing might have taken other paths.

Osborne 1

Adam Osborne had grasped something that few entrepreneurs understood in 1980. An author of computer books and head of a specialized publishing house, he observed a booming market where microcomputers remained confined to offices and laboratories. His idea seemed simple: create a computer that could be taken anywhere.

In April 1981, in the pages of *Creative Computing Magazine*, Osborne laid out his vision with remarkable clarity. He identified five fundamental needs that a personal computer had to satisfy: an affordable price, ease of use without sacrificing versatility, portability, software compatibility through standardized systems, and adaptability to developers' requirements. This analysis stemmed directly from his industry experience and his frustration with existing machines' limitations.

The Osborne 1 embodied this thinking. At 10.9 kg, the machine adopted a bold all-in-one design: a 5-inch screen, two floppy drives, a detachable keyboard, and power supply integrated into a single portable case. This approach represented a radical departure from traditional systems that required multiple components connected by a tangle of cables.

Technically, the computer proved sound. Its Zilog Z80A processor running at 4 MHz, 64 KB of RAM, and CP/M operating system offered respectable performance for the era. The screen displayed 52 characters across 24 lines, with horizontal scrolling capability up to 128 characters. Each 5.25-inch floppy drive provided 102 KB of storage. This configuration represented a judicious compromise between performance and mobility in 1981.

Osborne had especially understood the importance of software. For $1,795, the buyer received the machine along with a complete suite including WordStar for word processing, SuperCalc as a spreadsheet, and BASIC and CBASIC for programming. This software bundling strategy was unprecedented and transformed the purchase of simple hardware into acquisition of a complete working environment.

The commercial target revolved around three distinct segments. Mobile professionals found a means to manage their documents and spreadsheets while traveling, before synchronizing their work at the office via modem. The education sector discovered an economical solution for equipping institutions with computer equipment. Individual consumers finally accessed an affordable alternative to traditional desktop computers.

The success exceeded all predictions. Osborne Computer Corporation experienced meteoric growth, becoming one of the fastest-growing companies in American history. Sales reached 10,000 units monthly, generating over $70 million in revenue in the first year alone.

The machine's flaws became apparent. The 5-inch screen, constantly criticized, forced users into perpetual horizontal scrolling to view standard documents. Floppy storage proved insufficient for certain professional applications. Osborne responded by developing a double-density option increasing capacity to 200 KB per disk.

In 1982, the company announced a major improvement: the SCREENPAC, an 80/104 column display option. This hardware upgrade allowed display of up to 104 characters per line, bringing the Osborne 1 closer to industry standards. However, the premature announcement of this unavailable improvement triggered a perverse effect: sales of the existing model plummeted. This phenomenon, later dubbed the "Osborne effect", illustrated the dangers of poorly managed product communication.

The Osborne 1's influence on the computer industry far exceeded its commercial success. It established several lasting standards: complete component integration, systematic inclusion of software with hardware, and accessible pricing. Its impact manifested directly in the development of the first laptops from IBM, Compaq, and other manufacturers who subsequently dominated the market.

The machine introduced remarkable technical innovations. Its keyboard used *full travel membrane* technology offering excellent typing quality while resisting dust and liquid infiltration. Its modular design simplified repairs. Adoption of the S-100 bus and CP/M system guaranteed compatibility with a vast existing software ecosystem.

The device's ruggedness was impressive. Tests documented in *The Portable Companion* reported its survival of stairway falls and extreme operating conditions. This durability, coupled with its portability, made it a perfectly suited tool for mobile professionals.

The Osborne 1 transformed workplace organization in businesses. It enabled the emergence of the "mobile office" concept, allowing employees to carry their digital environment while traveling. This evolution foreshadowed the changes workplace organization would undergo with the widespread adoption of portable computing.

The fall was as swift as the rise had been meteoric. In September 1983, Osborne Computer Corporation filed for bankruptcy, victim of increased competition, production difficulties, and strategic errors. The company had underestimated the market's pace of evolution and the need for constant innovation. Within two years, the Osborne 1 went from revolutionary innovation to obsolete technology. Its impact on democratizing portable computing and establishing industry standards remains undeniable nonetheless.

The Osborne experience underscores the importance of portability, the necessity of a complete software ecosystem, and the delicate balance between technological innovation and market expectations. The Osborne 1 nevertheless wrote an essential chapter in the history of personal computing.

LZW

The LZW compression algorithm tells a singular story, that of a theoretical discovery that took years to find its practical form. It all began in 1977, when Abraham Lempel and Jacob Ziv published a hermetically-titled article in the IEEE Transactions on Information Theory: "A Universal Algorithm for Sequential Data Compression". Their work presented a method using phrase dictionaries that slide over already-read text, progressively building new patterns by adding symbols to existing sequences.

The following year, the two researchers returned to the same journal

with a refined version of their algorithm. These publications remained nonetheless highly abstract, laying theoretical foundations but leaving the reader wanting when it came to concrete applications. It wasn't until 1984 that a third party entered the scene and transformed this academic work into a usable technology.

Terry Welch was working at the Sperry Research Center when he published his article "A Technique for High-Performance Data Compression" in Computer magazine. His contribution revolutionized Lempel and Ziv's approach. Welch had the idea of pre-populating the dictionary with the 256 standard ASCII characters rather than starting with an empty dictionary. This modification enabled immediate processing of all simple characters, including on their first appearance. The remaining codes, from 256 to 4095 in a 12-bit dictionary, could thus serve exclusively for newly discovered sequences.

The LZW algorithm was born, bearing the initials of its three creators. Its strength lay in its ability to detect and eliminate redundancies while guaranteeing perfect restoration of the original data. Unlike other methods, LZW never sent raw characters but only codes, generating substantial gains on files containing repetitive patterns.

A remarkable characteristic of this algorithm was its autonomy during decompression. The decoder didn't need to receive the dictionary used for compression; it could reconstruct it identically by applying the same rules as the compressor. This property made LZW memory-efficient and fast to execute.

The computer industry embraced this innovation. In 1987, CompuServe launched the GIF (Graphics Interchange Format) format, which exploited LZW to compress images. This format quickly became indispensable on early networks and then on the Internet, notably thanks to its ability to store numerous images in a single file and manage transparency. UNIX also integrated the algorithm into its `compress` command, while the TIFF and PDF formats adopted it for their specific needs.

LZW's performance varied considerably depending on the type of data processed. Natural language texts, rich in redundancies, suited it perfectly, as did images containing vast uniform areas. On such content, the algorithm commonly achieved compression rates of 50% or more. Binary files, however, yielded more unpredictable results: some compressed better than ordinary text, while others stubbornly resisted any processing.

The practical implementation of LZW required solving some technical puzzles. The trickiest arose during decompression, when the decoder encountered a code before having been able to define it. This baroque situation occurred with sequences of the type (string, character, string,

character, string), where the algorithm emitted a code that the decompressor hadn't yet had the opportunity to create. A specific exception mechanism was therefore needed to handle these particular cases.

LZW's success story took an unexpected turn in the 1990s. Unisys, holder of a patent on the algorithm, began demanding royalties for any commercial use. This sudden claim provoked an outcry in the computing community and prompted the development of the PNG (Portable Network Graphics) format as a free alternative to GIF. The patent finally expired in 2003 in the United States and in 2004 elsewhere, at last freeing the algorithm from these legal constraints.

Dictionary size directly influenced compression efficiency. A 12-bit dictionary, capable of storing 4096 entries, often represented a satisfactory balance between compression rate and memory consumption. The algorithm gained performance with large files, as the dictionary had more opportunities to identify and reuse recurring patterns.

On small files, LZW showed its weaknesses. The storage cost of the dictionary could exceed the compression benefit, making the compressed file larger than the original. Variants like LZWS (Lempel-Ziv-Welch-Setia) attempted to dynamically adapt the strategy according to data size, with mixed success.

The LZW algorithm established enduring principles in the field of compression: the use of adaptive dictionaries, lossless compression, the importance of autonomy during decompression. It occupies a special place in the computing ecosystem. More recent methods certainly offer better compression rates, but its relative simplicity makes it an excellent pedagogical tool for understanding the fundamental mechanisms of data compression. It remains used in certain specific applications where its robustness and execution speed compensate for its limitations. In the history of information technologies, LZW remains one of the first compression algorithms to have achieved lasting commercial success and truly universal adoption.

AutoCAD

In 1982, computer-aided design was the preserve of the privileged few. CAD systems cost a fortune – several hundred thousand dollars – and required specialized machines that only large corporations could afford. A handful of programmers decided to change the game.

John Walker gathered around him about fifteen developers, most from

Information Systems Design, a company working on UNIVAC systems. They created Autodesk with a simple but bold idea: develop CAD software for personal microcomputers that were beginning to flood offices. The organization they established was unusual for the time: everyone worked from home, communicating by telephone and mail. With an initial investment of $59,000, they launched a handful of parallel projects, convinced that one of them would eventually break through.

The breakthrough came from Mike Riddle and his Interact program, written in a programming language he had designed himself. The Autodesk team purchased the source code and undertook a complete rewrite in C. Dan Drake and Greg Lutz handled this delicate conversion, progressively transforming this drawing software into what would become AutoCAD. The choice of C, still uncommon in the commercial world, proved judicious for the software's future portability.

In late 1982, at the Comdex show in Las Vegas, AutoCAD made its public debut. The software ran on an IBM PC equipped with 64 KB of memory and two floppy drives. To make an impression, the team organized a demonstration with an HP plotter borrowed for the occasion. The announced price caused a sensation: $1,000, certainly substantial for PC software, but trivial compared to existing solutions.

Mike Ford developed an original commercial strategy based on a network of trained resellers. These handled first-line technical support, thus freeing Autodesk from the constraints of direct customer service. This decentralized approach aligned well with the company's DNA and its remote work philosophy.

AutoCAD won over its target clientele: small architecture and engineering firms. Its ability to run on different hardware platforms and exchange files seamlessly represented a major asset in a highly compartmentalized computing world. Users appreciated this flexibility, which prevented them from being locked into a particular manufacturer.

The technical team maintained a frantic pace of innovation. Support for Intel's 8087 math coprocessor significantly accelerated calculations. Integration of a LISP language offered advanced users the ability to extend the software's functionality according to their specific needs. Dimensioning tools became increasingly sophisticated. Each week, during collegial technical meetings, the team decided on the product's future directions.

Three years after its creation, Autodesk went public. The company had never sought venture capital, developing solely through its own revenues. Business Week ranked it two consecutive years as the fastest-growing company. John Walker, little attracted to managerial aspects, handed the reins to Al Green, then to Carol Bartz in 1992.

The arrival of Windows 3.0 in 1990 transformed the technical landscape. Until then, AutoCAD ran primarily under DOS and was ported to numerous UNIX variants. Management decided to focus its efforts on Windows, anticipating this system's future dominance. This decision coincided with the emergence of 3D modeling as an essential feature, developed under Scott Heath's leadership.

AutoCAD succeeded where others failed through different ingredients: an exceptional technical team obsessed with quality and compatibility, an effective commercial strategy through resellers, accessible pricing, and a remarkable ability to follow the rapid evolution of computer hardware. By democratizing CAD and making it accessible to small organizations, the software transformed design practices in industry and architecture.

CD-ROM

At the turn of the 1980s, personal computing operated on floppy disk time. These small plastic squares, fragile and limited to a few hundred kilobytes, were the primary means of exchanging programs and data. The idea that a single disk could contain the equivalent of several hundred floppies belonged to the realm of science fiction. Yet this revolution was already taking shape in Philips laboratories.

The story begins with optical storage research conducted by the Dutch company during the 1970s. Philips engineers were working on the VLP system, a technology that etched analog video signals onto a plastic disc read by laser. These experiments, initially intended to replace video cassettes, would open a completely unexpected path. From this work would emerge first the audio Compact Disc, then its computing descendant: the CD-ROM.

The collaboration between Philips and Sony resulted in the 1983 announcement of the CD-ROM, an acronym for Compact Disc Read-Only Memory. The two electronics giants jointly defined the technical specifications in the famous "Yellow Book", a reference document that established the rules for the new format. The concept adopted the physical dimensions of the audio CD: a polycarbonate disc 12 cm in diameter and 1.2 mm thick, scanned by an infrared laser that deciphers information engraved as microscopic grooves.

But where the audio CD stores music, the CD-ROM organizes data differently. It offers two operating modes. Mode 1 prioritizes reliability for

computer data through an enhanced error correction system. Mode 2 is intended for compressed audio and video content, accepting minor errors imperceptible to the human ear or eye.

The storage capacity revolutionized habits: 650 MB at once, equivalent to 450 standard floppies. With this sudden abundance, software publishers could finally design ambitious applications without worrying about space constraints. Gone were the days of having to cut features to fit on a few floppies.

The first CD-ROM drives hit the market in 1985. These expensive and relatively sluggish machines displayed a transfer speed of 150 KB/s, dubbed "single speed" or "1X" in technical jargon. Adoption was initially limited, reserved for professionals and affluent enthusiasts. It would take until the early 1990s for the technology to truly find its audience.

The rise of multimedia changed everything. Video game and educational software developers discovered the possibilities offered by this new storage capacity. Where Windows 95 required no fewer than 13 floppies in its basic version, the CD-ROM version fit on a single disc and drastically simplified installation. No more juggling a stack of floppies, monitoring insertion order, or restarting the procedure when one proved defective.

The video game industry seized the opportunity. The CD-ROM enabled the integration of cinematic-quality video sequences, orchestral music, and detailed graphics previously unthinkable. Sony built the success of its PlayStation on this technology, abandoning expensive cartridges to bet on CD-ROM. Games took on new dimensions, studios could tell complex stories with multiplied production resources.

The race for speed was on. Manufacturers competed ingeniously to accelerate data reading. From 1X, speeds progressed to 2X, 4X, 8X and far beyond, reaching 52X in the early 2000s. This surge in performance was accompanied by a spectacular drop in prices. Drives, initially sold for several thousand francs, became accessible to the general public. Blank discs followed an identical trajectory, transforming the CD-ROM into an affordable archival medium.

Technical evolution didn't stop there. In 1988, CD-R appeared, finally allowing users to burn their own discs. This innovation democratized the production of custom CDs, paving the way for individual backups and data exchange between individuals. The CD-RW storage medium, introduced in 1997, took an additional step toward versatility by allowing data erasure and rewriting.

Standardization was the key to this success. Technical specifications were subject to strict international standards, with ISO 9660 notably defining the universal file system. This approach, inherited from the 1985

"High Sierra" format, guaranteed that a CD-ROM burned on one machine could be read on any other, regardless of the drive manufacturer.

The CD-ROM spawned a large family of derivative formats. CD-i targeted interactive multimedia applications, Video CD attacked the home video market, Photo CD revolutionized digital image storage. These variations met with mixed fortunes, but testified to the conceptual richness of the original medium.

The CD-ROM made possible the distribution of multimedia encyclopedias like Encarta, exhaustive documentary collections, and voluminous professional databases. It directly contributed to the multimedia explosion of the 1990s, a period when the personal computer ceased being a simple word processing machine to become a family entertainment center.

This hegemony began to falter in the early 2000s. The DVD, with its superior storage capacity, gradually relegated the CD-ROM to secondary status. The arrival of high-speed Internet and the widespread adoption of USB flash drives accelerated the process. The dematerialization of software, downloaded directly from publishers' servers, dealt an additional blow to physical media.

Intel 80286

February 1, 1982 marked the debut of the Intel 80286, officially designated iAPX 286 in Intel's terminology. This 134,000-transistor chip represented a turning point in the evolution of 16-bit microprocessors. Its destiny became intimately tied to that of the IBM PC/AT in 1984, and subsequently to the entire ecosystem of PC/AT compatibles that would dominate the computing landscape until the early 1990s.

The performance history of the 80286 tells a story of gradual power increase. Intel launched its first models at 6 MHz and 8 MHz, before offering faster versions reaching 12.5 MHz. AMD and Harris would later push this architecture to unexpected heights: 20 MHz for the former, 25 MHz for the latter. These figures mask a more nuanced reality: the processor executes on average only 0.21 instructions per clock cycle. Translated into raw power, this rate yields 0.9 MIPS for the 6 MHz model, 1.5 MIPS at 10 MHz, and 1.8 MIPS at 12 MHz.

Under the hood, Intel completely rethought the architecture. The 80286 nearly doubled performance per cycle compared to its predecessors, the 8086 and 8088. This technical achievement was no magic trick: it re-

sulted from targeted optimizations, such as handling complex base+index addressing modes through a dedicated circuit rather than the general arithmetic unit. Demanding mathematical calculations, particularly multiplication and division, now required fewer cycles.

The most spectacular leap concerned memory addressing. With its 24 address bits, the 80286 theoretically managed up to 16 MB of RAM, shattering the one-megabyte barrier that constrained the 8086. This capacity remained largely theoretical in daily practice, however. The prohibitive cost of RAM and the scarcity of compatible software limited most machines to a single megabyte. Accessing this extended memory from traditional real mode also imposed a significant performance penalty.

The 80286's major contribution lay in its protected mode. This innovation elevated it to the level of professional processors of the era. Beyond exploiting 16 MB of physical memory through its integrated memory management unit, this mode opened a logical addressing space of one gigabyte. Gone were the crashes caused by undisciplined applications writing anywhere. Memory protection became real. The system organized memory into distinct segments for data, code, and stack, with a privilege hierarchy that prevented low-level programs from interfering with higher-level ones.

This elegant mechanism concealed a critical flaw. Once switched to protected mode, the 80286 could only return to 8086-compatible real mode through a complete reset. IBM deployed considerable ingenuity in the PC/AT to work around this limitation: external circuits, specialized code in the ROM BIOS, a convoluted instruction sequence that triggered the reset while preserving memory and control. The solution worked, but performance suffered heavily.

These technical constraints directly influenced the software ecosystem. In January 1985, Digital Research partnered with Intel to present Concurrent DOS 286, an operating system designed to natively exploit protected mode in a multi-user and multitasking environment. The project ran into harsh realities: 8086 emulation on production chips revealed malfunctions. Intel responded by correcting documented errors in the E-1 stepping, and actually modifying the microcode in the E-2 stepping to accelerate emulation. These improvements allowed IBM to adopt DR Concurrent DOS 286 as the foundation for its IBM 4680 OS in 1986, intended for IBM Plant System products and point-of-sale terminals.

Criticism rained down on the 80286. Bill Gates, never short on memorable phrases, called it a "brain-dead chip", anticipating Windows's inability to run multiple MS-DOS applications in parallel. This position accentuated the rift between Microsoft and IBM, the latter stubbornly developing OS/2 for the 286 in text mode, a project initially shared by

both giants.

Despite its glaring imperfections, the 80286 democratized memory protection mechanisms previously reserved for mainframes and minicomputers. Where its NS320xx and M68000 competitors required external components to manage the MMU, the 80286 integrated these functions directly on its chip. This integration, coupled with substantial performance gains, propelled the x86 architecture and IBM PCs from entry-level systems to high-end workstations and servers.

The processor also enriched computing capabilities by handling different types of numbers: unsigned packed decimal, unsigned binary, unsigned unpacked decimal, signed binary, and floating point. These functionalities, combined with its multitasking capability, oriented it toward communications applications, real-time process control, and multi-user systems.

The 80286 saga perfectly illustrates the complexity inherent in microprocessor development, where each architectural choice resonates across decades of software evolution. Its imprint endures through subsequent generations of x86 processors, which retain its fundamental concepts while smoothing over its growing pains.

SMTP

In August 1982, the IETF published RFC 821. This technical specification, which might seem unremarkable in the constant flow of normative documents, would transform global electronic communication. It gave birth to the Simple Mail Transfer Protocol, the invisible architecture that carries billions of messages daily today.

The idea seems simple: create a common language between machines to exchange electronic mail. Before SMTP, each messaging system spoke its own dialect. A message sent from a UNIX server could not reach a recipient on a Windows machine. The interconnection of computer networks required a digital *esperanto*.

RFC 821 established the rules of the game. An SMTP client enters into conversation with a remote server, identifies itself, specifies the sender and recipients, then transmits the content. Each step generates standardized numerical response codes. This technical choreography, repeated millions of times every second across the planet, orchestrates the global exchange of messages.

The protocol sailed through the 1980s and 1990s without major modification. Its robustness was impressive because for nearly twenty years, the same technical specification supported the explosion of the Internet. But this longevity also revealed its flaws. Hackers discovered how to exploit open relays to flood the network with unsolicited mail. Spam was born and proliferated.

In April 2001 came RFC 2821. The IETF introduced ESMTP (Extended SMTP), an enhanced version of the original protocol. The new features changed the game: automatic delivery notifications, detailed error messages, support for non-standard characters in headers. Most importantly, ESMTP enabled encryption of communications between servers. Security finally entered the equation.

This evolution nevertheless preserved a cardinal principle: backward compatibility. All ESMTP servers accept classic SMTP connections. This strategy ensured a gradual transition, avoiding the chaos of an abrupt break with the existing infrastructure.

RFC 5321, published in October 2008, consolidated these achievements. It refined the rules for using TLS to secure exchanges and clarified certain ambiguous technical aspects of previous versions. But the protocol now faced challenges far more complex than its original specifications.

Spam transformed the Internet into a battlefield. Malicious automata systematically tested address combinations to collect valid targets. Directory Harvest Attacks automated this prospecting. Malware spread via attachments. Spoofing techniques enabled the falsification of message origins, sowing confusion about the true identity of senders.

Faced with this escalation, the technical ecosystem responded. With the SPF framework, domains officially declared which servers could send emails on their behalf. DKIM added cryptographic signatures to messages to authenticate their origin. Anti-spam filters analyzed content, consulted blacklists, applied sophisticated heuristics to separate the wheat from the chaff.

Yet the SMTP protocol carries certain structural vulnerabilities. It does not natively verify whether the sender's address matches their declared domain. Automatic notifications can confirm the existence of email addresses to spammers. The MIME format, initially designed to enrich messages with different types of content, also serves to conceal malicious content. A dual-use technology.

Despite these weaknesses, SMTP still dominates global electronic messaging. Its longevity stems first from its conceptual simplicity: developers quickly grasp how it works. Second, its flexibility: the protocol accepts extensions without breaking with existing implementations. Finally, the

inertia of the immense installed base of servers and clients that use it.

This technical history reveals a broader truth about the evolution of computing standards. A protocol rarely survives on its intrinsic qualities alone. It must adapt to new uses while integrating emerging security constraints and preserving compatibility with its ecosystem. SMTP illustrates this delicate balance between innovation and continuity.

Current concerns focus on systematic encryption of communications, strengthened sender authentication, and adaptation to mobile messaging. The protocol continues to evolve, forty years after its birth. This exceptional longevity in the computing universe testifies to its robust initial design and its capacity for continuous adaptation.

The complex architecture now surrounding SMTP—anti-spam filters, authentication systems, security protocols—forms a sophisticated technical ecosystem. This growing complexity contrasts with the protocol's original simplicity.

Commodore 64

In 1982, Jack Tramiel launched a bold gamble with the Commodore 64. Known for his philosophy "computers for the masses, not the classes", the businessman set the price at $595—half that of the competition. This machine takes its name from its 64 KB of RAM, an impressive amount that foreshadowed unprecedented possibilities.

The secret to this economic performance lay in a decision made six years earlier. By acquiring MOS Technology in 1976, Commodore secured control of its components: the MOS 6510 processor, derived from the famous 6502, but also two innovative chips. The VIC-II handled display with its 16 colors, 320x200 pixel resolution, and especially its eight sprites, those small graphical objects that would bring arcade games to life. As for the SID, this integrated synthesizer offered three synthesis voices with precise control of ADSR sound envelopes. Few computers of the era could rival such audio capabilities.

The C64's architecture revealed particular ingenuity. The system juggled between ROM and RAM as needed, freeing up memory space when necessary. The PETSCII character set included semi-graphical symbols that transformed the screen into a true creative canvas. A novice programmer could design rudimentary interfaces without knowing a single line of machine code. The integrated BASIC and KERNAL system offered the

accessibility Tramiel sought, without limiting experts who explored the machine's most technical depths.

Commercial innovation accompanied technical innovation. The Commodore company abandoned specialized stores to invest in big-box retailers like K-Mart. This groundbreaking strategy brought the C64 into American, then European, households. Parents discovered an affordable "family" computer. Children found a sophisticated game console. This dual nature appealed to a much broader audience than computer enthusiasts alone.

The numbers speak for themselves: between 12 and 17 million units sold through 1994 made the C64 the best-selling personal computer in history. This exceptional twelve-year longevity testified to a robust and scalable design. When IBM-compatible PCs dominated the professional market, the little Commodore continued to reign over home entertainment.

The software library literally exploded. Thousands of titles emerged, from classics like *Impossible Mission* to amateur creations distributed via BBS networks. These electronic bulletin boards, ancestors of our forums, connected users by modem. People exchanged programs, tips, and shared passions. Tools like the *Shoot'em Up Construction Kit* democratized video game creation. No need to be a programmer to design your own shooter!

The EasyScript word processor and CalcResult spreadsheet proved that the C64 wasn't limited to gaming. The arrival of GEOS in 1986 revolutionized the user interface. This graphical system offered windows, icons, and drop-down menus, rivaling Apple's Macintosh in an otherwise different market segment.

But it was the community that truly forged the C64's soul. Companies like HAL Laboratory continually pushed technical boundaries. The "demoscene" was born from this creative emulation. These artist-programmers created stunning visual demonstrations, blending impossible graphical effects, fluid animations, and refined musical compositions. Their works proved that a modest machine could rival far more expensive systems, provided one mastered every register, every processor cycle.

With this chip, a true electronic musical instrument, composers like Rob Hubbard and Martin Galway created a specific musical style: "SID music". Their melodies left a lasting mark on video game soundtracks and influenced emerging electronic music. Who would have imagined that a home computer would spawn a new musical genre?

Sinclair ZX Spectrum

In 1980, the personal computer remained a mysterious object for most people. Machines cost a fortune and seemed reserved for experts or businesses with deep pockets. Clive Sinclair harbored a different ambition. This British entrepreneur, who had already made his mark with pocket calculators, wanted to create a computer that everyone could afford.

In April 1982, Sinclair Research Ltd launched the ZX Spectrum, a small black box that would challenge many assumptions. With its price of 125 pounds sterling for 16 KB of memory or 175 pounds for the 48 KB version, this machine literally shattered market conventions. The design was disarmingly simple: a compact case with a membrane keyboard that plugged directly into any family television.

Under this minimalist hood beat the heart of a Zilog Z80A at 3.5 MHz. Sinclair had made clever technical choices. The real breakthrough lay in color management: eight different hues with two brightness levels each, hence the name "Spectrum" which sounded like a promise. The screen displayed 256 × 192 pixels, a decent resolution for the time. Sound was limited to a simple buzzer, but no matter! Programs loaded from ordinary audio cassettes, the very same ones used for listening to music.

The system embedded a version of BASIC language adapted by Nine Tiles Ltd, designed to leverage the machine's graphics and sound capabilities. No frills in the operating system, just the essentials to make everything work. This stripped-down approach actually concealed remarkable intelligence: making computing accessible without oversimplifying it.

Before reaching British store shelves, marketing began through direct sales from Sinclair's offices. The reception exceeded all predictions. People discovered they could finally afford a "real" computer, not a gadget. The enthusiasm was such that a video game industry emerged almost instantly around this little black machine.

The Timex factory in Dundee, Scotland, ran at full capacity. Production climbed to 50,000 units per month in 1983 to meet unwavering demand. The milestone of one million units sold was reached as early as 1983, then two million the following year. The Spectrum became the best-selling personal computer in Europe.

This commercial success enabled Sinclair to develop an entire universe around its machine. The ZX Printer thermal printer, joystick interfaces, and especially the Microdrive drives launched in 1983 completed the offering. While their high price limited adoption, these small cartridges offered faster and more reliable storage than cassettes.

Specialized press flourished with magazines entirely dedicated to the Spectrum. Readers found programs to copy line by line and accessible programming courses. A true software industry took shape around publishers like Ultimate Play The Game, Ocean, or Imagine. Games grew in sophistication, exploiting every available byte of memory.

British national education did not remain indifferent to the phenomenon. Seeing in this machine a way to prepare an entire generation for computing challenges, the government subsidized Spectrum purchases in schools. Thousands of schoolchildren thus discovered programming on these small membrane keyboards.

Facing intensifying competition, Sinclair launched the Spectrum+ in October 1984. Same electronics, but in a redesigned case with a real mechanical keyboard inspired by that of the QL, the brand's high-end computer. Sold at 179.95 pounds, it appealed to those who had been put off by the original's membrane keyboard. An upgrade kit was offered to owners of the first models.

The software catalog grew day by day, exceeding 10,000 titles in 1985. While games dominated largely, professional and educational applications could also be found. The developer community shared its discoveries and tricks, creating remarkable creative excitement around the machine's technical limitations.

But the market evolved quickly. The arrival of the Commodore 64 and Amstrad CPC changed the game. These competitors offered better keyboards and superior multimedia capabilities for comparable prices. Spectrum sales began to flag as early as 1985.

Sinclair Research's financial difficulties worsened. The dismal failure of other projects like the QL computer or the C5 electric vehicle weighed down the accounts. In 1986, Amstrad purchased the computer division. Spectrum production continued under this new owner until 1992, totaling more than 5 million units sold across all versions.

The Spectrum literally democratized personal computing in Europe, taking it out of laboratories and installing it in family living rooms. Entire generations of programmers and video game creators cut their teeth on this little membrane keyboard. Its simplicity encouraged experimentation and understanding of computing mechanisms.

A passionate community continues to create for the Spectrum. Emulators allow reliving the experience on our modern machines, while developers enjoy pushing the limits of this architecture over forty years old. The Spectrum has become the symbol of a golden era when programming remained within everyone's reach.

Its cultural impact is measured less in market share than in vocations

inspired. This little black box demystified computing by showing that a computer could be simple, affordable, and powerful all at once. Clive Sinclair's vision, making technology accessible to as many people as possible, finds its finest illustration here. Other computers certainly succeeded it, more powerful and more comfortable, but none will have had a comparable social impact.

Thomson TO/MO

In 1979, in the workshops of Thomson's factory in Moulins, José Henrard handled electronic circuit board prototypes. This economist and sociologist, a CNRS researcher, discovered computing through his trainer brother's computer. Thomson recruited him with an ambitious mission: to design a French home microcomputer. His vision went beyond a mere technological gadget. He wanted to create a machine that would demystify computing, teach, and communicate.

His prototype integrated a Motorola 6809 8-bit microprocessor, considered among the most powerful of its time. With 8 KB of RAM expandable to 32 KB and a graphical resolution of 320×200 pixels in 8 colors, the TO7 stood out with its light pen included as standard. Users could now interact directly with the screen, whereas most computers still relied solely on a keyboard.

In September 1980, Henrard presented his prototype to Thomson's executives. Their bewilderment was evident when faced with what they perceived as a gadget. Nevertheless, the project received approval. Production began in Moulins. Two years later, Thomson, freshly nationalized under Alain Gomez's leadership, unveiled the first 100 TO7s at SICOB. The specialized press applauded: remarkable graphics quality, acknowledged entertainment dimension, robust Microsoft Basic. The only drawback: the price of 7,000 francs.

The following year was strategic. Thomson created SIMIV (International Society for Microcomputing and Video) to orchestrate its microcomputer policy. Jean Gerothwohl, a former advertising executive and Alain Gomez's fellow student at Sciences Po, took charge. His strategy prioritized education and adopted a long-term approach. Sales took off thanks to bold marketing campaigns: mass-market advertising, targeted promotions, comprehensive press kits.

1984 saw the birth of two new models. The MO5, priced at 2,390

francs, challenged British machines like the Oric or Spectrum. The TO7-70 came equipped with 64 KB of RAM. Thomson secured a decisive contract with UGAP: 40,000 machines over five years for the Ministry of Education. A success that paved the way for a greater ambition.

In January 1985, Laurent Fabius, then Prime Minister, announced the "Computing for All" plan. By seeking to introduce all French students to computing and train 110,000 teachers in one year, the objective bordered on utopian. Thomson was the main supplier with a massive order of 108,400 machines intended to equip 24,000 educational establishments. The Lille-based company Léanord developed the "Nanoréseau", an ingenious system allowing up to 31 Thomson computers to connect to a more powerful master machine.

The TO7-70 revealed its technical innovations. Its RAM reached 48 KB, expandable to 64 KB. It now handled 16 colors versus 8 previously. Its proprietary operating system interfaced with Basic and Logo. The computer multiplied the interfaces: cassette reader for storing programs, cartridge port for software, SCART socket for television, audio interface, game controllers.

A rich software library developed around education, particularly for learning French and mathematics. Business applications arrived with the Colorcalc spreadsheet and the Pique-Fiche database. Colorpeint democratized computer-assisted drawing. Games completed this diversified offering.

However, Thomson struggled against international competition. Export attempts failed, except for some sales in Italy and a contract with Algeria. In 1986, the company launched the TO9, a more powerful model priced at 8,990 francs, but compatibility with previous models posed problems. Users questioned, developers hesitated.

The arrival of low-cost PC compatibles, led by Amstrad, brutally disrupted the market. Thomson abandoned its proprietary standard strategy in 1986 to venture into PC compatibles with the TO16 range. Too late. Sales collapsed: 160,000 machines in 1986, 100,000 in 1987, 60,000 in 1988. The Saint-Pierre-Montlimart factory closed its doors, eliminating 450 jobs.

On January 27, 1989, Thomson announced the definitive end of its microcomputing activity. This decision, made without consultation with users or the Ministry of Education, closed nearly a decade of French industrial adventure. The final withdrawal occurred on January 1st, 1990. Thomson preferred to refocus on consumer electronics and defense.

The legacy of Thomson computers marks the computerization of French society. The "Computing for All" plan enabled the first massive introduc-

tion of computing in schools. Their design prioritizing direct interaction via the light pen and the educational dimension testifies to a distinctive vision of personal computing. The bold technical choices, from the 6809 microprocessor to the Nanoréseau, illustrate the innovative capacity of the French industry in the 1980s.

The commercial failure resulted from a national market that was too narrow, the absence of international breakthrough, high prices, and especially the relentless emergence of the PC standard. This adventure nevertheless represents a significant stage in the French computer industry, blending technological ambition, political will for national independence, and democratization of access to computing.

PostScript

In 1982, two researchers from Xerox PARC decided to walk out. John Warnock and Charles Geschke had just spent months developing a printing technology called Interpress, capable of describing a page to be printed without worrying about the type of printer used. The concept was innovative, but Xerox saw no commercial interest in it. The two men then founded Adobe Systems and got to work on their own version: PostScript.

The era was hardly conducive to this kind of venture. The 1970s were steeped in a purely computational conception of computing. IBM boasted in its advertisements about the derisory cost of multiplications on its machines. But at Xerox PARC, another idea was germinating: the computer would become a communication tool. Everyone would soon have their own personal machine, connected to printers shared over the network.

Adobe's initial plan was to manufacture printers and open printing service centers. Bill Hambrecht, their investor, persuaded them to change course. Why not instead design complete systems combining workstations and laser printers? That's when Steve Jobs burst onto the scene.

Jobs discovered PostScript and immediately grasped its potential. His Macintosh revolutionized the user interface, but printing was a disaster. The ImageWriter barely exceeded 72 dots per inch, a derisory resolution for serious use. In 1985, Apple released the LaserWriter, the first printer to embed PostScript. This machine contained more computing power than the Macintosh: 512 KB of read-only memory just for the PostScript code, a veritable fortune.

The announced price of $7,000 shocked Apple's sales staff. Three times the price of the Macintosh for a printer! Jobs held firm, and the opportune drop in memory chip prices ultimately vindicated his gamble.

PostScript solved a complex technical problem with a strikingly simple approach. The language treated text as a special case of graphics. This unification allowed characters to be manipulated with total freedom: they could be rotated, resized, and deformed at will. The idea was all the more appealing because it transcended the resolution of the output device. A page description could be printed on different types of printers while maintaining its quality.

Adobe didn't stop at creating a printing standard. In 1987, the company launched Illustrator, a vector drawing software. The era was bold because no graphic designer used computers. Illustrator's interface, based on Bézier curves and their control points, was born from a family anecdote. Warnock's wife, a professional graphic designer, regularly asked him to hand-code in PostScript the logos she had to create. This domestic experience directly inspired the software's interface.

The following year, Adobe acquired a small program developed by the Knoll brothers. This software would become Photoshop. The gamble seemed crazy: the largest hard drives didn't exceed 20 megabytes, just enough to store one high-resolution photo. Adobe was betting on hardware evolution, and history proved them right.

In 1989, PostScript's dominant position faltered. Apple and Microsoft joined forces to create TrueType, a competing format for fonts. Adobe responded by rapidly developing ATM (Adobe Type Manager) for displaying fonts on screen. The technical quality of the solution and the trust Adobe inspired allowed it to maintain its lead.

Two years later, Warnock had a revelation. Why not use PostScript to create portable documents? All they had to do was capture the PostScript stream generated by any application to obtain a file readable on any platform. This idea gave birth to Acrobat and the PDF format. For success to come, the Internet had to develop so that exchanging formatted documents would become an obvious necessity.

Adobe's triumph rests on remarkable insights. The company knew how to identify needs in the market. It maintained impeccable technical standards in typographic rendering. Its commercial strategy, which consisted of licensing PostScript to manufacturers rather than producing its own machines, proved judicious.

The PostScript language established principles that remain relevant today, such as the separation between document description and its rendering, freedom from hardware constraints, and uniform treatment of text

and graphics. PDF, the direct heir of PostScript, reigns over global document exchange. Beyond these technical aspects, PostScript transformed the computer into a tool for graphic creation and communication, definitively wresting it from its status as a mere calculator.

SunOS

In 1982, when Bill Joy left the corridors of the University of Berkeley, he took with him far more than a master's degree in electrical engineering. This man had spent years working intensively on BSD, that version of UNIX born in California's university laboratories. He created the `vi` editor, developed the C shell, and knew the system's inner workings better than anyone. He co-founded a new company: Stanford University Network, shortened to Sun. Sun Microsystems. An era when AT&T was just beginning to commercialize UNIX, and when computing was experiencing a rare period of excitement.

The first version of SunOS appeared in 1983. Joy and his colleagues took BSD as their foundation, but they didn't simply reproduce what existed. They grafted remarkably efficient networking tools onto it and developed the Network File System, that NFS which would later become indispensable in the industry. Meanwhile, Sun worked on windowing solutions for UNIX. Their objective? To make this system, known for its austerity, accessible to a wider audience.

The 1980s established SunOS as a reference, especially in advanced technical fields. Computer-aided design engineering firms eagerly competed for Sun workstations. These machines, powered by SunOS, vastly outperformed the personal computers of that era. They offered a stable multi-user environment, which changed everything for businesses accustomed to the limitations of single-user systems.

But in 1988, a bombshell shook the small world of UNIX. AT&T acquired equity in Sun Microsystems. Competitors were outraged, convinced that Sun would benefit from privileged information about UNIX's evolution. They responded by creating the Open Software Foundation, while AT&T, Sun, Data General, and Unisys established UNIX International. These maneuvers revealed the tensions running through the computer industry, torn between technical standardization and commercial warfare.

The landscape changed again in 1993. Sun announced that SunOS 4.1.4 would be the last version built on BSD. The company switched to

System V Release 4, the result of collaboration with AT&T that combined the strengths of System V and BSD. This new version inherited the commercial name Solaris, with SunOS becoming the technical designation for the kernel. A strategic shift that was part of a broader standardization effort in the UNIX world.

The graphical interface represented another arena for innovation and battle. Sun developed the OPEN LOOK interface with AT&T, hoping to create a unified user experience for all UNIX variants. But this interface clashed with OSF/Motif, built on X Window, a system developed at MIT that offered a significant advantage: the ability to run programs on a remote machine while displaying their output locally, regardless of hardware architecture or operating system.

The confrontation went in Motif's favor. Sun took it in stride and adapted: starting with Solaris 2.5.1, the company offered a package combining both interfaces under the name Common Desktop Environment. This ability to adapt demonstrated Sun's agility in the face of market changes and the emergence of new standards.

On the technical front, SunOS brought innovations that left a lasting mark on computing. The fast file system, introduced in BSD 4.2 thanks to DARPA funding, revolutionized performance and combated file fragmentation. NFS, a purely Sun creation, became the reference solution for network file sharing. These advances, coupled with deep integration with Sun hardware, explain the system's commercial success.

The history of SunOS reflects that of computing as a whole. Born in the academic world as a BSD variant, heir to that research tradition characteristic of UNIX, it evolved into a complete commercial solution. It integrated emerging standards without denying its technical specificities. This transformation accompanied that of Sun Microsystems, which moved from daring startup status to industry heavyweight.

When SunOS gave way to Solaris, a chapter closed, but the legacy remained. The innovations it carried, particularly in networking and distributed file systems, continue to influence the development of modern operating systems. Its trajectory illustrates how technical choices, commercial strategies, and industry standards intertwine to shape technological evolution.

SunOS helped establish UNIX as a credible platform for professional applications, particularly in technical and scientific sectors. Its success in universities and research centers trained an entire generation of computer scientists in UNIX concepts.

Agat

In 1980, while the Apple II was enjoying resounding success in American schools, the Soviet Union embarked on an ambitious yet obstacle-ridden technological venture. Kremlin leaders became aware of a worrying gap: their children were growing up without ever touching a computer keyboard. Faced with this reality, they decided to create the Agat, an almost exact copy of Steve Wozniak's machine.

The Agat story begins with a painful realization for Soviet officials. In a society that prided itself on being at the forefront of scientific progress, personal computing remained largely unexplored territory. The Ministry of Education understood that swift action was needed to train a generation of computer users as a priority. Rather than starting from scratch, Soviet engineers made the pragmatic choice to replicate what was working elsewhere.

This decision revealed a broader strategy that the USSR had been practicing for decades. While this approach raised questions about the country's technological autonomy, copying Western innovations saved time on research. The Agat fit into this logic of accelerated catch-up, with all that this implied.

Production therefore began in factories discovering the intricacies of consumer electronics. The first machines rolled off the assembly lines with a design strongly reminiscent of the Apple II, but the similarities often stopped at appearance. The machine's heart beat to the rhythm of a K588 processor, the Soviet version of the famous 6502. RAM ranged from 16 to 32 kilobytes depending on the model, respectable figures for the time.

Released in 1983, problems accumulated from the first months. Locally produced electronic components failed to deliver their promised reliability. Failure rates climbed, turning each computer delivery into a technological lottery. Schools receiving these machines discovered that some simply didn't work, or broke down after a few weeks of use.

The software challenge proved even more formidable. Reproducing hardware was possible with good engineers and perseverance, but creating a complete software ecosystem required resources and expertise that were lacking. Programs designed for the Apple II stubbornly refused to run properly on the Agat. Each adaptation required considerable work, slowing down the development of a worthy software library.

The production figures speak for themselves. In three or four years,

barely 6,000 units left Soviet factories. For a country with tens of thousands of schools, these volumes remained derisory. The goal of massively computerizing education became a pipe dream. Only a few pilot institutions benefited from these rare machines, creating a glaring inequality among Soviet students.

This scarcity stemmed from the structural limitations of Soviet industry. Central planning, so effective at producing steel or tractors, struggled to adapt to the demands of precision electronics. Delivery times stretched out, specifications changed mid-course, and coordination between different component suppliers turned into a logistical nightmare.

Technological isolation worsened the situation. Western embargoes on sensitive technologies deprived the USSR of advanced components and precision machine tools. Soviet engineers had to reinvent solutions proven elsewhere, losing precious time in a race where every month counted.

The Soviet state harbored ambiguous feelings toward the computerization it officially promoted. On one hand, leaders understood the strategic importance of these technologies. On the other, they feared the uncontrolled circulation of information that computer democratization might encourage. This mistrust showed through in the usage restrictions imposed on the Agat: no question of leaving them in the hands of unsupervised users.

Education officials also discovered that owning computers wasn't sufficient. Training teachers, creating adapted educational programs, maintaining the computer fleet: so many ancillary questions that hadn't been anticipated. Many Agats ended up in closets for lack of qualified personnel to use them.

By the mid-1980s, initial enthusiasm gave way to realism. Soviet officials began looking toward IBM PC clones, easier to produce and benefiting from a richer software ecosystem. The Agat, which was meant to symbolize Soviet technological independence, progressively came to symbolize failure.

Perestroika accelerated this reassessment. The gradual opening to Western technologies made the computer self-sufficiency project obsolete. Why persist in reinventing the wheel when you could buy or copy more efficiently what already existed? This question, long obscured by ideology, resurfaced with force.

The abandonment of the Agat marked the end of an illusion. The USSR discovered that catching up technologically required more than political will and massive investments. Modern technological ecosystems rest on complex networks of innovations, skills, and know-how that cannot be decreed from Moscow.

This experience left traces in the post-Soviet technological landscape. The delay accumulated in the 1980s still weighs on ex-USSR countries today. Russia remains largely dependent on Western computer technologies, a situation whose origins lie in the failure of projects like the Agat.

Beyond its purely technical aspects, the Agat adventure tells the story of a system confronted with its own contradictions. The USSR wanted to modernize its education without losing control of information, innovate without challenging its rigid structures, catch up with the West without drawing inspiration from its methods. These tensions, long masked by space and military successes, came to light with the emergence of personal computing.

The Agat bears witness to an era when technological innovation was a major geopolitical issue. Its failure was only the beginning of broader difficulties that the USSR would face with digital technology. In this technological race, the rules of the game had changed: individual creativity, organizational flexibility, and freedom of information took precedence over central planning and state secrecy.

The history of this Soviet computer reminds us that technologies never develop in a vacuum. The Agat, in its successes as in its failures, is a mirror of a USSR in transition, torn between its ambitions of power and the limits of its political and economic model.

Apple Lisa

In the corridors of Apple, late 1979, the excitement was palpable. A delegation had just returned from Xerox PARC in Palo Alto, and their faces betrayed barely contained enthusiasm. What they had seen there would radically transform a project that had seemed well underway. The Lisa, whose development had begun the previous year as a simple professional computer with a custom processor, would undergo a radical metamorphosis.

Larry Tesler had shown them Smalltalk. This graphical interface, with its overlapping windows and mouse for navigation, represented everything Apple had been seeking without knowing it. Tesler, captivated by the visitors' sharp intellect, would join them in July 1980. From the initial project, only the code name would remain, along with a few components and a handful of engineers. Everything else would be reimagined.

Apple was betting big on this venture. Fifty million dollars vanished

into the development labyrinth, the equivalent of 200 person-years of work. The team swelled to 300 people, a hundred of whom devoted themselves to the project's core: making hardware and software communicate in perfect harmony. The custom processor gave way to the Motorola 68000, better suited to the demands of multitasking and graphics.

Apple's engineers invented what would become our everyday computing experience. The menu bar appeared for the first time, accompanied by a single-button mouse. The clipboard concept was born in these laboratories, as was that famous trash can that would swallow our unwanted files. True, the Xerox Star already used icons, but the Lisa went further by allowing them to be grabbed with the mouse, moved, and opened with a double-click. Windows naturally overlapped, creating this electronic desktop effect.

The obsession with detail pushed the team to heights of perfectionism. Novice users paraded through the laboratories to test each interface element. Debates over terminology dragged on: what to call this button, that function? Translators spent months seeking equivalents in other languages, aware that every word mattered in the user experience.

Seven applications accompanied the operating system: LisaWrite for text, LisaDraw for drawing, LisaCalc for organized tables. LisaGraph translated data into meaningful curves, LisaProject orchestrated complex projects, LisList organized information into structured databases, and LisaTerminal opened doors to the outside world. All shared a common philosophy: consistency trumped originality.

The 12-inch screen displayed its black-and-white pixels with unprecedented precision. The Motorola 68000 processor pulsed through 1 MB of memory, expandable to double. The "Twiggy" drives, named after the era's slender model, proved temperamental. Sony would replace them with its more reliable 3.5-inch drives. The 5 MB ProFile hard drive completed the ensemble, a colossal amount of storage.

The multitasking operating system stood as a technical feat. Virtual memory and error protection demonstrated a maturity rare in the world of personal computers. Bill Atkinson etched his name in history with QuickDraw, this graphics engine capable of displaying 4,000 characters and 800 lines per second. The file system's redundancy mechanisms promised to limit catastrophes in case of sudden failure.

January 1983: Apple unveiled its creation to the world. The price: $9,995. An astronomical sum that made even the wealthiest companies raise an eyebrow. Sales started reasonably well, roughly 80,000 units sold in eighteen months. But enthusiasm quickly faded. The price tag discouraged buyers, third-party applications were slow to arrive, and most

importantly, a little brother appeared: the Macintosh, more affordable and more compact, stole the show from 1984 onward.

April 1985 marked the end of the Lisa adventure. Production ceased, leaving Apple with an embarrassing inventory. The computer was briefly reborn as the Macintosh XL, modified to run its little brother's applications thanks to the MacWorks program. Sun Remarketing bought the unsold units and artificially extended the machine's career. As for the last remaining units, they ended up in a landfill, collateral victims of a shareholder lawsuit.

Yet the Lisa had sown seeds that would germinate everywhere. Its DNA found its way into the Macintosh, then into Windows, GEM, and Framework. Pull-down menus, dialog boxes, drag-and-drop became obvious features. QuickDraw migrated intact to the Macintosh, MacProject replaced LisaProject, LisaDraw transformed into MacDraw. The desktop manager inspired the Finder. Lisa Pascal continued its career as a reference language.

Beyond purely technical aspects, the Lisa redefined our relationship with computers. It proved that a complex machine could remain accessible, that power doesn't exclude simplicity. Computing stepped out of its circles of insiders to address the masses. The principles it established (direct manipulation, visual consistency, immediate feedback) crossed decades without aging. But innovation alone isn't enough: one must still find the right moment, the right price, the right audience. The Lisa failed commercially but triumphed conceptually.

Coleco Adam

Coleco Industries decided to leave the familiar world of video game consoles in 1983 to venture into the personal computer market. This American company, which had achieved success with its ColecoVision, launched the ADAM, a machine meant to be revolutionary in its design.

The initial concept was appealing: why force families to buy a computer, printer, keyboard, and storage media separately when everything could be integrated into a single product? The ADAM thus arrived on the market as a complete package for $750. In the box, buyers found the central unit equipped with 80 KB of RAM (expandable to 144 KB), a detachable 75-key keyboard with a professional feel, a daisy-wheel printer capable of producing quality correspondence, and two game controllers to honor the brand's gaming origins.

The storage system was one of the machine's most distinctive features. Coleco had developed a proprietary format called the "digital data pack", a sort of large high-density magnetic cassette capable of storing up to 500 KB of data. This format promised storage capacities exceeding the 5.25-inch floppy disks of the era, but this promise would turn into a nightmare.

From a technical standpoint, the ADAM adopted a highly original architecture. Rather than concentrating all the intelligence in the central unit, Coleco had distributed multiple microprocessors throughout the system. The computer contained two, the keyboard had its own, and the printer also had its own electronic brain. This fairly visionary network approach was theoretically supposed to enable multitasking operations that no other home computer offered at the time.

SmartWriter, the ADAM's flagship software, was directly embedded in the machine's ROM. This integrated word processor represented an undeniable competitive advantage: where Apple II or Commodore 64 owners had to pay hundreds of additional dollars for WordStar or similar software, the ADAM offered this functionality right out of the box. SmartBASIC completed the base software offering, with announced compatibility with Apple's AppleSoft BASIC. Coleco promised future support for the CP/M operating system, the gateway to the professional world.

The ADAM's software catalog reflected Coleco's educational ambitions. SmartLOGO, developed in collaboration with Seymour Papert from MIT, offered programming instruction for younger users. Sophisticated games like Buck Rogers Planet of Zoom sat alongside personal management utilities. Backward compatibility with ColecoVision cartridges added a welcome gaming dimension to this package.

Unfortunately, reality would dampen the initial enthusiasm. From the first months of commercialization, user reports multiplied signaling repeated malfunctions. The printer, presented as a major asset, broke down regularly. The magnetic tape drives proved temperamental, and the provided technical documentation was largely insufficient for understanding this complex machine's operation.

The tape storage system crystallized all of the ADAM's design problems on its own. Users discovered the hard way that cassettes had to be removed before turning the machine on or off, otherwise their data would be permanently corrupted. This constraint, never clearly explained in the manual, caused the loss of countless hours of work and fueled the machine's poor reputation.

The network architecture, so attractive on paper, generated unexpected limitations. The printer served as the general power supply for

the entire system, making it impossible to replace with a faster or more reliable model. The much-vaunted multitasking was actually quite limited: it was impossible to do anything else while printing was in progress.

Coleco's policy toward third-party developers did not foster the emergence of a rich software ecosystem. Unlike Apple, which actively encouraged external developers, Coleco imposed restrictive licensing agreements and provided little technical information about its system. This closed approach limited opportunities to enrich the available software catalog.

Performance tests conducted by BYTE magazine in 1984 confirmed early users' mixed impressions. While SmartBASIC proved indeed faster than Apple's AppleSoft BASIC for certain calculations, tape read and write operations were dramatically slow compared to floppy disk drives. The absence of an automatic backup system and the difficulty copying data from one medium to another significantly complicated the machine's daily use.

Faced with this failure, Coleco attempted to salvage the situation. A 5.25-inch floppy disk drive was announced, along with a 300/1200 baud modem and a 64 KB memory expansion. The company promised a complete overhaul of the technical manual and fixes for the most troublesome software bugs. But these belated improvements could not save a machine whose reputation was already permanently tarnished.

The ADAM disappeared from shelves in 1985, after less than two years of chaotic commercial existence. The considerable financial losses from this project precipitated Coleco toward difficulties that would lead to its bankruptcy in 1988. This story cruelly illustrates the pitfalls awaiting new entrants to the personal computer market of the 1980s. Technical complexity, the requirement for absolute reliability, and the need for a solid software ecosystem constituted significant barriers to entry.

In hindsight, the ADAM had nevertheless introduced forward-thinking concepts to home computing. The integration of a word processor directly into ROM foreshadowed our modern operating systems with their preinstalled applications. The "all-in-one" package concept would durably influence personal computer design.

The Coleco Adam adventure is valuable testimony to the importance of execution quality in a technological innovation's success. It also reminds us how crucial an active developer community is for establishing a viable computing platform. These lessons remain entirely relevant for understanding current technology market dynamics.

Compaq Portable

While IBM dominated the market with its PC, a young Texas-based company, Compaq Computer Corporation, was preparing to shake up the rules of the game in 1983. Its bet: to create the first portable computer fully compatible with the IBM PC architecture.

The idea wasn't completely new. Two years earlier, Adam Osborne had already commercialized his Osborne 1, presented as the first portable computer to achieve genuine commercial success. But Osborne's machine suffered from a fatal flaw: it couldn't run software designed for IBM PCs, which were establishing themselves as the market standard. This was exactly the problem that Compaq's engineers would solve.

The Compaq Portable came in the form of a 12 kg briefcase, equipped with a sturdy handle. Its design heralded what would soon be called "luggables", machines that one dragged more than carried. The 9-inch CRT screen stood at the center of the case, the keyboard detached to provide access to the 5.25-inch floppy drives. An external "power brick" completed the package, adding its considerable weight to that of the machine.

Under the hood, there was an Intel 8088 processor, the same one that equipped the IBM PC. This total technical compatibility allowed a portable computer for the first time to run exactly the same programs as a desktop machine. The monochrome screen, expandable RAM, and floppy drive completed a package sold for $3,590, a price that clearly positioned it in the high-end segment.

Success came quickly. Professionals finally discovered a machine that gave them the freedom to work on the go without abandoning their software habits. The build quality and system reliability won over a demanding clientele, accustomed to IBM standards. Compaq also relied on a carefully selected distribution network and impeccable technical support.

In 1986, the Texas firm made a major splash with the Portable 386. This new generation integrated the Intel 80386 processor clocked at 20 MHz, a world first that propelled Compaq to the rank of technological pioneer. Performance soared: 25% faster than 16 MHz machines, and up to 3.5 times faster than computers based on the 80286 at 8 MHz.

The machine came in two versions: the Model 40 with its 40-megabyte hard drive, and the Model 100 which doubled the storage capacity. The most spectacular innovation lay in the adoption of a dual-mode plasma screen, with a resolution of 640 x 400 pixels unprecedented for this type of machine. One megabyte of RAM equipped the base configuration, expandable to 10 megabytes.

Ergonomics received careful attention. The detachable keyboard gained 12 programmable function keys, ventilation doubled in efficiency, and shock-resistant mounts protected the hard drives. Dimensions shrank: 24.8 x 40.6 x 19.8 centimeters for a weight reduced to 9 kg, a significant decrease compared to the initial model.

Connectivity expanded with serial and parallel ports, an RGBI output for external monitor, and the possibility of installing an internal 1,200 or 2,400 baud modem. An optional expansion module added two 8/16-bit slots. For backup, a 40-megabyte magnetic tape unit completed the accessories offering.

These successes propelled Compaq to the heights of American industry. Between 1983 and 1986, the company broke all records by becoming the fastest company to reach the Fortune 500 ranking. This meteoric rise validated the existence of a market for high-end professional portable computers and prompted the competition to join the race.

The "luggable" concept naturally evolved toward greater lightness. In 1988, Compaq introduced the SLT computer, its first portable adopting the "clamshell" format that characterizes our modern laptops. At 6 kg, the SLT marked a step toward true mobility, even though current standards remained distant.

The adventure of the Compaq Portable reveals the importance of compatibility in the computer economy. By offering machines 100% compatible with the IBM PC standard, Compaq established a business model that the entire industry would adopt. This strategy accelerated market standardization of PCs and gave rise to a genuine personal computing industry.

The Compaq Portable matched the performance of a desktop machine, a true revelation at the time. Commercially, it validated the specialized distribution model and established quality standards that would durably influence the sector.

This period of intense innovation saw manufacturers seeking the balance between power, autonomy, and portability. The compromises of the Compaq Portable, with its power "brick" and its 12 kg, may seem archaic by our current standards. Yet, this machine paved the way toward modern mobile computing and proved that successful technological innovation always results from the convergence of technical advances, an accurate vision of the market, and a coherent commercial strategy.

Thanks to this winning combination, Compaq established itself as a major player in the computer industry, a position it would maintain until its absorption by Hewlett-Packard in 2002. The story of the Compaq Portable is that of a company that knew how to transform a technical

constraint—IBM compatibility—into a decisive competitive advantage.

GNU Project

The story begins in 1983 in the corridors of MIT's artificial intelligence laboratory. Richard Stallman, a seasoned programmer, experiences something that will change his life and, consequently, the history of computing. Faced with a malfunctioning Xerox printer, he encounters an outright refusal: no access to the source code to fix the problem. This frustration crystallizes a deeper unease: the software industry is shifting toward a closed model that breaks with the tradition of sharing that had characterized university laboratories until then.

This realization drives Stallman to announce, that same year of 1983, the launch of a project of rare ambition: GNU. The recursive acronym "GNU's Not UNIX" conceals a revealing objective: to build a complete operating system, UNIX-compatible certainly, but freed from any proprietary constraints. Every user must be able to study, modify, and share the programs they use daily.

To give substance to this vision, Stallman takes a decisive step in 1984: he resigns from MIT. This break guarantees him intellectual property rights over his future work. He first tackles a text editor, GNU Emacs, which achieves unexpected success in the computing community. The momentum created encourages what follows: GCC, the compiler, emerges in 1987, followed by the GDB debugger and a full range of system utilities.

In 1985, an institutional turning point occurs with the creation of the Free Software Foundation. This organization goes beyond mere technical support for the GNU project: it forges the philosophical and legal foundations of an emerging movement. Stallman defines four freedoms he considers inalienable: to use, study, modify, and redistribute software. These principles find their legal expression in 1989 with the General Public License, a license that cleverly turns copyright against itself to guarantee the permanence of the freedoms granted.

Development is organized according to a model unprecedented for the time. Small teams of geographically dispersed volunteers, coordination through mailing lists, file exchanges via FTP: the Internet makes possible this decentralized collaboration that is only the beginning of our modern software development methods. Each team works on specific components according to a meticulous plan, an approach that contrasts with the improvisation often criticized in community projects.

Around 1990, the GNU edifice approaches completion. Compiler, debugger, editors, system utilities: nearly all elements of an operating system are in place. Yet the centerpiece is missing: the core of the system, its kernel. HURD, the GNU kernel, becomes mired in unexpected technical complexity. This is where a young Finnish student, Linus Torvalds, comes in, launching the development of Linux in 1991. The meeting between GNU tools and this new kernel gives birth to a complete and free system that Stallman names GNU/Linux, thus emphasizing his project's contribution.

The GPL gradually becomes a de facto standard, adopted by thousands of developers worldwide. Companies like Cygnus Solutions, founded in 1989, prove that a viable business model can be built around free software by selling support and services. The collaborative methodology initiated by GNU inspires an entire generation of projects.

The questions raised by Stallman and the FSF fuel broader debates about intellectual property in the digital age. The concepts of software freedom spread to other domains: free culture, open data, open science. This conceptual expansion reveals the philosophical scope of an initiative born from technical frustration.

The 1990s and 2000s confront the GNU project with new challenges. Software patents, digital rights management, emergence of a more pragmatic "open source" movement: so many developments that test the movement's doctrinal coherence. The FSF maintains its uncompromising stance on freedom issues, sometimes at the cost of relative marginalization compared to less radical approaches.

Yet the facts prove this perseverance right. GNU/Linux establishes itself massively in web servers, conquers supercomputers and, through Android, infiltrates billions of mobile devices. GCC remains an essential technical reference. GNU tools continue to equip the majority of modern development environments.

The GNU project has demonstrated, beyond its technical achievements, that an initiative driven by strong ethical convictions could durably transform an entire industrial sector. By simultaneously creating high performance tools and a coherent legal-philosophical framework, Stallman and his collaborators founded a viable alternative ecosystem.

This historical contribution illuminates our contemporary issues. At a time when technology shapes our societies with growing intensity, the questions raised by the GNU project about freedom and control in the digital world resonate with particular acuity. The technical utopia of 1983 has transformed into economic and social reality, proving that ideas, even the most radical ones, can change the world when they meet the

appropriate technical tools.

DNS

The history of the Domain Name System begins with a file. A simple text file called HOSTS.TXT that the SRI Network Information Center laboriously distributed across ARPANET in the early 1980s. This makeshift solution worked as long as the network remained limited to a few dozen machines, but the explosion in the number of connected computers transformed this system into an administrative nightmare.

Paul Mockapetris and his team at the USC Information Sciences Institute observed this drift with concern in 1982. The HOSTS.TXT file grew larger every day, its distribution was costly, and nobody could keep up with the pace of modifications. Technological evolution worsened the situation: whereas the original ARPANET primarily connected shared central systems, the arrival of individual workstations exponentially multiplied the number of machines to reference. The scale shifted from the number of organizations to the number of users—a dizzying prospect for the time.

Faced with this impasse, Mockapetris explored existing alternatives. He studied IEN116 from DARPA Internet, examined XEROX's Grapevine, but no system truly met the specific needs of the Internet. The team then decided to design an entirely new architecture, distributed by nature: the Domain Name System. The first specifications appeared in 1983 in RFCs 882 and 883, technical documents that would transform the Internet.

The central idea of DNS lies in its tree structure. Each node of the tree carries a label, and a computer's full name is constructed by ascending from node to node up to the root. This natural hierarchy makes delegation of authority elegant because each organization manages its portion of the namespace without depending on a central body for routine modifications. The first top-level domains reflected this pragmatic philosophy: .edu for education, .com for businesses, .fr and .uk for countries.

The technical architecture rests on two pillars. On one side, name servers store and distribute information for their zone of responsibility. On the other, resolvers integrated into client systems query these servers to locate the requested information. Between the two, a sophisticated caching system temporarily stores responses according to a time-to-live defined by each administrator. This caching constitutes DNS's perfor-

mance secret: it avoids endlessly repeating the same queries to distant servers.

The transition from HOSTS.TXT did not happen overnight. The University of Berkeley took the lead in 1985, becoming the first organization to completely abandon the old system in favor of DNS. This pioneering migration immediately revealed practical problems: users struggled to adapt to new addressing formats, applications had to be reprogrammed to handle temporary failures inherent in any distributed system.

In 1987, approximately 300 domains had been delegated by the SRI-NIC. A year later, this number already exceeded 650, including 400 for normal namespaces and 250 for network address spaces. Seven root servers ensured the system's global redundancy, judiciously distributed across the Internet's main backbone networks. This exponential growth validated Mockapetris's intuitions: DNS absorbed without difficulty a load that the old system could never have supported.

The 1990s saw DNS usage explode with the arrival of the Web and the democratization of the Internet. The system proved its robustness in the face of this unexpected surge in demand. New record types were progressively added to handle new functionalities, such as MX records that revolutionized email routing. One of the protocol's major strengths lies in this capacity for evolution without breaking compatibility.

But success also attracted malicious interest. In the early 2000s, cache poisoning attacks (DNS spoofing) multiplied, and domain hijackings made headlines in specialized publications. The technical community responded with cryptography through DNSSEC, a protocol for securing DNS exchanges. Its deployment began symbolically in 2010 for the DNS root, before progressively extending to top-level domains and then to organizations.

Internationalization raised other issues, both technical and political. How could Chinese, Arabic, or Russian users be allowed to use their own characters in domain names? IDNs (Internationalized Domain Names) provided an elegant answer in the 2000s, but their implementation required profound modifications to ensure compatibility with billions of existing systems.

DNS governance underwent a major evolution in 1998 with the creation of ICANN. This non-profit organization took over from the U.S. government to coordinate the global domain name system. ICANN oversees the allocation of new top-level domains, establishes naming policies, and attempts as best it can to reconcile technical and commercial interests.

Between 2012 and 2024, DNS went through a new expansion phase. ICANN's new gTLD program released hundreds of novel extensions: .shop,

.paris, .blog, and thousands of others. Simultaneously, privacy concerns drove the development of DNS-over-HTTPS and DNS-over-TLS. These protocols encrypt DNS queries, limiting the possibilities for surveillance and censorship by intermediaries.

Forty years after its conception, DNS continues to operate on its original principles. This exceptional longevity is explained by Mockapetris's architectural choices: favoring simplicity over sophistication, flexibility over premature optimization. The caching mechanisms and hierarchical distribution of authority remain relevant despite technological evolution. DNS illustrates how an infrastructure can grow while preserving its compatibility with existing systems.

In 2024, DNS manages billions of domain names and responds to hundreds of billions of daily queries. This invisible infrastructure supports all global Internet traffic. Its current challenges concern security against denial-of-service attacks, query privacy, and root server resilience.

The DNS adventure tells how a technical innovation can restructure the Internet's architecture, transforming a simple directory into a critical global infrastructure. Its modular and evolutionary design is a model for all protocols aspiring to longevity on the Internet.

C++

In the 1970s, Bjarne Stroustrup discovered at Cambridge the joys and frustrations of Simula. He liked the language for its ability to structure code with classes, but was exasperated by its slow execution. For his thesis on distributed systems, he had to abandon Simula's elegance and return to BCPL, faster but less organized. This contradiction left a mark: why should one have to choose between performance and code clarity?

When Stroustrup joined AT&T's Bell Labs in 1979, he kept this question in mind. The projects he worked on—analysis of the UNIX kernel, network distribution—demanded both efficiency and structure. He set himself the bold challenge of grafting Simula's classes onto the C language. Thus was born "C with Classes", a hybrid that retained C's speed while adding object-oriented organization.

The first steps were modest. Stroustrup added classes, then constructors and destructors to automatically manage object initialization and destruction. Public-private access control appeared next, followed by virtual functions that enabled polymorphism. Each addition answered a specific

need encountered in the field.

In 1983, Rick Mascitti suggested a new name: C++. C's increment operator perfectly symbolized this evolution of the language. The name stuck and history would remember this designation. Two years later, in 1985, the first commercial version emerged from Bell Labs, accompanied by Stroustrup's foundational book, *The C++ Programming Language*. The Cfront compiler translated C++ code into C, an elegant solution to ensure maximum portability.

Success came quickly. Programmers discovered that they could retain their C habits while accessing the benefits of object orientation. No abrupt change, but a smooth transition. The language didn't slow down execution compared to C, didn't require a specialized environment like Smalltalk. This pragmatic approach won over a growing community.

Between 1985 and 1989, C++ grew richer. Multiple inheritance appeared in 1989, templates opened the way to generic programming, exception handling brought greater robustness. Each feature matured first in use before being integrated into the language. This caution forged C++'s reputation for stability.

The growth was impressive: from a few hundred users in 1985, the number rose to 400,000 in 1991. Application domains multiplied: operating systems, video games, graphical interfaces, scientific computing. Development environments appeared, specialized libraries enriched the ecosystem.

In 1989, the standardization process began under the auspices of ANSI and ISO. The work culminated in 1998 with the publication of the first international standard (ISO/IEC 14882:1998). This standardization freed C++ from its creator and ensured its longevity beyond Bell Labs.

The Standard Template Library deserves special mention. Alexander Stepanov developed a transformative approach: generic containers (vector, list, map) and reusable algorithms that work with all these containers. This library transformed the way C++ was written by favoring the reuse of proven code.

The `iostream` input-output system replaced the `printf` and `scanf` inherited from C. Type-safer, more extensible, it integrated naturally into the C++ philosophy. These additions to the standard library made the language richer without burdening the compiler core.

Stroustrup defended constant principles throughout the evolution: no performance degradation compared to C, backward compatibility preserved and support for varied programming styles. C++ accepts classic procedural programming, object-oriented with classes and inheritance,

generic programming with templates. This flexibility explains its adoption in very diverse contexts.

C++ popularized object orientation in domains traditionally reserved for procedural languages. Java and C# would draw heavily from its concepts, even if they made different architectural choices. Generic programming with templates inspired many subsequent languages.

C++ continues its evolution through successive standards. C++11 modernized the language with lambda expressions and smart pointers. C++14, C++17 and C++20 continue to expand possibilities: modules, coroutines, concepts. The language adapts to multicore architectures, parallel computing needs, and modern programming constraints.

This longevity stems from its designers' pragmatic approach. Rather than revolutionizing for revolution's sake, they always favored practical utility and preservation of the existing. C++ remains true to its origins: a powerful tool for those who want fine-grained control over resources without sacrificing code expressiveness. The history of this language illustrates how a good idea, nourished by field experience, crosses decades by adapting without losing its identity.

Objective-C

Brad Cox and Tom Love probably never imagined that their decision to hybridize the C language with Smalltalk concepts would give birth to one of the most enduring technologies in modern computing. Objective-C, created in 1983, would traverse the decades by constantly adapting.

The starting point was nevertheless simple. Dennis Ritchie and Ken Thompson had designed C in Bell Labs in the late 1960s to develop UNIX. This procedural language offered an interesting compromise between performance and readability, but its limitations became apparent in building complex systems. On the other side, at Xerox PARC, Alan Kay and Dan Ingalls were revolutionizing the programming approach with Smalltalk and its objects that encapsulated data and behaviors.

Cox and Love envisioned marrying these two worlds. Rather than creating a new language from scratch, they extended the existing C by grafting Smalltalk's object-oriented programming onto it. This pragmatic approach preserved the investment in C code while opening new possibilities for software design. The Free Software Foundation recognized the value of this innovation and integrated it into its toolbox.

Objective-C became a niche language appreciated by curious computer scientists. When Steve Jobs left Apple in 1985 to launch NeXT, he bet on Objective-C to develop NeXTSTEP. This technical choice transformed the language's destiny. NeXT pushed collaboration to the point of working with Sun Microsystems on OpenStep, a standardized version of their system. The Free Software Foundation followed suit with GNUstep.

Apple, bogged down in the 1990s, desperately sought to modernize Mac OS. After several failed attempts, two options presented themselves to the company: acquire Be, Inc. from Jean-Louis Gassée or buy back NeXT. The choice of NeXT in 1996 sealed Jobs's return and made Objective-C the technical heart of Mac OS X. A simple experimental language suddenly became the foundation of Apple's future.

The iPhone arrived in 2007. iOS, derived from Mac OS X, propelled Objective-C onto millions of mobile devices. The iPad three years later amplified the phenomenon even further. Developers worldwide discovered this hybrid language to create applications reaching hundreds of millions of users. This unexpected explosion revealed the robustness of the architecture imagined by Cox and Love twenty-five years earlier.

Faced with this massive adoption, Apple introduced substantial improvements, notably Automatic Reference Counting (ARC), which drastically simplified memory management. Gone were memory leaks and crashes related to lost pointers: ARC automated these tedious and error-prone tasks.

Objective-C's syntax bears the traces of this turbulent history. Traditional C coexists with Smalltalk's brackets in a cohabitation that surprises at first glance but reveals its coherence in use. The message system, inherited from Smalltalk, replaces direct method calls with a more flexible mechanism where objects communicate with each other. This approach enables dynamic method binding and facilitates system extension.

Frameworks accompany this syntactic richness. Foundation provides the basic building blocks: data manipulation, collections, file operations. These libraries form a coherent whole that accelerates development and standardizes practices. The `id` type perfectly illustrates the language's philosophy: it allows generic manipulation of objects while preserving type safety when necessary.

The ecosystem developed beyond Apple's borders. GNUstep maintains Objective-C's cross-platform portability, demonstrating that the concepts transcend the proprietary implementation. Xcode enriches the developer experience with Interface Builder and its visual tools. The community creates libraries, frameworks, and best practices that nurture a virtuous circle.

Swift arrived in 2014 as Apple's new champion. This announcement did not sound the death knell for Objective-C but testified to its maturity. Apple organized a gradual coexistence rather than an abrupt break, implicitly acknowledging the value of its historical language. The millions of lines of Objective-C code will continue to function for years.

Objective-C demonstrated that extending an existing language could succeed where creating from scratch often fails. Its message model inspired numerous object-oriented frameworks. Its ability to mix procedural and object paradigms opened paths explored by other languages.

Forty years after its birth, Objective-C remains a lesson in pragmatic engineering. Cox and Love knew how to identify the strengths of two different worlds and intelligently merge them. Their creation survived trends, acquisitions, mobile devices, and continues to power applications used daily by hundreds of millions of people. This longevity testifies to a solid design that knew how to adapt without losing its identity.

IPX/SPX

In the mid-1980s, when Novell was working on NetWare, its engineers faced a challenge: creating a network protocol capable of enabling communication between computers distributed throughout the enterprise. They didn't start from scratch. The XNS system that Xerox had developed a few years earlier served as their model. This pragmatic approach gave birth to IPX/SPX, a protocol suite that would leave its mark on an entire decade of network computing.

The era was marked by the first steps in networking. Companies were discovering the value of sharing files and printers across multiple workstations. NetWare offered an answer to these needs with its client-server architecture, where IPX/SPX played the role of messenger between machines. The protocol was structured in two distinct layers: IPX for routing packets from one point to another, SPX for guaranteeing their proper reception.

Novell had found an elegant trick to simplify network configuration. Where other protocols required complex mapping tables, IPX used the network card's MAC address directly as the machine identifier. Each IPX address combined a 32-bit network number, defined by the administrator, with this famous 48-bit MAC address. Ultimately, there was no longer any need for an additional protocol to link the logical and physical identities of the machine.

Routing relied on RIP (Routing Information Protocol), but not IP's RIP. The two protocols shared the name without being compatible. Every minute, IPX routers exchanged their routing tables using a particular metric called a "tick". A tick corresponded to one-eighteenth of a second, providing an estimate of the delay on each link. If two routes showed an identical number of ticks, the hop count would decide.

SAP (Session Announcement Protocol) added an extra dimension to the IPX/SPX ecosystem. This service announcement protocol transformed the network into a vast bulletin board where each server proclaimed its capabilities: file sharing, printing, messaging. As with RIP, announcements repeated every minute. Routers maintained a list of all available services, ready to inform any client seeking a particular resource.

IPX/SPX's adaptability manifested in its ability to coexist with different Ethernet standards. The protocol accepted multiple encapsulation types: Novell's proprietary format, IEEE 802.2 and 802.3 standards, Ethernet II, and SNAP. This flexibility facilitated integration into heterogeneous environments, provided consistency was maintained in the configuration of all equipment.

At the top of the protocol stack, NCP handled business aspects: file access, print control, security. NetBIOS found its place through an emulation layer that allowed existing applications to continue functioning. NLM modules extended the system's possibilities with new protocols, communication services, or database access.

The 1990s saw IPX/SPX flourish in enterprises. Its ease of implementation and good performance made it a natural choice for network administrators. But the Internet was growing and TCP/IP was establishing itself as the universal language of digital communications. Faced with this rise in power, Novell had to revise its strategy.

The company gradually integrated TCP/IP into NetWare. Later versions offered the ability to encapsulate IPX in UDP/IP packets to traverse networks based on the Internet protocol. The arrival of NDS in NetWare 4.0 reduced dependence on the SAP protocol by allowing clients to locate services through a centralized directory.

The transition accelerated in the 2000s. New applications were born with TCP/IP in their DNA, relegating IPX/SPX to the status of legacy technology. Novell eventually abandoned development of its protocol suite to turn toward IP-based solutions.

This evolution shouldn't obscure IPX/SPX's lasting influence. The direct use of MAC addresses, automatic service discovery—these concepts would resurface in more recent technologies. Modern protocols like mDNS or UPnP revisit ideas that SAP had already explored.

The story of IPX/SPX tells the tale of an era when network protocols competed to impose their vision of communication. Faced with the universal standardization of TCP/IP, proprietary solutions, whether technically accomplished or not, could only fade away. This lesson extends beyond the purely technical realm: in an interconnected world, openness and interoperability often prevail over local excellence.

IPX/SPX's fate ultimately illustrates an immutable truth of computing: no technology, however brilliant, survives without adapting to market changes. TCP/IP didn't triumph solely through its intrinsic qualities, but because it became the *lingua franca* of the Internet. In this battle of protocols, universality had the last word.

SMB

In 1984, IBM created a protocol that would transform how computers share their resources: SMB, or Server Message Block. At the time, no one imagined this technology would become the backbone of modern enterprise networks. Yet the concept seemed simple: enable remote machines to share files and printers via NetBIOS.

Microsoft seized this innovation. As early as 1987, the Redmond firm integrated SMB into its LAN Manager program, followed the next year by IBM with its OS/2 LAN server. These initial implementations marked the beginning of a system that would soon colonize offices worldwide. Local network resource sharing had finally found its path.

Nine years later, Microsoft reached a decisive milestone. Windows NT 4.0 arrived in 1996 with CIFS, the Common Internet File System. This dialectal version of SMB adapted to increasingly complex enterprise networks. The IETF published a draft standard the following year, while SNIA proposed a technical specification in 1999. The protocol gradually emerged from its proprietary constraints.

The millennium changed, and with it the protocol's designation. Windows 2000 returned to the SMB name while bringing considerable extensions. Dedicated TCP ports (445) were added to the traditional NetBIOS ports (137-139). This evolution accompanied NetBIOS's migration to TCP/IP, reflecting the growing maturity of network infrastructures.

The year 2006 marked a turning point. Windows Vista introduced SMB 2.0, a version that revolutionized the protocol's architecture. Microsoft streamlined the design: the 75 commands of the old version were

reduced to 19 instructions. Asynchronous operations appeared, reads and writes gained efficiency. HMAC SHA-256 hashing replaced the antiquated MD5, strengthening the security of a protocol that had become indispensable.

Windows 7 refined this foundation with SMB 2.1 in 2009. Cache management improved, high-bandwidth networks benefited. The file leasing system replaced the opportunistic locking mechanism, reducing network chatter while improving metadata management.

But 2012 witnessed SMB's finest transformation. Windows 8 and Windows Server 2012 delivered SMB 3.0, a version that finally addressed modern enterprise needs. High availability became possible through transparent failover. *Multichannel* aggregated multiple network connections, multiplying performance. The RDMA protocol integrated naturally, achieving previously unimaginable throughput. AES-CCM encryption secured the most sensitive exchanges.

The evolution didn't stop there. SMB 3.02 arrived with Windows 8.1 the following year, bringing support for asymmetric clusters and perfecting RDMA. Windows 10 introduced version 3.1.1, further hardening security against growing threats.

Meanwhile, another story unfolded in the free software world. Andrew Tridgell developed Samba, an alternative implementation that reconciled the Windows and UNIX universes. This titanic reverse-engineering project forced Microsoft to publish its specifications, democratizing a long-guarded protocol. Samba transformed SMB into a de facto standard, transcending operating system boundaries.

This four-decade evolution tells the story of enterprise networking. From a simple local sharing mechanism, SMB evolved into a complex system managing performance, security, and high availability. Each version responded to shifts in enterprise computing: virtualization, cloud, data protection.

Security remains the protocol's Achilles' heel. The early LM and NTLM authentication mechanisms showed their weaknesses. NTLMv2 and packet signing, introduced with Windows NT 4.0 SP3, patched the gaps. Recent versions integrate robust encryption and anti-attack protections, addressing contemporary threats.

Having become the universal standard for file sharing, SMB equips all modern operating systems. Its constant adaptation accompanies distributed architectures, cloud storage, and new cybersecurity threats.

The vulnerability discovered by the Cult of the Dead Cow in 2001 (CVE-2008-4037) necessitated major fixes. SMB relay attacks required

the addition of specific protections. Each incident forged a more resilient protocol.

Modern IT environments demand ever more: increased performance, enhanced security, adaptation to new storage architectures. This perpetual capacity for evolution explains the longevity of a protocol born forty years ago, still indispensable to 21st-century enterprises.

Lotus 1-2-3

VisiCalc reigned supreme in the spreadsheet market when Mitch Kapor and Jonathan Sachs decided to create their own software. It was 1982, and these two men knew well the universe they were about to disrupt. Kapor had worked at Personal Software, VisiCalc's publisher, where he designed VisiPlot and VisiTrend for graphical visualization. Sachs had already programmed spreadsheets, notably for Data General. This field knowledge gave them a head start: they knew exactly what existing solutions were missing.

The IBM PC, launched the previous year, brought new credibility to microcomputing and offered technical capabilities far beyond 8-bit computers. Its 640 KB of memory and 16-bit processor opened unprecedented possibilities. Kapor and Sachs decided to bet on this platform and design a spreadsheet that would fully exploit its resources.

Their technical approach broke with their competitors'. While others used high-level languages, they chose assembly language to maximize performance. They programmed direct interface with video hardware, bypassing the BIOS to accelerate display. They rethought the cell recalculation order to optimize the updating of interdependent formulas, resulting in execution speed five times superior to other spreadsheets.

The user interface benefited from remarkable innovations. Pull-down menus with mobile cursor and contextual prompts made the software accessible to novices. Graphics integration instantly transformed data into diagrams, creating spectacular demonstrations that sparked prospects' enthusiasm. Keyboard macros, added despite Sachs's initial reluctance, gave experienced users the ability to automate their tasks and develop true applications.

Lotus Development Corporation didn't settle for a good product: the company also innovated in its commercial strategy. It trained its resellers, took care of customer support, and accompanied the software with a

demonstration disk featuring an interactive tutorial. The documentation was carefully crafted. Above all, the company invested heavily in advertising in the business press, an unusual approach for a software publisher at that time.

The launch at Comdex in fall 1982 exceeded all expectations: $900,000 in orders in the first few days. In 1983, the first full year of commercialization, revenue reached $53 million. The following year, it jumped to $156 million, and staff grew from 20 to 750 people in just two years, making Lotus the world's largest software publisher.

This success was explained by a perfect alignment of the stars. The product answered exactly the needs of companies discovering microcomputing. Its compatibility with VisiCalc files facilitated migration. The IBM PC installed base was experiencing explosive growth. Product quality and commercial execution left no room for competitors.

But this lightning success created its own difficulties, making company growth unmanageable. Original teams moved to other projects. Development of new products proved complex. Symphony, an attempt to create an integrated office suite, disappointed. Jazz, the Macintosh version, failed against Microsoft's Excel.

Diversification attempts multiplied with Metro, Express or Manuscript, never recapturing the magic of 1-2-3. Only Notes, developed by Ray Ozzie, met with real success after IBM's acquisition of Lotus in 1995.

Technological evolution eventually caught up with the leader. The transition to Windows proved late and laborious. The assembly code, initially an asset, ultimately became a burden for evolving the product. Microsoft Excel, designed from the outset for graphical interfaces, gradually gained market share and established itself as the new standard.

Lotus 1-2-3 demonstrated the gigantic commercial potential of the professional software market. It established the standards for modern user interface with its pull-down menus and keyboard shortcuts. Macros enabled programming by end users. The DIF (Data Interchange Format) facilitated data exchange between applications.

Lotus 1-2-3 transformed the perception of personal computers in business, from experimental tool to indispensable instrument for individual productivity. This transformation accelerated the computerization of organizations and founded a true software industry. Kapor and Sachs's spreadsheet wasn't just a product: it was a catalyst that precipitated the business world's entry into the digital era.

Microsoft Word

November 1983. Readers of *PC World* discovered with surprise a floppy disk tucked inside their magazine. On that diskette: Multi-Tool Word, an unknown word processor from Microsoft. This bold distribution marked the commercial debut of software that would radically transform our relationship with digital writing.

The story begins ten years earlier, however, in the refined laboratories of Xerox PARC. Charles Simonyi, a young Hungarian computer scientist fresh from Stanford, joined this innovation hub in 1972. He worked on Bravo, a project that challenged conventions: for the first time, a word processor displayed text exactly as it would appear on paper. Gone were cryptic formatting codes, gone were surprises at printing. The concept bore a name that would make its fortune: WYSIWYG, "What You See Is What You Get".

Bravo remained confined to the laboratories. Xerox, the photocopying giant, failed to grasp the commercial potential of this research. Bill Gates, however, had his eye on it. In 1981, he recruited Simonyi with the specific mission of creating Microsoft's office applications. The Hungarian drafted a three-and-a-half-page memorandum that defined the entire strategy, an integrated software suite for individuals and businesses. This document, now kept in a vault, outlined what would become Office.

Word was born in this context. Its first MS-DOS version broke with contemporary practices. Where WordStar, the market leader, made do with the keyboard, Word bet on the Microsoft mouse. Advertisements touted this rudimentary graphical interface that displayed bold, italic, or underlined text. Novel for MS-DOS, but limited since changing fonts was impossible. Sales struggled and WordStar kept its faithful users, accustomed to its keyboard shortcuts.

Everything shifted with the Macintosh's arrival in 1984. Apple imposed its graphical interface, and Microsoft seized the opportunity. Simonyi's team, no more than twenty developers, adapted Word for the Mac. This time, Bravo's concepts found their medium of expression: multiple fonts, sophisticated layout, complete WYSIWYG. The software finally revealed its potential.

This development philosophy illustrated Microsoft's approach in the 1980s. Spot promising innovations, adapt them to the mass market, iterate relentlessly. Simonyi cultivated this deliberately small organization to maintain its coherence and agility, reminiscent of his years at PARC.

With Windows, Microsoft transposed the Mac version's achievements to its platform. Windows 3.0, launched in 1990, propelled Word to the

top. The commercial strategy proved successful with its bundled sale with Windows and slashed prices. WordPerfect and WordStar, though well established, lost their footing. Word was now indispensable.

Microsoft didn't merely dominate the word processing market. The company orchestrated integration among its applications: Word, Excel, PowerPoint now formed Office. This coherence met the expectations of businesses seeking compatible tools. The .doc format naturally established itself as the reference for exchanging documents.

The 1990s saw Word enriched endlessly: spell checker, thesaurus, mail merge, tracked changes. This accumulation of features proved divisive. Casual users got lost in endless menus. Professionals appreciated this functional richness. Simonyi questioned this drift of Word toward ever-increasing complexity.

Beyond the tool itself, Word transformed our writing practices. Sophisticated formatting, once the preserve of typographers and publishers, became democratized. Anyone could create professional-looking documents. This accessibility had its downside: built-in templates standardized presentation. Word influenced how we structure thought, how we organize ideas.

This criticism has accompanied Word since its beginnings. Does the software shape our creativity? Does it standardize our expression? These questions have traversed forty years of technological evolution. From MS-DOS to the cloud, from diskette to smartphone, Word has accompanied our digital transformations.

Office 365 propelled Word into the collaborative era. Documents live in the cloud, are shared in real time, follow the user across all devices. This omnipresence crowns four decades of continuous evolution.

Charles Simonyi left Microsoft in 2002. His creation survives him, transformed but faithful to its founding principles. The reference word processor, Word is used by hundreds of millions of people as their word processing software of choice. Its history encapsulates that of consumer computing, from laboratory innovation to universal tool, a technology that shapes our daily actions.

MSX

Personal computing in the 1980s resembled a technological *wild west*. Each manufacturer developed its own machines, incompatible with one an-

other, turning the purchase of a family computer into a veritable obstacle course. Software written for a Commodore wouldn't run on an Amstrad, and vice versa. This technological cacophony frustrated Kazuhiko Nishi, founder of ASCII Corporation in Japan (no relation to American Standard Code for Information Interchange) and vice president of emerging technologies at Microsoft.

The man had carefully observed the success of the VHS standard in the video industry. Where Betamax and VHS clashed, victory had gone to the format that managed to unite the greatest number of manufacturers. Drawing from this lesson, Nishi conceived the MSX in 1983. His idea boiled down to a few words: create a technical standard that different manufacturers could adopt, thus guaranteeing total compatibility between all machines bearing the MSX logo.

The chosen name revealed the project's ambition. "Machines with Software eXchangeability", according to its creator, expressed this desire to enable software exchange between computers of different brands. Other interpretations circulated: "MicroSoft eXtended BASIC" or "Matsushita Sony X-machine", testifying to the technological giants' commitment behind the initiative.

The MSX's technical architecture relied on already proven components. A Zilog Z80A processor running at 3.58 MHz formed the heart of the system, supported by a TMS9918 video chip capable of displaying 16 colors at 256x192 resolution, and an AY-3-8910 sound generator. RAM ranged from 8 to 128 KB, typically 32 or 64 KB depending on the model. This configuration allowed the use of three different media: cassettes, floppy disks, and cartridges.

Microsoft developed MSX-DOS, an operating system inspired by the MS-DOS of IBM PCs. It integrated an extended BASIC that simplified programming and guaranteed compatibility between machines. In a market where each manufacturer typically cobbled together its own solutions, this hardware and software standardization represented a genuine innovation.

Japanese electronics giants took the bait. Sony, Toshiba, Panasonic, Canon, Yamaha: all adopted the standard. Sony kicked things off with its HB-10 in 1983, initially dressed in red and white to appeal to the Japanese market, then in black for export. Over twenty manufacturers produced their own MSX computers, some like Sony and Philips offering up to a dozen models in their lineup.

In Japan, success was immediate. Consumers appreciated this compatibility guarantee that spared them unpleasant surprises. Video game publishers sensed the opportunity: Konami developed its Metal Gear se-

ries there, while Castlevania and Gradius found a privileged platform for expression on MSX. It was on this platform that the famous "Konami Code" was born, that legendary button combination in the video game universe.

The phenomenon spread far beyond Japanese borders. South Korea, Brazil, and Chile massively adopted the MSX. In Europe, the Netherlands became its stronghold, with Philips making it their flagship product. The Soviet Union succumbed: the Ministry of Education acquired hundreds of MSX1 and MSX2 computers to equip school computer labs, training an entire generation of programmers behind the Iron Curtain.

Yet America resisted. Only the SpectraVideo and the Yamaha CX-5M (marketed as a musical instrument) managed to cross the Atlantic. In 1984, the Commodore 64 reigned supreme over American territory, and the market saw little interest in adopting what it considered a technically inferior product. Across the Channel, competition from the ZX Spectrum and the prohibitive price of cartridge games compared to cheap cassettes severely limited its adoption.

Faced with this resistance, the standard evolved. The MSX2, launched in 1986, brought substantial improvements: more memory, a faster processor, a more powerful graphics chip. It maintained compatibility with its predecessor and integrated a floppy disk drive, an economical alternative to cartridges. In 1988, as European distribution ceased, the MSX2+ appeared in Japan and Korea, followed by the MSX TurboR, the ultimate evolution of the standard.

The TurboR marked a breakthrough by adopting a 16-bit processor while preserving Z80 compatibility. Alas, its high price and compatibility problems with certain earlier software broke its momentum. It was only distributed in Japan, with an exclusively Japanese BIOS that further limited its reach.

The emergence of the IBM PC and Microsoft's focus on Windows ultimately sounded the death knell for the MSX. ASCII Corporation attempted to keep the standard alive, developing the visionary idea of networking MSX computers to create a "primitive internet". But the refusal to adopt the CD-ROM format definitively buried hopes for the MSX3.

The MSX legacy endures in the retro computing enthusiast community. Its standardization philosophy and rich software library continue to inspire developers and hobbyists. Its influence can be found in *chiptune* music and *retrogaming* culture. Projects like the 1chipMSX, which concentrates all MSX logic into a single FPGA chip, perpetuate its spirit of innovation.

The MSX adventure illustrates a bold attempt at standardization that anticipated issues still relevant today. Though the standard didn't achieve

its goal of world domination, it demonstrated the viability of a collaborative approach to computer development.

Novell NetWare

In Utah in 1979, four men embarked on a computing adventure without imagining they would revolutionize how machines communicate with each other. Dennis Fairclough, George Canova, Darin Field, and Jack Davis founded Novell Data Systems Inc. in Orem, far from Silicon Valley's laboratories. Their financial backer, Safeguard Scientifics, bet on this team to develop CP/M-based systems. The early steps were arduous: competition was fierce and sales struggled to take off.

The idea that would change everything emerged almost out of necessity. Why not connect these microcomputers proliferating in offices, rather than persist in a saturated market? This insight didn't come from nowhere. Several team members, including newcomers Drew Major, Dale Neibaur, and Kyle Powell, had worked on government contracts that familiarized them with ARPANET. They understood the power of networks before the general public discovered the Internet.

The road remained fraught with obstacles. Safeguard's board of directors nearly shut down the operation, but the company secured new financing at the last minute. Raymond Noorda arrived as CEO in 1983. This experienced businessman renamed the company Novell Inc. and bet everything on NetWare, their network operating system.

NetWare's approach stood out from what existed at the time. A machine designated as the dedicated network server, orchestrating access to hard drives and printers shared by other workstations. The star topology relied on twisted-pair cabling, a more reliable technology than competing solutions. NetWare, initially called ShareNet, came in two versions: one for Intel 8086 processors, the other for Motorola.

The 1980s saw personal computing massively installed in businesses. NetWare rode this wave and became the reference for departmental local area networks. In 1986, Novell expanded its territory with GroupWise, a messaging and collaborative tools solution long before Microsoft Office 365.

The early 1990s established NetWare's dominance: nearly 70% of the network operating system market belonged to it. Version 4 radically transformed the system's capabilities. Novell Directory Services revolutionized

administration by consolidating multiple servers under unified management. The NetWare Administrator interface simplified network administrators' daily work. NetWare's file and print services had no equivalent at the time.

Everything shifted with the Internet explosion in the late 1990s. Local area networks interconnected, extending beyond department boundaries to embrace entire buildings, then continents via WAN connections. Novell responded with intraNetWare, which included an improved version of NetWare 4.11. This version facilitated installation, boosted performance, and introduced the first 32-bit client for Windows.

IntraNetWare adapted to new uses with an IPX/IP gateway for connecting workstations to IP networks, an integrated web server, native support for DHCP and DNS. NetWare still excelled in its area of expertise, but companies criticized its weakness in hosting business applications.

Microsoft chose this moment to strike hard. The Redmond firm ended its partnership with 3Com and developed Windows NT with integrated network features. This vertical integration strategy directly threatened Novell's model. Facing this offensive, the Utah company made a strategic mistake: it acquired WordPerfect and Quattro Pro to confront Microsoft on its own turf. This diversification scattered efforts and resources across markets that Novell poorly understood.

Microsoft played a war of attrition. Its considerable financial resources gave it the means to persevere until Windows NT prevailed over NetWare. Novell, engaged on too many fronts, had to divest its acquisitions and constantly revise its strategy without regaining its former glory.

The turn of the 2000s saw Novell embrace open-source philosophy. NetWare was reborn as Open Enterprise Server (OES), available in two versions: one retained the traditional NetWare kernel to ensure compatibility, the other relied on SUSE Linux Enterprise Server. NetWare 6.5 marked the end of an era: it corresponded to OES-NetWare Kernel with Support Pack 2. From then on, Novell recommended running NetWare in a Xen virtual machine hosted on SUSE Linux Enterprise Server.

This thirty-year trajectory tells the story of enterprise networking evolution. NetWare was born from an insight: connecting isolated machines to multiply their utility. It accompanied organizations' computerization, transforming standalone workstations into collaborative networks. Its fall against Microsoft illustrates the computing industry's brutality, where dominant position guarantees nothing against a determined and well-funded competitor. The final conversion to open source testifies to the sector's transformations, where openness and interoperability have also become imperatives for survival.

Oric-1

Cambridge, October 1979. Dr Paul Johnson and Barry Muncaster had just founded Tangerine Computer Systems in the effervescent atmosphere that characterized the microcomputer industry. Their company's name reflected a somewhat whimsical trend: after Apple, the age of fruit in computing had arrived. Their first offspring, the Microtan 65, paved the way for what would soon become a far more ambitious adventure.

Paul Kaufman joined the team during the summer of 1981 and took charge of the *Tansoft Gazette*. The company grew and restructured. In early 1982, after splitting off its Tandata Prestel division, Tangerine relocated to Cambridge Science Park and launched Tansoft, its software arm. It was at this point that Oric Products International was born in April 1982, with Tangerine serving as the research laboratory.

Initially, however, the idea seemed quite different: to design a desktop computer for executives, capable of interfacing with the Prestel network. Kaufman drafted specifications for a hypothetical Microtan 2, enhanced with sound and graphics capabilities. These reflections gave birth in late 1981 to the Tangerine Tiger project, an ambitious machine equipped with three processors: a Z80 for CP/M, a 6809 to manage peripherals, and a dedicated graphics chip. But this Tiger would never roar: H.H. Electronics acquired the project, which would sink into obscurity.

On 27 January 1983, it was in the sumptuous setting of Coworth Park Manor, the former residence of Lord and Lady Derby, that Oric Products International officially unveiled the Oric-1. Peter Harding, the 34-year-old commercial director, set the tone with disconcerting confidence: six major contracts signed with retail chains for over 200,000 units. His objective was to beat Clive Sinclair by offering much more for much less money.

The pricing strategy seemed bold. One had to shell out £129 for the 16k version, £169.95 for the 48k model. Behind this machine, a team of talented developers worked tirelessly. Andy Brown and Chris Shaw, two seasoned programmers, tackled the main ROM. Peter Halford developed the cassette routines and Oric Mon, while Kaufman coded the sound portions on a Microtan in Forth before Andy Brown transposed them into machine language.

The first reactions from the specialist press remained cautious. The Centronics printer port caused a sensation: "unusual, even unique for a machine at this price", noted one critic. But ROM bugs and cassette loading failures tarnished the image. Oric's reaction was swift: Cosma Sales,

the cassette duplicator, was shown the door, accused of being responsible for the tens of thousands of defective cassettes in circulation.

Across the Channel, success was slow to come. France proved more welcoming, with A.S.N. signing an exclusive distribution contract on 29 June 1983 based on 4,000 units per month. The French market developed a quality software ecosystem, making the Oric the French equivalent of the British Spectrum. An unexpected success in a country with a reputation for being difficult to conquer.

This upturn was short-lived. February 1985: Edenspring placed Oric into receivership. The figures were chilling: £5.5 million in debt against only £3 million in assets. The fall seemed inevitable when, in June 1985, S.P.I.D. emerged from France and acquired the company for "several hundred thousand pounds." The Atmos, successor to the Oric-1, came back to life in a Norman factory.

The new French owners attempted to relaunch the machine with the Telestrat, specially designed for the Minitel network. The idea seemed attractive on paper, but the high price and market specificity limited ambitions. December 1987: Oric International fell back into receivership, this time with substantial tax debts on the books. A year later, in December 1988, production lines shut down permanently.

The Oric-1 was based on the popular 6502 processor, a choice dictated as much by its reputation as by its moderate cost. Its memory handled text and high-resolution display with remarkable sound capabilities. Its serial graphics attributes proved well-suited to games, a significant asset. The machine featured modified Microsoft BASIC and a UHF modulator for television connection.

The Oric story reveals the pitfalls of the 1980s market. Despite honorable technical specifications and attractive prices, the company failed to control its growth and consistently supply the market with quality software. Broken promises accumulated, notably concerning the Microdisc floppy drive, announced in 1983 but only available in 1984, when it was almost too late.

Yet the Oric survived its own demise thanks to a community of passionate users. Publications like *Oric User Monthly* continued to appear long after the company's end. In France as in the United Kingdom, user groups still maintain the memory of this machine which, despite its flaws, introduced an entire generation to personal computing.

The Oric adventure embodies that era when technical boldness rubbed shoulders with commercial improvisation. While the ultimate failure was perhaps inevitable, it testifies to the creative vitality of the British computer industry in the 1980s, capable of offering credible alternatives to

industry giants in a constantly evolving market.

Amstrad CPC 464

In 1984, Alan Sugar was doodling on a paper napkin during a transatlantic flight. This sketch would give birth to one of the most significant machines in British home computing: the Amstrad CPC 464. The idea seemed simple on paper, but its technical implementation required far more than a few pen strokes. Roland Perry, Amstrad's technical director, inherited this sketch and transformed it into reality.

The challenge was substantial. Sugar wanted a computer that would work straight out of the box, without the usual headache of multiple connections that discouraged so many families. The CPC 464 integrated everything into a single unit: keyboard, cassette drive, and direct connection to its dedicated monitor. A single power cable sufficed. This simplicity contrasted with the practices of the era, when setting up a computer system often proved an ordeal.

Perry had initially set his sights on the 6502 processor, but British developers were more familiar with the Z80. This Zilog processor, running at 4 MHz, already powered numerous machines across the Channel. The decision proved judicious: the 64 KB of RAM came with remarkable graphics capabilities. Twenty-seven available colors, several display modes—enough to appeal to gaming enthusiasts and aspiring programmers alike.

Amstrad's commercial strategy deserves attention. Sugar offered two configurations: £200 for the monochrome screen version, £300 for color. These prices cleverly positioned between the Sinclair Spectrum—cheaper but limited—and the BBC Micro—more powerful but out of reach for many budgets. Most importantly, Sugar anticipated the pitfall that had killed many promising machines: the lack of software at launch.

Forty prototypes went to developers. Training sessions were organized, and a substantial software library awaited the first buyers. This meticulous preparation paid off. Users discovered a reliable, easy-to-install machine that delivered on its promises. While patience remained necessary with the interminable access times of magnetic storage, the integrated cassette drive simplified program loading.

This limitation prompted Amstrad to react quickly. 1985 saw the birth of the CPC 664, equipped with a 3-inch floppy drive. But this intermediate

model gave way to the CPC 6128, featuring 128 KB of memory and more professional ambitions. Evolution was underway, but the original 464 retained its appeal for the general public.

Games made the CPC's reputation. The "Roland" series paid tribute to Perry, while the graphics and sound capabilities enabled quality adaptations. The integrated BASIC opened the doors of programming to beginners, supported by careful documentation. Some dared venture into serious applications: Protext demonstrated that a proper word processor could run on the machine.

The international market was not forgotten. In France, Amstrad paid attention to localization: adapted keyboards, translated manuals, organized distribution. This attention to national specificities bore fruit in several European countries. The CPC established itself durably in households, opening the doors of the computer market to Amstrad. The company followed up with the PCW 8256 in 1985, then launched PC compatibles as early as 1986.

As the years passed, Amstrad attempted to reignite the flame. 1990 saw the arrival of the Plus models, 464 Plus and 6128 Plus, with improved graphics. A derived console, the GX4000, failed against the Japanese giants. Production faltered in the early 1990s, but the story did not end there.

An active community preserves the CPC's legacy. Emulators resurrect the software of yesteryear, while amateur developers create new programs. This loyalty testifies to the machine's enduring qualities: coherent design, comprehensive documentation, unshakeable robustness. Amstrad's pragmatism hit the mark: ease of use rather than a performance race.

Apple Macintosh

The story begins in 1979 in the corridors of Apple, where Jef Raskin, a veteran of the Apple II team, was mulling over an idea that seemed crazy at the time. He wanted to create a computer as simple as a toaster, sold for a thousand dollars and mass-produced. Apple's management, absorbed by the Apple III's troubles and the costly development of the Lisa workstation, greeted this proposal with a polite shrug.

Everything changed when Burrell Smith joined the team. This Apple II repairman from the maintenance department was hiding technical genius. After a visit to Silicon Valley that had electrified him, he had

started tinkering with microprocessors. His makeshift prototype—a Motorola 6809 jury-rigged onto an Apple II with a TV screen—impressed Raskin so much that he immediately brought him into the project. Steve Jobs, then vice president but sidelined from the Lisa project for lack of experience, sensed the opportunity and took charge of the Macintosh.

Jobs set up his team far from headquarters, in offices nicknamed "Texaco Towers" because they were located behind a gas station. The place looked more like a student apartment share than corporate offices. This physical proximity created a particular dynamic: Andy Hertzfeld, in charge of the operating system, worked side by side with Smith on the hardware. This close collaboration allowed them to exploit every available bit of memory.

The team had set themselves an ambitious goal: 300,000 units per year. Smith, obsessed with cost reduction, set out to revolutionize the architecture. Where the competition multiplied connectors and expansion slots, the Macintosh would have only two circuit boards. No buffers, no superfluous slots. Vital functions would be burned into read-only memory for improved reliability.

The approach was radical: general-purpose flexibility versus specialized architecture. The team methodically dissected competing products and found that their multiple connectors slowed performance while inflating prices. The Macintosh would rely on a fast serial port for future expansions.

The user interface drew inspiration from the Lisa, but in a streamlined version. Graphical windows allowed multiple programs to be displayed, icons replaced obscure commands, the mouse became king. However, everything had to run with less memory than the Lisa, and faster. The team rewrote the software from the ground up, abandoned the Lisa's multitasking in favor of an illusion created by small system programs like the calculator.

Bill Atkinson bequeathed them QuickDraw, his graphics management software created for the Lisa. This toolkit, stored in ROM, became the soul of the Macintosh. Developers could enhance the system by loading additional definitions from disk, thus preserving the limited memory.

The choice of the Motorola 68000 divided the team. This expensive processor far exceeded immediate needs, but Jobs bet on falling prices. The team optimized everything: only one processor mode used out of four available, three interrupt levels out of seven, file system with a single index.

Manufacturing required a revolutionary factory in Fremont, California. Inspired by Japanese methods, it applied just-in-time production and zero

defect tolerance. Suppliers delivered small batches frequently, errors were analyzed immediately. The factory was designed in three evolutionary phases, anticipating increased production rates.

In January 1984, the Macintosh arrived at $2,495. The price was far from the thousand dollars Raskin had dreamed of, but the computer made an impression. The Super Bowl commercial, a pastiche of Orwell's *1984*, presented Apple as a liberator against IBM uniformity. The user-friendly graphical interface and compact design created a new standard.

Criticisms were plentiful: high price, network limitations, insufficient memory. But the Macintosh had planted a seed. It demonstrated that a computer could be something other than a calculating machine: a design object, an extension of human thought.

The original team had established principles that would survive all trends: design first, deep hardware-software integration, obsessive simplicity. Jobs left Apple in 1985 after a conflict with the board of directors. The company sank into incoherence, multiplying models without a common thread.

Jobs's return in 1997 put these fundamentals back at the center. Refined design, streamlined lineup, total integration: these principles, applied to the iPod and then to the iPhone, allowed Apple to conquer the world. That beige machine from 1984, born from the meeting between a brilliant repairman and an obsessive visionary, had changed our relationship with technology.

HP-UX

In the early 1980s, Hewlett-Packard decided to venture into the UNIX world. The market was buzzing with activity, with manufacturers vying to offer their own variant of the system. In 1984, HP released the first version of HP-UX, a reworked UNIX System V that ran exclusively on HP 9000 servers in the 200, 300, and 400 series. These machines featured the Motorola 68000 processor, already proven in other contexts.

HP did more than simply adapt UNIX. The engineers immediately introduced access control lists, a more refined approach than the traditional UNIX permissions system. A logical volume manager was also part of the basic equipment, simplifying life for administrators facing storage challenges. These additions were no gimmicks: they addressed the concrete needs of professional users.

The evolution followed the pace of HP hardware. Version 5.0 from 1985 extended to the Integral PC, then to the 200/300 and 500 series. HP began unifying its approach, seeking consistency across its different product lines. This strategy paid off when the PA-RISC architecture arrived toward the end of the decade.

PA-RISC, these RISC processors developed internally by HP, would guide the system's technical evolution for years to come. In 1988, HP juggled two development lines: versions 3.x for the 600/800 series and 6.x for the 300 series. This situation reflected the delicate transition between the old architectures and the new PA-RISC generation.

Unification became concrete in 1990 with HP-UX 7.x, which covered the 300/400 and 600/700/800 series. Users gained in simplicity, administrators in consistency. A year later, version 8.x brought shared libraries, significantly improving performance and memory usage. Version 9.x from 1992 enriched the arsenal with the System Administration Manager and Logical Volume Manager, two tools that modernized system administration.

HP-UX 10.0 came out in 1995. This version unified the 700 and 800 series while adopting the SVR4 directory structure, aligning with the UNIX standards of the time. The following year, HP-UX 10.20 crossed into 64-bit territory with PA-RISC 2.0 processors. A technical feat that opened new horizons.

HP-UX 11.00 achieved a remarkable accomplishment in 1997: supporting 64-bit addressing without abandoning 32-bit applications. This backward compatibility spared companies the abrupt disruption of a migration. Three years later, HP-UX 11i v1 (11.11) introduced operating environments, customizing the system according to the specific needs of each type of usage.

The architecture underwent a new upheaval in 2001 with HP-UX 11i v1.5 (11.20), which integrated Intel Itanium support. This strategic decision accompanied the gradual transition from PA-RISC to Intel processors. HP-UX 11i v2 (11.23) from 2003 strengthened virtualization, while version 11i v3 (11.31) from 2007 added native *multipathing* and further refined virtual capabilities.

Over successive versions, HP-UX earned a solid reputation in IT services, software publishing, and finance. The VxFS file system and LVM manager became essential references. System administrators appreciated their flexibility and efficiency in managing storage, especially in complex environments.

As a consistent strength, HP progressively integrated security and intrusion detection directly into the kernel, developed secure partitioning,

and implemented role-based access control. These elements elevated HP-UX among the most secure UNIX systems on the market, a decisive advantage for sensitive applications.

Virtualization occupied an increasingly important place in the system's evolution. nPartitions arrived first, followed by vPars (Virtual Partitions), and finally Integrity Virtual Machines. These technologies allowed companies to optimize their hardware resources and consolidate their servers. The transition from PA-RISC to Itanium required considerable technical effort, but HP skillfully preserved application compatibility.

Clustering with HP Serviceguard and continuous improvements to administration tools kept HP-UX competitive against the changing demands of organizations. But Linux gained ground and cloud computing changed the rules of the game. HP-UX's market position found itself progressively called into question.

In 2017, HP announced its roadmap through 2025, reassuring its loyal user base. This announcement addressed concerns from companies relying on HP-UX for their critical operations. However, the aging of compatible hardware posed a problem: HP 9000 and Integrity servers were difficult to maintain, spare parts were becoming scarce.

In response, new approaches emerged. Hardware emulation, notably with Charon-PAR from Stromasys, offered an interesting alternative. Companies could continue running their HP-UX applications on modern x86 hardware or in the cloud, thereby escaping the constraints of aging proprietary hardware.

The history of HP-UX demonstrates constant adaptation to enterprise needs. Its innovations in logical volume management, security, and virtualization inspired the evolution of professional operating systems. Its reputation for reliability and robustness made it the tool of choice for numerous critical applications, from finance to healthcare and industry.

NFS

In 1985, Sun Microsystems released a technology that would transform how computers share files across networks: the Network File System, better known by the acronym NFS. At the time, the idea of connecting distant machines to access shared data posed a significant technical challenge. Yet this innovation would become one of the cornerstones of distributed systems.

The story begins with SunOS 2.0, which integrated the first public version of the protocol, NFSv2. But behind this release lay Sun's bold strategic vision. Where other companies would have kept their technology proprietary, Sun bet on openness. The NFS protocol only defined the message formats exchanged between clients and servers, leaving everyone free to implement their own solution. This decision spawned a thriving ecosystem: NetApp, EMC, IBM and many others joined the race, creating their own NFS servers while maintaining interoperability.

This strategy contrasted sharply with previous approaches. SVR3's Remote File System, for example, established a direct correspondence between client and server system calls. The approach worked, certainly, but revealed its limitations as soon as a server crashed: clients often found themselves stuck, forced to reboot to resume their activities.

Sun's engineers took a radically different direction. They designed NFS as a "stateless" system, where the server keeps no record of ongoing operations. Each request arrives with all the information needed for its processing. This architecture, counterintuitive at first glance, conceals formidable elegance: if the server stops abruptly, clients need only resend their requests once it has restarted. No complex state reconstruction, no delicate synchronization.

At the heart of this mechanism lies the "file handle", a true identity card for each file on the server. In NFSv2, this descriptor gathers a volume identifier, an inode number, and a generation number. This last element prevents confusion when the system reuses inode numbers: each file thus maintains its unique identity, even after deletion and recreation.

Of course, querying the server at every file access would have generated unsustainable network traffic. NFS therefore integrates client-side caching mechanisms, temporarily storing accessed data and metadata. This optimization immediately raises a thorny question: how to ensure that all clients see the same information? The answer lies in the "close-to-open" model. Modifications travel up to the server when files are closed, while clients verify data state at each opening. This approach favors performance while maintaining reasonable consistency.

The protocol would undergo several metamorphoses over the years. In 1995, NFSv3 arrived with performance improvements, notably better handling of asynchronous writes. But it was NFSv4, standardized in 2000, that truly revolutionized the architecture. Paradoxically, this version abandoned the stateless principle in favor of a stateful protocol, introduced cache delegations and considerably improved Internet compatibility.

Security was the Achilles' heel of early versions. The authentication

system relied on blind trust between clients and server, simply using UNIX numerical identifiers. This naivety was gradually corrected through Kerberos integration, then by the more sophisticated mechanisms of NFSv4.

But NFS development necessitated the creation of the Virtual File System interface, an abstraction layer allowing the coexistence of different file systems within the UNIX kernel. This innovation, still present in modern systems, greatly simplified the integration of new file systems.

The 1990s saw fierce competition emerge. Microsoft pushed its Server Message Block, later renamed Common Internet File System, which naturally established itself in the Windows universe. More ambitious systems like Andrew File System or, more recently, Lustre for supercomputers, offered advanced features for specific niches.

Yet NFS withstood these assaults. Its relative simplicity, proven robustness and near-universal availability make it a sensible choice for many organizations. Storage manufacturers continue to innovate around the protocol. NetApp, notably, developed hardware optimizations to accelerate synchronous writes, historically NFS's weak point.

NFS's technical legacy still informs the design of distributed systems. The stateless architectural principle, initially controversial, now inspires numerous network protocols. The lessons learned about cache consistency continue to fuel academic and industrial research.

As cloud storage reshapes computing's landscape, the challenges NFS addressed remain relevant. While solutions evolve, distribution, consistency and fault tolerance are issues that transcend time. The NFS adventure demonstrates that a pragmatic approach favoring robustness over sophistication can endure. In a field where technological changes succeed one another at a breakneck pace, this longevity commands respect.

POP3

In 1984, electronic communications faced a problem that is difficult for us to imagine today. Messages arrived successfully on servers through the SMTP protocol, but how could users retrieve them without a permanent Internet connection? This question concerned engineers of the era, who sought a practical solution for users still accustomed to intermittent telephone connections.

The POP (Post Office Protocol) emerged to address this need. The first version appeared in 1984. The idea was simple: create an electronic

"mailbox" system where messages wait until the user comes to collect them. POP2 followed in 1985 with some improvements, and finally POP3 arrived in 1988, introducing extension mechanisms and more robust authentication systems.

The operation of POP3 demonstrates a simplicity that took precedence over sophistication. The client connects to the server via TCP port 110, authenticates, and uses a handful of basic commands. STAT to check the number of messages, LIST to view their size, RETR to retrieve one, QUIT to leave. Nothing more, nothing less. This economy of means has largely contributed to its success among developers.

The integration of POP3 into Microsoft Outlook and other mainstream email clients sealed its fate. The protocol spread like wildfire, becoming the reference solution for individuals and small organizations. Its logic perfectly matched the habits of the time: people checked their mail on their personal computer, just as they emptied their physical mailbox.

But this approach conceals a trap that would prove problematic. By default, POP3 deletes messages from the server after downloading them. Once transferred to the living room computer, it becomes impossible to access them from the office or, later, from a mobile phone. This limitation, trivial in the 1990s, became a genuine handicap with the explosion of mobile computing.

Faced with these constraints, the IMAP protocol established itself as the modern alternative. Where POP3 empties the server's mailbox, IMAP keeps it intact. Messages remain accessible from any device, organized in folders, available offline if necessary. IMAP also offers server-side search functions and partial downloading of large messages.

Yet POP3 refuses to disappear. Its technical lightness remains an undeniable asset: it consumes few server resources and is easy to use. For certain specific uses, particularly when the user works primarily on a single device, POP3 retains its advantages. Messages stored locally can be accessed without an Internet connection and do not clutter the servers.

This persistence illustrates an interesting phenomenon in technological evolution. Two philosophies clash here: POP3 makes the personal computer the center of the digital universe, while IMAP favors a centralized vision where the server is the sole repository. Each approach addresses different needs and directly influences the user experience.

Recent history has proven IMAP right. The increase in storage capacity, the democratization of permanent broadband connections, and especially the simultaneous use of multiple devices have made synchronization indispensable. Who would accept today being able to check their emails only from their laptop?

The protocol has not remained static. Security improvements have emerged, such as the APOP authentication mechanism that strengthens exchange protection. These developments demonstrate an ability to adapt to contemporary requirements.

Standardization by the IETF played a decisive role in POP3's longevity. This normalization guaranteed its interoperability and facilitated its integration into numerous software applications. Paradoxically, this ubiquity may explain its survival: replacing such a widely adopted standard requires considerable effort.

The story of POP3 reveals that a technical solution does not need to be the most advanced to endure; it must primarily address a concrete need effectively. POP3 brilliantly solved the problem of email for the intermittent connections of the 1980s. The fact that this problem has since largely disappeared does not erase this initial success.

This persistence is also explained by very human factors. User familiarity with a tool, resistance to change, preference for proven solutions constitute obstacles to the adoption of new technologies. System administrators know this well: migrating to a new protocol requires time, money, and training.

The trajectory of POP3 is part of a broader dynamic in Internet evolution. Protocols are born, evolve, coexist, and sometimes disappear according to logics that transcend purely technical considerations. The history of email shows how different approaches can survive side by side, each retaining its niche of use.

This coexistence is not accidental but a characteristic of mature technological ecosystems. POP3 and IMAP respond to different philosophies of personal data management. One prioritizes autonomy and local control, the other relies on centralization and universal accessibility. Both visions remain legitimate depending on the context of use.

Understanding the history of POP3 means grasping how technologies become rooted in their time while maintaining a limited capacity for adaptation. It also means measuring the importance of initial design choices, which can determine the success or obsolescence of a standard over several decades.

IBM PCjr

In 1983, IBM dominated the professional market with its PC. Riding high on this success, the American giant cast its eye toward home computing, a territory dominated until then by machines like the Apple II or the Commodore 64. The idea emerged to adapt the PC for families by creating a scaled-down version dubbed "Peanut" internally. Rumors multiplied in the specialized press for months until the official announcement of the PCjr on November 1st, 1983.

Expectations ran high. IBM's research budget at the time exceeded that of certain nations, and its name alone was enough to trigger public enthusiasm. The machine reused the Intel 8088 processor from the PC and maintained substantial software compatibility with its bigger sibling. This deliberate connection aimed at two objectives: reassuring buyers about the sustainability of their investment and protecting sales of the professional PC.

Two versions came to market. The base model, priced at $669, featured 64 KB of memory. The enhanced model climbed to $1,269 with its 128 KB and integrated floppy disk drive. These prices immediately positioned the machine at the high end, well above the direct competition.

The PCjr's main innovation lay in its wireless keyboard, a world first for a personal computer. The infrared connection made the machine usable from up to 6 meters away, transforming the living room into an improvised office. But this innovation came with a questionable choice: flat keys inspired by calculators, devoid of inscriptions. The symbols and letters appeared on the chassis, between the keys, complicating typing and straining the eyes.

Performance-wise, IBM equipped its machine with remarkable graphics capabilities. Several display modes coexisted, from 320×200 pixels in 4 colors to 640×200 in two shades. The enhanced model supported up to 16 simultaneous colors. Sound was equally impressive with a sophisticated circuit producing three synthesis channels over 7 octaves, complemented by a white noise generator.

Sales began in January 1984, and IBM deployed unprecedented marketing firepower. But deliveries remained limited until March. Charlie Chaplin, already the PC's spokesperson, returned in television commercials. Over a hundred books appeared before the actual release, and four specialized magazines launched. Rarely had a machine benefited from such media hype.

Yet sales struggled to take off. Early reviews revealed glaring flaws. With its execution speed cut in half compared to the PC, the PCjr disap-

pointed users accustomed to its elder's performance. Memory, capped at 128 KB, severely limited upgrade possibilities. The internal architecture made adding a second floppy drive or hard disk impossible. Proprietary connectors forced buyers to purchase overpriced adapters to connect standard peripherals.

Facing this criticism, IBM attempted to right the ship in July 1984. A new, more conventional keyboard replaced the flat-key model. Prices dropped significantly. A major promotional campaign accompanied the 1984 holiday season, temporarily boosting sales to 17% of the home market. But the upturn was short-lived: by January 1985, that figure had plummeted to 4%.

The ax fell on March 19, 1985. IBM announced the end of production, just fifteen months after commercial launch. Between 100,000 and 350,000 unsold machines piled up in warehouses. The company slashed prices on remaining stock, even giving them away to schools purchasing PCs. The bill came to $45 million in losses.

Paradoxically, this commercial failure masked some lasting successes. Sierra On-Line developed *King's Quest* specifically for the PCjr, revolutionizing the graphic adventure genre and launching a legendary franchise. Tandy Computer drew directly on IBM's experience to create its Model 1000, correcting the major flaws in the process: professional keyboard, performance equivalent to the PC, expansion capabilities. This Tandy 1000 achieved considerable success and extended the PCjr's technical legacy.

A community of enthusiasts keeps the memory of this unique machine alive. Expansions like the jrIDE card add a hard drive and 640 KB of memory to surviving units. The DOSBox emulator resurrects its software catalog. In Japan, IBM had attempted a variant called the JX, developed with Matsushita, but this competing project sank even deeper with only 8,000 units sold.

The PCjr story illustrates the pitfalls of a poorly executed derivative strategy. By deliberately constraining their home machine's capabilities to protect professional PC sales, IBM's engineers created a flawed product, too expensive and too limited against agile and inventive competition. Exclusive distribution through professional networks, ill-suited to the family market, sealed the fate of a venture that marked the end of IBM's ambitions in home computing.

GNU Emacs

In 1974, in the hallways of MIT's Artificial Intelligence Laboratory, Richard Stallman tackled a problem that frustrated all programmers of the era: the TECO text editor, while powerful, remained cumbersome to use. He got the idea to improve this tool by drawing inspiration from the `E` editor developed at Stanford. He added real-time display and redesigned the command system.

Initially, this improvement didn't change much about TECO. Users launched the visual editing mode with Control-R, modified their text, then exited for other tasks. But one feature would transform everything: the ability to redefine commands to call programs written directly in TECO. This freedom unleashed users' creativity as they developed their own command systems. TECMAC, MACROS, RMODE, TMACS, Russ-mode, and DOC flourished in the community. Usage evolved so that exiting the editing mode became unnecessary, and newcomers no longer learned to manipulate TECO directly.

However, TECO showed its limitations. It lacked decent programming structures: no named functions, no proper variables. The first editors built on this foundation quickly became unmanageable. In 1976, the TMACS system attempted to add these missing features. The experiment worked, but performance suffered. Stallman observed, analyzed, and decided to create EMACS.

His approach combined two challenges: enriching TECO with the libraries and self-documentation necessary for readable programming, while designing an entirely new set of editing commands. He studied the numerous TECO-based editors and sought to minimize the number of keystrokes for common operations. By the end of 1976, the first EMACS was operational.

Development continued on Digital Equipment Corporation's PDP-10, under MIT's ITS system. Most of the new code was written in TECO, with Stallman adding only the bare minimum to the language: optimizations for search loops, symbolic expression parsing, and a few new input-output interfaces. In 1977, interest grew beyond MIT's walls. Mike McMahon from SRI International adapted EMACS to Digital's Twenex system. A hundred sites were already using it.

The term "Emacs" then became generic, designating an entire family of extensible editors. One of the most notable versions emerged under UNIX, developed by James Gosling. This implementation brought two remarkable innovations. First, it exploited UNIX's process management capabilities: filtering text through external programs, running different

tools in separate windows. Second, it introduced MLisp (Mock Lisp), an extension language independent of the C used for implementation.

This separation between implementation language and extension language revolutionized usage. With MLisp, simple and elegant, users created extensions without compilation or linking. The development cycle accelerated, encouraging the small improvements that make all the difference in a user interface. The language offered an abstract view of text, strings, processes, and windows.

In 1984, Stallman launched GNU Emacs as part of his GNU project. This time, he chose true Lisp as the extension language, more powerful and coherent than MLisp. GNU Emacs became dominant. An active community developed extensions for everything: email, discussion forums, specialized editing for programming languages, debugging interfaces.

Emacs proved that extensible and customizable software can adapt to everyone's needs. This model inspired numerous subsequent developments: a stable core combined with a powerful extension language. Modern development environments adopted this approach, integrating programming languages, interface customization, and task automation.

Emacs's legacy is found in its self-documentation, which provides access to command and function explanations from within the editor, and has become standard for online help in interactive software.

GNU Emacs continues to evolve, faithful to its original philosophy. In a world where technologies become obsolete in a few years, its longevity testifies to the soundness of its principles: extensibility, power, and flexibility. It adapts to contemporary needs without renouncing what made it successful.

Microsoft Paint

Some software programs leave their mark through technical sophistication or innovative prowess. Microsoft Paint is not one of them. Yet since 1985, this modest drawing application has accompanied millions of Windows users and has shaped, almost inadvertently, an entire digital aesthetic.

The adventure begins in the 1970s, when two philosophies clash in the field of computer graphics : vector programs that work with precise geometric objects, and digital painting software that directly manipulates

pixels. Dick Shoup, at Xerox PARC, develops SuperPaint in 1973, a system that captures hand movements on a pixel grid. This bitmap approach, less rigorous than vector graphics, offers a spontaneity that brings the computer closer to traditional artistic gesture.

Microsoft discovers this path when Apple triumphs with MacPaint on its first Macintosh computers. In 1985, Windows 1.0 includes Windows Paint, coded by Dan McCabe. The program offers 24 tools in a spartan interface such as pencil, brush, geometric shapes, and also some advanced functions like Bézier curves or isometric drawing. But these ambitions run up against the technical realities of the era. Computers struggle, screens lack resolution, and Windows 1.0 doesn't really convince anyone.

In the following years, Microsoft turns away from Paint. The company collaborates with IBM on Presentation Manager, a graphical operating system project that ultimately fails. Paint stagnates in Windows 2.0, victim of this strategic reorientation that leads nowhere.

The awakening comes with Windows 3.0 in 1990. Microsoft abandons the idea of developing all its applications in-house and calls upon existing publishers. Microsoft Paintbrush replaces Windows Paint, based on ZSoft's PC Paintbrush, created by Mark Zachmann. The transformation is spectacular : color makes its appearance with 256 shades available, the tool palette migrates to the left side, and colors line up at the bottom of the screen. The interface gains coherence and readability.

Windows 95 definitively transforms Paint. Renamed Microsoft Paint (or MS Paint for short), it becomes a standard component of every Windows installation. This ubiquity coincides with the explosion of the Internet, which mutates from a textual medium into a visual universe. Between 1995 and 2000, nearly one in two Americans discovers the Internet, and Paint is their first contact with digital graphic creation.

The tool then develops its particular aesthetic signature. The exclusive use of the mouse, combined with the bitmap approach, produces shaky lines, approximate contours, color areas with sharp borders. These technical "flaws" create a visual language recognizable among thousands. Where other software aims for perfection, Paint embraces its imperfections and transforms them into expressive characteristics.

Curiously, Microsoft leaves Paint in this state for more than ten years. From 1995 to 2007, features evolve little. This inertia is explained by a paradigm shift : Windows no longer needs to seduce through its integrated applications. Paint fulfills its role as a basic tool for simple tasks, without pretending to compete with Photoshop or Illustrator.

In the 2000s, the Internet sees the birth of the first visual memes. Communities on Something Awful or 4chan adopt Paint's "amateur" aesthetic

as a banner. The 2008 Rage Comics perfectly embody this movement : rudimentary comic strips with expressive characters, entirely created with Paint. The imprecision is deliberate, the amateurism claimed.

Microsoft does attempt to modernize the experience in 2012 with Fresh Paint, a touch application that simulates traditional painting. But this smooth and sophisticated version masks its digital nature instead of embracing it. Meanwhile, tools like Rage Maker proliferate to facilitate meme creation in authentic Paint style.

Paint's story reveals how constraints shape creativity. Its technical limitations, which could have constituted handicaps, have almost become distinctive assets. Paint has never figured in innovation rankings, but it has left its own mark on visual culture.

This trajectory illuminates the importance of "ordinary" technologies in computer evolution. Spectacular innovations capture attention, while modest tools quietly structure our daily practices. Paint survives in Windows, witness to a period when creating digitally became accessible to the masses, with its awkwardness and particular poetry.

X Window System

In 1984, at MIT, two seemingly unrelated projects would converge toward a major transformation in graphical display for UNIX systems. On one side, the Argus system at the Laboratory for Computer Science needed a debugging environment for its distributed applications. On the other, Project Athena had to manage thousands of workstations equipped with bitmap displays. Faced with the diversity of hardware – Digital VSlOO machines, IBM workstations, and other heterogeneous architectures – engineers realized they needed to completely rethink their approach to graphical display.

The inspiration came from Stanford, where Paul Asente and Brian Reid had created the W system as an alternative to VGTS. W already allowed transparent network access for display, thanks to the synchronous communication mechanism of the V system. This system provided text windows for ASCII terminal emulation and graphics windows built on a simple display list mechanism. Asente and Chris Kent adapted W for VSlOO machines at Digital's research laboratories, but a few days of experimentation revealed the limitations of this approach: a hierarchical window system accessible via network would be far more efficient, provided it wasn't limited to specific application modes.

X Window System was born from this reflection. Its client-server architecture reverses the usual logic: the server controls the physical display while client applications communicate with it via an asynchronous protocol carried over a reliable byte stream. This conceptual inversion changes everything. It becomes possible to run a program on any machine on the network and use any display, without worrying about the hardware architecture or underlying operating system. Performance remains remarkable: a VAXStation-II/GPX displays 19,500 characters per second and draws 3,500 short vectors per second, whether locally or via network.

Among the innovations introduced by the system at the time, windows organize themselves in arbitrary hierarchies, resize and overlap freely. The server encapsulates all hardware dependencies, making the communication protocol completely device-independent. An application runs without modification on a monochrome terminal or a sophisticated color display. The protocol minimizes network latency through asynchronous requests and fine-grained event management.

Color management reveals X's technical sophistication. The proposed abstract model adapts to monochrome screens as well as the most advanced pseudo-color displays. Multiple applications can share a common color table while maintaining their own colorimetric spaces. This approach allows the harmonious coexistence of multiple graphical applications without resource conflicts.

In 1987, DEC integrated X into its commercial products, validating the technical approach. MIT distributed the complete source code without restrictions, accelerating adoption by universities and companies. This open-source strategy before its time transformed X into a de facto standard. Academic researchers and commercial engineers collaborated on development, creating a rich and diverse ecosystem. The system was ported to more than seven different hardware architectures and sixteen display types.

The 1990s saw the birth of GNOME and KDE, desktop environments that demonstrated X11's robustness. But limitations appeared: the font system proved insufficient for modern needs, 2D graphics capabilities required improvements, accessibility for the visually impaired remained difficult to implement.

Xft solved typographical problems by allowing direct access to font files and introducing antialiasing. Fontconfig separated character management from the windowing system, simplifying internationalization and installation of new fonts. Cairo brought advanced graphics capabilities: Bézier curves, translucent image composition, high-quality vector rendering. These developments integrated naturally thanks to X's modular ar-

chitecture.

The Composite protocol transformed visual management by allowing translucent windows and sophisticated effects. XEvIE improved accessibility by intercepting and transforming input events. These extensions kept X relevant in the face of growing demands from modern interfaces, without breaking compatibility with existing systems.

Security was never neglected. X implements sophisticated authentication and authorization mechanisms from its inception. Standalone terminals request login services from remote hosts securely with the X Display Manager Control Protocol. Administrators can centralize access management while preserving the user experience of traditional terminals.

X's client-server model and modular design inspire contemporary software architectures. By establishing a standard for distributed graphical systems, X shapes the evolution of user interfaces on Linux and related systems.

In 2004, the X.org foundation took over project governance and continued development. Innovations continue: high-resolution display support, hardware acceleration via OpenGL, adaptation to emerging display technologies. Wayland appears as an alternative for certain use cases, but X Window System remains in 2024 the reference windowing system on many UNIX and Linux platforms.

X's architectural principles – modularity, extensibility, hardware independence – retain their relevance in the development of contemporary interfaces. More than a technology, X illustrates the effectiveness of the collaborative approach where universities, companies, and communities coordinate their efforts to create sustainable infrastructure.

Intel 80386

When Intel launched the 80386 in October 1985, no one really expected this processor to revolutionize computing. In the company's corridors, it was considered more of a transitional product, a stopgap solution until the arrival of the real flagship project: the iAPX 432. This ambitious processor kept accumulating delays, and the 386 had to fill the void. History would remember that it was ultimately this "temporary solution" that would permanently transform the computing landscape.

The project began in early 1982 under the direction of John Crawford, the chief architect. His team of about ten engineers faced a difficult

dilemma: should they break with the past to create a cutting-edge processor or maintain compatibility with existing systems at the risk of limiting performance? Discussions followed one after another. Intel's customers were clear: they did not want to rewrite their software. Total compatibility at the object code level was therefore naturally required.

This constraint did not prevent Crawford's team from thinking big. The 386 definitively abandoned the 16-bit architecture of its predecessors to move to 32-bit. Registers, arithmetic and logic unit, internal buses: everything shifted to the new dimension. The processor could now directly address 4 gigabytes of physical memory, a capacity that seemed gigantic at the time. Modern operating systems like UNIX or OS/2 would make virtual memory management with paging their reality.

Manufacturing the 386 required leading-edge technical innovations. Intel abandoned NMOS technology in favor of CMOS, reducing power consumption and improving signal quality. But it was primarily the two-level metal fabrication process that posed problems. This new technology generated significant production defects. Engineers discovered that the "forbidden zone" between metal traces greatly complicated production. Yields dropped, costs skyrocketed, and the team had to constantly adjust parameters.

Development resources remained modest. About ten engineers worked with often makeshift tools. The team made a bold decision by adopting UNIX as the development operating system, despite the absence of official authorization. This transgression proved successful. Automation of component placement and routing, achieved through software developed by a Berkeley student, significantly accelerated the design.

The commercial launch held a surprise. Compaq, not IBM, released the first personal computer equipped with the 386. IBM no longer controlled the market's technological evolution alone—this was a breakthrough in the industry. The Compaq Deskpro 386 became a reference that forced IBM to react. Intel discovered that demand exceeded its initial forecasts.

To meet this demand while controlling costs, Intel developed the 386SX. This simplified version retained the internal 32-bit architecture but used an external 16-bit bus. The compromise appealed to budget computer manufacturers who could offer modern architecture at a reduced price. The 386SX achieved significant commercial success, democratizing access to 32-bit capabilities.

The 386's performance evolved with the first version clocked at 16 MHz reaching 5 MIPS. Later versions could go up to 33 MHz for 9.9 MIPS. These figures represented a considerable leap compared to previous processors. The 386 transformed personal computing by enabling smooth

execution of graphical applications and multitasking systems.

The processor's success extended far beyond the PC market. UNIX workstations adopted it massively. Industry integrated it into automated control systems. Scientific research made it a reference component for its equipment. More surprisingly, the 386 found its place in space: the Hubble telescope and the SAMPEX probe carried hardened versions of the processor.

The first sophisticated mobile devices adopted the 386. The BlackBerry 950 and Nokia 9000 Communicator integrated low-power versions of the processor. This exceptional longevity testified to the robustness of the architecture designed by Crawford and his team. The 386 remained in production for years, well after the arrival of its successors.

The 386's influence exceeded its commercial success alone. Following this experience, Intel adopted a development strategy alternating architectural innovations and manufacturing optimizations. This approach, later called "tick-tock", still guides the company's strategy. At Intel, the principle of backward compatibility established with the 386 guarantees the sustainability of software investments—a true dogma.

The 386 perfectly illustrates how computing history sometimes proceeds through fortunate accidents. Designed as a temporary solution, this processor ended up establishing the foundations of modern computing. Its legacy endures in every personal computer, reminding us that in technology, pragmatic solutions often prevail over the most ambitious projects.

AppleTalk

In the early 1980s, Apple Computer embarked on an ambitious challenge: creating a network protocol in parallel with the development of the Macintosh. The idea seemed bold, but it addressed a concrete need. Users wanted to share their files and printers without wrestling with complex technical configurations. AppleTalk was born from this desire to simplify network computing.

The protocol concealed its sophistication behind a remarkably transparent interface. Computers communicated with each other without users needing to understand the underlying mechanisms. This philosophy stood in stark contrast to the systems of the era, often intimidating and reserved for experts. Apple bet on the invisibility of technical operations: only the result mattered.

Two main versions emerged. Phase 1 targeted small workgroups but suffered from constraining limitations. It was impossible to exceed 135 machines on a segment, and wide area networks remained out of reach. These restrictions hindered AppleTalk's expansion in enterprises. Phase 2 corrected these flaws by allowing up to 253 machines per segment and managing wide area networks. This evolution paved the way for more ambitious deployments.

AppleTalk's flexibility was expressed through its physical implementations. LocalTalk exploited Apple's proprietary twisted pair cabling, while TokenTalk relied on Token Ring networks. EtherTalk leveraged Ethernet, and FDDITalk exploited FDDI optical fibers. This modularity enabled technological transitions without a complete system overhaul.

The protocol architecture was methodically organized. The Datagram Delivery Protocol transported packets from one point to another. The AppleTalk Address Resolution Protocol established the correspondence between logical and physical addresses. The Routing Table Maintenance Protocol kept routing information up to date. Each protocol fulfilled its function without encroaching on the others.

Name management relied on ingenious mechanisms. The Name Binding Protocol associated understandable names with technical addresses, while the Zone Information Protocol organized networks into logical zones. The AppleTalk Echo Protocol verified that machines remained accessible. The AppleTalk Transaction Protocol guaranteed exchange reliability for critical applications such as file sharing or printing.

Dynamic addressing constituted one of AppleTalk's innovations. An identifier was automatically assigned to each machine upon connection. This approach prevented address conflicts while simplifying administration. The address combined three elements: a 16-bit network number, an 8-bit node identifier, and an 8-bit socket number to designate the relevant service.

The zone concept revolutionized network organization. Machines could be grouped according to functional rather than geographical criteria. The accounting department could thus gather its resources in a single zone, even if the computers were scattered across different buildings. This flexibility would appeal to many network administrators.

Transmission occurred without prior connection establishment. Packets traveled independently toward their destination. Higher-level protocols then added the necessary reliability mechanisms according to needs. This modular architecture facilitated system evolution and maintenance.

Cisco progressively enriched its AppleTalk integration. It added support for different EtherTalk versions, compatibility with VLANs, integra-

tion with WAN protocols like Frame Relay. These extensions pushed the protocol's limits far beyond Apple's initial intentions. The encapsulation of RTMP packets over IP also enabled AppleTalk traffic over the Internet.

Security remained AppleTalk's poor relation, with the protocol delegating it to applications—a questionable choice by current standards but consistent with the philosophy of the 1980s. Cisco attempted to fill this gap with its distribution lists, a rudimentary but useful mechanism for controlling the propagation of routing information.

Cisco's implementation presented a surprising peculiarity: it refused to transmit certain packets with identical local addresses. This restriction, contrary to Apple's specifications, aimed to preserve the integrity of address tables. The manufacturer prioritized network stability over strict standards compliance.

AppleTalk's decline began in the 1990s. TCP/IP progressively established itself as the universal standard for network communications. Its open nature and adoption by the Internet sealed the fate of proprietary protocols. In 2009, Mac OS X v10.6 Snow Leopard definitively abandoned AppleTalk. Apple thus came full circle by turning toward the open standards it had initially shunned.

With the AppleTalk protocol, a sophisticated network remained simple to use. Auto-configuration and automatic service discovery, central concepts of AppleTalk, still inspire current developments. Bonjour, AppleTalk's spiritual successor at Apple, perpetuates this philosophy of simplicity in a TCP/IP world.

NTP

In the world of distributed systems, synchronizing time between distant machines represents an ongoing technical challenge. The Network Time Protocol solved this equation with elegance that spans decades. Its origin dates back to 1979, when a public demonstration revealed the Internet to the world via a transatlantic satellite link. This first appearance already highlighted the importance that temporal coordination would assume in network architecture.

Two years later, in 1981, the Internet Engineering Notes dedicated their reference IEN-173 to this synchronization technology. RFC-778 followed with the first public specification of the protocol. The initial design took an unexpected form: it was directly integrated into the Hello routing

protocol, documented under RFC-891. This hybrid approach persisted for several years within Fuzzball, an experimental operating system that served as a testbed.

The year 1985 marked the official birth of NTP version 0. David L. Mills developed his version in Fuzzball while Louis Mamakos and Michael Petry simultaneously worked on a UNIX implementation at the University of Maryland. Their respective codes left lasting traces in current software, testament to the soundness of their architectural choices. RFC-958 documented this first formal version, focusing on the NTP packet header and the offset and delay calculations that remain unchanged today.

At that time, hardware constraints dictated performance. Modest networks and computers limited precision to around ten milliseconds on Ethernet. By comparison, transatlantic links generally maintained precision under 100 milliseconds.

Version 1 emerged three years later with RFC-1059. This document constituted the first complete exposition of the protocol and its algorithms, including the first versions of the filtering, selection, and clock discipline mechanisms. These developments built upon a series of experiments documented in RFC-956, where the theory of the filtering algorithm took shape. This version introduced client/server and symmetric modes, as well as the use of the version field in packet headers.

In 1991, the first academic transaction article devoted to NTP Version 1 appeared. This document presented the complete model to the technical community for the first time: architecture, protocol, and algorithms formed a coherent whole whose principles remain current. Scientific recognition of the protocol was achieved with this publication.

NTP Version 2 arrived in 1989 with RFC-1119. Dennis Ferguson at the University of Toronto completely rebuilt it to faithfully respect the specification. This version brought two major innovations: the NTP control message protocol for administering servers and clients, and the cryptographic authentication system based on symmetric cryptography.

The specification for NTP Version 3 appeared in 1992. Its 113 pages included a remarkable appendix proposing a formal error analysis accompanied by a detailed budget covering all error contributions from the primary source to the final client. These works established the foundations of maximum and estimated error statistics, rigorously characterizing synchronization quality.

Lars Mathiesen at the University of Copenhagen then undertook a meticulous revision of version 2 to conform it to the version 3 specification. More than a year of intensive exchanges between specification and implementation resulted in a coherent formal model, an exemplary process

of convergence between theory and practice.

The following years saw NTP continue to evolve. New features and algorithmic revisions accumulated while preserving interoperability with previous versions. This period saw the birth of Simple Network Time Protocol (SNTP) version 4, a simplified protocol compatible with NTP on IPv4, IPv6, and OSI, but lacking the sophisticated mitigation and discipline algorithms of its predecessor.

The worldwide deployment of NTP evoked the ham radio spirit. Each new country discovered using NTP represented a symbolic victory for the developers. Satisfaction reached its peak when a country's national standards laboratory established a primary NTP server directly connected to its national time and frequency ensemble.

Maintaining configuration scripts and patches libraries constituted a thankless but necessary task. Harlan Stenn, a dedicated volunteer, currently manages this process using modern autoconfiguration tools. New versions first undergo testing on the DCnet research network, then in broader environments like CAIRN before publication on www.ntp.org.

Technological progress radically transformed NTP performance. Precision improved from hundreds of milliseconds in the turbulent Internet of the 1980s to tens of nanoseconds in the Internet of the new millennium. This evolution blended historical lessons and technological revivals, preserving those ham radio resonances with each new country appearing on the Internet with active NTP.

The arrival of affordable atomic clocks revolutionized the field. Rubidium models, with their drift of one microsecond per day, democratized access to a high-quality temporal reference. OCXO oscillators offered the best alternative for obtaining precise clocks in compact packaging. With temperature exerting the primary influence on the crystal oscillation rate, thermostatization controlled and compensated for these variations, limiting daily clock drift.

NTP retains the title of the oldest distributed application in continuous operation on the Internet. Its development continues in the face of new temporal synchronization needs. Data centers and cloud computing demand ever finer temporal precision, where every nanosecond counts for modern applications. This perpetual quest for temporal precision perfectly illustrates the evolution of the Internet, from an experimental network to the critical infrastructure of our digital society.

Microsoft Excel

In 1978, Dan Bricklin and Bob Frankston developed VisiCalc, which sold nearly a million copies and revolutionized microcomputing. Microsoft sensed the market's potential, and in 1982 the Redmond firm launched MultiPlan, which adopted the R1C1 addressing system rather than VisiCalc's A1 format. The software found its audience on CP/M systems but failed against Lotus 1-2-3 on MS-DOS. This defeat pushed Microsoft to rethink its strategy.

Excel was born in 1985, but exclusively on Macintosh. The choice seemed bold: Microsoft bet on the graphical interface, pull-down menus, and the mouse when most programs still operated in text mode. This vision proved successful. The Windows version arrived in 1987, accompanied by a streamlined version of the operating system for less powerful machines.

The following years transformed Excel into a true Swiss Army knife of calculation. In 1990, version 3.0 added toolbars, drawing capabilities, 3D charts, and outline functions. Excel 4.0, in 1992, improved usability and introduced the fill handle, that small square that duplicates formulas into other cells with a simple drag. A gesture that became so natural that users struggle to imagine doing without it.

Multi-sheet workbooks changed the game with Excel 5.0 in 1993, but it was especially the arrival of VBA that revolutionized usage. Users discovered they could record their repetitive actions as macros. Overnight, Excel moved beyond simple calculation to become an accessible programming tool.

The transition to 32-bit architecture with Excel 95 brought stability and speed. Excel 97 blessed us with the Office Assistant, that animated paperclip that annoyed as much as it helped, and perfected data validation. More sophisticated, the VBA interface attracted a growing community of developers.

Excel 2000 improved the multiple clipboard and added the auto-repair function. Excel 2002 introduced file recovery after crashes, a lifesaving feature for anyone who has lost hours of work. Excel 2003 focused on XML support and list ranges.

2007 disrupted habits. The Ribbon replaced traditional menus, disconcerting seasoned users but winning over novices. The XLSX and XLSM formats, based on XML, modernized data storage. Excel 2010 added sparkline charts, these mini-graphs embedded in cells, and more powerful filters.

This technical evolution masked a more complex reality. While Excel certainly established itself in businesses for financial analysis, project management, and statistical modeling, its ease of use concealed formidable pitfalls. Errors easily crept into complex formulas, and their consequences could be dramatic.

JPMorgan Chase learned this the hard way in 2012. An error in an Excel model calculating risks cost the bank $6.2 billion. Fidelity Magellan experienced a similar incident in 1995: an accountant forgot a minus sign in front of a $1.3 billion loss, completely distorting dividend estimates. These accidents served as reminders that Excel, despite its apparent simplicity, often manipulates considerable stakes.

This dual nature constitutes both Excel's strength and weakness. On one hand, it democratizes data analysis and makes numerical computation accessible to millions of non-specialist users. On the other, it generates a sometimes dangerous dependency in fields where precision matters. The structured finance and quantitative analysis sectors owe their rise to it, but also some of their most bitter disappointments.

Recent versions attempt to reconcile these contradictory requirements. Artificial intelligence enters the formulas, visualization tools become more sophisticated, processing capabilities expand to handle ever-larger data volumes. Excel adapts to cloud computing and collaborative work, areas where it had fallen considerably behind.

Microsoft isn't letting go. Recent developments show a determination to keep Excel at the heart of modern professional practices. The integration of advanced programming languages like Python, improved analytics, and adaptation to new formats demonstrate this ambition.

More than forty years after VisiCalc, Excel embodies the evolution of professional computing tools. Its trajectory reflects the transformations of office work and the progressive computerization of society. Its persistence in a perpetually changing technological world demonstrates its unique ability to meet fundamental information processing needs, despite its imperfections and the risks it generates.

Microsoft Windows

In 1983, Bill Gates announced that Microsoft was working on a graphical interface. The idea was nothing truly novel—Xerox had already explored this path at PARC, Apple had just released Lisa and was preparing

the Macintosh. But Microsoft wanted its piece of the pie and fully intended to transform the user experience on PC.

November 1985 saw the birth of the first version of Windows. The result disappointed. This graphical layer placed on top of MS-DOS offered windows, pull-down menus, and a few icons. Users, accustomed to typing their commands, struggled to adopt this novelty. Sales remained minimal.

Two years later, Windows 2.0 corrected several flaws. Windows could now overlap instead of neatly aligning in a mosaic pattern. Microsoft was then collaborating with IBM on OS/2 and seeking visual consistency between its different projects. This version went largely unnoticed.

The real takeoff occurred in 1990 with Windows 3.0. This time, Microsoft had a product that worked. The interface displayed 16 colors, exploited the capabilities of 386 processors, and offered three components that would make history: Program Manager, File Manager, and Print Manager. These tools simplified users' lives and made the computer accessible to the general public. Sales soared and caught up with Apple's.

Windows 3.1 arrived in 1992 with enhanced stability. OLE technology made its debut by enabling the integration of objects from one application into another. The first multimedia elements appeared on PC. Windows for Workgroups 3.11 added network functions essential to companies that were beginning to connect their machines.

Microsoft developed Windows NT in parallel, a system designed for professionals. Released in 1993, NT broke with MS-DOS and offered an entirely new architecture. The company targeted the server market, UNIX's stronghold. NT's robustness earned it C2 certification from the U.S. government, a mark of credibility in the corporate world.

August 1995 marked a milestone in personal computing history. Windows 95 arrived with its Start menu and taskbar, two elements that would become familiar to millions of users. "Plug and Play" simplified hardware installation. The 32-bit architecture improved performance. Microsoft spent $300 million on advertising and used "Start Me Up" by the Rolling Stones to accompany the launch. Computing definitively entered households.

Windows 98 refined the formula the following year. The Internet was gaining importance, and Microsoft facilitated connections. USB made its debut, the FAT32 format increased hard drive capacity. The system gained stability and offered automatic maintenance tools.

In February 2000, Windows 2000 finally unified the consumer and professional branches. Based on NT, this system brought the long-awaited reliability. Active Directory revolutionized enterprise network management. Multiprocessor support became a reality on workstations.

Windows XP, released in 2001, completed this evolution. The "Luna" graphical interface modernized the system's appearance. Hardware compatibility improved, multimedia found its place. XP combined the user-friendliness of home versions with the robustness of professional editions. This version would remain popular for a decade.

The 2000s brought new concerns. The Internet was becoming widespread but brought its share of viruses, Trojan horses, and other malware. Microsoft responded with regular patches and strengthened security. XP's Service Pack 2 introduced an integrated firewall and security center. The war against computer threats began.

This technical history was accompanied by commercial controversies. The integration of Internet Explorer into Windows prompted antitrust lawsuits in the United States and Europe. Competitors denounced Microsoft's practices and dominant position. These legal battles marked the 1990s and 2000s.

Windows's evolution reflected that of personal computing as a whole. From the austere text mode of MS-DOS to modern colorful interfaces, Microsoft managed to maintain compatibility with existing applications while innovating. This continuity partly explains the system's persistent dominance.

Windows runs on 72% of personal computers worldwide in 2025. Four decades after its creation, Microsoft's system withstands assaults from its competitors. Smartphones and tablets have disrupted the market, but Windows maintains its place on desktops and in enterprises.

Amstrad PCW

In 1985, European personal computing took an unexpected turn with the arrival of the Amstrad PCW. Far from the colourful gaming machines that dominated the home market at the time, this British computer adopted a radically different philosophy: making word processing its sole purpose.

Alan Sugar, head of Amstrad, envisioned an integrated system comprising keyboard, screen, processing unit, and dot-matrix printer. His specifications could be summed up in a few words: simple, affordable, effective. The Z80 clocked at 4 MHz formed the technical heart of the system. This 8-bit processor allowed developers to build upon their existing expertise rather than starting from scratch with the fashionable 6502. The

256 KB memory of the PCW 8256 could be doubled in the higher model, a respectable capacity for the time. But it was the choice of the 3-inch floppy disk format that most marked this machine's history: proprietary, it temporarily protected it from competition whilst mortgaging its future against the 3.5-inch format that was becoming standard elsewhere.

The monochrome screen of 720×256 pixels, with its characteristic green tint, displayed text exclusively. This deliberate limitation corresponded perfectly to the intended use: LocScript, the integrated word processing software, transformed each PCW into a sophisticated electronic typewriter. The CP/M operating system opened access to an existing software library, inherited from previous professional machines.

Commercial success exceeded all predictions. At £400 with the dot-matrix printer included, the PCW appealed to an audience that nobody had really targeted before: independent secretaries, small businesses, writers, local government offices. In France particularly, these white and green machines colonised the offices of independent professionals and public services. Over one million units found buyers between 1985 and 1990, a remarkable figure for such a specialised product.

Successive iterations refined the initial concept without betraying it. The 1987 PCW 9512 replaced the dot-matrix printer with a daisy-wheel printer, finally producing letter-quality documents. Its entirely white casing modernised the aesthetics whilst preserving the proven ergonomics. The PCW 9256, the final version from 1991, brought some minor technical improvements without revolutionising the usage.

LocScript itself evolved, progressively integrating French accents, basic spell-checking, and simple table management. Third-party software enriched the ecosystem: accounting, client file management, with a few games for moments of relaxation. The LocoLink interface enabled document exchange with MS-DOS PCs, a valuable opening when these began to become more affordable.

Yet the concept's weaknesses appeared over time. No colour, no proportional fonts on screen, noticeable slowness compared to 16-bit machines. Compatible PCs, which became affordable in the early 1990s, offered more versatility for a comparable price. The PCW, prisoner of its specialisation, gradually lost its commercial relevance.

This machine proved that a personal computer could choose efficiency over sophistication, simplicity over raw power. Today, many PCWs still function perfectly, testament to their exceptional build quality.

The Amstrad PCW embodies a particular vision of computerisation: rather than imposing a complex and general-purpose system, it's better to design a tool suited to a specific need. This approach allowed thousands of

users to discover digital word processing without technical intimidation. Secretaries, craftspeople, associations: all found in this white machine their first gateway to professional computing.

This European success, facing American PC standardisation, shows that several possible paths existed to democratise computing. Amstrad explored one of them successfully, proving that demand existed for simple and affordable solutions.

Atari 520 ST

After leading Commodore International to the success of the Commodore 64, Jack Tramiel left the company in 1983 following a brutal price war against Texas Instruments that wore him out. This break led him to found Tramel Technology Ltd with his sons and a few loyalists, notably Shiraz Shivij, one of the brains behind the C64.

The small team embarked on the adventure of an affordable 16-bit personal computer. The choice of processor proved tricky : the National Semiconductor NS32000 failed to deliver on its performance promises. It was finally the Motorola 68000 that won the decision, despite its higher cost. Meanwhile, Atari Inc. was sinking. The 1983 video game crash cost Warner Communications, its parent company, 10,000 dollars a day. Jack Tramiel smelled an opportunity and bought Atari's consumer division in July 1984 for 240 million dollars in stock.

The technical team accomplished the development of the 520ST in just five months, a small miracle that seems impossible today. The name tells the machine's story : 520 KB of RAM and that "Sixteen/Thirty-two" architecture that blended the 16 and 32-bit worlds. Microsoft did offer to adapt Windows, but the schedules didn't align. Digital Research saved the day with a GEM license, which would give the Atari its characteristic graphical interface.

At the January 1985 Consumer Electronics Show, the 520ST's specifications caused a sensation. The 8 MHz Motorola 68000 hid a 32-bit internal architecture behind a 16-bit external bus. Three graphics modes coexisted : 320 x 200 in 16 colors for games, 640 x 200 in 4 colors for a balanced compromise, and especially that 640 x 400 high-resolution monochrome mode that would win over professionals.

Atari's engineers slipped a few clever finds into their machine. Those integrated MIDI ports for the modest sum of 75 cents, thanks to down-

graded Motorola chips, would transform the 520ST into an electronic music instrument. The 3.5-inch floppy drive stored 360 KB, a respectable capacity when most competitors still stuck with the 5.25-inch. The TOS operating system initially loaded from a floppy before permanently migrating to ROM.

May 1985 saw the first 520STs arrive in stores, narrowly beating Commodore's Amiga 1000. Jack Tramiel played the aggressive pricing card : 799 dollars with a monochrome monitor, 999 dollars in color. Against professional machines that easily exceeded 2,000 dollars, the argument carried weight. More than 50,000 units found buyers in six months, mainly in Europe where the reception was warm.

The machine seduced unexpected audiences. Musicians discovered the joys of MIDI with software like Cubase and Logic Pro that would go on to great careers. Amateur programmers played with GFA BASIC and STOS. As for desktop publishing professionals, they appreciated that high-resolution mode, especially since Macintosh emulation ran faster on ST than on a real Mac !

European success was no accident. This compact, integrated design matched Old World tastes. The slogan "Power Without the Price" hit the mark in the United Kingdom, won over by affordable home computers. In the United States, things were more complicated : Jack Tramiel maintained execrable relations with retailers, who returned the favor.

The 520ST's influence exceeded its mere technical performance. Its ability to read MS-DOS floppies facilitated exchanges with the PC universe. Those standardized MIDI ports created an entire ecosystem that still survives today. Jean-Michel Jarre, Fatboy Slim, and Madonna used it in the studio and on stage, creating a legend that stuck to the machine.

The software catalog expanded. Dungeon Master arrived in 1987 and revolutionized the first-person role-playing game. Publishers like Batteries Included explored the graphical possibilities with programs like DEGAS. The nascent "demo scene" pushed the machine to its limits, creating works of digital art that defied all technical logic.

The range evolved through versions. The 520STM added an RF modulator to connect to a television. The 520STFM integrated the power supply and drive in a redesigned, more family-friendly case. TOS gradually migrated from floppy to ROM, simplifying use.

In 1993, production of the 520ST stopped, swept away by the wave of IBM-compatible PCs. Yet the machine kept its faithful followers, especially among musicians who appreciated its low MIDI latency. It embodied that era when technical innovation rhymed with democratization of technology, when a handful of engineers could still shake up the computer

industry.

Toshiba T1100

Toshiba revolutionized personal computing with the T1100 in April 1985. For the first time, a laptop computer offered complete compatibility with the IBM PC. This 4 kg machine marked the birth of modern mobile computing.

The era seemed hardly conducive to such innovation. Personal computers, the Apple II since 1977 and the IBM PC since 1981, remained confined to desktops. A few manufacturers did attempt to create transportable machines: Compaq's Portable I weighed 12.5 kg in 1982, while Seiko released its 1.6 kg HX-20/HC-20 the same year, followed by NEC's PC8201A in 1983. But these attempts faced a technical dilemma: either maximize miniaturization with incompatible proprietary architectures, or aim for IBM PC compatibility while accepting bulk and limitations.

Toshiba's teams refused this compromise. Their motto "anywhere, anytime, anyone" expressed a clear ambition: to transform the desktop computer into a mobile companion. The company theorized this approach as a "revolution on the desk", a shift from machine-centered computing toward human-centered use. Three objectives guided development: abandoning closed architectures, making the computer truly portable, and enabling the use of standard software without specific programming.

The technical achievement was an engineering marvel. CMOS integrated circuits replaced power-hungry NMOS technology. Where the IBM PC required nearly 100 circuits for display, the T1100 made do with five. This radical optimization produced a compact device with unprecedented battery life. Energy was a field of innovation. Toshiba created from scratch a charge-discharge system and power management software. These developments would lay the groundwork for the *resume* function introduced in 1989 on the J3100SS001 model, then the ACPI standard developed with Microsoft and Intel.

The high-contrast LCD screen, mounted on a tilting stand, resulted from extensive studies on visual ergonomics. But the real commercial challenge emerged elsewhere: the choice of 3.5-inch floppy disks. While the IBM PC operated with 5.25-inch format, Toshiba bet on the compact format. Only publishers were reluctant to distribute their software on this still-marginal medium, despite the work of commercial teams who

conducted an intensive persuasion campaign to convince them to adopt 3.5-inch.

The marketing strategy broke with industry habits. No more technical arguments! Toshiba concretely demonstrated how to work mobile with the same software as at the fixed office. This pragmatic approach won over non-technical professionals and considerably expanded the market.

Commercial success confirmed this intuition. A study revealed two distinct expectations: portability on one side, more power and capacity on the other. Toshiba responded in January 1986 with two new models. The T2100 featured an i8086 processor and two 3.5-inch drives. The T3100 moved upmarket with an i80286 and a 10-megabyte hard drive. These machines required mains power, but *PC Week* validated this choice by recognizing the existence of a market for plugged-in laptops. *Byte* magazine called the T3100 the "King of Laptops". In 1987, Toshiba reached approximately 40% market share of the 130,000 laptops sold in Europe.

The economic impact exceeded all predictions. In 2008, the market for LCD screens under 17 inches represented $15 billion. That of 2.5-inch and smaller hard drives reached $19 billion. Lithium-ion batteries for mobile equipment generated $12 billion in revenue. A year later, laptop sales surpassed desktop machines in a market estimated at $150 billion.

The T1100 demonstrated that a user-centric approach transforms technology sectors. By prioritizing software compatibility, portability, and simplicity over raw performance, Toshiba took the computer out of IT departments and placed it in executives' briefcases. This democratization of mobile computing heralded the current ubiquity of portable devices in our lives.

Aldus PageMaker

In 1984 in Seattle, Paul Brainerd left his position at Atex to found Aldus Corporation. He was no stranger to the publishing world. A journalist by training, he had overseen the modernization of the Minneapolis Star and Tribune's systems before joining this manufacturer of computer systems for publishing professionals. This experience gave him firsthand insight into a problem: composition equipment cost a fortune, often several hundred thousand dollars, and remained inaccessible to smaller organizations.

With its graphical interface, Apple's Macintosh opened unprecedented

possibilities. Brainerd sensed that democratizing page layout tools was possible. He assembled a small team of engineers and developed a prototype in six months. Finding funding proved far more complicated: of fifty venture capital firms contacted, only one, Vanguard, agreed to bet on this project. At the time, investors were wary of software companies, deemed too risky in a sector where Microsoft had not yet gone public.

On July 13, 1985, PageMaker 1.0 was born. This date was no accident: it came seven months after Apple's LaserWriter printer launch. Brainerd understood that his software needed to leverage Adobe's PostScript language capabilities, integrated into this printer, to produce professional-quality documents. He spoke of the "three-legged stool" of desktop publishing, a term he actually coined during a meeting with his investors in late 1984.

This term "desktop publishing" hit the mark. Simple and effective, it perfectly captured the project's ambition: to bring professional page layout tools within everyone's reach. PageMaker's interface embodied this philosophy. The software simulated a real layout table where users could place and move text and images at will. The floating tool palette was a notable innovation, providing access to key functions. The WYSIWYG principle ensured that what appeared on screen exactly matched the printed result.

The success exceeded all expectations. Users flocked from unexpected sectors. Churches began producing parish bulletins sometimes printed in runs exceeding 600,000 copies. Steve Jobs had pushed to keep the price under 200, *butthe*495 ultimately chosen proved judicious: this pricing generated the margins necessary for ongoing development and technical support.

Aldus also distributed FreeHand, a vector drawing software created by Altsys, then acquired Silicon Beach Software with its range of consumer applications. These acquisitions gradually built a complete suite of graphic creation tools.

The competition responded. Ventura Publisher arrived in 1986 on PC with a different approach, based on style sheets and suited for long documents. QuarkXPress gained ground in the professional sector. Microsoft tried its luck but had to withdraw its product due to critical technical flaws.

Beyond the software, Aldus shaped industry standards. Steve Carlson developed the TIFF (Tagged Image File Format) at Aldus, which became the reference for digital image exchange. The company later created OPI (Open Prepress Interface) to streamline integration with professional printing systems.

By 1990, margins eroded under competitive pressure and the evolution of distribution channels. PageMaker grew more complex, torn between the needs of occasional users and professional requirements. This growing complexity complicated development and increased costs.

In 1994, Aldus merged with Adobe Systems. This operation, prepared with meticulous care, united both companies' technologies. The absorption proceeded remarkably well, creating a digital graphic creation giant.

PageMaker proved that an accessible computer tool could supplant prohibitively expensive specialized equipment. It established user interface conventions, but above all, it opened access to professional-quality publishing to individuals and organizations that had never been able to afford such luxury before.

Subsequently, PageMaker's code nourished Adobe InDesign, its spiritual heir that now dominates the desktop publishing market. Paul Brainerd turned to philanthropy after the Adobe merger. His Brainerd Foundation works to protect the environment in the American Northwest.

This story shows how a clear vision, supported by rigorous technical execution and favorable timing, can transform an entire sector. It also underscores the importance of open standards and cooperation between companies in creating new markets.

IBM AIX

During the 1970s, UNIX spread through universities, but without a standardized commercial version. Each installation required adapting the source code to the local hardware. This technical constraint led to the creation of multiple variants, each reflecting the specificities of its environment.

IBM took interest in the UNIX phenomenon and launched AIX (Advanced Interactive eXecutive) in 1986, developed with the help of Interactive Systems Corporation. The name actually came from ISC, which had already created the PC-IX system for IBM personal computers.

IBM was juggling two distinct Power architectures at the time. On one side, OS/400 equipped a range of machines, on the other, AIX established itself on the IBM 6150 RT workstation in 1986. The AS/400 followed the next year with OS/400.

AIX's real rise to prominence came in 1990 with Version 3, renamed AIX/6000. It accompanied the RS/6000 platform which definitively buried

the IBM RT. This RS/6000 generation inaugurated the era of POWER and PowerPC microprocessors at IBM. AIX finally found its machine of choice.

1994 marked an important technical milestone with AIX 4, which introduced symmetric multiprocessing on the first RS/6000 SMP servers. The system matured through version 4.3.3 in 1999, while IBM rethought its product strategy.

The year 2000 transformed IBM's organization. The company unified its server lines under the eServer banner: AS/400 became iSeries, RS/6000 transformed into pSeries. Beyond the marketing, this restructuring concealed a technical reality. The arrival of the POWER4 processor in 2001-2002 brought the hardware platforms closer together. The differences between "p" and "i" systems were now limited to the software layers.

POWER5 took an additional step in 2004 by making the System i5 570 strictly identical to the System p5 570. AIX 5.3 arrived in August, taking advantage of this standardization. POWER6 followed in May 2007, accompanied by AIX 6.1 in November. IBM pushed the logic to its conclusion in April 2008 by merging its two lines under the Power Systems designation. A single hardware, three system choices: IBM i, AIX or Linux.

The 2010s followed at a sustained pace. POWER7 arrived in February 2010, AIX 7.1 in September. POWER7+ landed in August 2012, then POWER8 in June 2014. This latest processor broke with tradition: each core handled eight hardware threads in parallel, a massively multithreaded architecture that exploited large caches to maximize memory and I/O throughput.

AIX 7.2, announced in October 2015, brought a remarkable feature: Live Kernel Update. Kernel patches could now be applied without disrupting running applications. This version also cleaned up obsolete components and restructured the network with bos.net.tcp.client, but it absolutely required POWER7 or a later generation.

POWER9 arrived in 2016 with its 14 nm fabrication and 3.9 GHz frequency. Performance gains reached 50% compared to POWER8, and doubled versus POWER7+. On AIX 7.2 TL 3, SMT8 mode activated automatically, fully exploiting the processor's capabilities.

Today, AIX powers the most critical infrastructures. Finance, industry, retail, telecommunications, healthcare, government and public sector: wherever reliability is non-negotiable. The system has proven its ability to evolve with new hardware architectures without sacrificing compatibility with existing systems.

This technical longevity perfectly illustrates IBM's strategy: converg-

ing its server lines toward a unified platform while preserving customers' software investments. AIX embodies this continuity in innovation, heir to UNIX but resolutely oriented toward the future of enterprise computing.

Eiffel

September 23, 1985. In the offices of Interactive Software Engineering, a small Californian company in Santa Barbara, Bertrand Meyer works on an internal project that will disrupt many established practices in the world of object-oriented programming. On that day, Eiffel is born, though it will not be until mid-1986 before the first operational version is running.

Meyer does not start from scratch. Since 1973, he has been working with object-oriented concepts using Simula 67. Jean-Raymond Abrial's work on formal specification fascinates him, particularly his original version of the Z language. Liskov, Zilles, and Guttag have broken ground in abstract data types, a field Meyer has explored. The prestigious lineage of Algol 60, Algol W, Pascal, and Ada shapes the general architecture of the new language, while Floyd, Hoare, and Dijkstra contribute their expertise in program proof and axiomatic semantics.

This particular alchemy produces something unusual. In September 1986, at the first "Object-Oriented Programming, Systems, Languages & Applications" conference in Portland, the presentation of Eiffel causes a sensation. The concepts developed surpass what exists at the time, both in industry and research laboratories. Commercialization follows with the internal compiler becoming available in December 1986. Companies and universities can finally get their hands on it.

Version 2 arrives in 1988, the year that sees the publication of "Object-Oriented Software Construction", the book that will propel Eiffel into the spotlight. The 1997 edition, considerably expanded, confirms the language's growing influence. Version 2.3 from 1990 marks the peak of ISE's original technology.

Meyer then makes a bold decision: in 1990, ISE releases the language definition into the public domain. This openness triggers a flourishing of projects—compilers and libraries spring up everywhere. The language undergoes a general overhaul, simplifying in some areas, enriching in others, but retaining its original soul. This remarkable stability makes Eiffel a language that has hardly changed since its 1985 design.

Between 1990 and 1993, the ISE team completely rethinks its technol-

ogy. Version 2.3 serves as the foundation for this complete reconstruction. ISE Eiffel 3 arrives in 1993 with its graphical development environment, followed by ISE Eiffel 4 in 1997 and Eiffel 5 in 2001. These versions bring their share of innovations: agents borrow from functional languages, introspection examines programs, the Precursor mechanism simplifies repeated inheritance, object creation gains flexibility, and conversion mechanisms are refined.

The BON notation emerges in 1995 with the publication of "Seamless Object-Oriented Software Construction" by Waldén and Nerson. This Business Object Notation extends Eiffel toward analysis and design, in a language understood by managers, analysts, and system architects.

The ecosystem grows. Object Tools in Germany develops Visual Eiffel, successor to Eiffel/S. The University of Nancy offers Small Eiffel as a free version, Halstenbach GmbH markets its own German solution. Libraries proliferate: 3D graphics, lexical analysis with Gobo, DirectX interfaces, variable precision calculation. Every domain finds its solution.

Industries discover Eiffel's virtues. Banking adopts it, finance takes interest, accounting experiments with it. Telecommunications, healthcare, CAD-CAM, simulation, and scientific computing place their trust in it. The Rainbow system at CALFP bank tells this success story: initially designed for derivatives trading, it ends up supervising the majority of banking operations.

Universities embrace the language with enthusiasm. Many faculties make it the first language taught to students. Others integrate it at different levels of the curriculum, encouraged by attractive packages from publishers.

Why this name, Eiffel? Meyer pays tribute to Gustave Eiffel, the engineer who did not merely erect the Parisian tower. The metal framework of the Statue of Liberty in New York, a railway station in Budapest—so many enduring constructions emerged from his workshops. The Eiffel Tower, begun in 1887 for the 1889 Universal Exhibition, respects deadlines and budget. It survives political hostilities, resists attempts at destruction, finds new uses with radio and television. Built from robust and elegant patterns, combined and varied to produce a powerful result, it perfectly embodies what Eiffel accomplishes in the software world.

"Design by Contract" forges Eiffel's identity. This approach advances software construction by founding it on contracts between clients and suppliers. Mutual obligations and benefits are made explicit through assertions, clearly establishing the responsibilities of each component. This philosophy produces more reliable and maintainable software, where each element knows its role and its limits.

Erlang

In 1986, in the offices of the Ericsson Telecom AB computer laboratory, Joe Armstrong faced a technical puzzle: how to program telephony applications that defied all the conventions of the time? Telephone switches handled thousands of simultaneous calls, these operations were naturally distributed across multiple machines, and the slightest failure made headlines. Nothing like a desktop application that crashes without consequence.

Armstrong first explored different approaches, developed prototypes in Smalltalk, and invented a graphical notation to describe telephony operations. It was Roger Skagervall who made the decisive observation: this notation resembled Prolog. Armstrong then switched to this language, and thus Erlang was truly born.

The technical context at Ericsson influenced the design of the new language. PLEX, created by Göran Hemdahl to program AXE switches, imposed its constraints: modifying code without stopping the system, avoiding at all costs the pointer errors that had handicapped previous generations. Armstrong and his team integrated these lessons into Erlang's DNA.

Robert Virding joined the project, and together they developed the first versions of the language, still in Prolog. The moment of truth came in 1989 with the ACS/Dunder project. The team developed 25 telephony features in Erlang, roughly one-tenth of those in the MD110. The results exceeded their expectations: depending on the features, development accelerated from 3 to 25 times compared to PLEX.

These successes attracted attention, and Erlang gained maturity. In 1993, Bogumil Hausman developed the Turbo Erlang system, renamed BEAM (Bogdan's Erlang Abstract Machine), which drastically improved performance. Claes Wikström added distribution support, enabling execution across multiple machines.

But it was an unexpected event that would propel Erlang beyond Ericsson's walls. In 1998, Ericsson Radio Systems banned the use of the language for new developments. This decision, which could have spelled the end of Erlang, produced the opposite effect. In December, Ericsson released the source code, and part of the original team left the company to found Bluetail AB, a company that would use Erlang as its core technology.

Erlang's technical architecture reflects its original mission. The lan-

guage structures programs around concurrent processes that share no memory and communicate through asynchronous messages. These processes belong to the language, not the operating system, which makes them remarkably lightweight. If one of them fails, the others continue their work as if nothing had happened.

The message reception system constitutes one of Erlang's most remarkable innovations. A process can selectively wait for certain types of messages while leaving others pending. This approach greatly simplifies the programming of complex communication protocols, a considerable asset in the telecommunications world.

The AXD301, an ATM switch developed by Ericsson, is Erlang's technological showcase. In 2001, this system comprised 1.13 million lines of code distributed across 2,248 modules. It achieved the coveted availability of 99.9999999%, demonstrating that the language was capable of handling large-scale industrial projects with exceptional reliability.

The OTP (Open Telecom Platform) system, developed from 1996 onward, enriched the Erlang ecosystem. This collection of libraries and design patterns offered "behaviours", abstractions that encapsulated common patterns such as the client-server model or event handling. OTP transformed Erlang from an experimental language into an industrial platform.

The arrival of multicore processors retrospectively confirmed the relevance of the initial choices. The concurrency model without data sharing naturally adapted to hardware parallelism. Erlang programs exploited multicore architectures without modification, a considerable advantage at a time when other languages struggled to take advantage of this evolution.

Beyond telecommunications, Erlang found its place in various domains. The ejabberd messaging server (Erlang Jabber Daemon) handled millions of simultaneous connections, CouchDB stored and distributed data at large scale, and RabbitMQ routed messages between distributed applications. These successes validated the original approach: the constraints of telephone systems were found in many contemporary applications.

The language initially addressed the particular requirements of Ericsson, but its solutions proved relevant for many current problems. The initial architectural choices, particularly the message-passing concurrency model, seemed radical but today enable natural exploitation of modern parallel architectures.

IMAP

In 1970, who could have imagined that electronic messaging would become one of the most universal applications on the Internet? At that time, messaging systems resembled internal address books, confined to users sharing a single machine.

The 1980s radically transformed the computing landscape. The arrival of individual workstations disrupted established practices. Users now demanded nomadic access to their messages from any workstation while maintaining a unified view of their mail. The POP protocol, launched in 1984, partially met this expectation by downloading messages locally. But this solution quickly showed its limitations when it came to managing emails stored on the server.

Mark Crispin observed these developments from Stanford University. In 1986, he developed IMAP, a radically different approach that placed the server at the heart of message management. The first public version, IMAP2, emerged in 1988. The major innovation lay in the ability to directly manipulate messages on the server without prior downloading. This architecture enabled the creation of multiple folders and offered sophisticated search functions within email content.

IMAP's technical architecture rested on a clear separation between two components: the message handling system (MHS) that orchestrated transfers, and the user agent (UA) that interacted with the interface. This modularity opened an immense field for email client developers, free to innovate on user experience without worrying about transport mechanisms.

The 1990s marked IMAP's gradual adoption by the Internet community. The protocol was enriched through successive versions: support for attachments via MIME, reinforced authentication mechanisms, disk space quota management. IMAP4 obtained Internet standard status in 1994, consecrating its technical maturity.

The University of Pittsburgh offers a striking example of this transition. In 1997, the institution launched a massive project: migrating 25,000 users to IMAP. The operation, completed in 2000, required developing custom tools to convert old messaging systems, train teams, and deploy server infrastructure equal to the stakes. This migration perfectly illustrates the organizational challenges that adopting IMAP at scale represented.

IMAP's technical sophistication shows through in its connection state management. The protocol juggles between three modes: connected, disconnected, and synchronization. This flexibility proved prophetic with the explosion of mobile usage. IMAP already knew how to handle intermittent connections, optimize data transfers, and maintain consistency

between client and server—concerns that would become central with the advent of smartphones.

In retrospect, IMAP anticipated cloud computing technologies: storage centralization, multi-device access, transparent synchronization. These characteristics explain its remarkable longevity and adoption by virtually all email providers, from web giants to local hosters.

Email clients contributed significantly to IMAP's success. Pine, with its spartan but effective interface, Mulberry and its advanced features, then Thunderbird and its democratization successively popularized the protocol. Native integration into operating systems and web browsers completed its universal distribution.

IMAP's impact on professional practices goes beyond the purely technical realm. Shared mailboxes, simultaneous message access by multiple collaborators, nomadic email consultation transformed work organization. Long before the smartphone era, IMAP made mobile messaging possible, an essential condition for today's professional flexibility.

The technical innovations carried by IMAP spread to other domains. Fine-grained network connection management, robust client-server synchronization, data flow optimization inspired many other protocols. This technical fecundity testifies to IMAP's quality of design.

Despite competition from webmail and the explosion of instant messaging, IMAP retains a prominent place in the Internet ecosystem. Its ability to evolve while preserving compatibility, its proven robustness, and its open standardization make it an enduring technology. Current developments focus on performance improvement, security reinforcement, and adaptation to new uses such as the Internet of Things.

NSFNet

By the late 1970s, growing frustration gripped the American scientific community. ARPANET connected a dozen privileged universities, but the Department of Defense refused to extend this network to other institutions. This situation created a technological divide between connected establishments and others, deprived of the benefits of digital communication.

Larry Landweber grasped the magnitude of the problem. In May 1979, he convened a meeting at the University of Wisconsin to seek solutions.

The assembly discovered that email and file transfer services were transforming the way researchers worked. Already, a few specialized networks were showing the way: THEORYNET brought together 200 theoretical computer scientists, SAMNET united 50 performance analysis specialists. These limited experiments revealed the potential of electronic exchanges for research.

Six months later, a consortium of universities knocked on the National Science Foundation's door. The project seemed attractive: create a national network accessible to all computer science departments, with reasonable costs and usage-based billing. The infrastructure would rely on commercial networks like Telenet. But the evaluators remained skeptical. Wasn't there overlap with ARPANET? How would such a project be managed? The NSF preferred to commission a detailed study.

The summer of 1980 saw the birth of a planning committee bringing together nineteen network experts. Two discoveries changed everything. The MMDF software, designed at the University of Delaware, transported messages across different media, including ARPANET and telephone lines. In parallel, DARPA adopted internet protocols that allowed communication between distinct networks. These technical innovations opened unprecedented perspectives: multiple physical networks could form a single logical organization.

A new proposal emerged in the fall of 1980. The consortium expanded to include the universities of Wisconsin, Purdue, Utah, Delaware, and the Rand Corporation. The National Science Board approved the project in early 1981, but imposed one condition: the NSF would direct the operation for two years before handing it over to an independent structure. Contracts were signed in the spring.

In 1985, the NSF launched a more ambitious challenge: connecting its supercomputing centers scattered across the country. NSFNET was born in 1986, with 56 kbps links connecting six strategic sites. The technical innovation relied on LSI-11/73 minicomputers equipped with Fuzzball software. This system integrated internet protocols with sophisticated routing and congestion management mechanisms.

The success exceeded expectations. As early as 1988, link saturation required an upgrade to 1.5 Mbps. The network expanded to regional academic networks, creating a complex web of interconnections. The Fuzzball nodes orchestrated adaptive routing, synchronized clocks with precision, and fairly distributed resources. Software agents filtered routing information between the network core and its periphery.

The rise of the internet transformed the landscape. Between 1993 and 1998, NSFNET evolved into a commercial architecture. Private providers

multiplied, pushing the NSF to rethink the network organization in 1993. This new structure endures today. Contracts signed in 1995 established interconnection points between commercial networks. In April 1995, the public NSFNET service closed its doors, replaced by a mesh of private networks.

This transition was accompanied by an unexpected transfer of responsibilities. Since the beginning, the Department of Defense had managed domain names for its military users. In the early 1990s, academic institutions represented the majority of new registrations. The Federal Networking Council then entrusted this mission to the NSF. Faced with exploding demand, registration became fee-based in September 1995. The figures were staggering: 7,500 domains in 1993, 2 million in 1998.

The year 1998 marked the complete privatization of internet's critical functions. ICANN (Internet Corporation for Assigned Names and Numbers) was created to supervise the domain name system. The NSF refocused on its core mission: supporting research on network protocols and applications. It continues to help isolated institutions connect.

NSFNET transformed the internet. From a confidential military and university network, it became the backbone of a global infrastructure. Its distributed architecture and public-private collaboration mechanisms shaped the current organization of the worldwide network. The technical innovations developed for this project—adaptive routing, precise time synchronization—remain pillars of the internet today.

Beyond technology, NSFNET invented a unique governance model. Universities, companies, and public agencies collaborate to preserve the network's openness while ensuring its growth. This success demonstrates how a public initiative can trigger the development of infrastructure that has become indispensable to the modern digital economy.

SCSI

In the 1960s, IBM was working on its mainframe computer, the 360, and developed an innovative I/O bus for its time: it could communicate simultaneously with multiple peripherals. This bus evolved to become the OEM Channel, but when IBM submitted it to ANSI for standardization, the institute refused. The reason? Its overly proprietary nature. ANSI preferred to see the emergence of a parallel I/O bus that would meet commercial needs without depending on a single manufacturer.

The story took an unexpected turn in the early 1980s. At Shugart Associates, a hard disk drive manufacturer, a few engineers who would later found Adaptec developed a parallel interface they named SASI (Shugart Associates System Interface). This specification circulated among manufacturers and met with great success. In 1982, building on this adoption, it was presented to ANSI as the basis for a new standard. The institute seized the opportunity, formalized and extended the SASI specifications, but changed its name to SCSI to avoid any reference to a particular manufacturer. On June 18, 1986, SCSI officially became an ANSI standard.

SCSI stood out for its versatility: it could control a multitude of peripherals, from hard drives to printers to CD-ROM drives and scanners. Its architecture interconnected small computers with their intelligent peripherals, particularly storage systems. Performance was remarkable: up to 4 MB/s depending on implementation, with a range that reached 25 meters thanks to differential drivers and receivers.

What made SCSI clever was its logical rather than physical addressing protocol for all data blocks. Each logical unit could be queried to determine its capacity in blocks. This abstraction considerably simplified the management and replacement of peripherals. The protocol provided for the connection of multiple initiators and targets, with a distributed arbitration system built directly into its architecture.

The 1990s revealed an interesting divide in the computer world. Macintosh computers massively adopted SCSI to connect their peripherals, while PCs remained faithful to the IDE/ATA interface. This difference was not trivial: almost all Mac components connected via SCSI, whereas IDE only handled hard drives. On PCs, users had to juggle specific controllers for CD-ROM drives (often integrated into sound cards) and connect tape drives to the floppy disk controller. SCSI avoided this proliferation of interfaces through its single bus.

The race for performance constantly pushed the technology to new heights. The original SCSI operated with an 8-bit bus clocked at 5 MHz, which yielded approximately 5 MB/s. To go faster, two paths were available: widen the bus or speed up the clock. These successive improvements gave birth to SCSI-2, which introduced variants such as Wide SCSI (16-bit bus) and Fast SCSI (10 MHz clock).

HVD (High Voltage Differential) appeared in the SCSI-2 standard to push distance limits. This technique used an additional pin that guaranteed precise transmission, allowing cable runs of up to 25 meters. Ultra SCSI, which fell under the SCSI-3 standard, refined the concepts of SCSI-2. Throughput continued its ascent with Wide Ultra SCSI, Ultra2 SCSI, and Wide Ultra2 SCSI. Ultra3 SCSI simply doubled the clock frequency

while maintaining its compatibility with SCSI-3.

SCSI's speed owed much to its underlying protocol, SCSI Block Commands (SBC). This system authorized continuous data transmission between peripherals with a single command. Take the example of an audio CD: the SCSI controller sent an SBC signal to the drive, which then streamed data to the host controller until the end of the disc or until receiving a stop signal. The drive could redirect its data directly to a hard disk, bypassing the controller and system bus to conserve bandwidth.

Yet, despite its undeniable qualities, SCSI ultimately lost the consumer market battle to IDE/ATA. The arrival of ATAPI (AT Attachment Packet Interface) changed everything: IDE peripherals could now handle devices other than simple hard drives. This extension of the IDE standard, coupled with the high cost of SCSI controllers, precipitated the gradual abandonment of this interface in personal computers. But SCSI did not disappear: it retained its place in professional systems and servers, where its technical characteristics remained essential.

Innovation did not stop there. iSCSI enabled SCSI commands to be sent over local, wide area, or Internet networks, revolutionizing remote access and management of SCSI peripherals. The influence of this interface was also found in other technologies: IEEE-1394 (FireWire) relied on a subset of SCSI-3 specifications to control its peripherals, and FireWire hubs bore a striking resemblance to simplified SCSI adapters.

Today, SCSI's legacy still permeates modern computer storage. Its logical addressing approach and command architecture have inspired the interfaces that succeeded it. Its ability to efficiently manage peripherals and its robustness in professional environments have made it a foundational technology.

Bash

In the UNIX shell universe, few stories rival that of the Bourne-Again Shell. Bash takes its name from a play on words with Steve Bourne, the man behind the original UNIX shell (/bin/sh) from Bell Labs' seventh edition. Brian Fox, working for the Free Software Foundation, launched this project with a clear ambition: to create a free command interpreter that would become the heart of the GNU system.

The exercise proved more challenging than it appeared. Recreating a Bourne-compatible shell wasn't limited to developing documented fea-

tures. Fox and his team discovered that Unix users exploited every corner of the original shell, including its most obscure behaviors and undocumented subtleties. The Bourne shell's grammar lacked precision, turning every detail into a technical challenge.

Despite these difficulties, Bash grew richer. The developers drew from the best of other shells: the advanced features of the Korn shell (ksh) and the innovations of the C shell (csh) came to complement the POSIX-compatible base. This hybrid approach gave birth to a tool that respected standards while offering a modern user experience.

When Chet Ramey from Case Western Reserve University took the reins of the project, Bash entered a maturation phase. Ramey brought a long-term vision and a rigorous development method. Under his direction, the shell gained stability and features. Version 1.13 adopted eight-bit clean operation. Until then, Bash used the eighth bit of characters to mark their status during expansions, which limited the handling of files containing accented characters or special symbols. This correction paved the way for better internationalization.

Bash's interface revealed the attention paid to user experience. The readline library transformed command input into a true integrated text editor. Users of emacs or vi found their familiar keyboard shortcuts directly in the shell. Automatic completion guessed file names, available commands, and environment variables. Command history, inherited from the C shell, transformed each session into a personal database of reusable instructions.

One of Bash's strengths lay in customization. Each user could shape their environment according to their preferences through the `.bashrc` and `.bash_profile` files. The command prompt adorned itself with colors and contextual information thanks to escape sequences. Some displayed the time, others the current directory or the system load level. This flexibility transformed the terminal from a simple interface into a true personalized dashboard.

Job control inherited from the Korn shell brought a multitasking dimension to the shell. Users juggled between multiple processes, suspended them, resumed them, or relaunched them in the background. Aliases shortened frequent commands, while brace expansion multiplied the possibilities for character string generation. Typing "file{1,2,3}.txt" automatically generated file1.txt, file2.txt, and file3.txt.

Bash's distribution illustrated the free software dissemination methods of the 1990s. Source code circulated via anonymous FTP from MIT servers and their mirrors scattered around the world. The Free Software Foundation sold magnetic tapes and CD-ROMs for those who preferred

physical media. Early Linux distributions adopted a dual strategy: a full-featured version coexisted with a minimal version intended to directly replace /bin/sh.

Technical evolution never stopped. Bash acquired support for one-dimensional arrays, copying this functionality from the Korn shell. Programmable completion grew more sophisticated, allowing developers to specify custom behaviors according to context. Dynamic loading of built-in commands extended the shell's capabilities during its execution, transforming the interpreter into an extension platform.

Thanks to its remarkable portability, Bash ran on OS/2, DOS, and Windows NT. QNX and Minix integrated it into their standard distributions. This ubiquity reinforced its position as the reference shell and familiarized a generation of users with its specifics.

The user community shaped Bash as much as its official developers. Thousands of experience reports fed the development cycles. Bug fixes arrived from around the world, transforming each version into a collective effort. This collaborative dynamic gave birth to an exceptionally robust tool adapted to real needs.

Bash ultimately became much more than a simple shell: it embodied the UNIX spirit in its free and democratized version. Its comprehensive documentation, detailed man pages, and active maintenance made it a reliable companion for generations of programmers and system administrators. In web servers, backup scripts, or continuous integration pipelines, Bash continues to silently process millions of daily tasks, a direct heir to a tradition several decades old.

GIF

In 1987, CompuServe launched the GIF format (Graphics Interchange Format) with a simple ambition: to create an image exchange system that would work across all platforms. At that time, sharing a graphic file between different systems was often an uphill battle. CompuServe bet on openness: the format specifications were freely published, which accelerated its adoption within the computing community.

The first version, dubbed GIF87a, incorporated the LZW compression algorithm developed by Terry Welch three years earlier. This process compressed images without losing information. The format supported up to 256 indexed colors, a limitation that seemed constraining but would later

reveal its unexpected creative potential. Two years later, CompuServe refined its format with version GIF89a, which added transparency and, most importantly, the ability to create animations through successive images.

The honeymoon ended abruptly in 1994. Unisys, owner of the patent on the LZW algorithm, suddenly decided to demand royalties for any commercial use. The news hit the software industry like a bombshell. Many developers found themselves trapped, forced to rethink their business models or simply abandon support for the format altogether. This crisis paradoxically stimulated innovation: it led to the creation of the PNG format in 1996, designed as a free alternative to GIF.

But GIF resisted and transformed itself. With the explosion of the Web, its supposed weaknesses became strengths. This 256-color palette, deemed archaic by purists, inspired artists who found in it a minimalist and expressive aesthetic. Loop animation, initially conceived as a simple technical feature, has since become the format's signature.

During the 2000s, creation tools became democratized while the first social platforms encouraged sharing of visual content. GIF found its calling: to become a universal language of instant emotion. "Reaction GIFs", these film or series excerpts transformed into expressive loops, revolutionized online communication. A few images were now enough to express what paragraphs sometimes struggled to convey.

Tumblr crystallized this phenomenon by encouraging content creation and reappropriation. On this platform, GIF transcended its status as a tool to become an artistic medium in its own right. Creators explored its possibilities, producing works that oscillated between kinetic art and social commentary. A digital generation grew up with this format, favoring immediate visual impact over narrative complexity.

The expiration of LZW patents between 2003 and 2004 definitively freed the format from its legal constraints. This emancipation coincided with the advent of smartphones and mobile social networks. GIF proved perfectly suited to the constraints of mobile communication: compact, universal, instantaneous. Its compatibility with all devices made it a common denominator in a fragmented technological ecosystem.

The art world eventually recognized the cultural importance of the format. MoMA exhibited GIFs in its collections, legitimizing this long-considered minor medium. This institutional recognition accompanied the emergence of new artistic practices such as the *cinemagraph*, which blends static photography and subtle animation to create unsettling images of suspended reality.

Despite the emergence of technically superior formats like WebM or MP4, GIF maintained its position. Its simplicity paradoxically consti-

tutes its strength: no need for complex codecs, no compatibility issues, no restrictive licenses. This technical accessibility is coupled with cultural accessibility that makes it a democratic expression tool.

The history of GIF illustrates a rare phenomenon in the technological universe: the social appropriation of a technical format beyond its original intentions. What was meant solely to exchange static images has become a central element of contemporary digital culture. The format embodies specific creative practices, a recognizable aesthetic, an entire shared visual vocabulary.

This exceptional longevity in a perpetually changing environment underscores the importance of open and simple standards. GIF demonstrates that lasting innovations are not necessarily the most sophisticated, but those that leave users the freedom to appropriate and repurpose them according to their needs.

The format occupies a singular position at the crossroads of art, communication, and popular culture. Its influence extends far beyond the technical framework to permeate contemporary social and media practices. Thirty-five years after its creation, GIF continues to evolve, driven by the creative uses of a global community that finds in it a means of expression that is both accessible and sophisticated.

Perl

In 1987, Larry Wall worked at NASA's Jet Propulsion Laboratory. A linguist by training, he spent his days facing the UNIX screens of the era, juggling between `sed`, `awk`, and shell scripts to process data. One day, tired of this daily gymnastics between disparate tools, he decided to create his own language. Perl was born from this practical frustration: making UNIX programs communicate without going through countless contortions.

He invented nothing new but drew from what already existed, such as sed's regular expressions, `awk`'s processing power, and C's familiar syntax. But he assembled these elements with a fresh perspective and made Perl into that "glue" language that transformed the drudgery of data conversion into child's play. Wall had shown his talents as an ingenious tinkerer with `rn`, one of the first Usenet newsreaders, and patch, that utility which revolutionized the distribution of software fixes.

Perl's philosophy boils down to a few words: efficiency trumps ele-

gance. Where other languages impose their rigid vision, Perl adapts to the programmer. Its credo, "There's more than one way to do it", is a flexibility that sometimes bewilders but seduces a growing community of system administrators and tinkerers of all kinds.

In the early 1990s, while Netscape popularized graphical browsing, Perl established itself as the indispensable tool for processing HTML forms, well before PHP. Its ability to parse text, its decent performance on small scripts, its availability on all platforms made it the ideal companion for early web developers. Amazon, Yahoo!, Slashdot: all these sites shaping the internet relied on Perl.

The language's evolution followed a steady but sure pace. Version 2 improved regular expressions, version 3 handled binary data. Then came 1994 and version 5. Wall rethought the syntax, added proper object support, introduced modules. This reference version is the one developers would use for decades to come.

But Perl wasn't limited to its creator's whims. A community formed around the language, organized in an original way. The "porters" shared maintenance duties, coordinated by the perl5-porters mailing list. For each version, one of them inherited the "patch pumpkin", overseeing modifications and releases. This informal governance worked remarkably well.

Perl's real treasure is CPAN. This archive of reusable modules transformed the language into a digital Swiss Army knife. Need to process images? Connect to a database? Parse XML? There's definitely a CPAN module for that. This wealth more than compensates for the base language's flaws.

Wall, with his linguistic training, infused Perl with a dose of humanity. The language tolerates ambiguity, adapts to context, forgives approximations. A Perl program sometimes resembles poorly punctuated English prose. This expressiveness seduces, but it also repels: reading Perl written by another developer sometimes borders on cryptographic deciphering.

The question of licenses then stirred the free software community. Wall settled the matter by proposing a dual license: GPL on one side, Artistic License on the other, everyone chooses according to their needs. This legal flexibility heralded the open source approach that emerged in the late 1990s.

Companies smelled opportunity. O'Reilly & Associates published manuals that would become references. ActiveState adapted Perl for Windows, breaking the UNIX monopoly. These commercial initiatives didn't undermine the project's community spirit. On the contrary, they strengthened it by bringing resources and visibility.

The new millennium brought its share of competitors. Python seduced

with its clarity, Ruby with its object elegance. These languages drew inspiration from Perl while correcting its supposed flaws. The Perl community didn't sit idly by. In 2000, it launched the Perl 6 project, an ambitious overhaul of the language. Too ambitious perhaps. The project got bogged down, changed its name to become Raku, while Perl 5 continued its course, unperturbed.

Because Perl 5 had found its niche. Biologists use it to analyze DNA, administrators for their daily scripts, some websites continue to rely on it. The language has lost its star status, but it retains a loyal and competent community. Its conferences still gather hundreds of enthusiasts, its modules continue to enrich CPAN.

Perl showed that a free project could rival commercial solutions, that a dispersed community could maintain complex software, that an imperfect language could render immense services. In the servers running today's internet, somewhere, Perl code surely continues to process your data, discreetly, efficiently.

Self

David Ungar and Randall Smith launched Self in 1987, a project that would challenge conventional wisdom. This new language deliberately rejected everything that seemed established at the time. Classes were gone, variables forgotten. Only objects and their conversations through messages remained.

This radical approach wasn't gratuitous. It arose from frustration with the growing complexity of object-oriented systems. Smalltalk-80, though admired for its elegance, carried the burden of its meta-classes and their dizzying interconnections. Self proposed a different vision: why bother with molds when you can sculpt directly in the material?

In this streamlined universe, each object is a container of named "slots". These slots store data and behaviors indiscriminately. Inheritance relies on simple "parent" pointers that create delegation chains between objects. A system so simple it's bewildering.

Object creation follows natural logic. Instead of consulting a class blueprint, Self directly clones an existing prototype. This approach resolves at once the famous infinite regression of meta-classes that gave Smalltalk designers cold sweats. One object serves as a model for other objects, period.

Uniformity constitutes the other pillar of this architecture. Whether an object requests stored information or triggers a complex computation, the syntax is identical: sending a message. This consistency blurs the traditional distinction between properties and methods. The programmer navigates a homogeneous world where every interaction follows the same rules.

Self's early implementations surprised with their performance. By 1989, the language caught up with and then surpassed Smalltalk in execution speed. This success concealed a discreet but decisive technical innovation. The team invented "maps", internal structures that optimize how objects are organized in memory. These hidden maps are what would later be called "hidden classes".

Customized compilation pushed optimization even further. Instead of producing generic code, Self generates machine code tailored specifically for each type of recipient. This dynamic specialization transformed the language's performance.

Polymorphic inline caches emerged in 1991. These PICs memorize multiple method resolutions for each call site. Concretely, when a message can lead to different processing depending on context, the system keeps previously found solutions in reserve. Polymorphic calls, traditionally expensive, suddenly became affordable.

The 1996 version reached a new milestone with "type feedback" and adaptive optimization. The first mechanism integrates dynamic calls directly into compiled code, reducing method invocations by 70%. The second observes the program in action and reworks on the fly the most heavily used passages.

These technical advances caught Sun Microsystems' attention. In 1991, the company recruited Ungar and Smith to continue their research. But the story took an unexpected turn when two key project figures, Urs Hölzle and Lars Bak, left Sun in 1994 to create Animorphic Systems. Their mission: develop Strongtalk, a high-performance Smalltalk.

Sun's acquisition of Animorphic in 1997 sealed the fate of Self's innovations. They migrated to HotSpot, the virtual machine that would propel Java to the heights. Lars Bak repeated the feat in 2008 with V8, Google's JavaScript engine, which also drew from Self's heritage.

This technical lineage constitutes only part of the story. Self also influenced language design. JavaScript adopted prototypes as its primary inheritance mechanism. Lua, Io, and other modern languages drew inspiration from this direct approach where objects beget other objects without intermediaries.

Self's philosophy of concreteness is reflected in its development envi-

ronment. Objects appear there as tangible entities, visible and modifiable in real time. This innovative graphical interface transformed programming into a tangible activity. You touch, you move, you transform. Abstraction gives way to direct manipulation.

Simplicity, uniformity, and concreteness are the three principles guiding this vision. Reduce the number of concepts, treat all operations identically, use comprehensible metaphors. These simple ideas produce an environment of surprising richness.

Self never conquered the programming masses. Too radical for some, too experimental for others, the language remained confined to research laboratories. Yet its influence spans decades. The optimization techniques it introduced now allow dynamic languages to rival their compiled cousins. Python, Ruby, PHP benefit from its innovations.

This singular history shows how a quest for simplicity can trigger a cascade of lasting innovations. Self demonstrated that sometimes everything must be torn down to move forward. Its minimalist design anticipated certain current trends where conceptual streamlining takes precedence over feature accumulation.

Self's technical legacy powers the most performant virtual machines on the market. Its philosophy inspires designers seeking elegance. Forty years after its birth, Self remains a source of inspiration for those seeking to rethink object-oriented programming.

Tcl

John Ousterhout and his students at the University of California, Berkeley, faced a recurring problem. They were designing interactive tools for integrated circuit design-Magic, Crystal, and others-and each required its own command language. The team devoted little effort to these languages. The result was predictable: limited syntaxes, inconsistent with each other, that eventually became cumbersome.

The solution germinated during the fall of 1987, while Ousterhout spent his sabbatical at the DEC research laboratory. Why not create a single interpreter, designed from the start as a library that any application could integrate? This language would provide the basics-variables, loops, procedures-and each program would add its own specialized commands. This extensibility would become the hallmark of Tcl, the "Tool Command Language".

The initial development began in early 1988. A few months later, a graphical text editor was already using Tcl. Ousterhout didn't expect other developers to take an interest in it-a misjudgment that would amuse him later.

Meanwhile, graphical interfaces were drastically complicating programmers' lives. The 1980s saw increasingly sophisticated visual environments emerge, but their development required massive investments. Ousterhout worried: would small research teams still be able to innovate? The only way out was to assemble complex systems from reusable building blocks.

This reflection led to Tk, the graphical extension of Tcl. The idea was simple: build interfaces by combining ready-made components, with Tcl as the glue between elements. By the end of 1988, Tk development was beginning. It would take two years to obtain something truly usable.

In 1989, Tcl emerged from obscurity. Ousterhout presented it in January 1990 at the USENIX conference, where the reception was enthusiastic. He then decided to put the source code on Berkeley's FTP server. The Internet did the rest: word of mouth spread Tcl at a surprising speed.

Don Libes of the National Institute of Standards and Technology created the first Tcl application to achieve resounding success. Expect automated interactive UNIX programs-a blessing for system administrators who spent their days juggling between terminals and scripts. Expect's success brought Tcl to attention far beyond academic circles.

Growth became exponential in the early 1990s. From a handful of users in 1989, the number grew to several tens of thousands in 1993. Two reasons explain this enthusiasm. Tk represented the most direct way to create graphical interfaces under UNIX-five to ten times faster than with Motif. And Tcl met the needs of developers who wanted to make their applications scriptable without the burden of creating their own language.

In 1994, Ousterhout left Berkeley for Sun Microsystems. The company gave him the means to pursue his ambitions: a dedicated team that reached twelve people in three years. This period was prolific. Tcl broke free from UNIX to conquer Windows and Macintosh. The I/O system was redesigned, network sockets integrated. A compiler now transformed scripts into *bytecode*. Safe-Tcl provided a security model, while a plugin allowed scripts to be executed directly in web browsers.

1998 marked official recognition. Tcl received the ACM award for best system software, a distinction reserved for programs that have had a lasting impact on computing. The same year, the USENIX STUG award came to honor the tool's technical excellence.

At the end of 1997, Ousterhout had created Scriptics to devote himself entirely to Tcl. The company developed TclPro, a suite of commercial

tools, while continuing to distribute the core of Tcl for free. The April 1999 version 8.1 introduced Unicode, thread management, and a new regular expression engine.

The name Scriptics disappeared in May 2000, replaced by Ajuba Solutions to reflect the orientation toward XML. Interwoven acquired the company in October. To guarantee the independence of development, an autonomous core team-the Tcl Core Team-took over during the summer of 2000.

The history of Tcl testifies to the unpredictable trajectory of technical innovations. A modest university project can become a major tool if its design meets real needs and if a community embraces it. Tcl proved the importance of making applications scriptable and demonstrated the value of reusable components for graphical interfaces.

Acorn Archimedes

In the fall of 1987, Acorn unveiled the Archimedes, a computer that would shake up the world of personal computing. Behind this name borrowed from the famous Greek mathematician lay a machine equipped with an ARM processor that ushered in the era of consumer RISC computing.

The story began a few years earlier, when Acorn's engineers were searching for a 32-bit successor to the venerable 6502. After examining the processors available on the market, they concluded that none met their requirements. The decision was made: they would design their own chip. This technical audacity would give birth to the ARM, a processor of deceptive simplicity but formidable efficiency.

The ARM's RISC architecture broke away from the growing complexity of processors of the era. Its 44 instructions mostly executed in a single clock cycle. This economy of means produced spectacular results: at just 4 MHz, the ARM delivered approximately four MIPS, outperforming many faster but less optimized processors. The 27 32-bit registers completed this elegant architecture, designed to work with standard dynamic memory.

Acorn developed the Archimedes in two distinct ranges. The A300 series targeted the education market, the natural successor to the BBC Micro that already equipped many British schools. These machines prioritized simplicity: no integrated hard disk controller, but the essentials for learning and discovery. The A400 series addressed more demanding users with its hard disk controller and expanded capabilities.

The A305 opened the lineup with 512 KB of RAM, while the A310 doubled that with 1 MB. At the top, the A440 impressed with its 4 MB of memory and 20 MB hard drive, a configuration that would put many competing machines to shame. These figures reflected Acorn's ambition: to create a personal computer capable of rivaling professional workstations.

The Archimedes' genius lay in its integration. Four specialized circuits orchestrated the electronic ballet: the ARM for calculations, the VIDC for display, the MEMC for memory, and the IOC for input-output. This approach drastically reduced the component count while optimizing performance. The results were visible on screen: 256 simultaneous colors, resolutions up to 640x512 pixels in color or 1152x976 in monochrome, and an eight-channel stereo sound system that made the Archimedes a multimedia machine ahead of its time.

In 1989, Acorn renewed its strategy with the A3000, a compact machine that integrated the keyboard into the main unit. This format recalled the Amiga 500 or Atari 520ST, but the ARM's performance made all the difference. The A400/1 series corrected the early models' teething problems and improved memory management. The A540, launched the following year, crossed a new threshold with its faster ARM 3 and expanded capabilities up to 16 MB of RAM.

The operating system followed this ambitious hardware evolution. The Archimedes' first OS, named Arthur, inherited the BBC Micro's spartan approach. Its command-line interface put off novices but won over programmers with its power. Acorn understood that the future belonged to graphical interfaces and launched RISC OS in 1988.

RISC OS transformed the user experience. Its cooperative multitasking interface favored fluidity over absolute robustness. Applications civilly shared the processor, an approach that worked remarkably well thanks to the ARM's performance. The context menu activated by the middle mouse button became an ergonomic signature, later imitated by other systems.

RISC OS's modular architecture enabled remarkable extensibility. Modules, whether residing in ROM or loading into memory, enriched functionality without touching the kernel. This flexibility facilitated the integration of new peripherals and system evolution. The unified file system erased the differences between floppies, hard drives, CD-ROMs, and the Econet network, a modern approach that anticipated future needs.

BBC BASIC V, considerably enriched compared to its predecessors, took advantage of the 32-bit architecture for unprecedented performance. Developers discovered a responsive programming environment that transformed learning to code into a pleasure.

The Archimedes conquered British schools, perpetuating the BBC Mi-

cro's educational tradition. Its computing power also attracted scientists and professionals who found a credible alternative to UNIX workstations. Acorn exploited this versatility by extending the ARM architecture toward the professional world with the R140 and R260 machines.

The ARM, born in Acorn's laboratories to equip the Archimedes, would conquer the world. ARM Ltd., born from this adventure, has since become one of the semiconductor industry's major players. Today, ARM processors equip virtually all our smartphones and tablets, retrospectively validating the British engineers' bold choices.

The Archimedes' legacy exceeds its modest sales. This machine demonstrated that an original approach to design could produce exceptional results. Its RISC architecture durably influenced the industry, its specialized circuits anticipated modern integration, and its operating system prefigured contemporary graphical interfaces.

The Archimedes remains a model of technical coherence. Every element, processor, memory, graphics, sound, operating system, participated in a harmonious overall vision. This global approach, rare in the computer industry, produced a machine with remarkable performance and refined ergonomics.

Apple Macintosh II

March 1987: Apple upends its established practices with the Macintosh II. Three years after the first Mac, this machine breaks away from the conventions set by Steve Jobs. Gone is the compact, closed case; in its place stands an architecture open to expansion. This about-face addresses a necessity: IBM is gaining ground with its PCs, and Apple must attract professional users.

The machine's heart beats with a Motorola 68020, clocked at 16 MHz. Apple pairs it with the 68881 math coprocessor, a combination that quadruples performance compared to previous Macs equipped with the 68000. Base RAM reaches 8 megabytes, expandable to 20 MB with a specialized kit. For storage, there's an 800 KB floppy drive and, optionally, a 40 to 80 MB hard disk.

But the true revolution of the Macintosh II lies in its graphics capabilities. Color display makes its debut in the Mac lineup with a theoretical palette of 16.7 million shades. Only 256 display simultaneously, admittedly, but this already represents a considerable leap. More boldly, the

machine accepts multiple monitors. Users choose between the 12-inch monochrome screen and the 13-inch color model, the latter exploiting Sony's Trinitron technology and its reputation for excellence.

The exterior appearance breaks with Mac tradition. The horizontal case evokes IBM PCs rather than the vertical elegance of early Macintoshes. This form houses a motherboard equipped with six NuBus expansion slots. These slots accommodate graphics cards, SCSI controllers, or CP/M and Pascal emulators. Apple maintains its legendary simplicity: no jumpers to position, no DIP switches to set. Cards configure themselves automatically.

Connectivity doesn't skimp on possibilities. Two serial ports, two Apple Desktop Bus connectors for peripherals, a SCSI interface, and a stereo audio output equip the machine. Sound deserves particular attention thanks to its custom chip. Four stereo voices with 44.1 kHz sampling: remarkable quality for the era.

MultiFinder accompanies this hardware power. This operating system allows simultaneous execution of multiple applications. Printing while downloading, calculating in the background during writing-all these possibilities represented innovation in 1987. The graphical interface retains Mac clarity while exploiting color.

The price reflects the product's ambition: $5,498 with hard drive, equivalent to over $14,000 today. This pricing clearly targets professionals and affluent users. Results follow: desktop publishing, graphics, and engineering sectors readily adopt this machine.

Beyond sales figures, the Macintosh II establishes lasting standards. The importance of display quality, controlled modularity, configuration automation: these principles still guide contemporary computer design. Proof that a computer can combine power with ease of use, a philosophy that spans decades at Apple.

Production ends in January 1990, replaced by the more powerful Macintosh IIx and IIfx. The legacy endures in the professional workstations that follow. This attention to display and sound heralds the multimedia era of personal computing.

The Macintosh II crystallizes a particular moment in Apple's history. The company demonstrates its ability to adapt to professional market constraints without abandoning its convictions. This search for balance between commercial openness and distinct identity still characterizes Apple's development strategy.

Commodore Amiga 500

Jay Miner had only one idea in mind: to build his dream computer around the Motorola 68000 processor. This Atari engineer kept hitting walls with his management, who found his project too expensive. A breakup was inevitable. In 1982, Miner walked out and founded Hi-Toro, soon renamed Amiga. The bet? Design a machine with unprecedented graphics and sound performance, featuring real multitasking, without breaking the bank for buyers.

The Amiga team worked in a bohemian spirit that contrasted sharply with the rigidity of large firms. Engineers would gather armed with foam bats to "vote" against bad ideas during their wild brainstorming sessions. This relaxed atmosphere unleashed creativity and pushed everyone to rethink personal computing. Prototypes piled up, made of gigantic printed circuit boards simulating future custom components. Innovation was boiling, but money was desperately scarce.

1984 sounded the alarm: Amiga was on the brink of bankruptcy. Atari offered a $500,000 loan, but the trap lay in the conditions. If the sum wasn't repaid by the end of June, Atari would recover all the technology. A nightmare scenario for Miner and his team. At the last moment, Commodore emerged from the shadows with a $24 million buyout offer. The rescue came at a price: Amiga's independence evaporated, but the technology survived.

Commodore brutally accelerated development. The Amiga 1000 arrived in 1985, a technically dazzling machine but a commercial failure. Its price discouraged families, while its positioning confused professionals. Thomas Rattigan took the helm at Commodore in 1986 and got straight to the point: the range would split into two distinct branches. On one side, the Amiga 500 to appeal to the general public. On the other, the Amiga 2000 for demanding users.

The Amiga 500 was born in 1987 from this differentiation strategy. Engineers took the 1000 model's architecture and cut costs mercilessly. The keyboard integrated directly into the case, the power supply moved outside, direct TV output gave way to a separately sold adapter. But the technological core remained intact: three specialized processors named Agnes, Denise, and Paula orchestrated an unprecedented multimedia spectacle.

Agnes managed memory and visual effects, Denise handled display, Paula supervised sound and input-output. This electronic trinity delivered 4,096 simultaneous colors when competitors struggled to display a handful. Hardware sprites animated game characters with striking fluidity. Stereo

sound across four sampled channels made other machines' electronic beeps forgettable. The Motorola 68000 processor clocked at 7.14 MHz executed preemptive multitasking worthy of professional workstations.

AmigaOS completely rethought interaction between human and machine . The Workbench interface impressed with its modernity, while the Exec kernel juggled concurrent tasks effortlessly. RJ Mical created with the Intuition API a masterpiece of programming that radically simplified interface creation. The system handled multiple screens of different resolutions, a unique technical feat at the time.

Game developers rushed to this unprecedented power. With its twelve parallax scrolling planes gliding at different speeds, *Shadow of the Beast* was mesmerizing. *Defender of the Crown* dazzled with its digitized illustrations of stunning realism. Musicians invented the MOD format, transforming the Amiga into a home recording studio. *Deluxe Paint* opened the doors of digital graphic creation to amateur artists.

A passionate community crystallized around the machine. Specialized magazines like *Amiga World* testified to this wild enthusiasm. Programmers exchanged tips and tricks to extract every last drop of performance from the hardware. Some completely disabled the operating system for total control, using the graphics processor as a debugger by changing the background color at strategic points in the code.

Europe massively adopted the Amiga 500. In the United Kingdom, it was the absolute reference for family gaming. Its affordable price and multimedia capabilities attracted budding graphic designers and musicians. Andy Warhol seized this new electronic canvas to explore the possibilities of digital art.

Yet Commodore squandered this success through disastrous management. The company accumulated strategic errors in the face of rising IBM-compatible PCs. Thomas Rattigan's dismissal in 1987 deprived the company of any clear vision. The Amiga 600 that followed disappointed with its lack of ambition, merely a cosmetic update of an aging architecture.

The Amiga 500 remains in history as the symbol of technical innovation that took precedence over marketing considerations. Preemptive multitasking, hardware graphics acceleration, multi-channel sampled sound: concepts that wouldn't become mainstream until a decade later. Its legacy endures in the video game industry and multimedia creation tools.

The loyalty of its community defies time. Teams like the Iraqi developers of *Babylonian Twins* pulled out their old Amiga projects years later to adapt them for smartphones. This determination testifies to the mark left by a machine that pushed the boundaries of personal computing, to

the point of making certain professional computers obsolete.

Sharp X68000

In 1987, while the Japanese computer landscape teemed with diverse machines – MSX, NEC PC-88, PC-98, Fujitsu computers – Sharp took a different direction. The company decided to create an uncompromising computer, even if it meant selling it at a premium price.

The Sharp X68000 displayed its ambitions from first glance. Its dual-tower case, equipped with a retractable handle, evoked high-end Hi-Fi equipment more than a traditional computer. This futuristic aesthetic drew inspiration from the Sharp X1 Twin, which already combined an X1 computer and a PC Engine console. The industrial design reached a milestone here: the computer became an object of desire.

Under this distinctive hood lay precision machinery. The Motorola 68000 processor ran at 10 MHz, supported by 1 MB of RAM expandable to 12 MB and 1 MB of dedicated video memory. These specifications placed the machine in a class of its own. The X68000 displayed 65,536 simultaneous colors and handled up to 128 hardware sprites – figures that made video game developers' mouths water.

The price reflected these capabilities: 369,000 yen, $6,000 in today's dollars. Sharp wasn't targeting the general public but an audience of knowledgeable enthusiasts and professionals. This strategy materialized through the development of tools and libraries specifically designed for game creators. The gamble paid off: the X68000 hosted versions often superior to the arcade originals of *Final Fight*, *Ghouls'n Ghosts*, and *Street Fighter II*.

The Human68k operating system, designed by Hudson Soft, borrowed from MS-DOS while adding its own innovations. Early machines used the VS interface, later replaced by SX-Window on subsequent models. A third interface, Ko-Window, resembled Motif and was developed by third parties.

The X68000's story unfolded over the years through a succession of models. In 1988 came the ACE, followed the next year by the EXPERT which included 2 MB of RAM as standard. The PRO model adopted a conventional PC case to accommodate more expansion slots. 1990 marked the arrival of the SUPER versions, which replaced the SASI interface with SCSI. The XVI of 1991 pushed the processor to 16 MHz, while the

Compact XVI of 1992 offered a more compact format with 3.5-inch drives. The series peaked in 1993 with the X68030, powered by a 68030 processor at 25 MHz.

The richness of the X68000 ecosystem remains impressive. MIDI expansion cards allowed connection to synthesizers like the Roland MT-32. The TS-6BGA graphics card combined acceleration and PCM sound. Some accessories now reach heights in terms of rarity, such as the POLYPHON card which integrated FPU, MIDI, PCM, and 8 MB of RAM in a single module. For controls, joysticks included the Cyber Stick for simulations and the surprising XE-1 with its unconventional design.

The creativity of the homebrew community produced remarkable works. *Cho Ren Sha 68k*, developed by two people in 1995, remains a striking example. This shoot-'em-up displayed 512 simultaneous sprites through advanced multiplexing techniques – a technical feat all the more remarkable as it was programmed primarily in C rather than assembly. This achievement testifies to the quality of the development tools made available.

Yet the X68000 adventure ended in 1993. Sharp had not significantly evolved the video and audio hardware for six years. Facing the rise of PCs equipped with powerful graphics and sound cards, the machine lost its appeal despite its intrinsic qualities. A PowerPC version project was also abandoned. But an unexpected gesture changed everything. In 2000, Sharp, Hudson, and other involved companies released the Human68k operating system, BIOS, and various tools into the public domain. This exceptional decision keeps this system accessible through legal and faithful emulation of the machine.

The X68000 embodies a particular Japanese vision of 1980s personal computing. Where others favored compromise between price and performance, Sharp chose technical excellence and bold design. Its ability to reproduce the arcade experience, exceptional graphics performance, and advanced architecture make it a machine apart. However, power supply reliability issues and its prohibitive price limited its distribution.

OS/2

In 1987, IBM and Microsoft forged an unlikely alliance to create OS/2, the operating system designed to run on IBM's new PS/2 computers. This collaboration between two giants stemmed from a simple observation:

DOS was showing its limitations against Intel 80286 processors. They needed to do better, to be more modern. The idea looked attractive on paper, but it already concealed latent tensions between two diametrically opposed visions of computing's future.

OS/2 delivered what users of the time were missing: a proper graphical interface and features previously reserved for mainframe systems. It shared certain traits with Windows, UNIX, or Xenix, but its ambitions exceeded those of its competitors. This first iteration gave no hint of the resounding divorce that would shake the industry.

The two partners were looking in opposite directions. Microsoft was eyeing Windows, which it considered the future. IBM wanted to continue with OS/2. This growing opposition finally exploded in 1992, causing what observers called a "divorce". Microsoft took its marbles and part of the code, which it recycled into Windows NT and Windows 95. IBM found itself alone with its system.

To bounce back, IBM turned to Commodore Business Machines and drew inspiration from AmigaOS. OS/2 version 2.0, launched in March 1992, introduced the Workplace Shell, a transformation in the user interface. This innovation marked the entry of object-oriented programming into the mainstream world of operating systems. A first that didn't go unnoticed.

The Workplace Shell shook up established habits. Dragging and dropping icons into folders, accessing menus through right-clicks, triggering printing by simply moving an icon: so many gestures that changed human-machine interaction. The catch came from hardware limitations, as computers lacked RAM, forcing users into costly upgrades to fully exploit these innovations.

1994 saw the birth of OS/2 Warp, the version that would mark the system's peak. This release appealed through its ease of use and 32-bit architecture, while maintaining DOS compatibility. A significant technical asset against the Windows versions of the time, mired in their 16-bit DOS base. Yet despite these undeniable qualities, OS/2 couldn't break through in the consumer market.

The system found refuge in specialized niches, particularly bank ATMs where its stability and reliability worked wonders. Microsoft, meanwhile, deployed an aggressive commercial strategy to impose Windows, relegating OS/2 to these technical niches. A frustrating situation for IBM, which saw its system confined to specialized uses.

IBM finally threw in the towel in 2005, stopping OS/2 production, followed by support in 2006. Die-hard fans of the system demanded its release under a free license, but this approach ran up against Microsoft's

intellectual property rights on certain parts of the code.

The story of OS/2 doesn't end there. In 2001, Serenity Systems obtained a license from IBM and marketed the system under the name eComStation. Later, in 2015, Arca Noae picked up the torch with ArcaOS.

OS/2 introduced innovations that are part of our digital daily life. Its rigorous design and pursuit of stability established new benchmarks. Yet its commercial failure reminds us that technical qualities aren't enough: commercial strategies, industrial alliances, and the ability to build an ecosystem largely determine a technology's success.

Unicode

In 1968, ASCII became a standard in the United States. The principle was to assign a numerical value between 0 and 127 to each character. The letter "a" takes code 97, "Z" takes code 90. Simple, effective... but terribly limited. It was impossible to write "naïve" or "café" correctly with this system designed for English.

During the 1980s, personal computing on 8-bit machines took off. The possible values extended up to 255. Manufacturers and publishers rushed to fill these 128 additional positions with accented characters. Each came up with their own solution. Commodore placed French accented letters in its own way, IBM too with its code page 437. A file created on one machine was unreadable on another. What looked like French on an Amstrad CPC turned into hieroglyphics on a PC compatible.

Conventions gradually emerged. The International Organization for Standardization standardized certain encodings, others gained dominance through commercial use. But the fundamental problem remained: 255 characters was laughable compared to actual needs. It was impossible to integrate French accents and the Cyrillic alphabet simultaneously. Users juggled between encodings depending on the language of the moment. KOI8 for typing in Russian, Latin1 for French. Writing a text quoting Tolstoy in the original version was a technical feat.

In the late 1980s, a few visionaries embarked on an ambitious project: creating a system capable of representing all characters from all human languages. Unicode was born from this ambition, with the initial idea of moving to 16-bit characters. Instead of 255 positions, 65,536 slots became available. Enough room for Chinese, Arabic, Hebrew, Cyrillic, and Latin alphabets within the same system.

ISO worked in parallel on its 10646 standard. Two projects, one objective. Fortunately, the teams eventually merged their efforts with Unicode version 1.1. The 65,536 positions proved insufficient, and the modern specification extends to 1,114,111 possible characters (0x10FFFF in hexadecimal notation).

Unicode goes far beyond a simple character catalog. The standard defines complex properties and usage rules. The responsible consortium publishes a manual of over 1,000 pages. Added to this are 14 technical annexes, 7 standards, 6 technical reports, and 4 stabilized reports. Hyphenation, bidirectional rendering, vertical layout: everything is covered. The scope of this documentation reveals the true complexity of multilingual text processing.

But in practice, how do you store these characters in memory? The brute force solution consists of reserving 32 bits per character. But it quadruples disk space and bandwidth compared to ASCII. Not to mention portability problems between processors that organize bytes differently.

UTF-8 provides the elegant answer. This format uses a variable number of bytes depending on the character. A single byte for ASCII (values below 128), two bytes for values up to 2,047, three or four bytes beyond. ASCII compatibility preserved, no troublesome null bytes, possibility of resynchronization in case of corruption: the advantages pile up.

UTF-16 also exists but remains less popular. It encodes characters on two or four bytes with a special mechanism called "surrogate pairs" for characters outside the basic multilingual plane. A byte order marker indicates the organization of data in memory.

Standard updates sometimes create turbulence. Amendment 5 to ISO 10646 moved and extended the Korean Hangul block, invalidating existing data. This episode, dubbed "the Korean mess", pushed standardization committees to commit against this type of incompatible modification.

Unicode manages sophisticated mechanisms. Characters combine, like an accent modifying its base letter. The standard defines bidirectional rendering rules for mixing Arabic texts (right to left) and Western texts (left to right) in the same document.

The adoption of Unicode transformed computing. Operating systems, Internet protocols, file formats: everything shifted toward this unified representation. Software internationalization became greatly facilitated. Exchanging a multilingual document finally became natural.

The Unicode Consortium continues its enrichment work. New characters to cover historical scripts, support for emerging needs: the standard constantly evolves. The arrival of emojis illustrates this capacity to adapt

to new uses of digital communication. Each addition requires meticulous care to preserve overall coherence.

Unicode marks a breakthrough in the history of textual computing. This standard establishes a solid technical foundation for multilingual processing. Its universal adoption proves that it is possible to create complex international standards that effectively serve all users on the planet.

FrameMaker

In 1986, an astrophysics student at Columbia University dropped out to embark on the adventure of technical publishing. Charles Corfield had just developed a prototype called /etc/publisher and was convinced he could do better than what existed at the time. Dissatisfied with the available tools and eager to secure a stable financial future, he left university to devote himself entirely to programming. This decision marked the birth of Frame Technology, a company founded by four individuals with complementary profiles. Alongside Corfield and his expertise in mathematics and development, Steven Kirsch brought entrepreneurial experience forged at Mouse Systems. David Murray mastered the intricacies of user interfaces and publishing, while Vickie Blakeslee excelled in organization and operational management. This combination of diverse expertise created fertile ground for innovation.

The technical publishing market was then dominated by Interleaf, which ruled the roost on UNIX workstations. Its system, sold for around $30,000, required complex installation and imposed its own interface standards. Facing this monopoly, Frame Technology chose a radically different approach. FrameMaker was offered at $2,500 with a native interface to operating systems and remarkably simple installation.

This disruptive strategy was built on remarkable technical innovations. FrameMaker succeeded in combining the functionality of word processors and desktop publishing software. The program handled automatic numbering of chapters and sections, real-time cross-references, offered sophisticated table management and advanced multilingual support. Its ability to handle large documents spread across multiple files appealed to technical writers. The architecture based on the concept of frames offered unprecedented flexibility in integrating text and graphics.

Success came quickly. From beta version 0.6, marketed in 1986, the company sold several hundred licenses. The first customer, an engineering group at John Deere, paved the way for rapid adoption. U.S. federal

agencies were attracted by this approach not tied to specific hardware. Partnerships with UNIX workstation manufacturers like Sun Microsystems accelerated this expansion.

Initially developed for UNIX, FrameMaker was adapted for Macintosh and then Windows. This portability was based on an innovative software architecture, the Device Independent Maker, which separated the core code from interfaces specific to each system. The MIF (Maker Interchange Format) exchange format ensured interoperability between platforms, a growing concern in an increasingly heterogeneous computing world.

Frame Technology also innovated commercially. The shared licensing system allowed multiple users to share a limited number of simultaneous access points, significantly reducing costs for organizations. The company developed strategic partnerships, notably with Toshiba, which generated over $5 million in royalties. This flexible approach to the business model greatly contributed to the software's adoption.

The company's growth was spectacular. From 5 employees in 1986, Frame Technology grew to 297 people in 1991. Revenue followed a similar trajectory: $3.4 million in 1987, $41.7 million in 1991. The February 1992 IPO valued the company at $146 million. Adobe Systems acquired Frame Technology in 1995 for approximately $500 million, thus cementing the success of this entrepreneurial venture.

The corporate culture was distinguished by its openness and diversity. The teams included a remarkable proportion of women, minorities, and LGBT people, including in management positions. This diversity nurtured creativity and stimulated innovation. Project management was inspired by the approach described by Richard Feynman in *Surely You're Joking, Mr. Feynman!*, favoring open discussion followed by clear and decisive decisions.

FrameMaker's impact on the industry far exceeded its creators' expectations. The software democratized access to professional technical publishing, previously reserved for large organizations with substantial budgets. It established new standards for user interface and operating system integration. Its modular architecture and advanced structured document management had a lasting influence on the development of future publishing tools.

Three decades later, FrameMaker remains a reference tool in technical publishing, used by over 700,000 people in 30,000 companies. Adobe continues to evolve the software, adapting it to contemporary needs with support for formats like XML and DITA, as well as multichannel publishing to HTML5 and digital formats.

IBM AS/400

While enterprise computing was fragmenting among a multitude of incompatible systems, IBM was secretly preparing a machine that would shake up the established conventions. The AS/400 would be born in 1988 in the Rochester, Minnesota laboratories, far from the major research centers of the American East Coast. This particular gestation partly explains why this architecture developed such unique characteristics.

Frank Soltis's team had been working since 1978 on the System/38, a computer with pioneering concepts but modest commercial success. The Rochester engineers had chosen a radically different path from that explored by the creators of UNIX, VMS, or the future designers of Windows NT. Their approach remained so confidential that IBM did not share all the secrets of this architecture with its other divisions.

When the AS/400 was unveiled on June 21, 1988, the announcement surprised by its scope. Six processor models were offered simultaneously, accompanied by more than 1,000 ready-to-use applications. This immediate software availability constituted a record in the computer industry. The promised performance was impressive: memory was multiplied by 24, storage by 48, and computing power tenfold compared to previous generations.

The real innovation lay in the internal architecture. The AS/400 was based on a technology-independent machine interface called TIMI. Programs no longer communicated directly with the processor but with a virtual machine. This abstraction enabled a technical tour de force in 1995: IBM replaced the 48-bit CISC processors with 64-bit RISC PowerPC chips without any existing application requiring the slightest modification. All programs continued to function as if nothing had happened.

The system organized everything as objects. Files, programs, printers, users: each element had a precise description of its authorized uses and function. This object-oriented design, unusual for the time, strengthened security and simplified management. An administrator could manipulate any system component with the same standardized commands.

Integration constituted the other pillar of this architecture. Where systems laboriously assembled databases, security, communications, and backup tools from different vendors, the AS/400 natively incorporated all these functions into its operating system. DB2/400 managed data, security controlled every access, communication protocols were integrated. This unity drastically simplified daily administration.

Memory management revolutionized usual practices. The "single-level store" created a unique 64-bit address space, approximately 18 quintillion bytes. Programs and their data received permanent addresses in this gigantic space. No more juggling between main memory and secondary storage: the system automatically handled placing objects in the right location according to their usage.

The 1990s saw the AS/400 evolve to adapt to new needs. In 1994, the Advanced Series integrated Lotus Notes and Internet access. Portable models appeared, as well as versions intended for small organizations. Reliability improved by a factor of 20 between 1988 and 1992, while performance progressed by 30% each year.

Commercial success accompanied technical excellence. 250,000 systems from previous lines (System/34, /36, and /38) were already installed at launch. In 1992, IBM delivered the 200,000$^{\text{th}}$ AS/400 to Heineken brewery. These figures testified to massive adoption in companies worldwide.

The machine's identity evolved through IBM's acquisitions and marketing strategies. AS/400e, then eServer iSeries, eServer i5, System i5, System i, and finally IBM i in 2008. This last name emphasized the system's new capabilities, now able to simultaneously execute IBM i, AIX, and Linux on the same POWER hardware.

The development environment progressively modernized. The Integrated Language Environment replaced the old programming model by improving modular program performance. RPG and COBOL, the platform's historical languages, coexisted with C/400, Pascal, and Java. The command-line interface recognizable by its mnemonics like WRKOBJ, work with object, survived all changes while enriching itself with modern graphical interfaces.

This exceptional longevity reveals the relevance of a different vision of enterprise computing. While most manufacturers developed modular systems assembling heterogeneous components, IBM Rochester prioritized integration, consistency, and ease of use. The AS/400 proves that an alternative approach to dominant standards can survive and prosper for more than three decades.

Internet Relay Chat

In 1988, in the premises of the University of Oulu, a Finnish student named Jarkko Oikarinen was working on a problem that frustrated him:

how to enable multiple people to converse simultaneously on BBS systems, those electronic bulletin boards that preceded the Web? Existing tools like `talk` or `rmsg` under UNIX were limited to two-person conversations. Oikarinen then imagined something different—discussion rooms where anyone could enter and participate in the general conversation.

This idea gave birth to Internet Relay Chat, better known by the acronym IRC. For the first time, dozens of people could exchange messages in real time in a virtual space. Oikarinen called these spaces "channels", identified by the hash symbol that would later become famous on other platforms.

IRC's first months remained confidential. The network was confined to Scandinavia, constrained by the fledgling international connections of the era. The situation changed when Mike Jacobs, from MIT, managed to connect to Oikarinen's system. This transatlantic contact opened the floodgates: the code circulated, new servers were added, the network grew.

In January 1991, during the Gulf War, IRC experienced its first massive use during a global event. Users gathered to relay war information in real time, reaching over 300 simultaneous users for the first time. The logs of these exchanges are preserved in the ibiblio archives.

In August 1991, during the attempted coup against Mikhail Gorbachev, IRC played its part. During the media blackout imposed by the putschists (August 19-21), users in Moscow used IRC to transmit information in real time to the rest of the world, thus circumventing censorship. The logs of these exchanges are also preserved in the archives. Sources sometimes mention IRC use during the Russian constitutional crisis of September-October 1993, but this information remains to be confirmed by primary sources.

Success attracted ambitions and disagreements. Quarrels erupted over the management of IRC's single network. Who could create new servers? According to what criteria? Tensions mounted until the breaking point. Some users broke away to create Undernet, leaving Oikarinen's original server to form the basis of EFnet. These two rival networks continued to exist, joined by hundreds of others born from the same divisions.

For IRC fragmented. Each network developed its own rules, its own servers, its own code. Programmers experimented, modified, adapted. From this ferment emerged three major technical families: Undernet's P10 protocol, hybrid for EFnet, bahamut for Dalnet. The technical diversity reflected IRC's decentralized philosophy, where no one controlled the whole.

This decentralization came at a price. IRC networks regularly suffered from *netsplits*, those temporary disruptions that divided a network into

several pieces. When two servers lost contact, their users found themselves separated until the connection was reestablished. These incidents, initially perceived as annoying failures, gradually became an accepted characteristic of the system.

IRC's architecture remained strikingly simple. Messages traveled in plain text, without encryption or persistent storage. This technical lightness partly explained the protocol's robustness, which also worked on sluggish connections with servers of limited resources. Where other systems demanded powerful servers and fast connections, IRC made do with little.

Users compensated for the protocol's limitations through their creativity. They developed chat bots, conversation logging systems, sophisticated graphical interfaces. The software ecosystem around IRC flourished. Everyone programmed their own tools, shared their scripts, improved those of others.

This culture of tinkering marked IRC. The protocol influenced the platforms that would succeed it. The @ of channel operators, the # of room names—these conventions born in the 1980s would reappear much later on Twitter and other social networks. IRC invented a symbolic language that the mainstream Web would eventually adopt.

Designed for a few hundred users at most, IRC exceeded its theoretical limits. At the height of its success, the four largest networks brought together over 500,000 simultaneously connected people. The servers held firm, the technology adapted, communities prospered.

Far more than a simple discussion tool, entire communities developed with IRC, with their codes, their habits, their rituals. Free software developers found a place for technical exchange. Computing enthusiasts gathered there to debate the latest innovations. Friendships were formed, projects were launched, collaborations were organized.

IRC survives discreetly in the shadow of the Web social giants. Younger generations often ignore it, preferring more colorful interfaces and richer features. But technical communities remain faithful to it. On thousands of channels, conversations continue, projects are developed, knowledge is transmitted. IRC embodies a certain idea of the Internet: decentralized, open, controllable by its users.

SNMP

By the late 1980s, computer networks were experiencing unprecedented growth. This expansion revealed a thorny problem: each manufacturer was developing its own management tools, creating a real headache for administrators. IBM had its solutions, Cisco had its own, and none spoke the same language. Juggling between these multiple interfaces bordered on acrobatics, especially since some tools were limited to simple cryptic commands.

The Internet Architecture Board grasped the magnitude of the challenge. In 1988, the organization published recommendation RFC 1052, which outlined the contours of a unified standard for Internet network management. The mission was assigned to the IETF with a tight deadline: 90 days to design what would become SNMP. The constraints were clear: simplicity of implementation and inspiration from ISO's CMIP protocol.

Three RFC documents emerged simultaneously that year. The first introduced the Structure of Management Information, an abstract language based on ASN.1 for formally describing management data. The second defined the Management Information Base, that tree-structured database where all monitored objects would reside. The third detailed the operational mechanisms of the protocol itself.

The architecture relied on manager-agent logic of disarming simplicity. On one side, agents dispersed across each device tirelessly collected local information. On the other, a central manager queried these agents on demand or received their spontaneous alerts, called traps. This asymmetry worked remarkably well in early installations.

SNMPv1's rapid success revealed, however, its congenital weaknesses. Security was merely a facade: community strings traveled in clear text over the network, accessible to anyone intercepting packets. This vulnerability was not an oversight but a deliberate choice. The designers had bet on the benign nature of exchanged information and counted on closed networks where trust prevailed.

This naivety pushed the IETF to develop SNMPsec in the early 1990s. This extension promised authentication, encryption, and sophisticated access control. But its complexity frightened developers, and it was overshadowed by SNMPv2 in 1993.

The protocol's second version corrected irritants from the first draft. The MIB gained grouping capabilities, making the description of complete equipment more natural. The GetBulk operation allowed massive data

retrieval, saving precious network round trips. Notifications gained reliability through the Inform mechanism, which required acknowledgment.

Unfortunately, internal feuds within the development community shattered the standard's unity. SNMPv2 fragmented into incompatible variants: SNMPv2, SNMPv2c, SNMPv2u, SNMPv2*. This technical cacophony discouraged adoption and maintained SNMPv1 in a paradoxically dominant position.

The IETF learned from these failures in designing SNMPv3, completed in 2002. This third version preserved SNMPv2's gains while offering a modular security architecture. The User-based Security Model authorized different encryption algorithms like DES or AES. The View-based Access Control Model offered remarkable granularity in defining access permissions.

Meanwhile, specialized extensions enriched the ecosystem. In 1991, RMON arrived with advanced network monitoring functions adapted to Ethernet networks. RMON2 extended this monitoring six years later to application layers. SMON completed the arsenal in 1999 with tools dedicated to switches and virtual networks.

Yet, defying all logic, SNMPv1 maintains the majority of deployments today. Its rusticity explains this surprising longevity. More recent versions, technically superior, have never managed to convince practitioners en masse, attached as they are to the proven simplicity of the original version.

The SNMP protocol validated the pragmatic approach against sophisticated but heavy solutions. It established enduring architectural principles such as separation between data format and content, and the manager-agent model that became a universal reference.

SNMP also testifies to the extensibility of Internet standards. Initially limited to network equipment, it now monitors servers, UPS systems, air conditioning systems, and a multitude of other devices. This versatility has consolidated its position in the IT ecosystem.

The protocol's limits nevertheless surface with infrastructure evolution. SNMP collects elementary data but delegates their intelligent processing to the manager. This philosophy shows its weaknesses against contemporary networks, generators of torrential information flows. Technologies like NETCONF are emerging to address these new constraints, without challenging SNMP's omnipresence in existing infrastructure.

ZIP

In 1989, could Phil Katz and Gary Conway have imagined that their compression format would durably revolutionize computing? The history of ZIP begins in controversy. PKWARE, their company, had just been sued by Systems Enhancement Associates for allegedly copying the ARC archiving system. Rather than defending themselves, Katz and Conway decided to create something radically new.

The name ZIP reflects this ambition. Robert Mahoney, a friend of Katz, suggested this term to evoke the speed of their new solution, far superior to competing formats. On February 14, 1989, PKWARE and Infinity Design Concepts announced the release of their format into the public domain. This strategic decision, a true stroke of genius, instantly transformed ZIP into an open standard. The technical specification, documented in the `APPNOTE.TXT` file, was accessible to everyone. Developers seized this opportunity, multiplying compatible tools and libraries.

The early versions of the format reflected the hardware constraints of the era. PKZIP 2.04g could contain only 16,383 files per archive, with a maximum size of 2 GB. These limitations corresponded to the capabilities of available machines at the time. The rapid evolution of computer hardware nevertheless pushed designers to constantly rethink their format.

In 1993, version 2.0 appeared, introducing DEFLATE compression and traditional PKWARE encryption. This iteration laid the foundations for the modern architecture of ZIP files. Three years later, Deflate64 improved compression performance. But it was in 2001 that a major transformation occurred with version 4.5 and the introduction of the ZIP64 format. Gone was the 4 GB limit, finished was the restriction to 65,535 files per archive.

The year 2002 marked an opening toward new encryption methods: DES, Triple DES, RC2, and RC4 joined the available arsenal. In 2003, AES made its appearance. Version 6.3.0 of 2006 crossed another threshold by integrating Unicode for file names and new algorithms like LZMA and PPMd. Each evolution testified to a remarkable capacity for adaptation to users' changing needs.

The internal structure of the ZIP format reveals clever design. The central directory, placed at the end of the file, catalogs all elements of the archive. This organization allows adding or removing files without reprocessing the entire archive. In the era of floppy disks, when each write operation took considerable time, this characteristic represented a decisive advantage.

The format's success pushed operating system vendors to adopt it natively. Microsoft integrated ZIP support into Windows in 1998 via the

Windows Plus! pack for Windows 98, naming this functionality "compressed folders". Apple followed in 2003 with Mac OS X 10.3. Free software systems were quick to follow suit.

Since 2006, Microsoft has used the compressed ZIP format as the basis for its Office Open XML. The .docx, .xlsx, and .pptx files are nothing more than ZIP archives containing XML files and various resources. This repurposed use illustrates the versatility of a format that has become invisible infrastructure of modern computing.

Official recognition came in 2015 with the publication of ISO standard. This standardization, ISO/IEC 21320-1, defines a minimal compressed archive format, compatible with different standards such as OpenDocument, Office Open XML, and EPUB. The standard imposes restrictions, however: only DEFLATE compression is authorized, encryption is prohibited.

Security issues have long troubled the ZIP format. The original encryption system, ZipCrypto, proved vulnerable to known-plaintext attacks. These weaknesses motivated the introduction of more robust methods. A controversy erupted in 2003 when WinZip integrated its own AES encryption system while PKWARE maintained its competing specification. The agreement reached in 2004 between the two companies facilitated mutual support of their respective formats.

The legacy of ZIP can be read in the adoption of the DEFLATE algorithm by other technologies such as gzip and zlib. The format continues to evolve: version 6.3.8 of 2020 integrated modern compression methods such as Zstandard, MP3, and XZ. This capacity for evolution, made possible by a modular design accepting new fields, ensures the format's longevity.

Thirty-five years after its creation, ZIP remains ubiquitous. Contemporary implementations support ZIP64 and its theoretical archives of 16 exabytes. This exceptional longevity is explained by a rare combination of ease of use, backward compatibility, and technical adaptability. Phil Katz, who died in 2000, would not see his format cross into the new millennium and establish itself as one of the silent pillars of modern computing.

Intel 80486

The year 1989 witnessed the birth of the Intel 80486 processor, a chip that marked a breakthrough in the world of microprocessors. For the first time in the history of the x86 family, one million transistors were

etched onto a single component. This technical feat was accompanied by innovations that radically transformed the performance of personal computers.

The true breakthrough of the 80486 lay in its redesigned architecture. Intel directly integrated a floating-point calculation unit onto the chip, previously confined to a separate coprocessor, the 80387. This merger simplified motherboard design and considerably accelerated complex mathematical calculations. Added to this innovation was a unified 8 KB cache memory, positioned at the heart of the processor. Gone were the slow external cache accesses that penalized the 80386.

The introduction of an instruction pipeline gave the processor the ability to simultaneously process different instruction stages. Simple arithmetic operations now executed in a single clock cycle, halving the time required compared to the 80386. This new efficiency propelled performance well beyond the gains brought by simply increasing frequency.

The first versions of the 80486 ran between 16 and 33 MHz. In 1991, Intel attempted to push the frequency up to 50 MHz, but encountered heat dissipation problems. The solution involved reducing the etching fineness to 0.8 micrometers. Despite these adjustments, this fast model struggled to attract buyers, notably due to its limited compatibility with the local buses essential to graphics cards of the era.

The 80486 range diversified. The 486SX, a stripped-down version without a floating-point unit, targeted the consumer market concerned with savings. More cleverly, the 486DX2 inaugurated a technique that would become standard practice: doubling the internal frequency relative to the system bus. This increased performance without requiring a complete overhaul of existing motherboards. The DX4, despite its misleading name, didn't quadruple but tripled the bus frequency.

A legal element disrupted Intel's strategy. The company no longer had the right to trademark purely numerical designations beginning with "80". This constraint pushed Intel to rethink its product communication and heralded the era of brand names like Pentium.

The commercial success of the 80486 attracted competition. AMD, IBM, Texas Instruments, Cyrix, UMC, and STMicroelectronics developed their own versions. AMD distinguished itself by offering frequencies absent from Intel's catalog, such as the 40 MHz bus. The firm marketed original models: 486DX-40, 486DX/2-80, and 486DX/4-120. In 1995, AMD's Am5x86 was the fastest 486 ever designed, clocked at 133 MHz, while experimental versions reached 150 and 160 MHz.

Cyrix adopted a different strategy by developing its chips through complete reverse engineering, without relying on Intel's plans. The first

models, 486DLC and 486SLC, constituted hybrid solutions compatible with 386 sockets. Handicapped by their mere 1 KB cache, they struggled against Intel and AMD's 8 KB models. Later Cyrix versions, equipped with more generous cache, unfortunately arrived too late to upset market shares.

The golden age of MS-DOS games coincided with the 80486's peak. The DX2-66 MHz model was the reference for video game enthusiasts in the early 1990s. This dominance faltered with the emergence of real-time 3D. These new graphics intensely taxed the floating-point unit and demanded considerable memory bandwidth. Developers began optimizing their creations for the Pentium's P5 architecture, progressively condemning 486 processors.

In the personal computer world, the 80486 survived into the 2000s in budget configurations. The withdrawal of Windows 95 support and the growing demands of subsequent operating systems accelerated its obsolescence. Paradoxically, Intel maintained production until September 2007 to supply the embedded systems market.

This chip embodied a period when computing performance progressed at a breakneck pace, enabling new applications and accelerating the democratization of computing tools.

Intel i960

Intel delivered a major blow in September 1989 with the launch of the i960CA, its first 32-bit superscalar processor dedicated to embedded systems. This chip marked a breakthrough in the embedded computing world by executing instructions simultaneously on each clock cycle. The result came quickly with 66 MIPS, a figure that caused a sensation at the time.

The philosophy behind the i960 surprised with its boldness. Intel married the traditional RISC core with its 32 registers to an enriched instruction set that borrowed from the CISC world. This genre-mixing, far from being a shaky compromise, responded precisely to the particular constraints of embedded applications. Intel's engineers knew how to balance this hybridization to create a processor that draws the best from both worlds.

The family expanded, the i960CF succeeded the CA and doubled its performance thanks to a redesigned cache memory. Intel then declined

its creation according to a well-established commercial logic: the SA/SB for tight budgets, the KA/KB for the mid-range, and the hardened MC versions for military applications. This product line strategy testified to a mature industrial approach.

Under the hood, innovations abounded. The interrupt controller didn't just manage priorities, it did so autonomously. Local registers saved themselves without intervention during subroutine calls, eliminating a traditional source of latency. Instruction caches equipped all models, while certain versions benefited from data caches that accelerated repetitive processing.

Data transfers reached 160 MB per second on high-performance buses, a remarkable speed. This capability transformed information flow management in demanding applications. Object code compatibility between different models constituted a major asset: developers could migrate their applications without starting from scratch, a guarantee of sustainability that was appreciated.

Integration pushed innovation further. The VH model brought together on a single chip a 32-bit PCI v2.1 interface at 33 MHz, a memory controller, and various peripheral functionalities. This concentration reduced costs, footprint, and consumption, three critical parameters in embedded systems.

The i960's application domains drew an impressive map. Networks adopted it for their communication controllers, bridges, and routers. Medical imaging, particularly ultrasound, exploited its processing capabilities. Industrial automation, robotics, and computer vision found an excellent ally in this processor. The aerospace sector integrated it into its flight control equipment and satellite navigation systems.

Intel didn't limit itself to silicon. The Solutions960 program federated more than 200 tools developed by 70 partner companies. Optimized compilers, operating systems, debugging tools, evaluation boards: the ecosystem took shape around the processor. This software richness transformed i960 adoption into a smooth experience for developers.

The integrated DMA controller impressed with its performance: 59 MB/s in fly-by transfers, 32 MB/s in two-cycle transfers. The interrupt controller managed up to 248 external sources with 32 programmable priority levels. Lockable cache memory guaranteed optimal execution of critical algorithms, a valuable feature in real-time systems.

Specialization guided the development of different versions. Network models processed packets without saturation, while those dedicated to imaging adapted to different bus widths and data conventions. This targeted approach reinforced processor attractiveness according to their ap-

plication domain.

Superscalar execution became democratized, peripheral integration on the main chip became the norm, the importance of a complete development ecosystem remains relevant. The late 1990s saw the birth of new competing architectures that disrupted the landscape. The i960 nevertheless paved the way toward increasingly integrated and high-performance solutions.

World Wide Web

At the end of 1989, Tim Berners-Lee was working at CERN in Geneva. The British computer scientist faced the same problem daily: how to find information in what had become a Tower of Babel at the European nuclear research organization? Researchers were accumulating documents about their experiments, equipment, and discoveries, but everything remained scattered and inaccessible. Berners-Lee then conceived a system to connect this dispersed information.

For decades, visionaries had dreamed of machines to organize human knowledge. Paul Otlet had sketched his Mundaneum as early as 1934, a connected universal library. Vannevar Bush theorized the Memex in 1945 in "As We May Think", a machine capable of navigating between documents linked by associations of ideas. Ted Nelson coined the term "hypertext" for this non-linear navigation in 1965 with his Xanadu project, which never came to fruition despite decades of development. Apple brought part of these dreams to life in 1987 with HyperCard, the first commercial success of hypertext.

Berners-Lee took decisive action. Rather than designing a complex system, he focused on simplicity. In 1990, with Robert Cailliau's help, he laid the foundations of his architecture: the URL uniquely identifies each resource, the HTTP protocol organizes exchanges between machines, the HTML language structures documents and their links. Shortly after, he developed the first Web browser, which he called WorldWideWeb, before renaming it Nexus.

The first website went live on August 6, 1991 at the address info.cern.ch. Tim Berners-Lee explained what the World Wide Web was, how to create pages, and how to install a browser. The information spread through Usenet forums. The following year, Les Cernettes, a music group formed by CERN employees, saw their photo become the first non-scientific image published on the Web. A nod to the democratization to come.

The turning point came on April 30, 1993. On that day, CERN relinquished its copyright on Web technologies and placed them in the public domain. This decision, seemingly trivial in retrospect, changed everything. Anyone could now use, modify, and distribute the code without constraints. Universities seized the technology, and businesses would follow.

Marc Andreessen seized the opportunity. In 1994, his Mosaic browser revolutionized the user experience with its intuitive graphical interface. Gone were command lines and austere interfaces: the Web became accessible to the general public. That year also saw the birth of the World Wide Web Consortium under Berners-Lee's direction. This organization established the technical standards of the Web.

1995 marked the explosive entry of commerce onto the Web. Yahoo! launched its directory, meticulously classifying sites by categories. Jeff Bezos opened his Amazon bookstore from his Seattle garage. Microsoft integrated Internet Explorer into Windows 95, triggering the first browser war against Netscape. The Web was no longer just a research tool; it had become a marketplace.

Three years later, two Stanford students disrupted the game. Larry Page and Sergey Brin created Google with their PageRank algorithm, which ranked pages according to their popularity measured by incoming links. This mathematical approach to relevance radically transformed information retrieval. The Web became the universal gateway to digital knowledge.

Evolution continued in successive waves. The early Web, from 1989 to 2005, resembled an immense library. Sites remained static; users were content to read. HTML, HTTP, and basic protocols were sufficient to operate this still-simple universe.

Between 2004 and 2016, everything shifted toward interaction with what would later be called Web 2.0. Users no longer simply consulted; they created, shared, and commented. Facebook opened its doors in 2004, YouTube followed in 2005. JavaScript, XML, and Ajax enriched interfaces, making them dynamic and responsive.

From 2015 onward, a third generation emerged: the Semantic Web. The ambition changed scale: the goal was to make information understandable by machines themselves. RDF, RDFS, and OWL structured data so that computers could grasp the meaning, not just display it.

The HTTP protocol accompanied these transformations. Its version 1.1, standardized in 1997, introduced persistent connections that avoided renegotiating each exchange. HTTP/2 arrived in 2015 with stream multiplexing, which allowed multiple simultaneous requests on a single connec-

tion. HTTP/3, finalized in 2022, abandoned TCP in favor of QUIC based on UDP, gaining speed.

The appearance of Apple's AppStore in 2008 redefined Web access. Mobile applications complemented the traditional browser, sometimes replacing it. The Web moved out of computers and into our pockets, cars, and watches. This ubiquity was unthinkable in 1989.

The architecture conceived by Berners-Lee has withstood all these transformations. The separation between content and presentation, unique identifiers for each resource, the standardization of protocols: these technical choices enabled continuous expansion without major disruption. The client-server model and REST architecture provided a framework flexible enough to accommodate all uses.

Of course, today's Web bears little resemblance to the documentary system envisioned at CERN. It has transformed how we communicate, inform ourselves, conduct commerce, and work. While certain developments are concerning (concentration of services among a few players, exploitation of personal data, disinformation), Berners-Lee's vision of a universal and decentralized information space has largely been realized.

Microsoft Office

On August 1st, 1988, in the bustling atmosphere of Las Vegas, Bill Gates took the stage to announce an office productivity suite. At the time, no one truly grasped the magnitude of what was unfolding. Yet this announcement launched a venture that would revolutionize working habits worldwide.

Two years later, in 1990, the first version of Microsoft Office was released. It brought together three programs that can now be considered basic: Word for writing, Excel for calculations, PowerPoint for presentations. This combination was far from trivial. Microsoft had identified the need to have all the necessary tools for daily professional work at hand, without juggling between different publishers and their sometimes contradictory approaches.

The history of Word actually begins much earlier, in 1981, when Microsoft hired Charles Simonyi to develop a word processor. The first version came out in 1983, but it was confusing. WordPerfect reigned supreme and users didn't understand this very different interface. Microsoft didn't give up. A Macintosh version arrived in 1985, then in 1987 the RTF for-

mat appeared, finally simplifying document exchange between different systems.

Excel, for its part, established itself as the spreadsheet reference. The software could handle complex calculations, draw graphs, and analyze data through pivot tables. With the arrival of Visual Basic for Applications, users could program automations and free themselves from repetitive tasks.

PowerPoint has a unique genesis. Initially named Presenter by Forethought, Microsoft acquired it in 1987 for 14 million dollars. Robert Gaskins then suggested the new name. This software radically transformed the way presentations were conceived and delivered. In its first 1990 version, it was impossible to go back through slides, and customization options remained limited.

Microsoft released Windows 95 and Office 95 simultaneously... in 1995. This synchronization was no accident: the deep integration between the operating system and the office suite created a unified work environment that massively appealed to businesses. Sales skyrocketed and Microsoft established its dominance.

The suite gradually expanded. Access arrived to manage databases, Publisher to create marketing materials, OneNote for digital note-taking. Outlook completed the ensemble by managing emails and calendars. Each addition strengthened the ecosystem's appeal.

Microsoft had to adapt to technological developments. Versions were released for Windows and macOS. The company developed mobile applications like Office Lens and Office Remote to keep pace with new usage patterns. SharePoint and Skype for Business Server extended collaborative capabilities to enterprise servers.

The advent of cloud computing changed the game. Office Web Apps offered a lightweight version accessible through browsers. Office 365 then revolutionized the business model by switching to subscriptions. This transformation addressed contemporary users' expectations for mobility and real-time collaboration.

The suite's consistency strengthened with each version. The spell checker was shared, data flowed between applications, the interface became harmonized. Learning one application in the suite made mastering the others easier. The .docx format, introduced with Office 2007, improved document compatibility and security.

Education became a strategic conquest ground. Generations of pupils and students learned on Office, creating a user base familiar with these tools. Availability in 35 languages accelerated worldwide adoption. It

initially confused longtime users, but the 2007 ribbon interface modernized access to features.

Office shapes contemporary professional practices. Word establishes its standards for document formatting. Excel revolutionizes data analysis in businesses and financial management. PowerPoint redefines corporate communication codes, and these practices persist in the digital universe.

This story reflects the transformations of office work since the 1990s. From a collection of three programs, Office evolved into a complete ecosystem blending cloud services, mobile applications, and collaborative tools. This capacity for continuous adaptation explains its longevity in an otherwise volatile sector.

NeXTSTEP

When Steve Jobs left Apple in 1985, he took with him an obsession: to create the perfect computer. This obsession would give birth to one of the most fascinating adventures in computing history, that of NeXTSTEP. Here was an operating system that would break the mold.

In the computing world of the mid-1980s, graphical interfaces were still finding their way, systems crashed at the slightest misstep, and programming was an uphill battle. Jobs and his team at NeXT Computer decided to start from scratch. Their gamble was to marry UNIX's robustness with a new interface. They chose BSD as their base system, grafted the Mach kernel onto it, and there was their war machine ready to challenge the industry.

But NeXTSTEP's true boldness lay in its entirely object-oriented architecture. Where its competitors were tinkering with procedural code, NeXT bet everything on Objective-C. This decision seemed crazy at the time for anyone trying to understand this approach. Would developers follow? The team persisted and transformed every system component into an object: windows, buttons, files, processes. This architectural consistency transformed the user experience.

NeXTSTEP's interface was striking in its elegance. Display PostScript ensured surgical precision in graphics rendering. What you saw on screen matched exactly what the printer would produce. The famous Dock made its first appearance, that icon bar that simplified application access. A more discreet but equally remarkable innovation: the Workspace Manager offered column navigation that revolutionized file management.

NeXTMail pushed multimedia integration further. Send an image in an email? A sound file? A formatted document? Nothing could be simpler. This modern vision of messaging was years ahead of its time. Other systems struggled to display simple text properly when NeXT already offered the equivalent of the rich emails we know today.

Interface Builder revolutionized software development. Gone was the laborious coding of interfaces: programmers now assembled their windows visually, through drag-and-drop. This approach still inspires current tools. Services introduced unprecedented interoperability between applications. Retouching an image from within a word processor became child's play.

NeXTSTEP's file system anticipated our contemporary needs. Extensible metadata allowed enriching each file with contextual information. The Digital Librarian indexed all documents and offered sophisticated searches across the entire machine. Google Desktop and Spotlight would adopt this idea twenty years later.

Tim Berners-Lee chose a NeXT workstation to invent the Web at CERN. This choice was no accident, as NeXTSTEP's development tools made it possible to prototype complex applications. The first web browser and the first web server were born on this platform. Wall Street also took an interest in the system. Investment banks discovered its stability and adopted NeXT workstations for their mission-critical applications.

But technical excellence alone was not enough. The prohibitive price of NeXT machines hindered their adoption. Only wealthy professionals and a few universities could afford them. The general public remained out of reach. In 1994, NeXT abandoned hardware and refocused on software. OPENSTEP, the portable version of NeXTSTEP, ran on different architectures. This strategy broadened the audience but perhaps came too late.

The acquisition by Apple in 1996 revived everything. NeXTSTEP was reborn as Mac OS X. Jobs's system DNA shaped Apple's new operating system: object-oriented architecture, Cocoa frameworks, user interface philosophy. iOS directly inherited from this lineage. The iPhone and iPad owe much to NeXTSTEP's innovations.

This story proves that in computing, technological advancement always pays off in the end. The best ideas survive commercial failures and find their way to success.

VIII

1990

The Dawn of a Connected World

The destruction of the Berlin Wall in November 1989 disrupted the global balance. This fracture in history created fertile ground for digital technologies that were waiting to spread. The redrawn geopolitical landscape changed the stakes for computing and its dissemination.

When Soviet republics opened their doors to Western technologies, they unlocked unsuspected markets for computer firms. Eastern mathematicians, renowned for their excellence, shared their approaches with their Western colleagues. This intellectual cross-fertilization enriched computer science thinking on both sides.

The 1990s saw the global economy break free from traditional barriers. Large corporations had to adapt to this new reality: managing planetary operations required information systems capable of processing mountains of data and synchronizing activities across time zones.

Finance underwent a radical transformation. Transactions became dematerialized, algorithms invaded trading floors. Stock exchanges multiplied the pace and volume of exchanges, demanding ever faster machines and ever more robust networks.

The Internet left the closed circle of academics and military personnel to invade the daily lives of individuals. Internet service providers flour-

ished, first with modest modem connections that whistled and crackled, then with increasingly generous bandwidths. This opening gave birth to new services and modes of communication.

Citizens created their digital spaces of expression. Forums, electronic messaging, and personal websites wove invisible links between individuals. Virtual communities formed, bringing together enthusiasts across distances. This new socialization transformed user interfaces, which now had to appeal to an audience unfamiliar with computing intricacies.

Computers became fixtures on every professional desk. Companies massively invested in hardware and training. Word processing buried the typewriter. Spreadsheets conquered management. Slideshows revolutionized meetings by replacing transparencies and their overhead projectors. This rapid computerization created a voracious appetite for software adapted to professional realities.

In schools and universities, computing found its place in curricula. Computer labs became familiar places for students. This evolution responded to the demands of a job market seeking technical skills.

Video gaming transformed into a major industry during this period. Consoles pushed technical boundaries, with increasingly striking graphics. This sector energized research on graphics processors and brought computing into households through the gateway of entertainment.

Digital works flourished with the advent of multimedia. CD-ROMs made it possible to combine text, images, sounds, and videos in an accessible format. Digital encyclopedias revived knowledge. Entertainment applications breathed new life into personal computing.

Electronic components continued to shrink, giving birth to the first consumer laptops and personal digital assistants. Mobility became a requirement, forcing the adaptation of interfaces and operating systems.

Mobile telephony experienced explosive growth. Digital networks deployed across cities and countryside. Text messages invented a new language, concise and immediate. The future fusion between telephone and computer was becoming apparent.

Administrations modernized their IT infrastructures, developed the first online services for citizens, and digitized their dusty archives. These large-scale projects stimulated the market for professional solutions.

With the multiplication of connections, security became a central concern. The first viruses targeting personal computing sowed panic, followed by the development of antivirus software and firewalls. The protection of personal data emerged as an issue for the dawning 21st century.

Free software found its way with GNU and Linux. This alternative philosophy to proprietary software disrupted established models by proposing

a collaborative and open approach to software development.

The final years of the decade saw the flourishing of the first online stores and the swelling of the Internet bubble. Investors rushed toward anything related to digital technology, sometimes with more passionate than rational fervor. The bursting of this bubble awaited at the turn of the millennium.

This decade changed our relationship with computing. Computers settled into our daily lives, transformed our professions, modified our leisure activities. They were no longer these strange entities confined to air-conditioned laboratories. The household equipment rate soared, reaching 23% in France in 1999, compared to barely 8.5% in 1990.

The world discovered itself connected, without yet fully measuring the consequences of this transformation.

Computing in the 1990s cannot be reduced to a simple technical evolution; it constitutes an anthropological rupture whose magnitude we are only beginning to perceive.

Adobe Photoshop

In 1987, Thomas Knoll was working on his doctoral thesis in computer vision at the University of Michigan. On his Macintosh Plus, launched in January 1986, he tinkered with a small program he called Display, designed to show grayscale images on his black-and-white bitmap screen. Just an academic distraction, nothing more. Thomas had no idea he had just planted the first seed of what would become the world's reference tool for image processing.

John Knoll, Thomas's brother, discovered this program. John worked at Industrial Light and Magic, Lucasfilm's special effects division. At ILM, they were searching for solutions to process images digitally. John immediately saw the potential of his brother's small software and proposed they work together. This fraternal collaboration was about to change the game: Thomas adapted his code to handle color on the new Macintosh II, while John developed processing routines that foreshadowed the software's future filters.

In 1988, John sensed the commercial potential of their creation. Aware of the titanic work required to transform a personal program into a true commercial application. John's optimism prevailed, Thomas hesitated. The two brothers set to work, enriching the features, adding support for

various file formats, and developing sophisticated selection tools. After numerous naming attempts, the program received its definitive name: Photoshop.

Finding a publisher proved more complicated than expected. The Knoll brothers knocked on several doors. Barneyscan, a scanner manufacturer, agreed to market an initial version under the name Barneyscan XP. Approximately 200 copies of this version 0.87 accompanied their devices. The software circulated in the small world of imaging professionals. At Apple, engineers were enthusiastic and shared copies with their colleagues.

In September 1988, Adobe Systems entered the picture. Russell Brown, the company's art director, fell in love with the software. Negotiations resulted in a distribution license agreement rather than an outright purchase. A judicious decision for the Knolls, who would receive royalties on each sale. The formal signing occurred in April 1989.

In February 1990, Photoshop 1.0 hit the market. Despite some early bugs, the reception was enthusiastic. Against the competition, the software distinguished itself through its optimized code and intuitive interface. Russell Brown delivered spectacular demonstrations, revealing the program's creative possibilities and ease of use.

Version 2.0 in 1991 marked a decisive milestone. CMYK support revolutionized the printing industry by democratizing color separation techniques, previously reserved for professionals equipped with expensive hardware. Version 2.5 reached a new milestone by arriving on Windows, multiplying the potential audience.

1994 saw the birth of Photoshop 3.0, and with it, layers. This feature radically transformed digital image manipulation. Users could now overlay and modify elements independently. Contrary to rumors at the time, this innovation didn't copy competitor Live Picture but stemmed from Thomas Knoll's earlier research.

Version 4.0 in 1996 disrupted established habits. Andrei Herasimchuk redesigned the user interface to harmonize it with other Adobe products. These changes initially provoked ripples in the user community, but ultimately convinced through their coherence.

With version 5.0 in 1998, Mark Hamburg revolutionized the creative process by introducing the History palette. This innovation broke with the traditional linear logic of undoing, allowing non-sequential backtracking through modifications. Simultaneously, the direct integration of color management tools simplified the production of images intended for different media.

Photoshop's success exceeded all predictions. The software established itself in photography, advertising, publishing, and web design. Adobe saw

its revenue soar from $16 million in 1986 to over $1 billion in 1999. A flourishing ecosystem developed around the program: plugins, training courses, specialized publications.

By democratizing image manipulation, Photoshop nourished contemporary visual arts and transformed our relationship with photography. The verb "to photoshop" entered common vocabulary, proof of the software's cultural omnipresence.

This exceptional success resulted from Thomas Knoll's technical genius, John's commercial intuition, Adobe's marketing efficiency, all carried by the rise of personal computing and digital design. Thomas would never complete his thesis, but his "distraction" revolutionized the world of digital imaging. He continues to collaborate with Adobe as a consultant, while John pursues his career in special effects at Industrial Light & Magic.

IBM PS/1

The PS/1 story begins in 1987, when IBM watched with concern as PC clones gained ground. These compatible machines were eating into its market share, selling for far less than Big Blue's computers. Home users and small businesses turned away from IBM, more sensitive to price than brand prestige.

IBM was not ignoring this segment. Three years earlier, the company had taken a shot at it with the PCjr, a computer meant to appeal to American households. This machine offered interesting innovations: game cartridges, television output, and a wireless keyboard called the "Freeboard". But the PCjr crashed and burned in 1985, leaving IBM empty-handed in the consumer market.

Meanwhile, the company put all its eggs in the PS/2 line, aimed at businesses. These computers matched the competition in performance and offered extensive expansion possibilities, but their pricing made them unaffordable for home users. With over twenty models in the catalog, the lineup was incomprehensible for anyone simply looking for a family computer.

In 1990, IBM launched the Personal System/1. This time, the approach changed completely. The first models adopted a bold all-in-one design: the color monitor and system unit formed a compact assembly, connected by thick cables. The power supply was housed in the monitor,

creating total dependency between the two components. This configuration recalled Apple's compact Macintosh computers, except that Apple integrated everything into a single case.

Later versions would abandon this aesthetic choice to return to separate components, foreshadowing the future Aptiva line. The PS/1 ran PC-DOS, IBM's version of the MS-DOS system. To simplify life for novice users, IBM developed a graphical interface nicknamed "4-quad". Windows or OS/2 remained technically installable, but IBM discouraged their use due to the performance and stability problems this caused.

A technical peculiarity distinguished these early machines: unlike typical installations on hard disk or floppy disks, the operating system and interface resided directly in read-only memory. Tandy had experimented with this approach on certain models in its 1000 line. The advantage proved immediate: fast startup and increased reliability.

Technical evolution accelerated over the years. The launch models featured an Intel 80286 processor at 10 MHz, modest performance but sufficient for the era's office applications. Four years later, when IBM ceased production, the final PS/1 models integrated an Intel 486 DX2 running at 66 MHz, multiplying computing capacity.

This particular arrangement of the power supply in the monitor was not a first. The Amstrad PC1512 had adopted this solution, as had the Coleco Adam, which housed its power supply in the printer. But this configuration required using a proprietary connector between monitor and system unit, limiting replacement or upgrade possibilities.

IBM did not simply ship a machine. The company deployed an ambitious marketing strategy to recapture the consumer market. A telephone support service operated seven days a week, weekends included. PS/1 owners could interact with each other through a dedicated online service, the "Users' Club", prefiguring the user communities that would flourish with the Internet.

The PS/1 adventure ended in 1994. IBM halted production, marking a turning point in its strategy. The following year, the PS/VP and PS/2e variants also disappeared from the catalog. These decisions reflected IBM's gradual withdrawal from the consumer personal computer market, a sector where component standardization and price wars drastically reduced margins.

Haskell

About a dozen functional languages coexisted in the late 1980s, each with its peculiarities but all sharing similar semantic foundations. This dispersion frustrated researchers who struggled to gain adoption for their ideas beyond narrow circles. The notion of unifying these efforts emerged during a meeting in September 1987, at the Functional Programming Languages and Computer Architecture conference in Portland. The participants decided to create a committee to design a common language that would serve as a stable reference.

This collective approach seems surprising nowadays. It's hard to imagine that a committee could produce an elegant language, as design by consensus appears doomed to compromises and inconsistencies. Yet Haskell proves otherwise. The secret lies in the alignment of individual objectives and the importance placed on mathematical beauty.

John Backus had paved the way in 1978 with his Turing lecture. The creator of Fortran presented functional programming as a credible alternative to the von Neumann model. Coming from such a figure, this endorsement transformed the perception of the functional paradigm, which ceased to be seen as an academic curiosity.

Lazy evaluation fascinates Haskell's designers. This technique, discovered independently by several teams in the 1970s, radically changes the way programs are conceived. Dan Friedman and David Wise at Indiana, Peter Henderson and James H. Morris Jr. at Xerox PARC, David Turner at St Andrews and Kent: all explore this promising path. Turner demonstrates its elegance in SASL and KRC, using lazy lists to simulate complex behaviors with disarming simplicity.

April 1st, 1990 marks the publication of the first Haskell report. This date, chosen by chance, would fuel numerous jokes. Who would have thought that a language born on April Fools' Day would become so influential? Development continues for fifteen years, primarily via email. An era when technical exchanges still passed through thoughtful messages rather than incessant notifications.

Miranda influences Haskell's design. David Turner commercializes this non-strict functional language through his company Research Software Limited, with notable success: 250 universities and 50 companies adopt it. Haskell inherits many syntactic characteristics from Miranda but distinguishes itself through significant innovations.

The type class system constitutes the first of these innovations. Philip Wadler introduces it in February 1988, elegantly solving the problems of numeric operator overloading. More systematic than the solutions adopted

by Miranda or SML, this modular approach has since served as a model for other languages.

Purity represents Haskell's second pillar. As a pure functional language, it guarantees that a function always returns the same result for identical arguments, without any side effects. This constraint, directly related to lazy evaluation, initially complicates input-output handling. But this apparent difficulty leads to the invention of monadic input-output, recognized as a major contribution to computer science.

Monads transform a technical problem into a conceptual solution. They encapsulate side effects within a rigorous mathematical framework, preserving the language's purity while allowing interactions with the outside world. This theoretical elegance hides a practical complexity that sometimes discourages newcomers.

In 1999, "Haskell 98" stabilizes the language. The community was calling for this standardization, tired of the permanent evolutions that compromised code portability. The committee then dissolves, letting the language evolve freely. This two-speed approach works remarkably well: Haskell serves both as a laboratory for exploring advanced ideas and as a practical tool for real applications.

Successive versions progressively enrich the language. Version 1.1 in 1991, 1.2 in 1992, 1.3 in 1996, 1.4 in 1997: each iteration brings its share of improvements. This sustained pace testifies to the vitality of a community engaged in exploring new conceptual territories.

Haskell's innovations inspire numerous languages and frameworks. Advanced type systems, effect management via monads, lazy evaluation: all concepts that now radiate throughout the computing ecosystem. Imperative languages also integrate functional elements, testifying to the lasting influence of this approach.

The name "Haskell" honors Haskell B. Curry, a mathematician and logician whose work on combinatory calculus influences programming language theory. This choice is not trivial as it anchors the language in a rigorous mathematical tradition, reminding us that functional programming draws its roots from formal logic.

Haskell occupies a singular position. Neither a mainstream language nor an academic curiosity, it maintains a subtle balance between theoretical rigor and practical utility. Its influence is measured less by its direct adoption than by its capacity to evolve programming practices. In a world where elegance often gives way to efficiency, Haskell reminds us that other paths exist, more demanding but infinitely more satisfying.

HTML

The HTML language was born in 1991 at CERN, in Tim Berners-Lee's office. This British physicist was working on a concrete problem: how to share documents among researchers scattered across the world? His solution consisted of a few lines of code that would revolutionize our relationship with information. HyperText Markup Language, its full name, was inspired by an existing standard, the Standard Generalized Markup Language (SGML), standardized by ISO in 1986. Berners-Lee added one ingredient: the ability to create links between documents.

The idea represented an immense conceptual leap at the time. Imagine a text capable of taking you to another text with a simple click, then to a third, and so on, creating a web of interconnected information. This is exactly what was made possible with HTML's anchor tags. The language works through markers enclosed in angle brackets: `<p>` for paragraphs, `<h1>` for main headings. These tags serve a dual role: they structure content and determine its on-screen appearance.

As early as September 1991, a mailing list called www-talk brought together the project's first enthusiasts. Dave Raggett, an engineer at Hewlett-Packard, Marc Andreessen from the National Center for Supercomputing Applications, and many others contributed to enriching the language. Their exchanges gave birth to HTML+, an enhanced version of future developments.

This creative ferment concealed a trap. Lynx for text terminals, Mosaic for graphical interfaces, Arena as a European alternative, each browser developed its own extensions. "Proprietary tags" multiplied, threatening the universality dear to Berners-Lee. A document readable on Mosaic became unreadable on Lynx. The web risked fragmentation before truly existing.

To avoid this balkanization, Berners-Lee founded the World Wide Web Consortium, the famous W3C, in 1994. The initiative received financial support from CERN, the American DARPA, and the European Commission. Three laboratories hosted this new institution: MIT in the United States, INRIA in France, and Keio University in Japan. This geography reflected the desire to make the web a truly global project.

The W3C's first task was to standardize what existed. HTML 2.0 was released in November 1995, cataloging and harmonizing the tags used by different browsers. This version coincided with a pivotal moment: the price of personal computers dropped below $5,000, making them accessible

to the general public. In the United States, political initiatives encouraged internet development, sensing its economic potential.

But standardization had to contend with conflicting interests. Netscape and Microsoft, which dominated the browser market, sought to impose their own innovations to stand out from the competition. Their engineers invented spectacular tags: `<blink>` made text flash in Netscape, `<marquee>` made it scroll in Microsoft. These additions provoked the ire of W3C purists, who saw in them a dangerous confusion between document structure and visual presentation.

The debate went beyond pure technology. It opposed two visions of the web: on one side, content creators demanding more control over their pages' appearance; on the other, pioneers defending a strict separation between content and form. This tension still runs through discussions about web standards evolution.

The solution emerged in late 1996 with CSS (Cascading Style Sheets). This new standard delegated visual aspects to a specialized language, freeing HTML from this responsibility. The division of labor was formalized with HTML 4.0 in December 1997: HTML for structure, CSS for styling.

The W3C became professionalized. A "Process Working Group" including Netscape, HP, IBM, and Microsoft formalized standardization procedures. Their work resulted in 1999 in a "Process Document" that established the consortium's governance rules. These bureaucratic mechanisms may seem tedious, but they guarantee transparency and fairness in the standardization process.

HTML's history reveals the political stakes hidden behind apparent technical neutrality. The interoperability advocated by the W3C collides with browser vendors' commercial strategies. The ideal of a decentralized web confronts attempts by certain dominant players to maintain control. These questions still resonate in the face of the growing influence of Google, Apple, or Meta on standards evolution.

Yet HTML has kept its original promise. It has given birth to an open web where anyone can publish without prior authorization. This vision follows in the lineage of the cybernetic utopias of the 1960s, which dreamed of horizontal and collaborative digital spaces. While the contemporary web sometimes strays from these founding ideals, HTML remains a pillar of internet architecture alongside the HTTP protocol and the URL system.

The W3C continues its mission. The arrival of actors like the Motion Picture Association of America or scientific publishers broadens the spectrum of interests to reconcile. The entertainment industry demands anti-piracy measures, publishers want to protect their content, while digital rights defenders fear a restriction of the web's openness.

This permanent tension between innovation and preservation of founding principles constitutes HTML's DNA. Thirty years after its creation, this language continues to evolve while maintaining its original simplicity. A tag is a tag, a link is a link. In a digital world in perpetual flux, this stability is reassuring. It reminds us that the internet is not just a marketplace or a technological playground, but also a public space that must be preserved and passed on.

Gopher

At the University of Minnesota in 1990, Mark McCahill and Farhad Anklesaria led a team facing a practical problem, similar to Tim Berners-Lee's: how to organize information on campus? The official committee proposed a solution that the team deemed too cumbersome and unsuitable. They decided to create their own system.

In the spring of 1991, the first version of Gopher was born. The name refers to two things: the university's mascot (a gopher) and the idea of "going to fetch" information. The team designed a hierarchical architecture, similar to a file system. Users navigated through menus to access documents, just as they would in a traditional library where books are organized by subject.

The interface was deliberately simple. Text-based menus, designed to work even on slow connections. The creators integrated a full-text search engine using NeXT computers, making it possible to find the desired content. This combination of structured navigation and free search was immediately appealing.

The distribution strategy followed the free software model: the server source code circulated freely, but the team kept control of the clients to maintain system consistency. Administrators installed their servers easily and created links to other Gopher resources.

The success exceeded all expectations. By November 1994, more than 8,000 Gopher servers were running on the Internet. Librarians massively adopted the protocol, appreciating its logical organization and ease of use. The team developed extensions like Gopher+ to handle metadata and different document formats.

This story took place during a major transition period. Computing was gradually abandoning mainframes in favor of distributed systems. McCahill's team, coming from microcomputer support, advocated a decentral-

ized vision that opposed that of the central computer administrators. This tension reflected the paradigm shifts of the time.

Gopher democratized access to digital information before the arrival of the Web. "Gopherspace" was an information space that users navigated intuitively. The protocol elegantly solved the problem of organizing digital content by combining hierarchical navigation and textual search.

In 1993, the university made a strategic mistake. It attempted to monetize Gopher by imposing licensing fees on commercial users. This decision, driven by budget constraints, caused controversy. But the real challenge came from elsewhere: Tim Berners-Lee proposed to the team to merge Gopher and the World Wide Web. The apparent complexity of the Web made them hesitate.

This hesitation proved costly. Mosaic popularized the Web and gradually relegated Gopher to the background. The limited support for the protocol in this browser, combined with the growing richness of Web pages, diverted users toward the new standard. McCahill's team lost control of the user experience, a determining factor in a protocol's evolution.

The ideas developed for Gopher nonetheless survived. The hierarchical organization of content still influences many current systems. The constraints that guided its design—limited bandwidth, ease of use—have found renewed relevance with mobile applications and resource-constrained environments.

The experience taught several important lessons. First, the importance of controlling the user interface in the success of a network protocol. Second, the difficulties of monetizing Internet protocols. Finally, the impact of architectural choices on a system's longevity. With this project, the University of Minnesota developed recognized expertise in distributed information systems.

The questions raised by Gopher remain relevant: how to organize access to information? How to navigate digital spaces? How to balance simplicity and functionality? System designers continue to seek elegant answers to these issues.

PCMCIA

In 1985, the JEIDA (Japan Electronic Industries Development Association) identified a problem that would mark the computer industry. Portable computers were beginning to gain widespread adoption, but they

suffered from a frustrating limitation: it was impossible to easily add peripherals as one would on a desktop machine. Each manufacturer developed its own memory card formats, creating a technological cacophony where nothing was compatible.

Everything changed in San Jose, California in 1989. About twenty American companies gathered around a table to create PCMCIA: Personal Computer Memory Card International Association. Thise acronym concealed the ambition to ensure that all memory cards would speak the same language. Poqet Computer, a pioneer in this field, had envisioned a computer running solely on removable memory cards. But convincing software publishers to distribute their programs on these media proved to be an uphill battle without a unified standard.

The first PCMCIA standard was released in June 1990. The result of collaboration with JEIDA, it defined credit-card-sized cards equipped with a 68-pin connector. This version 1.0 was limited to governing memory cards, but the engineers on the technical committees quickly grasped the broader potential of this format. Why not add modems or network cards?

One year later, in September 1991, version 2.0 crossed the threshold. PCMCIA cards transformed portable computers into modular platforms serving as true extensions. The standard then featured three different thicknesses: Type I cards at 3.3 mm for memory, Type II at 5 mm for communication peripherals, and the substantial Type III at 10.5 mm intended for miniature hard drives.

The software architecture envisioned by PCMCIA's designers demonstrated remarkable vision. Three layers worked together harmoniously: Socket Services managed hardware at the lowest level, Card Services handled resource allocation and automatic configuration, while card-specific drivers topped the ensemble. Inserting and removing cards without powering down the system became possible.

PCMCIA transformed mobile computing. Manufacturers competed ingeniously to offer modems, network cards, wireless adapters, and multiple peripherals in this compact format. Portable computers finally gained the flexibility they had been lacking, becoming true electronic Swiss Army knives.

Technical evolutions followed in rapid succession according to market needs. 1995 marked the arrival of version 5.0 with the 32-bit CardBus, capable of transferring up to 132 MB/s. Low-power 3.3V cards appeared, power management became more refined, and multifunction cards emerged. The following year, Zoomed Video created a direct link between PCMCIA card and graphics controller, freeing the processor from video decoding tasks.

What impresses about PCMCIA is its early technical sophistication. Each card contained its own CIS information system, a sort of electronic identity card describing its capabilities to the operating system. This plug-and-play approach preceded the arrival of USB by several years. The computer automatically recognized the inserted card and configured the necessary resources, without user intervention.

The physical robustness of PCMCIA cards established lasting benchmarks. Specifications required 10,000 insertion cycles in office environments and 5,000 under harsh conditions. The electrical design protected components: power pins connected first and disconnected last, preventing destructive voltage surges.

At the turn of the 2000s, CardBay modernized the interface by introducing a high-speed serial link based on USB, while maintaining compatibility with existing systems. This approach illustrated the maturity of the standard, capable of evolving without breaking the ecosystem built around it.

The concepts of hot-swapping and automatic configuration from PCMCIA directly inspired USB and PCI Express. The cooperation between American and Japanese industrialists demonstrated that a worldwide standard could emerge from collaboration rather than commercial warfare.

Beyond portable computing, PCMCIA spread throughout consumer electronics. Digital cameras, television decoders, and automotive embedded systems adopted this format for its reliability and ease of use. This versatility testified to the quality of the original design.

However, PCMCIA did not escape the limitations of its era. Its simple bus, inherited from a direct memory interface, lacked elaborate synchronization signals. The 16-bit width restricted performance for demanding applications. These technical constraints prepared its gradual replacement by more recent standards like ExpressCard.

PCMCIA belongs to computer history, but it proved that an industrial consortium could create a durable standard, capable of stimulating innovation while solving complex interoperability problems.

MP3

In 1987, in the laboratories of the Fraunhofer Institute in Germany, researchers were working on Digital Audio Broadcasting, a project that seemed unremarkable. No one suspected they were laying the groundwork

for a revolution that would shake the global music industry. The CD had just taken its first steps five years earlier, and the question of digital audio compression was being raised.

Karlheinz Brandenburg, a graduate in electrical engineering and mathematics from the University of Erlangen, led this research with an obsession: how to drastically reduce the size of audio files without destroying their quality? The answer lay in the quirks of our auditory system. The human ear has flaws that Brandenburg intended to exploit. Some sounds mask others, certain frequencies disappear in the shadow of more dominant ones. These psychoacoustic phenomena would become the pillars of the future MP3.

In 1988, ISO created the Moving Pictures Experts Group. This international consortium had the mission of establishing audio and video compression standards. Fraunhofer's work finally found its institutional framework. Four years of development passed before the MPEG-1 Layer 3 standard officially came into being in 1992. This unwieldy name would soon transform into three letters that would change the world: MP3.

The magic works through a process of formidable complexity. The audio signal first undergoes meticulous temporal segmentation, then passes through a hybrid filter bank combining modified discrete cosine transform and polyphase filtering. A psychoacoustic model then calculates masking thresholds for each frequency band. Everything the ear does not naturally perceive simply vanishes from the final file. Only truly audible sound information remains, encoded with the necessary care.

This surgical approach produces spectacular results. A piece of music occupies about 10 megabytes per minute on a CD. With MP3 compression at 128 kilobits per second, that same piece fits into 1 megabyte. Ten times less space for quality that most listeners find acceptable. This technical feat would soon meet the Internet.

The mid-1990s saw an explosive convergence. The Internet expanded, connections improved, and MP3 found its way. In 1995, the format circulated on computer networks. WinAmp, a small audio playback software, accompanied this diffusion and transformed the personal computer into a hi-fi system. But this democratization came with an unexpected phenomenon: music piracy exploded.

Napster shook the record industry. This peer-to-peer exchange platform connected millions of Internet users eager to share their digitized music collection. Record labels discovered with horror that their business model was faltering. Lawsuits rained down, debates about copyright ignited. MP3 became the symbol of an unprecedented legal and economic battle.

The industry responded by creating legal offerings. Saehan Information Systems marketed the first portable MP3 player in 1998. Three years later, Apple made a big splash with the iPod and iTunes. Steve Jobs succeeded where many had failed: creating a viable ecosystem combining hardware and legal digital music distribution. The iPod was the essential technological object of the early 21st century.

The technical flexibility of MP3 partly explains its success. The format offers a range of bitrates from 32 to 320 kilobits per second. Each user adapts the size-quality tradeoff according to their needs and constraints. Three compression modes coexist: CBR maintains a constant bitrate, VBR varies the bitrate according to the complexity of the musical passage, ABR finds a compromise between the two approaches. This diversity satisfies both the demanding audiophile and the hurried user.

Under the hood, the technology impresses with its sophistication. The spectral analysis relies on the fast Fourier transform which decomposes the signal into its frequency components. A "bit reservoir" mechanism dynamically redistributes the bitrate between frames according to the needs of each audio segment. This intelligent management optimizes every available byte.

However, MP3 does not escape the compromises inherent in all compression. Artifacts can appear, like those pre-echoes that precede brutal instrumental attacks. High frequencies sometimes lose their clarity. These defects, imperceptible to many, annoy purists and motivate the development of more efficient successors like AAC or Opus.

The legal adventure of MP3 found its epilogue in 2017. The patents of the Fraunhofer Institute and Thompson Licensing expired that year. Their creators officially announced the end of support for the format, encouraging the adoption of more recent technologies. A page turned after twenty-five years of dominance.

MP3 has transformed our relationship with music. Gone are the CD shelves, finished are the walkmans and their fragile cassettes. Music is mobile, accessible everywhere, infinitely copyable. This new freedom upends listening habits but raises questions about the valuation of artistic work. How to fairly compensate creators in a world where their work becomes dematerialized?

Beyond music, MP3 perfectly illustrates how fundamental research can transform society. Work on psychoacoustics and signal processing revolutionized an entire industry. This technology also demonstrates the power of open standards and public research in technological innovation.

The legacy of MP3 is found in all modern audio codecs. Spotify, Deezer, Apple Music rely on the principles it established. Technically

surpassed, MP3 remains the reference format in the collective consciousness.

WAVE

In 1991, when Microsoft and IBM joined forces, they created far more than just an audio file format. WAVE was born from this collaboration, at a time when personal computers were truly discovering multimedia. Machines were beginning to produce something other than electronic beeps, and there was a need to store this new sonic richness somewhere.

The technical choice centered on RIFF, a file architecture designed as a large modular container. Each element found its place in chunks, small segments identified by four characters. An approach that transformed multimedia data management. Developers appreciated this flexibility of being able to add, remove, and reorganize information without breaking the overall structure.

The WAVE philosophy can be summed up in a few words: preserve the integrity of sound. No compression, no artifice, just the raw digital representation of the analog signal. The format consists of just two essential elements: the "fmt" segment describing the technical characteristics, and the "data" segment containing the audio samples. This apparent simplicity actually conceals remarkable robustness.

The 1990s saw WAVE gradually establish itself. Its natural compatibility with Windows gave it a definite advantage, but Macintosh and Linux systems adopted it as well. Audio professionals discovered a reliable tool, capable of reproducing exactly what they had recorded. Gone were the degradations caused by compression algorithms: what you heard at the output faithfully corresponded to what you captured at the input.

The European Broadcasting Union quickly grasped the format's potential. In 1997, it launched the Broadcast Wave Format, an extension that enriched WAVE with specialized metadata. Suddenly, an audio file could tell its story: who created it, when, under what conditions, with what equipment. This traceability quickly became indispensable in the audiovisual industry, where every second of programming must be documented and verifiable.

But WAVE also revealed its limitations. The 4 GB boundary, inherited from the original RIFF structure, proved problematic when studios moved toward high resolution. A concert recording at 96 kHz could easily exceed

this limit. The EBU responded in 2009 with RF64, an extension that pushed back these technical constraints. Archivists and producers could finally work without worrying about the duration of their recordings.

Data representation in WAVE follows precise logic. 8-bit samples use unsigned bytes, from 0 to 255, while 16-bit adopts two's complement, between -32,768 and 32,767. This technical consistency greatly facilitates programmers' work and guarantees faithful conversion to the analog signal.

The archival world massively adopted the format. The International Association of Sound and Audiovisual Archives made it an official recommendation. Their reasoning? A simple, documented format without compression guarantees readability fifty years from now. The British Library digitizes its audio collections in WAVE, as does the Library of Congress. These institutional choices definitively anchored the format in digital preservation culture.

In the 2000s, FLAC arrived and compressed losslessly, halving file sizes while preserving audio quality. Yet WAVE didn't falter. In recording studios, editing suites, and archives, it remained the reference. Its simplicity ultimately constituted its strength: no complex algorithm to decode, no patent to circumvent, just raw, readable data.

The format evolved discreetly. Metadata became enriched with INFO segments, XMP or ID3 tags. These additions gradually transformed WAVE from a simple audio container into a true digital document, capable of carrying rich contextual information. Modern professional workflows take advantage of this flexibility to automate content management.

Tools like JHOVE accompanied this evolution by offering validation and technical analysis. Verifying WAVE file compliance, extracting its metadata, detecting potential corruption—these are indispensable functions when managing thousands of hours of audio archives. The community thus developed an entire ecosystem around the format.

More than thirty years after its creation, WAVE has weathered the ages without showing its age. Its longevity stems from a subtle balance between technical simplicity and versatility of use. Where other formats disappear with their original technologies, WAVE persists because it meets the need to preserve sound in its purest form, without artifice or compromise.

Microsoft and IBM's 1991 audio format continues to capture our most precious sounds, proof that certain innovations transcend generations without losing their relevance.

Python

The story begins with a weekend in December 1989. Guido van Rossum, a researcher at the National Research Institute for Mathematics and Computer Science in the Netherlands, uses his holiday break to tinker with a new programming language. Nothing too ambitious at first: he wants to create something more practical than ABC, a language he had worked on, something that works better with UNIX without being chained to it.

The name he chooses reveals his taste for British humor. Python is a nod to *Monty Python's Flying Circus*, the troupe that left its mark on English comedy. This choice announces the mindset that will inhabit the developer community: a blend of technical seriousness and welcome irreverence in a sometimes too austere world.

He publishes the first version in February 1991 on alt.sources, the forum where programmers share their creations. Version 0.9.0 contains what will make Python unique, namely a syntax that prioritizes readability, an approach that contrasts with the languages of the time. Indentation, a syntactic element in its own right, is a direct inheritance from ABC. This idea grates at first. Programmers accustomed to braces and semicolons find it strange. Yet this constraint will prove liberating: it forces clean and readable code.

The philosophy crystallizes around a simple principle: "There should be one—and preferably only one—obvious way to do it." This maxim guides all design decisions. Where other languages multiply syntaxes for the same operation, Python chooses simplicity. The code is more predictable, therefore easier to maintain and understand.

Python 1.0 arrives in January 1994 with a solid functional arsenal. Whether lambda, map, filter, or reduce, these tools borrowed from functional programming enrich expressive possibilities. The language finds its footing, attracts its first followers. Version 1.5 in December 1997 consolidates these foundations and prepares the ground for what follows.

The year 2000 brings Python 2.0 and its novelties. The garbage collector automates memory management, freeing the programmer from this chore. List comprehensions introduce an elegant syntax for manipulating collections: `[x**2 for x in range(10)]` advantageously replaces several lines of traditional loops. These additions don't betray the original spirit but refine it.

Python's growing success reveals a different approach to development. The language encourages modularity without brutally imposing it. Modules combine naturally, code reuses without gymnastics. This fluidity appeals to programmers tired of other tools' complexity.

Then comes 2008 and Python 3.0, a version that disrupts habits. Van Rossum and his team choose to break compatibility to correct design flaws. This courageous decision temporarily fractures the community: old programs no longer work directly with the new version. But this boldness pays off in the long run. Python 3 establishes healthier, more consistent foundations.

Python's dynamic typing facilitates rapid prototype writing. No need to declare a variable's type before using it, the language handles it. This flexibility accelerates initial development while sometimes demanding more rigor in testing. Error handling, inspired by Modula-3, structures programs without weighing them down.

Object orientation integrates naturally into Python from its early versions. No conceptual shock, just a pragmatic approach that leaves the choice to the programmer, whether writing simple procedural code or organizing programs into sophisticated classes according to needs.

The richness of the standard library distinguishes Python from its competitors. "Batteries included", as they say in the community. Email, web, cryptography, compression: most common needs find an answer in the base distribution. This abundance avoids searching for third-party libraries for standard tasks.

Accessibility makes Python a language of choice for learning. Its syntax, close to natural language, demystifies programming. Books like *Python for Kids* popularize its pedagogical use. Universities adopt it massively to introduce their students.

PyPI, the Python package index, transforms code sharing into child's play. A simple `pip install` command suffices to add functionality to a project. This ease nourishes a dynamic ecosystem where everyone contributes and benefits from others' work.

PEPs (Python Enhancement Proposals) organize the language's evolution democratically. These improvement proposals structure debates, document technical choices. This open governance maintains coherence while welcoming external contributions.

Google adopts Python from its early days for parts of its search engine. This legitimization by a major technology company opens doors. Other organizations follow, attracted by the language's productivity and its ecosystem's quality.

The explosion of artificial intelligence propels Python to the forefront. TensorFlow, PyTorch, scikit-learn: specialized libraries multiply. Data scientists massively adopt the language for its analysis and visualization capabilities. NumPy transforms Python into a serious competitor to MATLAB for scientific computing.

Django revolutionizes web development in Python. This framework offers a complete approach: integrated ORM, automatic administration interface, sophisticated template system. Instagram, Pinterest, Mozilla use Django for their high-traffic sites, proving its industrial robustness.

Recent versions refine the user experience. Python 3.8 introduces the walrus operator (:=) which assigns a value while using it in an expression. Python 3.9 improves dictionary performance, the language's central data structure. These evolutions reflect growing technical maturity.

Portability remains a major asset: a Python program runs identically under Windows, macOS, or Linux. This universality, combined with free cost and the open source model, eliminates many barriers to adoption. Companies appreciate this freedom that protects them from technological lock-in.

Integration with other languages opens perspectives. Cython compiles Python to C for performance gains. Jython transpiles to Java, PyPy recompiles on the fly. These bridges allow optimizing critical parts without abandoning Python's productivity.

Python establishes itself in various domains: system automation, web development, data analysis, artificial intelligence, and scientific computing. This versatility reflects van Rossum's initial design choices. By prioritizing simplicity and readability, he created a tool that adapts to computing's changing needs. Thirty years after that December 1989 weekend, the language continues to evolve while keeping its soul: making programming accessible without sacrificing power.

Linux

UNIX was an operating system written in C that could adapt to different hardware. It was a way to work around the headache of computers that couldn't communicate between manufacturers, such as those from IBM and Burroughs, for example. Only the kernel required specific adjustments for each architecture. Clever, but UNIX licenses cost a fortune: resellers inflated prices up to ten times the initial rate.

Andrew Tanenbaum, an American professor working in the Netherlands, then took a different approach. He wanted to show his students how an operating system actually worked. His MINIX, designed for Intel 8086 processors that were flooding the market, didn't match the performance of commercial systems. Its strength lay elsewhere: Tanenbaum

published the 12,000 lines of code in his book "Operating Systems: Design and Implementation". This practice was already seen for BASIC programs on personal computers, like games, but not for entire operating systems. Curious programmers could finally look under the hood.

Linus Torvalds, a second-year computer science student in Helsinki, was among those passionate readers. This self-taught Finn was simultaneously discovering Richard Stallman's ideas. Stallman had started his career at MIT's AI lab, where he created the Emacs editor. In the early 1980s, he observed with bitterness how software companies poached the best programmers by imposing draconian confidentiality clauses. In his view, software should remain free, copyable, and modifiable by all.

The GNU system project (a recursive acronym for "GNU is Not UNIX") was born in 1983. Stallman began with the GCC compiler in 1984, a technical masterpiece that surpassed the achievements of entire teams of commercial developers. But the system kernel was still missing.

Torvalds, inspired by MINIX, took on the task, and his first Linux version (0.01) appeared in September 1991, followed by 0.02 in October. The reception was immediate: programmers from around the world downloaded the code, tried it, improved it, and sent their contributions back to Torvalds. An unprecedented collaborative dynamic was set in motion.

Tanenbaum was not kind to this newcomer. In 1991, he harshly criticized Linux's monolithic kernel architecture, calling it a major design flaw. "Linux is obsolete", he asserted. Torvalds took the criticism in stride and continued his work. History would prove him right.

The association with GNU programs transformed Linux into a complete system. The GPL license ensured that the source code would remain accessible, creating a virtuous circle: more contributors, more improvements, more users. Students and programmers flocked to this open playground.

Red Hat and Debian launched the first commercial distributions, assembling precompiled software to simplify installation. The KDE and GNOME graphical interfaces made the system accessible to non-experts. Linux was leaving the labs to conquer desktops.

Tux the penguin, the project's mascot, tells a delightful story. During a vacation in the southern hemisphere, Torvalds encountered a penguin that bit his hand. This amusing incident later inspired the choice of symbol. Humor was never far away in this adventure.

Linux developed its own philosophy. Each tool accomplishes a single task but does it perfectly. The interface treats files and input-output devices uniformly. Users combine these simple tools to create sophisticated functionality, customizing their environment to their liking.

The robustness is impressive: a failing program doesn't bring down the entire system. Each process's memory remains compartmentalized, preventing unauthorized interference. The development environment natively integrates a full suite of programming tools.

Linux represents the transformation of a student project into a global phenomenon. It runs web servers, supercomputers, and our Android mobile devices. An international community of developers proves it can create and maintain a sophisticated operating system, challenging established proprietary models. Linus's small kernel has grown to challenge industry giants, championing collaborative development and open source code.

Visual Basic

Visual Basic began with a simple idea: making Windows programming accessible to a broader audience than just C or assembly language experts. In May 1991, Microsoft made a major breakthrough by launching Visual Basic 1.0. For the first time, creating a graphical interface no longer required writing hundreds of lines of obscure code. All it took was dragging and dropping buttons, text boxes, and menus directly onto the screen.

This visual approach revolutionized established practices. Where previously one had to master the intricacies of the Windows API and manipulate complex structures, Visual Basic offered an intuitive logic: what you see on screen corresponds exactly to what the end user will see. A button placed in a corner of the window will appear in the same location in the finished application.

Microsoft didn't stop there. In September 1992, a DOS version was released, attempting to transpose this ease of use into the text environment. Extended ASCII characters simulated graphical elements as best they could, but the experience was rudimentary compared to the Windows version.

Version 2.0 of November 1992 corrected initial flaws and accelerated program execution. The development environment gained fluidity, which encouraged even more programmers to take the plunge. With version 3.0 in summer 1993, Microsoft introduced a major innovation: direct database access. Integration with Access simplified the creation of management applications, a rapidly expanding market.

Visual Basic 4.0 was released in 1995. Windows 95 democratized 32-bit systems, and Microsoft adapted its language accordingly. This version

navigated between two worlds: it generated 16-bit code for legacy systems while exploiting the capabilities of new 32-bit processors. Object-oriented programming made its appearance, timidly admittedly, but it opened new architectural perspectives.

Two years later, Visual Basic 5.0 definitively abandoned 16-bit support. This bold decision demonstrated Microsoft's confidence in the future of 32-bit computing. Native compilation replaced traditional BASIC interpretation, significantly accelerating program execution. Developers discovered the ability to create their own reusable controls, paving the way for an ecosystem of third-party components.

Visual Basic 6.0, released mid-1998, ventured into web territory. Internet Explorer evolved into a full-fledged development platform, enabling the creation of hybrid applications blending desktop and web. This version enjoyed phenomenal success and still appears to be used in some companies, despite official support ending.

The new millennium brought us Visual Basic .NET in 2002. Microsoft executed a complete break with the past. Managed code, automatic memory management, and integration with the .NET Framework transformed the nature of the language. This abrupt transition unsettled part of the community, accustomed to the simplicity of previous versions.

Visual Basic .NET 2003 consolidated this new direction. Support for the .NET Compact Framework opened the doors to mobile development, which was confidential at that time. Microsoft provided automatic migration tools, but the transformation was often laborious.

In 2005, Microsoft simplified the nomenclature and dropped the .NET suffix. Visual Basic 2005 recovered part of its lost identity. The "Edit and Continue" feature revolutionized debugging by allowing real-time code modifications. The "My" namespace facilitated access to system resources, recapturing the spirit of simplicity from the early versions.

Subsequent versions accompanied the evolution of the .NET Framework: 2008 exploited version 3.5, 2010 relied on 4.0, 2012 used 4.5. Each iteration brought its share of technical improvements, but the essence of the language remained faithful to its initial mission: democratizing software development.

Event-driven programming constitutes the soul of Visual Basic. Unlike traditional programs that follow a linear path from beginning to end, Visual Basic applications operate at the rhythm of user interactions. A button click triggers a procedure, a keystroke activates another function. This reactive approach transforms the logic of programming.

This event-driven philosophy reflects the evolution of personal computing. Computers in the 1990s became increasingly interactive, and users

expected responsive and intuitive interfaces. Visual Basic perfectly met this expectation by offering a development model aligned with this new reality.

The massive adoption of Visual Basic testifies to its relevance: by the late 1990s, nine out of ten Windows applications came from this environment. This dominance is explained by the explosion of the computer market, the standardization of Windows, and above all the accessibility of the language.

Visual Basic transformed the software development landscape by making the creation of Windows applications accessible to a much broader audience than just seasoned computer professionals. Accountants created their own management tools, engineers developed calculation applications, teachers programmed educational software. This democratization reshuffled the cards of the software industry.

DEC OSF/1

In 1992, Digital Equipment Corporation was going through a period of transformation. The company, once a giant of minicomputing, sought to establish itself in the enterprise server market against increasingly fierce competition. It then launched the development of a particular version of OSF/1, adapted to its new Alpha architecture and intended for AlphaServer servers.

The project was born out of necessity: to create a system capable of competing with existing UNIX solutions while fully exploiting the capabilities of Alpha processors. The engineering teams set ambitious goals: managing at least two processors, achieving 120 transactions per second on TPC-A benchmarks, and supporting at least 1 gigabyte of memory. These specifications represented considerable technical work.

The chosen architecture relied on a system bus with a maximum of eight nodes. Seven slots accommodated up to four processors each, with the eighth reserved for the system-to-PCI bridge. This modular design responded to a clear philosophy: offering scalability without compromising performance. The engineers integrated emerging technologies such as PCI and EISA buses, a risky bet that would prove successful for compatibility with the PC ecosystem.

Memory management posed particular constraints. The system had to juggle processors of different speeds while maintaining perfect cache

coherence. The adopted solution employed a 1 MB secondary cache using 15-nanosecond SRAM components. This choice resulted from a delicate compromise between performance and cost, with faster memories being prohibitively expensive for a commercial product.

Stream buffers constituted one of the system's most remarkable innovations. These devices monitored transaction addresses for reads and anticipated needs by preloading frequently used data, dropping access times from 9 to 7 cycles—a significant gain that impacted overall performance.

On the I/O side, DEC adopted a hybrid approach between PCI and EISA. This conservative decision was explained by economic constraints. Although older, EISA remained less expensive than PCI while offering acceptable performance for certain applications. The system thus supported up to 18 network ports.

The system boot process integrated carefully designed high-availability features. Hot-swap storage devices allowed on-the-fly replacement of failing components. Even more impressive, the system automatically managed corrupted flash ROMs without human intervention—a technical feat that would spare system administrators many nightmares.

Multi-OS compatibility represented another significant asset. Windows NT and OpenVMS coexisted with DEC OSF/1 on the same machine, thus meeting the heterogeneous needs of enterprises. This versatility, rare at the time, would attract many customers concerned with preserving their existing software investments.

Version 3.0, unveiled in 1994, marked the culmination of these efforts. Full support for symmetric multiprocessing placed the system among the major players. Developers had rethought synchronization mechanisms and kernel algorithms to take advantage of parallelism. The result exceeded expectations: measured performance far surpassed initial objectives.

The market reception was enthusiastic. The AlphaServer 2100 was equipped with DEC OSF/1 and set the benchmark for price-performance ratio. Independent tests confirmed the system's superiority, consolidating DEC's position in the enterprise server market.

This technical success had a lasting influence on the industry. The concepts developed for DEC OSF/1—advanced memory management, optimized multiprocessor support, modular architecture—spread to other systems. The balanced approach between innovation and pragmatism would serve as a model for subsequent developments.

In 1995, DEC renamed its system Digital UNIX, then Tru64 UNIX after the acquisition by Compaq in 1998. HP, which absorbed Compaq in 2002, continued development until 2012. This longevity testified to the solidity of the foundations laid by DEC's teams in the early 1990s. In

retrospect, DEC OSF/1 perfectly illustrates the challenges of this pivotal decade. Faced with the rise of RISC architectures and the explosion of distributed computing needs, DEC managed to create a system that lived up to its ambitions.

Blowfish

In 1993, as the Data Encryption Standard was showing its first signs of weakness with its 56-bit key becoming vulnerable to brute-force attacks, Bruce Schneier introduced Blowfish to the world—a symmetric encryption algorithm that would revolutionize the approach to computer security.

He had not chosen this path by chance. Proprietary solutions dominated the market, often accompanied by restrictive patents and prohibitive costs. The idea of a free, no-cost, and high-performance algorithm was taking shape in the mind of this American cryptographer who wanted to democratize access to strong encryption. Blowfish was born from this vision: to offer a credible alternative to established standards, free from legal or financial constraints.

The technical specifications of Blowfish broke with the practices of the time. Unlike DES with its fixed key, this new algorithm accepted variable-length keys, ranging from 32 to 448 bits. This flexibility addressed the diverse needs of users, whether individuals seeking basic protection or governments requiring maximum security. The heart of the algorithm relied on a 16-round Feistel network, a proven architecture enhanced by a remarkable innovation: key-dependent S-boxes.

These substitution boxes constituted Blowfish's signature. Rather than using fixed tables like its predecessors, the algorithm generated its own S-boxes from the provided key. Initialization began with the decimals of π, ensuring the absence of secret backdoors, then modified them through a complex process involving the key used. This approach made each implementation unique while preserving the desired cryptographic properties.

The official presentation took place in 1994 at the Fast Software Encryption workshop in Cambridge. The reception was mixed: while the technical innovation impressed, the cryptographic community remained cautious about this newcomer. The first security tests followed one another. Serge Vaudenay identified classes of weak keys in versions reduced to fewer than 14 rounds, but these vulnerabilities did not affect the standard version. Vincent Rijmen developed a second-order differential attack

against a 4-round variant, a brilliant technical demonstration that did not in any way challenge the robustness of the complete algorithm.

The commercial adoption of Blowfish exceeded its creator's expectations. Citi-Software Ltd's Access Manager integrated it into its password manager, leveraging its execution speed to secure sensitive data. The AEdit word processor made it its document encryption engine, while Markus Hahn's Blowfish Advanced CS used it for secure file deletion. This proliferation of implementations testified to the confidence the industry placed in this free algorithm.

Performance was one of Blowfish's major assets. On 32-bit processors, it required only 18 clock cycles per encrypted byte, a remarkable performance compared to DES's 45 cycles or IDEA's 50 cycles. This efficiency came from the judicious choice of simple operations: XOR, addition on 32-bit words, access to pre-calculated tables. The subkey arrays fit in the cache of processors like the 80486 or 68040, thus optimizing memory accesses.

However, this execution speed came with a downside: initialization. Subkey generation required the equivalent of 521 encryption iterations, a long process that penalized applications requiring frequent key changes. This characteristic naturally directed Blowfish toward uses where the key remained stable: file encryption, established secure communications, data storage.

The theoretical security of Blowfish impressed with its mathematical solidity. Schneier had calculated that a 197-bit key would resist even if all the energy produced by the Sun were converted into computational operations. These projections, though hypothetical, illustrated the considerable security margin offered by the algorithm. A 128-bit key required billions of powerful computers for millions of years to be broken by brute force.

The key-dependent S-boxes reinforced this security by complicating differential cryptanalysis. Each key generated its own substitution tables, making it difficult to establish exploitable patterns. The Feistel network ensured optimal diffusion: after a few rounds, each output bit depended on all input bits, a central property for resisting statistical attacks.

Despite its undeniable qualities, Blowfish revealed certain limitations over time. Its 64-bit block size, standard in the 1990s, seemed insufficient given the growing security requirements that now favored 128-bit blocks. The algorithm's memory footprint, with its multiple tables, posed problems on resource-constrained embedded systems such as smart cards.

These constraints did not prevent Blowfish from establishing itself durably. Thirty years after its creation, the algorithm still powers nu-

merous systems, testament to its solid and balanced design. Its successor Twofish, a finalist in the 1998 AES competition, adopted several of its innovations while correcting the identified limitations. But Blowfish retained its followers, attracted by its ease of integration and proven performance.

Blowfish's impact went far beyond the technical realm. It demonstrated that a free cryptographic algorithm could rival the most sophisticated commercial solutions. This success inspired developers and researchers, contributing to the rise of the free cryptography movement. OpenSSL and other projects like GnuPG relied on this demonstration to legitimize their collaborative approach.

The public analysis that Blowfish received also validated Kerckhoffs's principle, according to which the security of a cryptographic system must rely only on the secrecy of the key, not on that of the algorithm. This transparency, far from weakening security, strengthened it by subjecting the algorithm to the critical scrutiny of the international scientific community.

Blowfish remains an essential pedagogical reference for understanding the mechanisms of modern symmetric cryptography. Its history illustrates the transition from hardware encryption to software encryption, the democratization of cryptographic tools, and the growing importance of performance on consumer architectures. This successful synthesis between theoretical security and practical efficiency made it a model for many subsequent algorithms.

JPEG

In the 1980s, a paradoxical situation characterized the digital world: images proliferated but no standard existed to compress and exchange them. Telecommunications were taking their first steps toward multimedia while computing discovered the joys of color. Faced with this normative void, an international initiative emerged in 1986: the Joint Photographic Experts Group, better known by its acronym JPEG.

This collaboration brought together researchers from ISO and CCITT (Comité consultatif international télégraphique et téléphonique, the predecessor of ITU-T). Their mission seemed clear on paper: to invent a universal compression method. In reality, the challenge proved far more complex. How could visual quality and storage space economy be reconciled? How could a format be created that would work equally well for color fax transmission and for future applications not yet imagined?

The work began with an all-encompassing exploration phase. In June 1987, twelve different compression techniques competed during comparative tests. The atmosphere must have been electric: each team defended its method with the conviction that theirs would change the world. After this first confrontation, three approaches stood out and warranted further investigation.

The three finalists underwent a new battery of evaluations in January 1988 in Copenhagen. It was there that the technique which would dominate the following decades emerged: the Discrete Cosine Transform, better known by the acronym DCT. This mathematical approach, developed in the 1970s, finally found its ideal application field.

The JPEG principle resembles a sophisticated recipe. The image is first divided into small 8×8 pixel squares, as if cutting it into a mosaic. Each piece undergoes the famous DCT transformation which converts pixels into spatial frequency coefficients. This step reveals the ingenuity of the process: rather than storing each pixel individually, the variations and repetitions in the image are encoded.

Then comes quantization, the most delicate step in the process. Here, the algorithm makes choices: it eliminates information that the human eye perceives poorly or not at all. This ruthless selection constitutes the heart of lossy compression. Huffman encoding completes the chain by compacting the resulting data according to their frequency of appearance.

The results exceeded expectations. Compression ratios of 10:1, even 20:1, are commonplace without visible alteration of quality. For the first time, hundreds of images could be stored on the digital equivalent of a few floppy disks.

Standardization stretched over years, revealing the complexity of the exercise. The JPEG committee opted for a modular architecture that authorized different operating modes. The baseline mode constitutes the minimal foundation: every decoder worthy of the name must support it. This pragmatic approach avoided the pitfall of standards that are too rigid or too permissive.

The first drafts circulated as early as 1990 in laboratories and companies. ITU-T validated the specification in 1992 under reference T.81, followed by ISO in 1994 with standard ISO/IEC 10918-1. These official dates mask a more nuanced reality: the format became established in practice well before its administrative validation.

The committee made a decision that would prove decisive for the future: the baseline mode components remain royalty-free. Only certain optional features could be subject to RAND licenses. This policy contrasted with an era when patents often constituted barriers to adoption.

The Independent JPEG Group published a complete implementation under a free license in 1991. This source code, distributed under terms similar to BSD, became the technical reference. Developers worldwide could finally integrate JPEG into their applications without complex license negotiations.

The IJG didn't stop there. Its developers regularly refined their code, corrected bugs, and added advanced features from the standard. This collaborative approach heralded the open source spirit that would triumph a few years later.

The first applications remained faithful to the initial objectives: fax and videotex exploited the new format's capabilities. But it was the Web explosion in the mid-1990s that propelled JPEG toward planetary success. Web pages became enriched with photographic images without becoming impractical on 56K modems.

The arrival of digital cameras amplified the phenomenon. Kodak, Nikon, and Canon massively adopted the format for their devices. Memory cards, expensive and limited in capacity, directly benefited from JPEG compression. A camera from 2000 stored several hundred images on a 64 MB CompactFlash card.

At this time, the number of JPEG images in circulation was already counted in billions. Each day brought its share of new digital photographs. The format was invisible, so naturally did it establish itself. Users manipulated .jpg files without knowing of the committee's existence that made them possible.

Yet limitations appeared progressively. At high compression rates, characteristic artifacts manifested: blocking effects, oscillations around sharp edges, degradation of uniform areas. These defects were troublesome when compression was pushed to the extreme, while they remained imperceptible in moderate use.

HDR images revealed other shortcomings of the original format. Traditional JPEG is confined to 8 bits per color channel, a ceiling that seemed derisory compared to modern sensors capable of 12 or 14 bits.

These limitations motivated the development of JPEG 2000, a complete overhaul based on wavelets rather than DCT. This new version promised better quality at equal compression rates and natively supported HDR images. It also integrated advanced features like progressive resolution enhancement.

Paradoxically, JPEG 2000 never managed to dethrone its predecessor. Technological inertia played its role: why change a format that works? The quality gains, real but subtle, did not justify the migration cost for most applications.

Classic JPEG still dominates digital photography. Smartphones generate trillions of images in .jpg format each year. Instagram, Facebook, and Twitter rely heavily on this technology that is over thirty years old.

This lasting success is explained by favorable factors. Efficient compression is the main argument, but other elements count just as much. The standard's flexible architecture allowed its adaptation to very diverse uses. The open policy regarding patents facilitated industrial adoption. The free nature of IJG democratized access to the technology.

Thirty-five years after its creation, the JPEG committee continues its work. The modular approach, rigorous comparative testing, and balanced management of intellectual property constitute lessons for today's standardizers. It now explores new avenues such as artificial intelligence applied to compression. But its firstborn remains its finest legacy: a format that has become so universal that it is no longer noticed.

IBM ThinkPad

In IBM's corridors during the 1920s, a small black notebook accompanied every employee. On its cover, a simple word: "THINK". This pocket notepad embodied the company's philosophy and its constant call to reflection. Seventy years later, this spirit would give its name to one of the most influential laptop lines in computing history.

IBM's awakening to the realities of the portable market dates back to 1980. A team of internal analysts delivered a bold prediction: laptops would surpass desktop machines in sales volume before 1996. This revelation prompted John Akers, CEO at the time, to restructure the organization. In 1992, he created the Personal Computer Company, an independent division entrusted to Robert J. Corrigan to catch up with the accumulated lag behind Japanese and Californian manufacturers.

The first steps proved laborious, but the ThinkPad 700T was unveiled in April 1992. Equipped with an Intel 386SX/20 processor and a 10-inch monochrome STN screen, it struggled to convince. Its 4 or 8 MB of memory and two 10 MB storage units seemed paltry against market expectations. But this experience laid the groundwork for a more ambitious project.

At IBM's Yamato laboratory, near Tokyo, Arimasa Naitoh orchestrated a passionate team. This Japanese engineer enjoyed an exceptional reputation for solving the most intricate technical problems. Alongside

Ken Yonemochi and Koichi Higuchi, he leveraged the Japanese art of miniaturization to rethink every component. Meanwhile, Richard Sapper, a German designer consulting for IBM since 1980, made a radical decision. Gone was the ubiquitous beige in computing: the future ThinkPad would be black.

This unusual color reflected an aesthetic ambition that transcended conventions. Sapper drew inspiration from Japanese elegance and German functionality to design a different computer. The ThinkPad 700C revolutionized established norms.

While competitors still settled for monochrome displays, IBM bet on color. The 10.4-inch color TFT screen constituted its first major breakthrough. This technological risk-taking was accompanied by a more controversial innovation: the TrackPoint. Ted Selker, a scientist at IBM, designed this small red button located at the heart of the keyboard. Pressure-sensitive, it replaced the traditional mouse. The idea divided opinions. Some users fell in love with it immediately, others hated it. But this originality forged the ThinkPad identity.

Under the hood, the specifications impressed. The IBM 486 SLC processor clocked at 25 MHz, memory expandable up to 16 MB, and a 120 MB hard drive placed the machine at the peak of performance. The 3.8-hour battery life challenged the standards. Marketed at $4,350, the 700C displayed a price 15% lower than its direct rival, the Toshiba 4400. This pricing aggressiveness paid off: 100,000 orders poured in during the first two months.

Success encouraged IBM to multiply experiments. The 1993 ThinkPad 500 explored ultra-portability with its 7.24-inch screen. Exclusively in Japan, the 550BJ model integrated a Canon printer, anticipating total mobility needs. The 750P laid the groundwork for touch technology with its pressure-sensitive screen, a concept perfected by the 360P through a rotating mechanism. The 755CD marked another milestone by incorporating a CD-ROM drive, equipment that would remain standard for two decades.

But it was the ThinkPad 701C that crystallized IBM's genius. John Karidis, an engineer at Big Blue, imagined the unthinkable: a keyboard that unfolds when the screen opens. Two mechanical parts articulate to form a full keyboard in a compact chassis. This achievement earned the 701C, nicknamed "Butterfly", a place in the permanent collections of New York's Museum of Modern Art. Cruel paradox: despite this artistic consecration, the model quickly disappeared from the catalog, victim of its mechanical complexity.

The 1990s saw the range structure itself. The T series established

itself as the professional reference, combining robustness and performance. The X line targeted demanding nomads, prioritizing lightness without sacrificing reliability. The A models addressed the brute power needs of mobile workstations. This clear segmentation transformed the ThinkPad into a true cultural phenomenon. Owning a black ThinkPad became a social marker in corporations. In the first year alone, sales exceeded one billion dollars.

Irony struck in 2004. IBM, inventor of the PC and creator of the ThinkPad, recorded a one-billion-dollar loss in its personal computing division. Markets evolved too quickly, Asian competition pressed hard. In May 2005, Lenovo acquired IBM's PC business for $1.75 billion. The ThinkPad changed ownership but retained its soul.

The T60, the first Lenovo-branded ThinkPad, reassured the worried. Its visual identity endured, the TrackPoint persisted, the robustness remained. Intel dual-core processors made their appearance, the magnesium alloy cage reinforced the structure. Lenovo understood it had inherited precious heritage.

Transformations nevertheless began. The switch to 16:10 screens with the 2008 T400 sparked passionate debates on specialized forums. The adoption of the *chiclet* keyboard on the 2011 T430 divided the user community. But it was the removal of physical buttons from the touchpad on the T440 that provoked the most virulent uprising. Lenovo partially backtracked on the T450, proof that the ThinkPad legacy cannot be handled with impunity.

The X1 Carbon series, launched in 2012, embodied assumed modernity. Its carbon fiber chassis pushed the limits of lightness without compromising the brand's legendary solidity. These technical innovations were accompanied by faithfulness to the original aesthetic codes: matte black, the red TrackPoint, the sobriety of lines.

More than 150 million ThinkPads circulate worldwide. This exceptional longevity transcends trends. Peter Hortensius, an executive who navigated between IBM and Lenovo, offers an enlightening explanation: the ThinkPad embodies "less an object than a value system". Reliability, sobriety, efficiency: these principles span decades and resist the sirens of marketing.

OpenGL

In the computer graphics world of the 1990s, each manufacturer had its own programming interface, its own rules, its own vision of 3D rendering. Silicon Graphics used IrisGL, others had their in-house solutions. Developers who wanted to run their applications on multiple platforms had to rewrite their code as many times as there were different systems.

It was amid this chaos that Kurt Akeley and Mark Segal, both at SGI, had an idea that seemed almost utopian: create an open and universal standard for 3D graphics programming. They didn't just dream about it, they made it happen. In June 1992, OpenGL was born, the first truly cross-platform specification for 3D.

To prevent OpenGL from becoming another proprietary SGI product, the company immediately established a supervisory board: the Architecture Review Board. Digital Equipment, Evans & Sutherland, Intel, IBM, Hewlett Packard, Intergraph, Microsoft, Silicon Graphics, and Sun Microsystems all sat down at the table together. Imagine the scene: direct competitors agreeing to work together on a common project.

OpenGL's gamble lay in a few simple guidelines. First, standardize access to graphics hardware without meddling in what didn't concern it. Window management? Left to the operating system. User interface? Not our problem. This minimalist approach became its strength.

The creators made bold choices. They refused to include functions that couldn't be hardware-accelerated. Everything related to convenience or data management was delegated to higher-level libraries. GLU, the utility library that accompanied OpenGL, handled the matrices and NURBS surfaces that developers were asking for.

In January 1996, OpenGL 1.1 reached an important milestone. Extended textures, logical operations in RGB mode, and vertex arrays were game changers. These improvements addressed the pressing demands of an industry beginning to understand the potential of real-time 3D.

Adoption didn't happen overnight. First came scientific applications, followed by industry. CAD software understood OpenGL's value: front-face culling, dashed lines, stencil buffers—all these features were there, ready to use.

The video game world remained skeptical until John Carmack of id Software made a decision that would change everything. In 1996, he decided to use OpenGL for a new version of Quake, even though PCs equipped with affordable 3D acceleration could be counted on one hand. Yet this version of Quake demonstrated that an open standard could com-

pete with any proprietary solution. Graphics card manufacturers took notice.

OpenGL grew with its era. When GPUs became programmable, the 2004 version 2.0 introduced GLSL, the shader language that gave developers direct control over vertex and pixel processing. This major evolution reflected the emergence of a new generation of graphics processors.

The mobile device explosion gave birth to OpenGL ES. This streamlined version, tailored for embedded systems, became the graphics soul of smartphones and tablets. Its success strengthened the OpenGL ecosystem as a whole, creating a virtuous circle between desktop and mobile.

In 2006, the Khronos Group took over the reins of OpenGL. This non-profit organization ensured the standard's continuity while adapting it to new technologies. The governance change marked a new stage in OpenGL's maturity.

Thirty years after its birth, OpenGL's application domains are staggering. Medical imaging relies on it to visualize 3D scans. Flight simulators use it to recreate realistic environments. Hollywood employs it for special effects in film and television. Adobe integrated it into After Effects, Premiere Pro, and Photoshop. Major television networks like CBS, NBC, CNN, and BBC used OpenGL for their election coverage.

OpenGL's extension mechanism partly explains this exceptional longevity. Manufacturers can add new features without breaking existing functionality. These extensions, once widely adopted, eventually join the official standard. This flexibility allowed OpenGL to evolve without ever losing backward compatibility.

OpenGL remains the only truly universal graphics API. Windows, macOS, Linux, embedded systems—wherever there's a screen and a graphics processor, OpenGL can be deployed. WebGL brings its capabilities to web browsers, Vulkan takes over for high-performance applications. The legacy continues.

IBM Simon

When IBM unveiled the Simon Personal Communicator in 1992, no one fully grasped the significance of this invention. The idea seemed almost outlandish: merging a mobile phone with a personal digital assistant in a single device. IBM engineers envisioned a mobile terminal that would

do everything at once: make calls, organize appointments, send electronic messages. A vision that exceeded market expectations.

BellSouth Cellular launched it commercially in 1994. The price tag: $900, a hefty sum equivalent to approximately $1,960 today. The device impressed with its dimensions: 20.3 by 6.4 by 3.8 centimeters. Some compared it to a military *walkie-talkie*, so massive it appeared. Yet beneath this black chassis lay cutting-edge technology: an x86-compatible processor, integrated fax modem, Type II PCMCIA card slot, and most notably an 11.4 by 3.8 centimeter LCD touchscreen.

For this was the Simon's true revolution: its touchscreen. Gone was the traditional physical keyboard; users interacted directly with their fingers or a stylus. This approach upended established conventions. The graphical interface featured eleven pre-installed applications covering daily needs: calendar, address book, calculator, notepad, email, fax, and of course telephony. The Simon incorporated a web browser, a feature virtually nonexistent on mobile devices in 1994.

The custom-developed operating system offered an intuitive interface with clear icons. Switching from one function to another required just a simple tap on the screen. Users could scribble handwritten notes, sketch drawings, compose and send emails. The onboard memory stored all this data while the PCMCIA slot opened interesting expansion possibilities.

Alas, success did not follow. Barely 2,000 units rolled off the production lines. Most were returned to BellSouth and eventually destroyed. The reasons for this commercial failure? First, the price, prohibitive for many. Second, the battery life, disappointing. Third, the weight, a deal-breaker for daily use. Finally, the ergonomics, despite innovations, were perfectible.

But the real problem lay elsewhere. In 1994, networks weren't ready. Mobile infrastructure remained rudimentary, data rates paltry, coverage uneven. How could one fully exploit a web browser when the connection struggled to display a text page? Users sought above all mobility and simplicity. A phone that makes calls, period. The Simon arrived ten years too early.

This unfortunate experience nevertheless concealed a remarkable conceptual achievement. IBM anticipated uses that wouldn't become widespread until some fifteen years later. Apple's iPhone in 2007 adopted many of the Simon's ideas: touchscreen, intuitive graphical interface, convergence of functions. The first Android smartphones also followed this lineage.

The Simon's story illustrates a truth about technological innovation: technology alone isn't enough. A product can be technically perfect yet

commercially disastrous if the ecosystem isn't mature. Networks, usage patterns, and mindsets must evolve together. IBM learned this the hard way, but this lesson enriched understanding of mobile markets.

Thirty years later, the Simon still fascinates computing enthusiasts. As the first device to marry communication and personal organization, it laid the conceptual groundwork for today's smartphones. Its commercial failure doesn't erase its historical value. It bears witness to an era when manufacturers dared bold technological gambles without specific needs, even at the risk of suffering crushing setbacks.

IBM had envisioned pocket computing ahead of its time. Engineers dreamed of a single device capable of managing all its owner's digital needs. The Simon didn't win over consumers of its era, but this premature vision from 1994 now dominates the mobile market.

Adobe PDF

In 1990, John Warnock set an ambitious goal for Adobe Systems: to create a file format that would preserve the exact formatting of documents, no matter where they were viewed. This internal project, called "The Camelot Project", was born from a daily frustration: exchanging digital documents was an uphill battle. Fonts would disappear, layouts would fall apart, and systems refused to cooperate.

Warnock and his team had a major advantage: PostScript, the language Adobe had developed in the 1980s. But transforming this programming language into a practical file format required a different approach. Three years later, in 1993, PDF emerged in a structured binary form, abandoning PostScript's complexity in favor of enhanced performance for interactive display.

The beginnings were modest. Adobe launched two complementary products: Acrobat to create PDF files, and Acrobat Reader distributed free of charge to read them. This dual strategy—paid on one side, free on the other—eventually paid off, but adoption remained gradual during the early years.

The Internet changed everything. Around the mid-1990s, the Web exploded and PDF found its natural place: a document could now travel across the network while retaining its original appearance. Companies, government agencies, and then individuals adopted the format for their digital communications. PDF fulfilled a simple promise: what you see is

what your recipient will see.

Adobe didn't rest on its initial success. Successive versions enriched the format with features that transformed it from a simple container for static documents into a true interactive platform. Annotations made it possible to mark up texts, hyperlinks to navigate between sections, forms to collect data, and digital signatures to authenticate documents. Multimedia made its appearance, integrating videos and sounds into files.

The technology followed this functional evolution, as it should every time. PDF developed a sophisticated approach to compression, adapting its methods to the content: JPEG for photographs, CCITT Group 4 for black-and-white images, LZW for text and graphics. This technical flexibility maintained quality while controlling file size, a delicate but essential balance.

Font management illustrated the format's ingenuity well. Rather than being at the mercy of operating systems and their variable font collections, PDF offered two solutions: directly embedding fonts in the document or using intelligent substitution mechanisms. The document thus retained its appearance on a computer lacking the original fonts.

The year 2008 marked a turning point. Adobe entrusted the PDF specification to the International Organization for Standardization, transforming its proprietary format into the open standard ISO 32000-1:2008. This strategic decision confirmed the format's maturity and paved the way for even wider adoption.

Specialized versions emerged for particular uses. The PDF/X format conquered the graphics industry and professional printing, PDF/A addressed long-term archiving needs, PDF/E adapted to engineering documents, and PDF/UA integrated accessibility requirements. Each variation testified to the format's ability to adapt to the specific constraints of different sectors.

PDF 2.0 arrived in 2017 with the ISO 32000-2 standard, bringing improvements in digital signatures, metadata, and multimedia support. This version strengthened security and improved interoperability with contemporary technologies, proof that the format continued to evolve.

During these decades of evolution, the PDF ecosystem grew well beyond Adobe solutions. Open-source libraries, online services, and third-party applications offered their own approaches for creating, modifying, or viewing PDF files. This diversification consolidated the format's status as a universal standard.

PDF gradually integrated Unicode for international text, XMP metadata to describe documents, and digital rights management mechanisms

to secure content. It adapted to mobile screens and new viewing modes, proving its ability to adapt to technological transformations.

PDF dominates practically all professional and personal digital exchanges. Its success lies in a promise: to guarantee that documents retain their exact appearance, wherever they are viewed. This visual fidelity, combined with universal compatibility and a constant capacity for evolution, has made PDF much more than a file format—it's a true common language of digital communication.

Common Gateway Interface

The early Web resembled an immense frozen library. Consulting a page was like leafing through a book: one could read, but interaction was impossible. This situation changed radically a few years later when users began demanding more. They wanted to fill out forms, search for information, personalize their experience. Static pages were already reaching their limits.

Rob McCool worked at the National Center for Supercomputing Applications on the NCSA HTTPd web server. In 1993, he conceived an elegant solution to this problem: creating a bridge between the web server and external programs. This idea gave birth to the Common Gateway Interface, better known by the acronym CGI. The term *gateway* perfectly reflected its purpose: connecting two previously separate worlds.

The genius of CGI lay in its simplicity. When a visitor clicked a link or submitted a form, the server no longer simply returned an existing file. It launched a program on the server that generated a customized response, then transmitted this response to the browser. Suddenly, the Web came alive. Developers could write their scripts in the language of their choice, provided it respected a few basic rules.

These rules defined a standardized communication protocol. The server transmitted request information via environment variables—called meta-variables. The script retrieved this data, processed it, and generated a response conforming to the HTTP protocol. This universal approach worked with any programming language, from C to Python to Perl.

Perl indeed became CGI's champion. Its ability to manipulate text, its permissive syntax, and its availability on all platforms won over web developers. The first scripts processed HTML forms, queried databases, and assembled personalized pages. It was an era of experimentation.

These experiments gave rise to collections of shared scripts. Matt Wright, a high school student in Colorado, created "Matt's Script Archive" in 1995, which became an essential reference. His FormMail script allowed form content to be sent by email and was downloaded thousands of times. Problem: Wright was young, inexperienced, and his scripts contained glaring security flaws. The Perl community responded by creating "Not Matt's Scripts", offering more robust alternatives.

For security posed a constant challenge. Each CGI script represented a potential gateway into the system. Developers had to scrupulously validate input data, avoid buffer overflows, and guard against malicious code injection. Administrators generally confined these scripts to a special directory, the famous cgi-bin, to limit damage in case of problems.

CGI's architecture suffered from a congenital defect: each request triggered the launch of a new process. On a lightly trafficked server, this went unnoticed. But as soon as traffic increased, performance collapsed. The machine spent more time creating and destroying processes than actually handling the requests themselves.

Solutions emerged to work around this limitation. FastCGI kept processes alive between requests, eliminating the startup overhead. Mod_perl integrated the Perl interpreter directly into Apache, transforming scripts into persistent modules. These optimizations gave CGI a new lease on life.

Meanwhile, other approaches gained ground. PHP, created in 1994 by Rasmus Lerdorf, offered native integration with the web server. No more external processes needed: code executed directly within Apache. Microsoft developed ASP for its IIS servers, while Sun worked on JSP for the Java universe. Each sought their path toward dynamic web programming.

These new technologies did not erase CGI's legacy. They adopted its fundamental concepts: separation between server and application logic, information transmission via the environment, request-response model. CGI had blazed the trail, others were widening it.

Evolution accelerated at the turn of the 2000s. Modern web frameworks brought higher-level abstractions, sophisticated state management, service-oriented architectures. Ruby on Rails revolutionized web development in 2004, Django did the same for Python. These tools rendered CGI obsolete for most uses.

Yet CGI refused to disappear completely. Its official standardization in RFC 3875 in 2004 recognized its historical importance. This specification codified ten years of practical experience, detailing every technical aspect of the interface. Apache and other servers continued supporting it,

preserving compatibility with legacy applications.

This longevity is explained by the robustness of CGI's conceptual model. The idea of a standard interface between web server and external programs remains relevant today. Modern microservices architectures and REST APIs take up this principle on a larger scale, including the *stateless* philosophy.

FreeBSD

To understand FreeBSD's history, we must return to the 1970s, when the University of California at Berkeley received UNIX source code from Bell Labs. Researchers then began developing their own improvements to this system, creating what was called BSD, for Berkeley Software Distribution. This academic work resulted in several major versions that marked history, the last being 4.4BSD-Lite.

This 4.4BSD-Lite version concentrated a decade of technical innovations. The socket interface for network communications appeared there, accompanied by the reference implementation of the TCP/IP protocol. The fast file system improved performance, while NFS support and the `mmap` virtual memory model laid foundations that remain current. These technical elements were not mere academic exercises: they transformed how computers communicated and managed their resources.

In the early 1990s, the situation was particular. The computing community sought a free and above all functional operating system. A group of developers decided to create a complete distribution from 4.4BSD-Lite. But they rejected the usual model of single leadership. Seven people formed what they named the *Core Team*, a collective direction that would define the project's culture for decades to come.

The initial technical choice focused on the Intel 386 architecture. Developers aimed for stability and performance, particularly for networking and storage. The kernel integrated sophisticated mechanisms inherited from Mach technology for virtual memory management. The programming interface remained faithful to UNIX standards, facilitating the migration of existing applications.

Governance evolved in 2000. The nine Core Team members became elected every two years by active contributors. This democratic system allowed regular renewal and the emergence of new ideas. The project equipped itself with centralized development tools: version control, bug

tracking. These infrastructures made remote collaboration much more fluid than before.

The ports system represented a major innovation. This collection of software prepared for automatic installation radically simplified the addition of third-party applications. Users no longer had to worry about dependencies or compilation problems. The pkg system later further improved package management, making the user experience comparable to the most advanced Linux distributions.

Documentation received unusual attention. Developers established a team dedicated to writing and maintaining manuals. These contributors obtained the same rights as programmers, a rare recognition that underlined the importance placed on documentation. This approach bore fruit: FreeBSD became renowned for the quality of its technical documentation.

In 2000, the FreeBSD Foundation brought institutional support to the project. This non-profit organization provided technical infrastructure and funded specific developments. It now employs about twenty people who work on development, documentation, and promotion of the system.

The choice of the Berkeley license was strategic. Less restrictive than Linux's GPL, it allowed companies to integrate the code without obligation to publish their modifications. This flexibility attracted numerous companies in embedded systems and network devices. Apple found there the basis of Darwin, the core of iOS and macOS.

Exchanges with other BSD projects enriched development. NetBSD brought its multi-architecture expertise and automated testing methods, OpenBSD contributed significantly to security with the SSH program and encryption components for HTTPS. This collaboration between cousin projects created a coherent technical ecosystem.

Internet infrastructure adopted FreeBSD. Its stability and network performance made it the preferred choice of access providers in the 1990s. Yahoo! and Hotmail used it for their servers, demonstrating its capacity to handle massive loads. Web hosts followed, finding in this system a reliable foundation for their services.

Community culture played a determining role. Developers established respectful communication rules, avoiding the conflicts that paralyzed other projects. This inclusive approach encouraged participation from contributors worldwide. The community became diverse and productive, far from the ego wars that harmed certain competing projects.

Thirty years later, FreeBSD remains relevant. The system continues to integrate modern technologies without sacrificing its stability. Its code appears in varied commercial products, from routers to game consoles.

Network storage systems use it for its robustness. This discreet but omnipresent presence testifies to the quality of work accomplished.

This longevity is explained by mutually reinforcing factors. The technical foundation inherited from BSD was solid. Community organization proved effective. Complete documentation facilitated adoption. The permissive license attracted companies. But above all, constant attention to technical quality and balanced governance allowed weathering the storms of the computing world without losing course.

Intel Pentium

In 1993, Intel abandoned its numerical nomenclature. The processor that was supposed to be called 586 ultimately took the name Pentium. This break was far from trivial, as it reflected a defensive strategy against AMD, which had marketed its Am486 by exploiting the similarity in naming. Intel attempted to register "586" or "i586" as trademarks, but was denied. A simple sequence of numbers lacked distinctive character in the eyes of the competent authorities.

The first Pentium was based on the P5 architecture. This superscalar processor operated between 60 and 66 MHz, had a 16 KB L1 cache, and used a system bus clocked at the same frequency range. Its manufacture using 800-nanometer technology represented a technical feat for the time. Two years later, the Pentium MMX integrated new instructions dedicated to multimedia processing. Intel developed the P6 architecture in parallel, marketed under the name Pentium Pro in 1995. This version introduced out-of-order execution and incorporated a level-2 cache in a multi-chip package.

The family expanded in 1997 with the Pentium II, which combined the advances of the Pentium Pro and MMX instructions. This model adopted a new physical format, the SECC (Single Edge Contact Cartridge) cartridge, which simplified assembly and testing. The Pentium III arrived in 1999 with the SSE (Streaming SIMD Extensions) instruction set and 128-bit registers enabling simultaneous processing of four floating-point numbers.

Intel switched to the NetBurst architecture with the Pentium 4 in 2000. This new design favored high clock frequencies through an extended pipeline. Mobile versions emerged for laptops, with optimizations for power consumption. In 2005, the Pentium D ushered in the multi-core era by integrating two Pentium 4 processors in a single package.

Intel modified its strategy in 2006. The Pentium brand now positioned itself between entry-level Celerons and the new high-end Core series. Recent Pentiums use the same chips as Core processors, but limited: reduced frequencies, partially disabled L3 cache, advanced technologies removed. The processors are less expensive while maintaining compatibility with the x86 architecture.

Technical developments necessitated hardware adaptations. Early models inserted into Socket supports, then Slot formats appeared to accommodate SECC cartridges. The increase in required connections led to the development of LGA 775, where pins are located on the socket rather than on the processor.

Miniaturization progressed, with manufacturing process technology advancing from 800 nanometers in 1993 to 32 nanometers for recent versions. This evolution enabled an increase in the number of transistors, improved performance, and reduced power consumption. In 2011, the Sandy Bridge architecture integrated graphics capabilities directly into the processor. This integration addressed the growing needs of common applications for graphics rendering.

Pentium processors contributed to democratizing personal computing. Their x86 architecture established itself as a de facto standard and created a vast software ecosystem. Backward compatibility maintained across generations allowed users to keep their applications when upgrading their hardware, even though we know in hindsight that this has its limits. The Pentium's influence extends beyond personal computers. These processors served as the foundation for developing other product lines: Celeron for entry-level, Xeon for servers and workstations, embedded versions such as the EP80579 for systems-on-chip.

Industry shifts are reflected in the evolution of the Pentium. The rise of multimedia applications motivated the addition of specialized instructions. The growing importance of energy efficiency led to the development of optimized mobile versions. The emergence of parallel computing drove the adoption of multi-core architectures.

The Pentium illustrates a remarkable form of longevity in the computer industry. This brand, created in 1993, still exists in 2025, adapting to technological shifts and market needs.

Lua

In 1993, three Brazilian researchers from the computer graphics technology group at the Pontifical Catholic University of Rio de Janeiro, Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes, were working on computer projects for Petrobras, the national oil company. They had no idea they were about to create one of the most influential scripting languages in video game history.

Between 1977 and 1992, a strict protectionist policy made importing foreign software difficult in Brazil. Local companies had no choice but to develop their own tools. This is how Tecgraf had designed two specialized languages: DEL for data entry, and SOL for generating lithological reports intended for Petrobras. These two precursors contained the foundations of what would become Lua.

The first version of the language merged the capabilities of DEL and SOL into a more general approach. The name "Lua", which means "moon" in Portuguese, echoes SOL ("sun"), thus creating a poetic continuity with the earlier work. The creators first chose a restrictive license, limiting commercial use, before gradually adopting a more open philosophy starting with version 2.1, and the MIT license in 2002.

Simplicity forms the heart of Lua. The language relies on a few fundamental concepts: tables (associative arrays), functions, and coroutines. This economy of means translates into a remarkably compact implementation of approximately 17,000 lines of C code, capable of running on nearly all platforms, from microcontrollers to supercomputers. But it is primarily its ability to integrate with other languages, specifically C, that constitutes its strength. The interface between Lua and C, called the C API and a central element of the language, transforms Lua into an excellent extension tool for existing applications.

Successive versions mark significant technical milestones. Version 2.1 from 1995 introduced semantic extensibility mechanisms, giving programmers the ability to adapt the language's behavior. Version 3.0 from 1997 unified C and Lua functions into a single type. Version 4.0 from 2000 completely redesigned the C API to make it more consistent and reentrant. In 2003, Lua 5.0 revolutionized the internal architecture with the introduction of full lexical scoping, coroutines, and a register-based virtual machine, a remarkable innovation for a scripting language at that time.

The video game industry discovered Lua in 1998 when LucasArts used it for *Grim Fandango*. This pioneering adoption triggered a movement that would never stop. *World of Warcraft*, *Angry Birds*, *The Sims*, and hundreds of other titles integrated Lua to manage their game logic. The

reasons for this success? Easy integration, exceptional lightness, satisfactory execution speed, and an accessible learning curve.

Beyond games, Lua established itself in unexpected domains. Adobe integrated it into Photoshop Lightroom, where over 40% of the code relies on this language. It can be found in network routers, digital televisions, scientific instruments, and all sorts of embedded applications. This versatility stems from a particular design philosophy: providing mechanisms rather than imposing policies. Instead of dictating a way to program, Lua offers general tools allowing developers to build their own solutions, whether in procedural, functional, object-oriented, or data-oriented programming.

The language's development reflects a coherent vision maintained by a stable team since its inception. The creators systematically favor simplicity and consistency over the accumulation of features. Each new version undergoes rigorous testing, with alpha versions already displaying great stability and beta versions being virtually final. This caution guarantees the language's reliability.

The Lua community is primarily organized around a mailing list created in 1997. This technical forum maintains a high level of discussion while remaining open to beginners. The 2003 publication of the book *Programming in Lua* greatly contributed to the language's dissemination and the training of new users.

Lua's internal architecture testifies to this pursuit of minimalism. The compiler operates in a single pass without intermediate representation, thus efficiently processing large data files. This characteristic proves valuable in the video game domain, where Lua often serves to describe game resources.

The history of Lua demonstrates that a project born from specific local constraints can become a universal tool. Its success rests on clear design principles, rigorous implementation, and a stable development team that never lost sight of the language's original qualities: simplicity and efficiency. What was meant to be a temporary solution to a Brazilian problem became a worldwide standard, proving that the best ideas sometimes emerge from the most constraining situations.

R

Two statisticians from the University of Auckland, Ross Ihaka and Robert Gentleman, were seeking a statistical environment for their teaching laboratory on Macintosh. Their reading of Abelson and Sussman's book, *The Structure and Interpretation of Computer Programs*, and their interest in the Scheme language led them to develop a minimalist C interpreter of approximately one thousand lines of code. What began as an experiment would become one of the most widely used tools in statistics and data analysis.

The choice to adopt a syntax close to the S language, created at Bell Labs by John Chambers and his team, was natural. This decision ensured a degree of familiarity for statisticians accustomed to S, while leaving the freedom to innovate technically. R retained specific features inherited from Scheme, particularly in its memory management with fixed-allocation garbage collection that limited paging problems. Lexical scoping rules allowed functions to access variables defined at their creation, a characteristic that distinguished R from its contemporaries.

In August 1993, Ihaka and Gentleman deposited their first binary versions on StatLib and announced their work on the s-news mailing list. Martin Mächler from ETH Zurich, intrigued by the project, encouraged them to release the source code under the GNU GPL license. This suggestion was initially met with caution. The two researchers hesitated to fully open their code. Yet they took the leap in June 1995. This decision radically transformed the nature of the project: from a closed development between two collaborators, R became an international collaborative initiative.

The creation of automated mailing lists at ETH Zurich in 1996 accelerated external contributions. Bug reports, suggestions, and patches flowed in. The language progressively gained new functionalities. Faced with the scale of contributions, a larger group of core developers formed in 1997, establishing an organizational structure that endures.

The year 2000 marked an important milestone with the creation of the R Foundation for Statistical Computing, a non-profit organization based in Vienna. This foundation, formed by members of the development team, assigned itself three objectives: supporting the continued development of R, providing a reference point for interactions with the community, and managing the copyright of the software and its documentation.

R established itself as a comprehensive statistical tool with extensive capabilities for data manipulation, computation, and graphical visualization. Its modular architecture allowed the addition of functionalities

through packages, constantly enriching its application possibilities. The system integrated powerful operators for matrix calculations, a collection of statistical analysis tools, and sophisticated graphical functionalities. The documentation adopted a format close to LaTeX, ensuring comprehensive documentation accessible both online and in print. This approach, combined with the availability of source code, gave users the means to understand function behavior.

R's development model illustrates the advantages of free software in the scientific domain. International collaboration among developers, peer review of source code, and thorough testing in various real-world situations contributed to the software's robustness. The size of the user community, estimated in tens or even hundreds of thousands of people, multiplying code verifications and improvements, reinforced this solidity.

The development cycle follows a regular rhythm with major versions published annually since 2013. Each version undergoes rigorous testing, including alpha, beta, and release candidate phases, concerning both source code and precompiled binary versions for different platforms. The team maintains a version management system based on Subversion, with distinct branches for the stable version and the development version. Bug fixes are integrated into the stable branch, while significant new functionalities are developed in the development branch.

R's impact on statistical research and data analysis extends beyond the academic framework. Its adoption in industry, particularly in the finance, pharmaceutical research, and big data analysis sectors, testifies to its technical maturity. The availability of interfaces with other languages such as C, C++, and Fortran extends its capabilities to intensive computing.

R's legacy lies in its combination of a truly functional programming language with a comprehensive statistical environment. This association gives statisticians the means to develop and test new analytical methods while offering end users a practical and extensible tool. The project's sustainability rests on a distributed infrastructure, including a worldwide network of CRAN mirrors (Comprehensive R Archive Network) that ensures resource availability. This network distributes the core software as well as thousands of complementary packages developed by the community.

DHCP

TCP/IP networks in the 1980s remained modest and their configuration was entirely static. Administrators manually assigned an IP address to each machine, which stored this information in its secondary memory. Any modification required direct intervention at the console, typically followed by a system reboot. This hands-on approach suited the infrastructures of the time, but it would soon show its limitations.

Network growth and the arrival of affordable workstations without secondary memory disrupted this organization. It became urgent to centralize the administration of links between IP addresses and computer hardware. The RARP (Reverse Address Resolution Protocol) protocol emerged as the first answer: a machine connected to a network segment could now discover its IP address and initiate TCP/IP communications normally. Meanwhile, BOOTP (Bootstrap Protocol) facilitated the configuration of diskless stations by retrieving all TCP/IP parameters and system data needed for startup. The introduction of BOOTP relay agents made it possible to cross the boundaries of a single network segment. BOOTP already integrated an extension mechanism using the last field of the frame for specific data, an idea that DHCP would adopt.

RFC 1531 defined DHCP as a standard in October 1993. This extension of BOOTP corrected two major weaknesses: the requirement for manual intervention to add configuration information for each client, and the inability to reuse IP addresses. The protocol gained popularity, which led to successive clarifications. In 1997, RFC 2131 became the reference for IPv4 networks, a status it still holds today.

The arrival of IPv6 required the development of DHCPv6, documented in RFC 3315. This was not a simple transposition of DHCPv4 to IPv6 addresses, but a substantially different protocol. The most notable change concerned client identification: DHCPv4 relies on the MAC address, while DHCPv6 introduces the DUID (DHCP Unique Identifier). The design of DHCPv6 abandons the principle of a single address per device, allowing devices to request multiple addresses. RFC 3633 added prefix delegation, a novel functionality with no equivalent in DHCPv4. RFC 3736 extended the protocol's capabilities to support configuration of clients using stateless address autoconfiguration.

The Internet Systems Consortium (ISC) played a decisive role in the development of DHCP. The organization maintains two major systems: ISC DHCP and Kea. Ted Lemon and Vixie Enterprises wrote the first implementation, ISC DHCP, as a reference for the new protocol. Version 1.0 was released in June 1998, followed a year later by version 2.0. Version

3.0, published in 2001, integrated support for the IETF failover standard and asynchronous DDNS updates. Version 4.0, in 2007, brought IPv6 support.

A dedicated engineering team has been working on ISC DHCP since 2004. Ted Lemon and Shawn Routhier, former ISC employees, contributed to the project for many years. Thomas Markwalder has been the primary maintainer since 2016, while Francis Dupont has been actively involved in maintaining the software since 2007. The community has enriched the project with an LDAP lease storage system and a lease display script.

Kea represents an entirely new implementation, intended to replace the aging ISC DHCP. Initially designed within the BIND 10 application framework to support multiple DNS and DHCP applications, the project refocused on DHCP after DNS development was discontinued in 2014. Tomek Mrugalski and Marcin Siodelski led the initial development of Kea, which stands out for its modern REST management interface and modular architecture. Unlike ISC DHCP, Kea separates the DHCPv4, DHCPv6 and dynamic DNS daemons, and offers optional libraries to extend the core DHCP server functionality.

The community has organized itself around complementary tools. The `perfdhcp` software, distributed with Kea, evaluates DHCP server performance by generating heavy traffic from multiple simulated clients. It tests both IPv4 and IPv6 servers, providing statistics on response times and lost requests. The Anterius project, part of ISC's Google Summer of Code program, demonstrated the possibility of developing a lightweight management dashboard for Kea. More recently, the Stork project, launched in 2020, offers a robust and extensible web interface, integrating with the Prometheus time-series database and the Grafana visualization tool.

DHCP illustrates the adaptability of network protocols to the changing needs of computing. From a simple automatic configuration tool, it has evolved into a sophisticated system managing dynamic address allocation in modern networks. Its continued development, driven by an active community, ensures its lasting relevance in the face of contemporary network infrastructure challenges.

NCSA Mosaic

NCSA Mosaic arrived in 1993 and permanently transformed how people access resources on the Internet. The story begins a year earlier,

when Marc Andreessen, a student at the University of Illinois, spent more time at the National Center for Supercomputing Applications than in lecture halls. The timing was no coincidence: supercomputing centers were undergoing a major transformation. Cray computers, having become prohibitively expensive compared to increasingly powerful microprocessors, were giving way to computer networks connecting researchers and educators.

The Internet at that time resembled a patchwork of disparate protocols and services. FTP transferred files, Gopher organized menus, WAIS searched for information. Tim Berners-Lee had just created HTTP and HTML at CERN, but the available tools didn't really take advantage of these innovations. A few pioneering browsers like Erwise or ViolaWWW offered interesting features, without progressing beyond the prototype stage confined to specific platforms.

Andreessen and his colleague Eric Bina developed the alpha version of Mosaic during Christmas break. Their browser stood out from the start: it displayed images directly within the text, radically changing the experience. The intuitive graphical interface enabled navigation through simple clicks on hypertext links. The takeoff was meteoric. From twelve users at launch in early 1993, Mosaic had hundreds of thousands by mid-year.

This massive adoption stemmed from concrete technical reasons. The browser ran on UNIX, Windows, and Macintosh, handled the various Internet protocols, and installed easily thanks to a single executable. Internet users appreciated the bookmarks that saved their favorite pages and the browsing history. Image loading remained dependent on the modest bandwidth of that period, but the display speed was adequate.

The success prompted the NCSA to grant licenses for the source code. Spyglass became the primary distributor and supplied a version to Microsoft, which would serve as the starting point for Internet Explorer. Andreessen left the NCSA in late 1993 to start Mosaic Communications with Jim Clark, a company later renamed Netscape. Their commercial strategy was striking: distribute the browser free to individuals and educational institutions, charge businesses.

The impact on the Internet was dramatic. Web traffic jumped from 1.5% to 23.9% of total volume on NSFNet in two years. The number of accessible servers exploded: around fifty in January 1993, over 1,500 in June 1994. This growth raised governance questions. The NSF then entrusted domain name management to Network Solutions Inc., creating a commercial monopoly that would generate considerable controversy.

The Mosaic browser established the graphical interface standards ado-

pted by all its successors: address bar, navigation buttons, integrated image display. It popularized hypertext navigation among the general public and inspired the freemium business model adopted by so many online services. The issues it revealed regarding security, with the subsequent addition of the SSL protocol, or privacy protection, remain highly relevant today.

In computing history, Mosaic marks the moment when the Internet transitioned from an academic tool to a mass medium. Its ease of use and technical innovations made the modern Web possible.

Secure Socket Layer

When the World Wide Web opened to the general public in early 1990, no one truly grasped the security challenges that would follow. However, by 1994, Netscape Communications understood that Internet communications needed protection. The company then launched SSL (Secure Socket Layer) version 1.0, a protocol designed to secure communications between web browsers and servers.

The underlying idea was nothing new: back in 1978, Loren Kohnfelder had already proposed using digital certificates to guarantee the authenticity of public keys. This theoretical insight would serve as the foundation for SSL, which combines asymmetric and symmetric cryptography to ensure confidentiality and integrity of transmitted data.

The first public version, SSL 2.0, was released in 1995. But it came with a host of flaws: message authentication relied solely on MD5, the keys used for authentication and encryption were identical, and TCP connection closures exposed the protocol to truncation attacks. Worse still, nothing truly protected the initial negotiation against man-in-the-middle attacks.

Netscape responded quickly and released SSL 3.0 in 1996. This new iteration significantly improved key generation from the master secret. The protocol adopted a preliminary version of HMAC for message authentication and now mandated support for DH/DSS algorithms and Triple-DES.

Three years later, the IETF took over and standardized the protocol under the name TLS (Transport Layer Security) 1.0. Netscape's proprietary technology became an open standard. TLS 1.0 was compatible with SSL 3.0 but strengthened security through complete use of HMAC.

Versions then followed at a steady pace: TLS 1.1 in 2006, TLS 1.2 in

2008, then TLS 1.3 in 2018. Each iteration abandoned obsolete mechanisms and corrected vulnerabilities discovered in the meantime. SSL 2.0 was officially deprecated, SSL 3.0 followed suit. From 2015 onward, TLS 1.0 no longer sufficed for systems processing banking data.

Behind SSL and TLS lies a complex public key infrastructure (PKI). Certificate authorities (CAs) issue X.509 certificates to entities operating web servers with specific DNS names. These certificates are typically signed by intermediate CAs, creating a chain of trust that traces back to a root present in certificate stores of operating systems or browsers.

Certificate validation by CAs has evolved over time. Domain validation (DV) simply verifies authority over a domain name. Organization validation (OV) authenticates the entity requesting the certificate. Extended validation (EV), introduced later, requires CAs to follow a strict protocol to verify the applicant's identity.

SSL and TLS adoption remained sluggish for a long time, until Let's Encrypt arrived in 2015. This automated and free certificate authority removed the financial barriers that had hindered certificate acquisition. Google and other web giants began favoring secure sites in their search results, which significantly accelerated protocol adoption.

SSL and TLS operation follows four main stages. The session begins with negotiation where client and server exchange their cryptographic capabilities. Next comes authentication, typically from server to client only. The third phase establishes a shared key via asymmetric cryptography. Finally, the session uses this key to encrypt exchanged data with a symmetric algorithm.

Web browsers have gradually integrated visual indicators to signal a secure connection. The padlock, which became the universal symbol of SSL and TLS security, appeared in the earliest versions of Netscape Navigator. Modern browsers display these indicators in the address bar, with color codes and explicit messages that inform users about the security level.

Early SSL applications mainly concerned e-commerce and online banking services. The protocol then expanded to secure email, virtual private networks, and many other services requiring confidential communications.

The evolution of computer threats has led to enriching TLS with new features. The HSTS (HTTP Strict Transport Security) security policy mechanism now forces the use of the secure protocol. Certificate Transparency promotes the detection of fraudulent certificate issuance.

The history of SSL and TLS clearly shows that in security matters, nothing is ever guaranteed. Successive versions of the protocol have had to continuously strengthen communication protection while maintaining

compatibility with existing systems. This technology has established itself as the de facto standard for securing Internet communications, contributing to the growth of e-commerce and online services that demand confidentiality.

Microsoft Windows NT 3.51

Microsoft launched Windows NT 3.1 (for "New Technology") in July 1993. This first version inaugurated a family of operating systems that would reshape the professional computing landscape. The platform targeted Intel x86 and RISC processors. IDC analysts then defined an advanced operating system by its 32-bit APIs, preemptive multitasking managing processes and threads, integrated networking, and demand-paged virtual memory.

Feedback on Windows NT 3.1 pushed Microsoft to develop version 3.51, which was released in 1995. Two variants appeared: Windows NT Workstation and Windows NT Server. The former sought maximum responsiveness for interactive applications, while the latter optimized network performance. This choice reflected a clear segmentation strategy between workstations and servers.

Windows NT Workstation 3.51 reduced its memory footprint by 4 to 8 MB compared to its predecessor. 16-bit applications gained 25 to 50% performance on typical desktop configurations. On RISC systems (Alpha AXP and MIPS), Intel emulation for Windows 16-bit and MS-DOS applications received substantial improvements.

Windows 16-bit applications could now run in separate DOS virtual machines (VDMs). A crashing application no longer compromised the stability of other programs. Integration mechanisms like DDE (Dynamic Data Exchange) and OLE (Object Linking and Embedding) continued to function across VDMs.

Microsoft enriched Windows NT 3.51 with OpenGL, this 3D graphics function library initially developed by Silicon Graphics. A committee bringing together Digital Equipment Corporation, IBM, Intel, and Microsoft validated this technology. CAD, industrial design, and scientific analysis applications leveraged these new 3D capabilities.

Network connectivity improved significantly. The system integrated a compatible NetWare redirector that facilitated access to files and printers on Novell servers via IPX/SPX. The TCP/IP stack doubled its perfor-

mance. PPP and SLIP enabled TCP/IP connections over asynchronous lines, attracting the UNIX and Internet communities.

Security was reinforced with account lockout after several failed login attempts. This protection blocked brute force attacks on passwords. NTFS gained per-file and per-directory compression, reducing data size by 40 to 50% depending on their nature.

Microsoft improved reliability. When a fatal system error occurred, Windows NT 3.51 automatically saved the memory state to a debug file and restarted. This approach, borrowed from UNIX systems, maximized the availability of network-connected machines.

The system targeted four categories of users. First, enterprises sought its stability to limit the costs of frequent restarts. With 2,000 workstations restarting 4 times per month for 5 minutes, an organization lost 8,000 hours of productivity per year. Developers appreciated its robustness for creating Win16 and Win32 applications. Technical users (engineers, scientists, statisticians) exploited its performance for their intensive calculations, often ported from UNIX. Computing enthusiasts adopted its innovations without losing productivity.

Peripheral management extended to PCMCIA cards for high-end laptops. The system now handled SCSI scanners and HP plotters. Absolute input mode facilitated the use of digitizers and touch screens. Windows NT 3.51 also integrated drivers for Cinepak and Indeo video formats.

The user interface adopted Windows 95's common controls. File Manager, Print Manager, and other system applications inherited a modernized presentation with tooltips. The login sequence was customizable, allowing for example bank smart cards.

Microsoft offered different technical support formulas, including telephone assistance available 24 hours a day for critical incidents. Enterprises could choose different service levels, from basic support to Premium contracts with guaranteed response time. A network of authorized centers completed the system.

Administration was simplified with five backup modes: normal (complete copy), copy (without marking), incremental (modified files), differential (changes since the last complete backup), and daily (same-day modifications). The system documented each backup with the tape name, date, operator identity, and sequence number.

Windows NT 3.51 marked an important milestone in Microsoft systems history. Its solid architecture, performance, and advanced features attracted enterprises and demanding users. The system established technical foundations that would influence future versions of Windows, demonstrating Microsoft's ability to finally design reliable professional systems.

Apple Power Macintosh

The announcement of an alliance between Apple, IBM, and Motorola in May 1991 would have made any computer industry observer smile. These three companies, two of which were fierce competitors in the personal computer market, had nevertheless decided to collaborate. IBM was Apple's historical enemy, the one against which the company with the apple logo had built itself since its beginnings.

This unlikely union had its roots in Apple's growing difficulties with its Motorola 68000 processors. Intel was steadily eating away at market share with its x86 chips, which were less technically elegant in the eyes of purists but increasingly more powerful. Motorola could no longer keep pace with its American competitor. The new Macintosh computers were falling worryingly behind, and their competitiveness was eroding.

Apple had to respond. The company turned to RISC architecture, an acronym for Reduced Instruction Set Computing. This technology came from IBM, which had experimented with it since the 1970s in its 801 project, conducted in a building of the same name in Yorktown Heights, New York State. John Cocke led a team that was developing a modular processor architecture capable of powering both personal computers and larger machines.

On July 3, 1991, Apple and IBM officially sealed their partnership. Motorola joined them as the third party. This AIM alliance was to give birth to a new generation of RISC processors intended for future Macintosh computers. IBM and Motorola would design the chips together. Apple and IBM would collaborate on an object-oriented operating system. The PowerPC 601 emerged from this collaboration. This first chip was derived from IBM's RSC processor. Apple wanted computers faster than Intel-based PCs without sacrificing price competitiveness. The Power Macintosh 6100/60 launched the line with this processor clocked at 60 MHz.

The RISC architecture brought tangible benefits. The processors simplified their instruction set to execute programs more efficiently. They included 32 general-purpose registers, compared to only eight for Intel x86 chips. This richness reduced external memory accesses and boosted performance.

The migration to PowerPC represented a technical puzzle for Apple. It was necessary to preserve compatibility with existing applications written for the 68000 architecture. The company achieved this through a software emulator that allowed users to continue running their usual programs on

the new Power Macintosh computers with acceptable performance.

Apple streamlined its product line around three models: the 6100, 7100, and 8100 series. This simplification contrasted with the previous proliferation of references that complicated customer choice. The machines were distinguished by their clock speed, indicated after the serial number: 60 MHz for the 6100/60, 66 MHz for the 7100/66, 80 MHz for the 8100/80.

The Power Macintosh computers offered advanced features: 16-bit audio stereo, serial ports compatible with LocalTalk and GeoPort, integrated Ethernet connectivity. RAM reached up to 72 MB for the 6100/60 model, 136 MB for the 7100/66, 264 MB for the 8100/80. These characteristics placed them in a strong position against competing PCs.

A curious detail deserves attention in this story. Motorola never manufactured a single PowerPC 601 processor, despite its communications suggesting otherwise. IBM Microelectronics handled production alone. Motorola nevertheless participated in the development of subsequent generations such as the 603, 604, and 620.

This arrangement satisfied everyone. IBM penetrated the personal computer market with its RISC technology, which was initially designed for its RS/6000 workstations. Apple obtained powerful processors at a good price. Motorola maintained a foothold in the Macintosh ecosystem, although it lost its status as exclusive supplier.

The Power Macintosh computers achieved notable commercial success. They gave Apple the ability to sell machines faster than equivalent PCs in certain applications, particularly graphics. A Macintosh Quadra equipped with a 40 MHz 68040 processor already outperformed a PC with a 66 MHz 486. The new Power Macintosh computers widened the gap.

This technical success did not overturn Apple's market position. The company maintained a share of approximately 10 to 15%, facing the overwhelming dominance of IBM-compatible PCs. Professional users, guided primarily by budget considerations, remained loyal to the Intel/Windows platform.

The PowerPC architecture continued its evolution for years. The processors gained in power and energy efficiency. Apple used it until 2006, before switching to Intel processors. This transition closed a singular technological chapter, where historical competitors had joined forces to create an alternative to established architectures.

CD-RW

The idea of storing information on an optical disc dates back to the 1950s. Americans such as David Paul Gregg and James Russell envisioned writing with electron beams and reading with laser beams. The principle of the rotating disc and a reflective surface transformed these intuitions into concrete possibilities.

Around 1970, Hollywood became interested in optical discs for distributing films. MCA and Philips joined forces and launched the first consumer laser videodisc, LaserVision, in late 1978. Helium-neon lasers read the molded pits on a 30 cm disc, whose size recalled that of vinyl records. The video information was encoded in the variable spacing between the edges of these pits, arranged in a spiral.

In 1974, Philips laboratories embarked on developing an optical disc audio system. Their engineers relied on existing technologies while betting on future advances in integrated circuits and semiconductor lasers. The project gained momentum, and the team concluded that digital technology would surpass analog recording. The scale of the undertaking prompted Philips to seek a partner: Sony joined the venture in 1979.

The compact disc adopted a diameter of 120 mm, much more compact than LaserVision. The designers knew that contemporary semiconductor lasers delivered approximately 1 mW at 800 nm. They adjusted the optics accordingly. The laser beam passes through a transparent substrate of 1.2 mm before reaching the data engraved on the aluminum layer of the disc.

The first audio CD players arrived in stores in 1982. The technology found its place in the computing world. Philips and Sony announced the CD-ROM in 1984, and the first drives were delivered the following year as peripherals for large systems. International organizations validated the standard in 1985. However, data organization remained proprietary until 1988, when ISO 9660 became the reference.

Research on recordable and rewritable discs accelerated in the 1970s in the United States, Europe, and Japan. The limited power of lasers hindered progress. In France, Thomson-CSF and later Alcatel Thomson Gigadisc experimented with glass discs coated with layers containing malleable gold. Writing created microscopic bumps, but repeated laser readings deformed these features.

Another approach proved more fruitful for WORM (Write Once Read Many) media: coating glass or plastic with polymer dye mixtures. The optics were those of read-only discs, provided peak power of 50 to 100 mW was available. Philips and Sony defined the recordable CD (CD-R) in

their 1988 "Orange Book". By the late 1990s, the required lasers became affordable and CD-R burning spread throughout personal computing.

Two technologies competed for the rewritable disc market: magneto-optical recording and phase change. The former got off to a strong start in the early 1970s. It relied on synchronizing laser heating with magnetic field modulation. Read/write heads were complex, but the media tolerated a virtually unlimited number of cycles.

Phase-change media use a thin layer of chalcogenide alloy, such as AgInSbTe or GeSbTe. This layer is stable in two states: amorphous and microcrystalline, each exhibiting distinct reflectivity. A brief, intense laser pulse melts the layer, which cools in an amorphous state. A longer but less energetic pulse heats the film without melting it, triggering crystallization. From the 1970s to the 1990s, research refined alloy compositions and deposition processes.

In the mid-1990s, Korean and Taiwanese manufacturers entered the market. The first CD-ROM production in Taiwan and Korea began in 1994. Two years later, Taiwanese manufacturers accounted for 12% of global production. In 1996, LG alone represented nearly 10% of worldwide sales. This competition drove down drive prices.

The number of optical drive manufacturers rose from 2 in 1983 to 16 in 1985, climbed to 65 around 1995, then dropped back to 44 in 1999. Newly created specialized companies initially stimulated technical progress. Before 1988, the year of ISO 9660, the fastest drives relied on proprietary formats. After standardization, electronics and computing giants dominated the performance race.

Prices for optical burners plummeted: $15,000 in 1991, $5,000 in 1993, and under $1,000 in 1995. CD-RW established itself firmly in the computing landscape until the arrival of rewritable DVD in the 2000s. The technology left its mark on the history of digital storage.

IBM Aptiva

IBM launched the Aptiva in September 1994. The computer succeeded the PS/1 line in the consumer market. Manufacturers were trying to stand out in a standardized world built around Intel and Windows, the famous "Wintel". But the task would prove difficult.

The first Aptiva computers received Intel 80486 processors before evolving to Pentiums and AMD chips. IBM manufactured most machines in-

house, except for the E series entrusted to Acer. The computers were sold as complete solutions: system unit, monitor, speakers, keyboard, and mouse. The first generation ran on IBM PC DOS 6.3 and Windows 3.1. Pentium versions offered Windows 95, and on certain models a "select-a-system" option that let users choose between PC DOS 7/Windows 3.1 and OS/2 Warp.

On the M, A, C, and S models, IBM integrated an in-house Mwave card to handle sound and modem. This proprietary solution accumulated compatibility and performance problems. IBM eventually abandoned it in favor of standard components. The company had to settle a dispute by financially compensating buyers for purchasing compatible peripherals.

In 1996 came the S "Stealth" series, which changed aesthetically. IBM abandoned the traditional beige of computers for a black design that broke with convention. Consumer electronics now influenced the appearance of machines. The S series separated the floppy and CD-ROM drives from the main case containing the motherboard. These elements were housed in a thin media console that served as a base for the monitor. A 1.8-meter cable connected everything to the system unit. This architecture freed up workspace: the main case could slip under the desk or into furniture while keeping the drives within reach.

IBM reduced the number of configurations to five models. The entry-level model featured a 166 MHz Pentium for $2,499. The high-end model used a 200 MHz Pentium sold for $3,099, monitor not included (budget between $499 and $799 depending on size). All machines had at least 16 megabytes of RAM, hard drives from 2.5 to 3.2 gigabytes, and 28,800 bits per second modems.

Between 1994 and 2001, IBM released the Aptiva in several series identifiable by a letter: M (Magic), A, C (Courageous), E, L, and S (Stealth). The first models adopted a desktop format (reference 2144) or tower (2168). Production then shifted exclusively to tower format. Many machine types followed: 2134-2138, 2140-2144, 2151-2159, 2161-2168, 2170-2178, 2193-2198, 2255, 2270-2274, 6832, and 6864.

IBM discontinued the Aptiva in 2001 without offering a direct replacement. This decision was part of the company's withdrawal from the consumer market. Customers were redirected to the NetVista line, rather intended for professionals. This strategic shift reflected the difficulties traditional manufacturers faced with market standardization and the arrival of new players.

Netscape Navigator

In 1994, Jim Clark and Marc Andreessen founded Netscape Communications with a clear vision: to create a universal interface providing access to the Web from any device. Clark had made his fortune with Silicon Graphics, while Andreessen had just graduated from the University of Illinois where he had led the development team for Mosaic, the first browser designed for the general public. Their partnership would revolutionize how the world accesses the Internet.

The first version of Navigator was released in December 1994, and achieved immediate success. Internet users adopted it massively. A year later, Netscape's market value reached $7 billion. The company wasn't limited to the browser: it developed server solutions that relied on Internet protocols to run intranets, extranets, and other business applications.

Navigator stood out through its technical advances. Its rendering engine loaded text and images in parallel, whereas competing browsers displayed text first before showing images. A caching system accelerated the loading of regularly consulted pages. Security received particular attention with the integration of the SSL (Secure Socket Layer) protocol to protect sensitive data. Anti-spyware and anti-advertising mechanisms reinforced this protection.

Navigator 1.1's interface offered nine ways to browse. Users could directly enter a URL, follow hypertext links, use breadcrumbs to retrace their path, click on shortcut buttons, go back, consult history, manage bookmarks with annotations, preview links, or rely on visual markers. This functional richness made the Web accessible to those with no technical training.

Microsoft quickly understood that Navigator threatened its empire. In 1995, Bill Gates launched Internet Explorer and integrated it free of charge into Windows 95. The strategy paid off. Navigator's market share, which had approached 90% in early 1996, began to decline. By the end of 1997, it fell below 50%.

Financial losses accumulated. In November 1998, Netscape agreed to be acquired by America Online (AOL) for $4.3 billion. AOL simultaneously concluded an agreement with Sun Microsystems worth $1.25 billion to market Netscape software and take over its divisions.

Netscape released Navigator's source code under the name Mozilla project in March 1998. It was the first time a publicly traded company joined the free software movement. Thousands of people downloaded the code as soon as it went online. Developers from around the world contributed voluntarily to the project, like young Pavlov from Georgia. Netscape was

counting on this community to improve the browser and counter Microsoft. The adopted license was inspired by the GPL (GNU Public License) while authorizing commercial exploitation.

Navigator's codebase exploded, growing from a few hundred thousand lines to over 2 million. Engineers worked back-to-back shifts, some sleeping at the office to meet deadlines. Multi-platform compatibility further complicated matters: the code had to run on Windows, Mac OS, and various versions of UNIX.

In 1997, Netscape launched an ambitious overhaul with the Communicator 6.0 project. The goal: restructure the code into modules that were easier to maintain and rewrite certain parts in Java to simplify multi-platform development. In early 1998, the project was abandoned. Java didn't offer the expected performance. Modularization continued more gradually in versions 4.5 and 5.0.

The merger with AOL in 1999 marked the end of Netscape's independence. Many executives sold their shares during the final year, but CEO Jim Barksdale increased his stake and converted his securities into AOL shares worth over half a billion dollars at the time of the merger. Several engineers left the ship, disappointed by AOL's marketing-oriented direction.

Netscape's legacy remains immense. The company made the Web accessible to the masses by simplifying navigation. Its innovations in security and performance established standards that endure. The Mozilla project inspired numerous open source initiatives and gave birth to Firefox.

The Netscape adventure illustrates how quickly things can shift in the software industry. In less than five years, the startup had become a mere division of a large corporation. This story also demonstrates the strategic importance of web browsers, which have become the universal interface for accessing online services.

In 2003, Navigator's market share was negligible. AOL maintained development until 2008, when the project ended definitively. The developers joined the Firefox team, which inherited Navigator's patents. This succession testifies to the durability of Netscape's innovations in the contemporary Web landscape.

World Wide Web Consortium

In 1994, four years after Tim Berners-Lee created the Web, the World Wide Web Consortium (commonly referred to as W3C) was established. The international organization set itself a mission: to lead the World Wide Web to its full potential through the development of protocols and guidelines that ensure its long-term growth. In the 1990s, the rapid proliferation of Web usage made this structure indispensable.

Three technical elements form the foundations of the Web. The HTTP protocol handles communications, URIs enable universal resource identification, and HTML structures document markup. From 1993 onward, the combination of these technologies triggered explosive expansion. The free and open nature of these technologies largely explains this success: they remain accessible without usage rights, whereas other systems like Gopher failed after the University of Minnesota attempted to impose a paid license.

The W3C adopted an original structure. This international consortium brings together companies, universities, and public institutions. This configuration associates the various Web stakeholders in a neutral forum, away from direct commercial interests. Three institutions host the organization: MIT in the United States, ERCIM in France, and Keio University in Japan. Eighteen regional offices complete the setup and ensure a global presence.

Standards development forms the core of the activity. Technical recommendations follow a rigorous process. Working groups, composed of experts from member organizations, develop proposals. These are then subjected to public comment periods. This method ensures technical quality and adequacy to real needs. The early years produced fundamental standards: XML, CSS style sheets, the DOM document model.

In 2001, the W3C proposed allowing the inclusion of patented technologies in its standards, subject to Reasonable And Non-Discriminatory (RAND) licenses. The developer community, attached to free software principles, strongly opposed this proposal. After months of intense debate, the organization finally adopted a policy that favors royalty-free technologies. The episode illustrates the W3C's ability to incorporate community feedback.

Areas of intervention expanded over the years. Accessibility is a central concern; recommendations emerged to make the Web usable by everyone, regardless of physical or mental capabilities. Internationalization constitutes another line of work, particularly Web usage in different languages and writing systems. The consortium simultaneously developed standards

for adaptation to mobile terminals, anticipating the diversification of access methods.

The Semantic Web represents a central direction of work since the early 2000s. This evolution aims to enrich web content with machine-understandable metadata. The RDF and OWL technologies, standardized by the consortium, describe relationships between information and simplify automated processing. Specialized groups, such as the one dedicated to life sciences and health (HCLS), explore practical applications of these technologies in various domains.

The organization continuously adapts its working methods. Working groups, which initially brought together only member representatives, gradually opened up to invited experts. Public comment periods took on increasing importance and enabled the integration of feedback from a broader community of developers and users. This evolution reflects the desire to maintain a balance between technical expertise and field needs.

The 2010s saw the emergence of new technical initiatives. Native multimedia capabilities introduced by HTML5 reduced dependence on proprietary technologies. Standards for progressive web applications enabled the creation of sophisticated applications directly in the browser. The W3C simultaneously developed recommendations for privacy protection and security, in response to growing concerns in these areas.

The consortium's governance follows an original model that combines technical direction and broad consultation. Tim Berners-Lee, as director, maintains a strategic guidance role while fostering the emergence of consensus. A permanent team of approximately fifty people ensures activity coordination, while over six hundred experts participate in various working groups.

The production of technical standards is accompanied by significant documentation and education work. The W3C publishes usage guides, organizes workshops, and maintains educational resources. This action contributes to the dissemination of best practices and helps with the adoption of new technologies. Translations of main documents are produced in numerous languages, which strengthens the international accessibility of standards.

The history of the W3C demonstrates that it is possible to collectively manage a global technical infrastructure. Through its operating mode that combines technical expertise, public consultation, and consensus building, the organization maintains coherent Web development while preserving its open and universal character.

Yahoo!

In 1994, two Stanford electrical engineering students, David Filo and Jerry Yang, began compiling a list of websites they enjoyed. Their small personal project, which they initially named "Jerry and David's Guide to the World Wide Web", addressed a simple need: finding one's way around an emerging Web. The name Yahoo! came to them later, an acronym for "Yet Another Hierarchical Officious Oracle", a phrase that captured the humor and casual attitude of the two founders.

Word-of-mouth did the rest. In April of that year, the directory listed a hundred sites and attracted a thousand visitors per week. Five months later, the numbers surged: 2,000 sites indexed, 50,000 page views per day. Netscape, which had just launched its browser, decided to display a link to Yahoo! on its homepage, and traffic exploded.

Stanford's servers couldn't keep up. Marc Andreessen, from Netscape, offered to host the service for free. In March 1995, Yahoo! incorporated and received one million dollars from Sequoia Capital. The two students left the university, set up offices in Mountain View, and hired their first employees.

Online advertising began in August 1995. General Motors and Visa were among the first advertisers. Yahoo! experimented with targeted advertising based on users' interests, an innovative approach for the time. The site simultaneously added news, weather, stock quotes, and other free content.

The April 1996 IPO turned euphoric. The stock rose 154% on the first day, valuing the company at $848 million. A performance that surpassed Netscape's a few months earlier. Yahoo! used these funds to expand internationally, first in Japan with Softbank.

Acquisitions followed in rapid succession: Four11 for email, GeoCities for personal webpage hosting, Broadcast.com for multimedia. The group diversified its offerings while maintaining its guiding principle: a free, general-purpose web portal. The Internet bubble propelled it to great heights, with its market capitalization exceeding $125 billion in 2000. Then the bubble burst, advertising revenues collapsed. Most importantly, Yahoo! struggled to adapt. Google dominated with its PageRank algorithm. Facebook redefined social interactions. Turnaround attempts, notably under Marissa Mayer's leadership between 2012 and 2017, failed to reverse the trend.

Verizon acquired Yahoo!'s Internet operations in 2017 for $4.5 billion,

a fraction of its former value. Nevertheless, Yahoo! remains a case study of the early days of commercial Web. The company made the Internet accessible to the general public with a simple portal. It established viable business models, particularly through targeted advertising.

The strategic mistakes speak for themselves. Yahoo! outsourced its web search to Google and later to Microsoft, choosing not to invest heavily in this area. The culture of innovation withered, giving way to an unresponsive bureaucracy. Costly acquisitions often resulted in integration failures.

Some Yahoo! brands survive at Verizon. Yahoo! Finance still has a notable audience. This company's history illustrates how quickly dominant positions erode in technology. It underscores the imperative of continuous innovation and adaptation to user behavior. Yahoo!'s journey embodies both the promises and pitfalls of commercial Web, the difficulties of building lasting technology companies.

Yahoo! is tied to the history of the Internet. By democratizing the Web in the 1990s, the company participated in spreading this technology. Its model of a free portal funded by advertising inspired numerous services, even though Yahoo! did not maintain its dominance.

AltaVista

In 1995, Paul Flaherty, a researcher at DEC, had an idea during his vacation. He wanted to demonstrate the power of the Alpha processor that his company had just developed. From this intuition would emerge AltaVista, the first true Web search engine. Two DEC engineers, Michael Burrows and Louis Monier, took charge of developing the project.

The infrastructure relied on two evocatively named machines: Scooter and Turbo Vista. The first, equipped with a 20-gigabyte hard drive and one gigabyte of RAM, crawled web pages. The second, with 250 gigabytes of storage and two gigabytes of RAM, stored data and returned results. This configuration represented impressive computing power at the time.

AltaVista brought a major technical breakthrough. For the first time, a search engine indexed the full text of web pages. Its predecessors limited themselves to titles and headers. The interface, deliberately minimalist, nonetheless offered advanced features: keyword search, exact phrase search, and the ability to restrict results to a particular domain.

Success was immediate. From launch, 300,000 queries poured in each

day. Two years later, this figure reached 80 million. In 1998, a study conducted among professional researchers revealed that AltaVista was their preferred search engine at 45%, far ahead of HotBot which garnered 20% of votes.

The speed of query processing, made possible by the Alpha processor architecture, made all the difference. The search engine offered innovative features such as multilingual search and automatic translation through its Babel Fish service. Its database contained more than 16 million web pages in 1996, a figure that kept growing.

But AltaVista's story illustrates how a technological lead can melt away for lack of strategic vision. In 1998, Compaq acquired DEC. The following year, under Rod Schrock's leadership, AltaVista abandoned its streamlined interface to transform into a web portal. The company wanted to compete with Yahoo!. This decision, however, diluted AltaVista's core expertise in information retrieval.

In 1999, Compaq sold 83% of AltaVista's shares to CMGI, owner of the Lycos search engine. CMGI prepared an IPO, but the bursting of the dot-com bubble forced the company to abandon it. Meanwhile, Google, founded in 1998, was gaining ground thanks to its exclusive focus on search and its PageRank algorithm.

In 2003, Overture Services acquired AltaVista, before being bought by Yahoo!. The pioneering search engine found itself integrated into the Yahoo! platform, losing its own identity. On July 8, 2013, Yahoo! ended AltaVista's existence, redirecting its domain to its own search engine.

AltaVista's legacy is nonetheless significant in Web history. Its technical architecture influenced subsequent search engines. It established the standard for large-scale full-text search. Its simple and effective interface inspired numerous competitors, particularly Google.

AltaVista's failure can be explained by several factors. DEC initially considered the search engine merely as a technology demonstrator rather than a commercial opportunity. Frequent ownership changes prevented any coherent long-term strategy. The transformation into a web portal betrayed a misunderstanding of user expectations, who favored simplicity and search efficiency.

On the technical level, however, AltaVista had everything right. Its crawler, named Scooter, indexed millions of pages. The search engine offered advanced search operators such as NEAR to measure term proximity. It was the first to analyze web page metadata and to use linguistic analysis to improve results.

AltaVista's decline coincided with Google's rise, which refined concepts introduced by its predecessor. Google added the notion of page popularity

to its ranking algorithm, whereas AltaVista focused mainly on textual relevance. This evolution better addressed the needs of users confronted with a rapidly expanding web.

AltaVista's disappearance marked the end of a pioneering Web era, where pure technical innovation was no longer sufficient to guarantee success against competitors who better mastered commercial and marketing aspects.

Apache Web Server

The Apache web server was born in 1995, but its history begins two years earlier with NCSA HTTPd. In 1993, Rob McCool developed the first portable HTTP server at the National Center for Supercomputing Applications at the University of Illinois. Distributed free of charge with its source code, the software enjoyed immediate success. Website administrators adopted it en masse, preferring this free solution to Netscape's commercial servers, which cost several thousand dollars. NCSA HTTPd was then the most widely deployed server on the Internet.

But in 1994, the original team left NCSA to join Netscape, McCool included. Development came to an abrupt halt. Users found themselves with frozen software-functional, yes, but improvable. They had grown accustomed to fixing bugs themselves, adding features according to their needs, and then sharing these modifications on mailing lists in the form of *patches*. Brian Behlendorf was one of these active contributors; he notably created a password authentication system for the Hotwired website.

Faced with the project's abandonment, a small group of *webmasters* decided to take up the torch. In February 1995, eight volunteer programmers including Brian Behlendorf and Roy Fielding created the "new-httpd" mailing list to coordinate their work. Two months later, in April, they released their first version: 0.6.2, built on NCSA HTTPd 1.3 and enriched with patches developed by the community.

The origin of the Apache name remains contested to this day. The foundation has long explained that it paid homage to the Apache Native American tribe, known for their endurance and warrior skills. Yet between 1996 and 2001, the project's FAQ offered another explanation: "a patchy server", a direct allusion to the numerous patches applied to the original code. Brian Behlendorf has successively validated both versions, suggesting that perhaps they both coexist.

Apache 1.0 arrived in December 1995. Robert Thau led this complete code rewrite, internally dubbed "Shambhala". This version introduced a modular architecture that changed everything and simplified the addition of new features. Development became decentralized, innovation came from everywhere in the community, and results followed swiftly. By April 1996, Apache was the most widely used web server on the Internet. It has never relinquished that top position since.

The project drew inspiration from the IETF's operating principles and its motto: "rough consensus and running code". The rules remained simple. Anyone could contribute, but only core developers voted on official releases. Entry of new voting members required a nomination and unanimity among existing members. This system preserved the project's coherence while remaining open.

Apache prevailed through its technical innovations. Virtual hosting, available from the summer of 1995 onward, addressed a pressing need for Internet service providers. This feature could manage multiple websites on a single server, up to 10,000 sites on one machine. Commercial solutions offered nothing comparable. The accessible source code provided another decisive advantage: users modified the software according to their needs, without waiting for a publisher to deign to implement a new feature or fix a bug.

IBM took interest in Apache in 1998. The computing giant was seeking a web server for its Websphere line. Rather than its own solutions, it chose Apache, recognizing its technical superiority. IBM understood where the value resided: not in the server, but in services and proprietary extensions like e-commerce systems. This collaboration marked the beginning of a lasting relationship. IBM in turn contributed to Apache's development, enriching the project with its expertise.

The project's structure took a new step in 1999 with the creation of the Apache Software Foundation, a non-profit organization. This evolution resolved legal and administrative aspects while preserving the project's independence and collaborative culture. The ASF gradually developed a portfolio of projects, including Tomcat, the reference implementation of the Java J2EE architecture.

The core development team numbered about fifteen people spread across the United States, Great Britain, Canada, Germany, and Italy. Contrary to stereotypes, these developers were not passionate teenagers but seasoned professionals: PhD students, doctors of computer science, experienced developers, company executives. Core developers produced about 80% of new features, but more than 400 people contributed to the code and over 3,000 reported issues.

Apache dominated the market despite fierce competition. Microsoft integrated its Internet Information Server into Windows NT as early as 1996, never exceeding 30% market share. Other players positioned themselves in niches, Zeus for very high-traffic sites for example. Some disappeared, like Netscape's server, which had nevertheless reached 10% of the market.

In September 2009, Apache served more than 54% of global websites and more than 66% of the most visited sites. This dominance became self-sustaining: widespread adoption generated feedback and contributions that strengthened the software's quality and stability, in turn attracting new users.

This success inspired countless other projects and shaped open source development practices.

BeOS

In 1990, Jean-Louis Gassée and Steve Sakoman left Apple to found Be Inc. Their project was to create an entirely new operating system capable of taking advantage of modern machines without dragging along the burden of backward compatibility. Apple was going through a difficult period at the time, with its aging system showing its limitations in the face of hardware advances.

BeOS was born from this desire to start from scratch. Be's engineers built a system designed primarily for multimedia, without worrying about the usual compromises. Multithreading runs through all layers of the system, with each task divided into multiple execution threads to make the best use of resources. The 64-bit BFS file system incorporates database capabilities and journaling to protect data. Symmetric multiprocessor management proved refined, which meant that BeOS ran faster on a Pentium 233 with 64 MB of memory than Mac OS X on machines eight times more powerful.

The 1995-1996 demonstrations were impressive. BeOS simultaneously ran multiple real-time 3D rendering windows, played video, and browsed the web without faltering, where competing systems struggled. Apple, desperately seeking a modern replacement for its aging system, entered into discussions to acquire Be Inc. in 1996. The negotiations failed over price. Apple turned to NeXT, bringing Steve Jobs back to the company he had created.

Be Inc. continued on its path and targeted the Intel PC market start-

ing in 1998, while keeping its PowerPC version. The system won people over with its elegant interface, responsiveness, and technical innovations like dynamic queries in the file system. Users appreciated the stability, execution speed, and fine-grained file attribute management.

But BeOS failed to break through. Applications were sorely lacking. Developers hesitated to invest time in a platform with few users, and users waited for applications before adopting the system. A vicious circle. Windows dominated the PC market, Mac loyalists stayed on their platform, and space was tight for a newcomer.

Be Inc. attempted a pivot in 1999 toward emerging Internet appliances. Too late. Investors grew weary of funding a project that couldn't find its audience. The company closed in 2001 and sold its patents to Palm.

The story doesn't end there completely. Haiku, an open source project, took up the torch and reconstructed BeOS in spirit. Some of the system's ideas, advanced metadata management or multimedia-oriented architecture, spread elsewhere.

BeOS illustrates the difficulties of a market where technical excellence is not enough. Timing plays its role: BeOS probably arrived too late to disrupt established systems. What remains is the mark of a modern architecture, an attention to performance that continues to inspire. The system shows that by freeing oneself from historical constraints, remarkable results can be achieved.

The lesson concerns the software ecosystem. Without popular applications, a system, however brilliant, withers away. This reality still weighs on current strategies, where building a developer community matters as much as technical prowess.

BeOS's commercial failure does not erase its technical contribution or the questions it raised about market dynamics.

DVD

In 1994, the industry sought a successor to the CD-ROM. Storage needs were growing, digital video was advancing, and the CD was showing its limitations. Two camps formed around competing technologies: Super Disc on one side, Multimedia CD on the other. Nobody wanted to relive the same fratricidal war that had left its mark in the 1980s between

VHS and Betamax. The manufacturers eventually agreed on a common standard.

This convergence brought together players who could have been at odds: Japanese consumer electronics manufacturers, Hollywood studios, American computing giants. Together they created a disc the size of a CD but capable of storing seven times more data. The Japanese discovered the first players in November 1996. Americans had to wait until August 1997.

The DVD relies on a simple but effective technical trick. Where the CD stacks 1.2 mm of polycarbonate in a single layer, the DVD superimposes two 0.6 mm substrates. This architecture allows the use of a laser with a reduced wavelength: 650 or 635 nanometers versus 780 for the CD. The tracks tighten, the pits shrink. A single-layer DVD contains 4.7 GB. The dual-layer reaches 8.5 GB per side. A double-sided disc stores 17 GB in theory, but this format remains marginal.

Hollywood embraced the DVD enthusiastically. MPEG-2 compression fits a two-hour feature film on a single layer, with an image that surpasses that of VHS tapes and LaserDiscs. The studios appreciate the interactive menus, multiple audio tracks, and bonus features. Viewers choose their language, activate subtitles, discover deleted scenes. It's no longer just a film you watch: it's an experience you explore.

Manufacturers learned from the past. Every DVD player can read existing CDs. This backward compatibility reassures consumers who keep their collections of audio CDs and game CD-ROMs. The transition happens painlessly, without the obligation to replace everything. Microsoft integrates DVD support in Windows 98, standardizing its use in the PC world.

Computing benefits from this story. DVD-ROM drives progressively replace their predecessors in computer towers. Video game and multimedia software developers suddenly have generous spaces for their creations. Myst and Riven had to make do with multiple CDs; their successors will fit on a single DVD.

The market becomes complicated with the arrival of recordable formats. DVD-R appears in 1997 for single-write use. Then come DVD-RAM, DVD-RW, DVD+RW, each with its supporters and technical specificities. This profusion creates confusion but also stimulates innovation and drives prices down. Consumers sometimes get confused, but multistandard burners eventually prevail.

Content protection obsesses the studios. IBM and Disney collaborate to develop the Content Scrambling System, an encryption meant to prevent pirated copies. The CSS system doesn't impact playback perfor-

mance but gives film producers the guarantee they demand. Without this protection, many of them would likely have been reluctant to adopt the format.

The sales figures speak for themselves. Between 1997 and 2002, more than 27 million players found buyers in the United States. The entry price plummeted: more than $1,000 in 1997, less than $200 in 2001. This lightning-fast democratization sealed the fate of VHS. Video stores transformed their shelves, households renewed their equipment.

The DVD attempted a few ventures outside its natural territory. DVD Audio, with mixed success, wanted to dethrone the CD with superior sound quality and multimedia functions. DVDPlus, also called DualDisc, offered a DVD side and a CD side on the same disc. These variations enriched the ecosystem without really taking hold, as DVD remained primarily associated with home video.

Beyond entertainment, the format found its place in education and professional training. Its ability to combine high-quality video, multichannel sound, and interactivity made it a valued pedagogical tool. Libraries used it to digitize and distribute their audiovisual collections. Companies distributed their product catalogs on interactive DVDs.

From the early 2000s, laboratories explored the future. Holographic DVD promised 100 GB per disc. This research prepared the arrival of Blu-ray and high definition. The DVD accomplished its mission: democratizing digital video, standardizing practices, preparing the ground for subsequent generations.

GPU

In the 1980s, graphics processing remained the domain of rudimentary VGA controllers. These components merely received image data, organized it, and transmitted it to a monitor. Nothing more.

The IBM Professional Graphics Controller was released in 1984. This card integrated an Intel 8088 microprocessor specifically dedicated to graphics tasks, thereby freeing up the main processor. At $5,500 and with limited compatibility, it achieved only limited distribution. But the idea took root: entrusting graphics calculations to a specialized processor.

The OpenGL standardization by SGI stimulated an entire ecosystem of hardware solutions dedicated to graphics rendering. The following decade accelerated the movement. SGI introduced the RealityEngine in 1993,

whose architecture established the foundations of the modern graphics pipeline. This pipeline principle, which transforms 3D coordinates into 2D pixels through successive stages, still structures GPU operation today.

Then came the 3dfx Voodoo in 1996. This consumer card dedicated to 3D acceleration revolutionized video gaming on personal computers. One million transistors, 4 MB of 64-bit DRAM memory, a frequency of 50 MHz: these specifications delivered unprecedented performance at the time.

NVIDIA made a strong impact in 1999 with the GeForce 256. It was the first card marketed under the GPU name. This terminological choice was far from trivial: it emphasized the complete integration of graphics processing functions on a single chip, including geometric transformation and lighting. Its 23 million transistors and 32 MB of memory inaugurated the modern era of the graphics processor.

The early 2000s brought programmability. In 2001, developers could program certain pipeline stages via shaders on NVIDIA's GeForce 3. This newfound flexibility opened horizons for visual effects and 3D rendering.

The year 2002 saw the arrival of a generation of fully programmable GPUs with the GeForce FX and the Radeon 9700. Developers could now program operations per pixel and per vertex, precisely controlling graphics rendering. The GeForce FX, with its 80 million transistors and 128 MB of DDR memory, testified to the growing complexity of these components.

NVIDIA's GeForce 8800 in 2006 marked a turning point. Its unified architecture replaced specialized processing units with versatile processors capable of executing different types of graphics calculations. CUDA accompanied this evolution: this programming environment authorized GPU usage for general calculations, well beyond simple graphics.

The Fermi architecture, which NVIDIA unveiled in 2009, confirmed this orientation toward general-purpose computing. The first GPU architecture designed for scientific computing, it integrated a cache memory hierarchy, ECC error correction, and improved double-precision performance. GPUs moved closer to conventional processors while retaining their massively parallel processing power.

This convergence trend could be observed everywhere. AMD launched its Fusion line in 2011, which integrated GPU and CPU on a single chip. Intel developed Larrabee, an architecture combining x86 cores with wide vector units. The convergence between graphics and general-purpose processors accelerated.

Modern GPUs embody the power of parallel processing. Their architecture, the result of continuous evolution from the simple graphics controllers of the 1980s, now runs an extensive range of applications:

3D rendering, scientific computing, artificial intelligence. This versatility, combined with considerable computing power, makes GPUs indispensable components of contemporary computer systems.

Microsoft Internet Explorer

The mid-1990s saw the Web take off. The first mainstream browsers appeared, and Microsoft decided to join the fray. On August 16, 1995, the Redmond company launched Internet Explorer 1.0. This marked the beginning of an adventure that would last more than twenty years and leave its mark on Internet history.

Internet Explorer was not born in Microsoft's laboratories. The company took a shortcut by licensing the source code of Spyglass Mosaic, a browser developed by Spyglass. Other companies made the same choice at the time. Spyglass Mosaic was based on the work of the National Center for Supercomputing Applications, which had created NCSA Mosaic, the first graphical browser to achieve widespread success.

Initially, the team assigned to the project was tiny: half a dozen developers at most. Microsoft distributed its browser in several ways. It was included in the Microsoft Plus! pack for Windows 95, a set of extensions and utilities sold separately. Computer manufacturers received an OEM version to preinstall on their machines. Users could install it via the Internet Jumpstart Kit provided with the Plus! pack. An interesting point: it ran on the original version of Windows 95, without any system updates.

A few months later, Microsoft released Internet Explorer 1.5 for Windows NT. This version added basic support for HTML tables, an element that was becoming essential for building Web pages.

Microsoft's strategy evolved quickly. The company realized that the Internet was not a passing fad and changed its approach. In subsequent versions of Windows 95, the browser became increasingly integrated with the operating system. This growing integration would trigger controversies that would mark the browser's history.

Some Microsoft's internal documents reveal that the company had not planned to include a browser with Windows 95. In early 1994, the plans mentioned only the addition of basic elements for connecting to the Internet: the TCP/IP stack and support for PPP and SLIP protocols. Steven Sinofsky wrote in June 1994 that no client software like Mosaic or Cello was planned for Windows 95.

The change occurred in stages. In 1995, Microsoft was still hesitating. Should the browser be marketed in the "Plus!" pack or integrated directly into Windows? Internal discussions show questions about the relevance of this integration, particularly due to space constraints on installation diskettes.

Technically, Internet Explorer 1.0 reflected its era. The graphical interface remained simple, the features limited. Users navigated page by page, without tabs or ad blocking. But this matched the needs of 1995 users who were discovering the Web.

Computer manufacturers contributed to the browser's distribution. Microsoft signed agreements with them to preinstall Internet Explorer on new machines. This approach ensured massive distribution of the software.

The software architecture laid the groundwork for future versions. The code inherited from Spyglass Mosaic was reworked to adapt to Windows. Developers modified the interface and optimized performance for Windows operating system. Documentation shows, however, that Microsoft considered Internet Explorer as a product distinct from Windows. Internal and external communications presented the browser as a standalone application, with no connection to the operating system.

This first version constituted a milestone in Web history. Microsoft entered a rapidly expanding field with a pragmatic approach: acquiring existing technology rather than developing everything in-house.

Java

In 1991, a team of engineers at Sun Microsystems launched the "Green" project under the leadership of James Gosling. Their ambition was to create a distributed system that would enable electronic devices to communicate with each other in a heterogeneous network. The idea was to offer consumer electronics manufacturers a technology suited to their needs.

The engineers began their work with C++. But the need to adapt code to different processors led them to consider compiler modifications. Despite possible extensions, C++ proved too complex to meet their requirements. This dead end led them to design a new language: Oak, named after the tree that Gosling could see from his office window.

Oak was created to program interactive television control devices and home automation systems. After several months of intense work, the team

produced a complete operating system, a toolkit, an interface, an innovative hardware platform, and three custom chips. All of this culminated in the "*7" (Star 7), which Gosling described as a "handheld remote control".

The Green team, which became FirstPerson Inc., responded to a 1993 call for tenders for a set-top box operating system and video-on-demand technology. Their solution stood out for its technical excellence, but the market went to SGI for reasons that Gosling considered non-technical. FirstPerson persisted in the set-top box sector until 1994, when reality set in: this market was not viable. The company disappeared, and half of its staff joined Sun Interactive to work on digital video data servers.

Everything changed in mid-1994. The Internet and the World Wide Web exploded. The team then decided to reorient Oak toward web applications. The language's characteristics, particularly its architectural independence and platform neutrality, were remarkably well-suited to this environment.

In January 1995, Oak became Java, a name available for registration. The language transformed into a tool for creating web applications. To demonstrate its potential, the team designed HotJava, a web browser written entirely in Java, capable of executing Java applications (Applets) directly integrated into web pages. Netscape and Microsoft would adopt this functionality. Sun then released the initial development kit (JDK) along with HotJava.

Java's simplicity was immediately striking. Code could be read and written easily, reducing the risk of errors. The runtime environment automatically handled memory allocation and garbage collection, eliminating a major source of bugs that plagued C or C++.

Java enforced a rigorous object-oriented approach that structured and modularized programming. Its interpretation allowed programs to run on various platforms without recompilation, thanks to the Java Virtual Machine (JVM). Programs compiled to *bytecode*, an intermediate format independent of hardware. This characteristic guaranteed the portability of Java applications on any system equipped with a Java interpreter.

The language's robustness stemmed from its strong typing system and thorough compilation checks. The absence of pointers, those recurring sources of problems in other languages, reinforced program reliability.

Java then evolved into several editions. Java 2 Standard Edition (J2SE) targeted standard desktop computers. Java 2 Enterprise Edition (J2EE) addressed enterprise applications. Java 2 Micro Edition (J2ME) covered mobile and embedded devices.

Improvements followed one after another. Java 1.02, released in 1996, suffered from significant limitations, such as the inability to print. Sub-

sequent versions progressively enriched the functionalities. Java 1.2, published in December 1998 and renamed Java 2 three days later, notably introduced more sophisticated graphics libraries.

The developer community embraced Java for various reasons. The language simplified the creation of network applications, a legacy of its initial design for distributed systems. Its integrated security model protected against malicious programs. Its multitasking capabilities facilitated the development of high-performance and responsive applications.

The architecture of the Java Virtual Machine testified to the technical sophistication of the system. The class loader, method area, heap, and Java stack composed this runtime environment. The execution engine combined an interpreter and a Just-In-Time compiler that optimized performance by translating *bytecode* into native code.

Java transformed web programming by introducing applets, which executed code in browsers. This innovation enriched the user experience on the Internet and inspired new approaches to web development.

The "Write Once, Run Anywhere" philosophy influenced the design of subsequent languages and platforms. Java's security model, automatic garbage collector, and object-oriented programming approach established standards that endure in contemporary software development.

JavaScript

In 1995, the Web was undergoing rapid expansion. Netscape and Microsoft were engaged in an intense commercial battle over their respective browsers. Brendan Eich, an engineer at Netscape, received an unusual assignment: design a programming language in ten days, a tool that would execute directly in the Netscape browser. But this request was part of a broader strategic vision. Netscape no longer viewed the browser as a simple application; the company considered it, along with the server, as a new form of distributed operating system.

At that time, Sun Microsystems' Java was establishing itself as the reference solution for complex Web applications. This compiled language generated *bytecode* for its virtual machine, adopted object-oriented principles from C++, and promised performance comparable to native languages. However, Netscape identified a gap: a second language was needed, lighter and interpreted, that would complement Java. An accessible tool for amateur programmers that would integrate seamlessly

into Web pages.

Brendan Eich then designed JavaScript as a language with multiple influences. The syntax borrowed braces and semicolons from C, along with data structures, while the models drew inspiration from Smalltalk. The symmetry between data and code, characteristic of LISP, found its place in the design. For event handling, Eich turned to HyperCard. As for the object-oriented approach, it relied on runtime semantics using prototypes, in the manner of the Self language, rather than on class syntax like Java.

The language's early years remained modest. JavaScript was mainly used to scroll messages in the browser's status bar or animate a few images. Nothing particularly spectacular. But the language contained enough fundamental elements to endure. Its initially limited adoption gave it time to evolve gradually, particularly through the ECMA standardization process, which improved its performance and robustness.

The real turning point came between 2004 and 2005 with the arrival of Ajax. JavaScript could now retrieve data from servers and update HTML documents without reloading the entire page. Microsoft introduced this functionality in Internet Explorer via the XMLHttpRequest object, which was adopted by other browsers. The user interface partially shifted to the browser, creating much richer experiences. Gmail and Google Maps illustrated this transformation.

This change quickly revealed the weaknesses of browser JavaScript engines. Web pages no longer restarted every few minutes. They maintained extended sessions, manipulated large volumes of dynamic data, and communicated continuously with servers in the background. Google developed Chrome and its V8 interpreter to meet these new performance requirements. The market followed suit. JavaScript interpreter performance improved across all vendors.

In 2012, Mozilla launched asm.js, a strict subset of JavaScript optimized for executing code compiled from C or C++. Thanks to this innovation, complex games were ported to the Web. Epic Games provided a spectacular demonstration by adapting its Unreal Engine 3 to JavaScript in just a few days. The success of asm.js would directly influence the creation of WebAssembly, a more efficient binary format for executing native code in browsers.

Node.js then expanded JavaScript's playing field beyond the browser. The language moved into server application development. Its event-driven nature, present from the beginning, facilitated the creation of highly scalable Web applications without the complexity of multithreaded programming.

Ongoing standardization through ECMAScript regularly brought new

features. ES6 modules are one example. All while maintaining the backward compatibility essential to the Web's longevity. JavaScript gradually transformed from a simple animation tool into a versatile language capable of handling complex applications, whether client-side or server-side.

The ecosystem surrounding it constantly enriched itself. Transpilers enabled the use of the latest features while remaining compatible with older browser versions. Package managers like npm facilitated code sharing among developers. React, Angular, and Vue.js standardized the development of complex user interfaces.

JavaScript's success owes much to its accessibility and adaptability. Despite its initial imperfections, linked to its rushed creation, the language managed to respond to the Web's changing needs. Its direct interpretation, flexibility, and native integration in browsers make it a central element of modern Web development.

JavaScript's dominant position does not, however, signal the end of its evolution. WebAssembly opens new perspectives as a universal compilation format for the Web. Other languages can now execute in the browser. JavaScript nevertheless retains its role as the privileged interface with the DOM and Web APIs. It demonstrates its ability to coexist with new technologies.

PHP

In 1994, Rasmus Lerdorf, a Canadian programmer of Greenlandic origin, developed a few Perl scripts to manage visits to his online CV. These tools, dubbed "Personal Home Page Tools", initially served to display his curriculum vitae and track traffic. Nothing spectacular at first, just a practical solution to a personal need.

Working as a web development consultant in Toronto, Lerdorf noticed he was constantly rewriting the same CGI scripts in C for his various clients. This repetition led him to consolidate his code into a C library that he integrated with the NCSA web server. He added a template system to simplify calls. The idea remained pragmatic: save time and work more efficiently.

On June 8, 1995, he released the first public version under the name "Personal Home Page Tools version 1.0." He had rewritten his scripts as CGI binaries, adding functionality to process web forms and interact with databases. The whole thing then took the name "Personal Home

Page/Forms Interpreter", or PHP/FI. Lerdorf mainly sought to detect bugs faster and improve his code through feedback from other developers.

This first iteration already contained Perl-inspired variables, form handling, and the ability to embed directly in HTML. The syntax borrowed from Perl while remaining more accessible, though consistency wasn't always present. Web developers began taking interest, attracted by this simpler way of creating dynamic pages.

Lerdorf didn't consider PHP his property. Rather, he saw it as a tool for solving problems. When other developers asked to use his scripts, he willingly shared them. His value lay in his ability to solve his clients' problems, not in exclusive code ownership.

The real turning point came in 1997. Zeev Suraski and Andi Gutmans, two Israeli developers from Technion IIT, rewrote the parser. This overhaul gave birth to PHP 3, where the name became a recursive acronym: "PHP: Hypertext Preprocessor." Public testing began, and the official version was released in June 1998. The two Israelis continued their work by rewriting PHP's core, thus creating the Zend engine in 1999. They founded Zend Technologies in Ramat Gan.

PHP 4 was released on May 22, 2000, powered by Zend Engine 1.0. This version significantly improved object-oriented programming support and overall performance. The language matured and established itself as a real tool for building dynamic websites.

Version 5 arrived on July 13, 2004, with Zend Engine II and a new object model. Object-oriented programming became truly usable, and PHP Data Objects (PDO) offered a unified interface for database access. In 2008, PHP 5 was the only actively developed stable version.

PHP's history shows how much the community matters in its evolution. The language grew richer through contributions from numerous developers. The PHP Extensions Community Library (PECL) served as an incubator where groups of volunteers experimented with new ideas before they eventually integrated into the main distribution.

Project management followed a meritocratic logic. Code took precedence over theoretical debates. If a developer proposed a working implementation, it would probably be adopted, regardless of its degree of perfection. This philosophy accelerated innovation and feature additions, sometimes at the expense of overall consistency.

PHP's success owed much to its ease of use and proximity to HTML. Developers could directly insert PHP code into their web pages, making the language accessible to beginners. This ease of learning largely explains its growing popularity.

Development continued with PHP 7.0 in 2015. The Zend Engine 3 significantly boosted performance. Execution speed improved dramatically, and new features appeared such as scalar type declarations and the combined comparison operator.

PHP 8.0 was released in 2020, marking a new stage with the JIT (Just-In-Time) compiler, union types, and match expressions. These developments strengthened the language's robustness without sacrificing its ease of use.

Over the years, PHP has become essential in web development. Its evolution reflects changes in programming practices and the needs of the modern web. From a simple personal tool, it has transformed into a complete language supporting object-oriented programming, exception handling, and numerous advanced features.

Ruby

On February 24, 1993, two Japanese computer scientists, Yukihiro Matsumoto and Keiju Ishitsuka, discussed online what to name their new programming language project. They wanted a gemstone. After considering "Coral", they chose "Ruby". One character shorter, thus simpler to type.

Yukihiro Matsumoto, whom everyone calls Matz, wanted to create a truly enjoyable object-oriented scripting language. He borrowed flexibility from Perl, object-oriented vision from Smalltalk, rigor from Eiffel, reliability from Ada, and expressiveness from Lisp. From this blend emerged a language distinguished by its elegance.

The first public version, 0.95, was released on December 21, 1995 on Japanese forums. Success in Japan was immediate. In 1997, netlab.jp, a Japanese company specializing in open source, hired Matz to work full-time on Ruby. Version 1.0 appeared in 1996, version 1.1 in August 1997. Development accelerated.

Matsumoto designed Ruby according to the principle of least surprise. Methods bear names drawn from common English vocabulary that naturally describe their function. For character strings, we find verbs like `strip`, `split`, `delete`, or `upcase`. Learning is intuitive, code maintenance easier.

In Ruby, everything is an object. Truly everything, including numbers and primitive types. This approach, inherited from Smalltalk, unifies

the syntax. Developers apply the same principles to all elements they manipulate. The language gains consistency.

Ruby allows modification of its essential components, enabling programmers to redefine and enrich existing classes and add methods to built-in classes. For example, one can enrich the number class with a `plus` method to make code more expressive. This flexibility appeals.

In 2001, Ryan Leavengood began developing RubyGems, a tool to simplify library distribution. The project stopped when Leavengood left at version 0.4.0, but a new team took it over in 2003 and completely rewrote it. RubyGems then became a central element of the ecosystem.

The year 2004 changed everything. David Heinemeier Hansson created Ruby on Rails, a web framework that propelled Ruby onto the international stage. This platform attracted developers with its pragmatic approach and productivity. Apple integrated Ruby on Rails into Mac OS X v10.5 "Leopard". This recognition by an industry giant accelerated the language's adoption.

Ruby's popularity peaked in 2006, when it received the title of "Programming Language of the Year". In 2016, it reached eighth place in the TIOBE index. Between 2013 and 2020, Ruby evolved from version 2.0.0 to version 3.0.0. The language reached technical maturity.

The community remains dynamic. In 2022, there were 2.4 million Ruby developers worldwide. Airbnb, Shopify, Coinbase, and GitLab use Ruby and contribute to its development. The Rails Foundation, recently created, works on documentation, training, and language promotion.

Version 3.0, released in 2020, brought significant performance improvements. The "Ruby 3x3" project aims to triple the performance of version 3.x compared to Ruby 3.0. Developers optimize the MJIT compiler, refine the garbage collector, and improve concurrency support through Ractors.

Several alternative implementations enrich the ecosystem. JRuby runs on the Java virtual machine, TruffleRuby offers high performance thanks to GraalVM. Matz leads the development of mruby, a lightweight version designed for embedded systems. These variations expand the language's usage possibilities.

SSH

In 1995, at the University of Helsinki, Tatu Ylönen had just experienced a network attack on his institution's systems. Passwords were

circulating in plain text, intercepted by attackers exploiting the glaring vulnerabilities of protocols like Telnet, rlogin, or rsh. Faced with this concrete threat, the Finnish researcher decided to design a radically different solution: a protocol capable of encrypting communications and robustly authenticating the parties involved. SSH, for Secure Shell, was born from this immediate necessity to protect exchanges over the Internet.

The first version was released in July 1995 as free software. The success was immediate in academic and technical circles. Everyone understood the urgency of securing remote connections, and SSH met this need. Given this enthusiasm, Ylönen founded SSH Communications Security in December 1995. The company commercialized professional versions while maintaining a free version for non-commercial uses, a delicate balance between economic logic and broad dissemination of the protocol.

The following year marked a turning point with SSH-2, a complete redesign of the initial version. This new iteration adopted a three-layer architecture: the transport protocol handled encryption and server authentication, the authentication protocol verified user identity, and the connection protocol multiplexed communication channels within a single secure tunnel. This modularity brought increased flexibility and robustness.

The IETF launched a working group dedicated to SSH standardization in 1997. Between 2004 and 2006, several RFCs specified SSH-2 standards. This standardization promoted interoperability between implementations and established SSH as the de facto standard for remote system administration. But an event would disrupt the ecosystem in 1999: the appearance of OpenSSH, a free implementation developed within the OpenBSD project. This free and open alternative accelerated the universal adoption of the protocol. OpenSSH quickly became the reference, integrated by default into most UNIX and Linux systems.

On the technical level, SSH relies on a hybrid approach combining asymmetric and symmetric cryptography. When establishing the connection, public-key mechanisms authenticate and exchange keys. Once the link is established, symmetric encryption takes over to protect data with better performance. The protocol introduced the "known hosts" mechanism: clients memorize the public keys of contacted servers, thus detecting any man-in-the-middle attack attempts. SSH also enabled tunneling of other TCP/IP protocols via port forwarding, considerably extending its scope of application.

The flexibility of SSH-2 allowed support for various authentication methods: passwords, public keys, host-based authentication. This adaptability made it possible to calibrate security according to contexts. Data

compression and channel multiplexing optimized bandwidth usage, making the protocol even more efficient.

In the 2000s, SSH established itself as a key building block of Internet infrastructure. Its use extended far beyond remote access to encompass secure file transfer with SFTP, entire system management, and application deployment. Companies massively adopted it to secure their administrative operations. The protocol continued to evolve, integrating more recent cryptographic algorithms such as AES and elliptic curves. The developer community remained mobilized to adapt SSH to emerging threats and use cases.

SSH's impact on modern computing remains considerable. As a practical and secure solution for remote administration, the protocol has accompanied the growth of the Internet and the explosion of cloud computing. Its modular design has inspired other security protocols. SSH remains ubiquitous in development environments, data centers, and cloud services. It illustrates how a technical response born from a specific incident can become an indispensable international standard.

SSH's lasting success lies in its ability to combine robust security with ease of use. By automating cryptographic complexity while remaining accessible, the protocol has democratized security best practices in system administration.

Lynx

When discussing web browsers, the image that spontaneously comes to mind is that of a colorful, animated, interactive graphical interface. Yet, there has existed since 1992 a program that defies this representation: Lynx, a fully text-based browser born at the University of Kansas. Its history reveals how accessibility and simplicity can span three decades without aging a day.

Initially, Lynx was nothing like a web browser. Developers at Kansas were seeking to build an interface to access their local information system. Hypertext was gradually grafted onto it, then the Gopher protocol made its appearance. Only after these successive additions did Lynx learn to speak the language of the Web. This layered genesis explains the code structure: a mosaic of pieces assembled over time, without an initial master plan. Some might see this as a flaw; others, the mark of organic adaptation to real needs.

The program's name refers to the lynx, a feline renowned for its keen vision. The analogy is not trivial: where other browsers multiplied visual ornaments, Lynx got straight to the point. Early users discovered a raw, stripped-down, yet functional Internet. This deliberate bareness proved a major asset for blind or visually impaired people, who could finally access the web via screen readers.

Navigation in Lynx relies on the keyboard. Arrow keys move the cursor between links, the Enter key activates the selected one, and a few shortcuts provide access to advanced features. Nothing more. This economy of means nevertheless conceals genuine technical sophistication. The program can handle HTML forms, understands cookies, and offers extensive configuration via the .lynxrc file. You can customize colors (yes, even in text mode), define your preferences, or hook up your favorite text editor to modify a page directly.

Image handling well illustrates the project's philosophy. Rather than ignoring them outright, Lynx displays their alternative text—that "alt" attribute that web developers too often forget. This approach, born from a technical constraint, has become a reference point for accessibility. It forces one to ask: what does someone who cannot see actually see?

In today's web, saturated with JavaScript and complex style sheets, Lynx retains its usefulness. System administrators working on the command line use it regularly. Low-bandwidth connections benefit from it. And then there are those who prefer to read without being distracted by animations or flashing advertisements. That said, the reading mode of today's browsers serves this purpose perfectly well too.

Web developers still use Lynx as a diagnostic tool. Displaying a page in this browser means seeing its bare HTML skeleton. If the content is comprehensible under these conditions, that's a good sign. This method is part of what's called progressive enhancement: starting from a simple functional base, and enriching the experience for those with more advanced technical means.

That Lynx has survived this long is no accident. Its open source code and community of developers have managed to evolve it without betraying its founding principles. Updates maintain compatibility with modern standards, while the interface remains faithful to the original spirit. Few software programs can pride themselves on such consistency over three decades.

MySQL

In 1979, Michael "Monty" Widenius was working at TcX, a small company, when he developed a reporting tool in BASIC. The hardware constraints of the era were severe: 16 KB of RAM, a 4 MHz processor. In this Spartan environment, Widenius learned to write lightweight and fast code. This experience shaped his approach to software development. His position as co-owner at TcX gave him a rare freedom for a programmer: control over his code. This combination of technical skill, intellectual property, and autonomy would prove crucial.

During the 1990s, TcX clients demanded an SQL interface to manipulate their data. Widenius explored several options: buying a commercial database license, integrating mSQL code... None truly satisfied him. He decided to create his own solution. In May 1996, MySQL version 1.0 was released, but remained confined to a limited circle. The first public version (3.11.1) arrived in October 1996, initially only for Solaris. A month later, Linux was supported.

MySQL gradually extended to other operating systems. Its features grew richer with each version. The distribution model relied on a particular license: commercial use was free as long as you didn't redistribute the software with your own products. TcX also sold technical support. These revenues funded development.

MySQL version 3.22 already offered a substantial portion of the SQL language. Its optimizer was impressive, especially for a project driven mostly by a single person. The system excelled in speed and stability. APIs multiplied, making it usable from almost any programming language. But gaps remained: no transactions, no subqueries, no foreign keys, no stored procedures or views. Table-level locking sometimes considerably slowed operations.

Around 1999-2000, MySQL AB emerged as a separate company. The team grew with the recruitment of developers. A partnership with Sleepycat was formed to create an SQL interface to Berkeley DB files. The goal was to provide MySQL with transactional capabilities. Version 3.23 resulted from this work.

The Berkeley DB integration didn't go as planned. Stability was never really achieved. But this effort wasn't in vain: MySQL's source code was now equipped to accommodate different storage engine types. In April 2000, with support from Slashdot, master-slave replication was introduced. The old ISAM engine was reworked to become MyISAM, which brought improvements including full-text search.

Around the same time, Heikki Tuuri proposed integrating InnoDB,

his own storage engine with similar functionality. The table management interface, born from the Berkeley DB work, facilitated this integration. MySQL version 4.0, combined with InnoDB, was declared stable in March 2003.

But the real innovation of 4.0 was the query cache. This feature drastically improved the performance of many applications. The replication code on the slave was rewritten with two threads: one for network I/O from the master, the other to process updates. The optimizer gained efficiency. The client/server protocol became SSL-compatible.

During the development of 4.1, work on the 5.0 branch progressed. This version brought stored procedures, server-side cursors, triggers, views, XA transactions, and major optimizer improvements. Why create a separate branch? The developers wanted to prevent the addition of stored procedures from delaying the stabilization of 4.1. Version 5.0 was released in alpha in December 2003. Two alpha branches temporarily coexisted, creating some confusion.

Between 2005 and 2006, Oracle successively acquired Innobase and then Sleepycat, the two transactional storage engine providers for MySQL. These acquisitions raised concerns in the community. Larry Ellison, Oracle's CEO, casually commented that he had spoken to almost every company in the sector. He described MySQL as a tiny company with revenues oscillating between 30 and 40 million dollars, compared to Oracle's 15 billion in annual revenue.

Yet MySQL AB's commercial trajectory followed an upward curve. In 2001, Mårten Mickos took over as CEO and transformed a technical startup into a viable business. Revenues climbed from 6.5 million dollars in 2002 to 75 million in 2007. The business model initially relied on dual licensing for OEM integrators, then shifted toward support subscriptions for end users with the launch of MySQL Network in 2005. The company counted 8 million active installations in 2006, employed 320 people spread across 25 countries, 70% of whom worked from home. That year, Mickos announced preparations for a 2008 IPO with a target of 100 million in revenue.

But Sun Microsystems struck first. In January 2008, the California-based company announced the acquisition of MySQL AB for approximately 1 billion dollars. Jonathan Schwartz, Sun's CEO, called this the most important operation in his company's history. He saw it as an opportunity to reposition at the heart of the web economy. The numbers spoke for themselves: 75% of MySQL installations ran on hardware from manufacturers other than Sun, 80% used Linux rather than Solaris. The acquisition gave Sun access to a massive installed base to which it could

sell servers and software. With this billion dollars, MySQL established a new valuation standard for open-source software companies.

The integration at Sun proved chaotic. Widenius and Axmark, two of the three founders, left the company shortly after the acquisition. In 2009, Mickos resigned in turn to become entrepreneur-in-residence at Benchmark Capital. Within a few months, Sun lost the technical and commercial leaders who had built MySQL's success.

That same year, Oracle announced the acquisition of Sun for 7.4 billion dollars. MySQL thus became the property of the main commercial competitor in the database field. This prospect raised many questions. Would Oracle continue to develop MySQL or seek to stifle it to protect its own database? Could companies that depended on MySQL still trust it? The European Commission closely examined the operation before giving its approval, not without demanding guarantees on maintaining MySQL's development.

Widenius himself didn't remain idle. As early as 2009, he started developing MariaDB, a fork of MySQL. This new database maintained full compatibility with MySQL while adding its own improvements. The nonprofit MariaDB Foundation was created in 2012. The name this time came from Maria, the founder's other daughter. This fork gradually became the reference alternative for those wary of Oracle. Wikipedia switched to MariaDB, Google too. Linux distributions began replacing MySQL with MariaDB in their default repositories.

MySQL nevertheless continued its evolution under Oracle's governance. The team remained largely the same as before the acquisition. NoSQL features appeared. Oracle maintains MySQL under dual GPL and commercial licensing.

RealPlayer

In April 1995, the Internet was breaking into households worldwide. 56k modems were being deployed and struggled to transmit .wav files, and no one yet imagined being able to listen to music or watch a video without waiting for it to download completely. Rob Glaser, who had just left Microsoft after ten years working on applications and multimedia, saw things differently. He founded RealNetworks with an idea that seemed almost crazy at the time: streaming audio directly from the Internet.

The first RealAudio Player disrupted usage patterns from its release.

For the first time, people could listen to audio content via streaming, a word that wasn't yet part of everyday vocabulary. Success came quickly: three years later, Microsoft integrated the player into Windows 98. This achievement becomes even more impressive when you realize that YouTube wouldn't appear until ten years later.

The RealNetworks teams worked on two fronts. On one hand, they developed compression techniques that delivered decent quality despite ridiculously low bandwidth. On the other, they invented an original way of combining connection-based and connectionless Internet protocols to smooth out streaming. It was sophisticated tinkering, but it worked.

The company chose a strategy that would later become common: a free player for users, paid servers for those wanting to broadcast. Jim Breyer, from Accel Partners, joined the board of directors and investors poured in. In 1996, against the advice of some team members, RealNetworks launched a premium version of the player. The gamble paid off handsomely: more than three million copies sold, most of them online.

Video arrived in 1997 with RealSystem. Connections remained slow, but Glaser anticipated their improvement. To prove his technology was solid, he partnered with Spike Lee, who directed the first short films designed for the Internet. More than 500,000 people watched them in a month. The message got through.

The software changed names several times over the years—RealAudio, RealOne Player, RealPlayer G2—each version bringing its share of new features. Format conversion, video downloading, playlists, audio extraction... The features piled up. In 2015, the company tried something different with RealTimes, a mobile app for creating video montages from photos. It was a way of keeping up with new usage patterns, while having its core business elsewhere.

Glaser retained from his time at Microsoft a certain vision of convergence between technology and media. He pushed his company toward constant innovation, even at the risk of taking chances. In 2002, RealNetworks launched Helix, a platform capable of streaming any audio or video format. The company even opened part of its source code to developers, to encourage collective improvement of solutions.

RealPlayer transformed our relationship with online media, moving us from tedious downloading to instant streaming. Current platforms owe it a great deal, though few people remember this. The company lasted twenty-five years by constantly adapting.

Sun UltraSPARC

Research conducted at Berkeley between 1980 and 1984 on RISC architectures, carried out in parallel with Stanford's work on MIPS, led to the creation of Scalable Processor ARChitecture (SPARC) by Sun Microsystems in 1985. Four years later, Sun founded SPARC International to open this architecture to other manufacturers.

The first version, SPARC V7, was released in 1986. V8 arrived in 1990 with hardware support for multiplication and division, an MMU, and 128-bit floating-point operations. V9 represented a major milestone: it handled 64-bit registers and operations, redesigned exception handling, and integrated prefetching.

In 1995, Sun launched the first UltraSPARC, the first processor to integrate SPARC V9. Marc Tremblay participated in its design. This superscalar microprocessor executed instructions out of order and processed four simultaneously. Its pipeline had eight stages. Engineers had simplified the execution unit compared to the SuperSPARC to gain frequency, particularly by modifying the ALU branching.

The processor featured 32 64-bit registers divided into eight windows, totaling 144 registers. This technique prevented called functions from wasting time saving and then restoring registers. Seven input registers and three output registers provided access to two ALUs and the memory management unit. A single ALU handled multiplication and division.

The floating-point unit was divided into five blocks: addition and subtraction, multiplication, division and square roots, followed by two blocks dedicated to SIMD instructions of the Visual Instruction Set. Thirty-two 64-bit registers served these operations, including five for inputs and three for outputs.

The primary cache was divided into two 16 KB sections, one for instructions and one for data. An external unified secondary cache was to complement the system, with a capacity between 512 KB and 4 MB, accessible in one cycle. This cache used synchronous SRAM clocked at the processor speed.

Texas Instruments manufactured UltraSPARC using the EPIC-3 CMOS process at 0.5 µm on four metal layers. The manufacturer abandoned BiCMOS, which offered little advantage at this level of miniaturization. The chip integrated 3.8 million transistors in a 521-pin PBGA package.

Ten years later, the UltraSPARC T1 broke with this approach. Sun embodied its concept of *throughput computing*, which favored multiplying cores and threads over increasing frequencies or making pipelines more complex. In its full version, the T1 had eight cores each executing four

threads, for a total of 32 threads. Sun sacrificed floating-point performance and cache size in favor of parallelism.

The T1 used a short in-order execution pipeline: fetch, thread selection, decode, execute, memory, and write-back. Its ALUs operated with a latency of one cycle, multipliers and dividers over multiple cycles. A single floating-point unit was shared among the eight cores. Each thread had its own instruction buffer.

The processor switched between threads according to several rules. By default, it alternated between available threads using an LRU policy. Long instructions triggered a thread switch to keep the pipeline active. The L1 cache was flow-through, with the L2 including the L1. The T1 deliberately made do with a reduced cache: the large number of threads masked memory latencies.

The UltraSPARC T1 embedded a hypervisor that added a privilege level above user and supervisor modes. This lightweight software layer offered a complete virtualization interface to guest systems. Sun fully documented this API and released the T1 VHDL sources under an open license, renamed OpenSPARC T1.

The T1 consumed no more than 70 watts, half that of a Xeon or an Itanium. This efficiency mattered for data centers where air conditioning could cost more than the hardware itself.

This architecture suited web servers, client-server applications, and certain databases that exploited its high parallelism. Sun acknowledged that scientific workloads, especially those making heavy use of floating-point operations, were not these processors' forte.

UltraSPARC reflects the evolution of processor architectures, from early RISC to massively multicore designs. It demonstrates the strategies explored to increase performance: pipeline complexity on one hand, massive parallelization on the other.

Wiki

In 1994, Ward Cunningham invented something that would change how we create content on the Internet. His "Wiki Wiki Web" was initially just a tool to facilitate teamwork at Tektronix, but it would become much more than that.

Cunningham had grown up in the American Midwest during the Sputnik years, tinkering with electronic circuits from childhood. After studying

electrical engineering at Purdue and earning a master's degree in computer science, he became passionate about object-oriented programming. While working with Smalltalk and Lisp, he noticed that these languages produced code that could be understood by reading it anywhere, without needing the full context. This idea of immediate readability would influence him.

The name wiki comes from Hawaiian and means "fast". Cunningham had discovered it while taking the Wiki Wiki shuttles at Honolulu airport. It was exactly what he was looking for: a word that conveyed the speed with which web pages could be modified. Gone were the cumbersome processes requiring writing, review, correction, and finally publication. With the wiki, you wrote, published immediately, and others could review and correct afterward. A real game-changer.

Technical simplicity was at the heart of the system. A simple web browser was enough. No need for complicated software or knowledge of obscure markup languages. The syntax remained basic: to create a link to another page, you simply wrote a word in CamelCase, with a capital letter in the middle. The system automatically created the link.

At launch, Cunningham wrote about a hundred pages on computer programming. This trick gave the impression that a community already existed, that others were contributing. Visitors saw how it worked and felt encouraged to participate. He invited developers to create their personal pages, gradually familiarizing them with collaborative editing.

Developers embraced the concept. Within the first few weeks, clones of the system appeared. The wiki contrasted sharply with traditional content management systems that imposed validation at every level. Here, freedom was total.

Many feared vandalism and inevitable abuse. But the community self-regulated naturally. Cunningham explained it simply: removing inappropriate content required less effort than creating it. This asymmetry favored the emergence of relevant information. The system relied on mutual trust between users, built through complete transparency of modifications and the ability for anyone to contribute.

The wiki's impact exploded in 2001 with Wikipedia. Jimmy Wales took the principles of the Wiki Wiki Web and added an appropriate license and a multilingual global vision. The wiki became a worldwide cultural phenomenon.

Research conducted at the University of Ulster demonstrated the educational effectiveness of wikis. Over 80% of students found the navigation intuitive and appreciated the freedom to edit. Without technical barriers, they focused on content rather than tools.

The wiki introduced lasting technical innovations: page versioning, automatic link creation, full-text search. These features have become standards of modern collaborative work. It proved that an online community could organize itself effectively without rigid hierarchy. This philosophy of self-organization has inspired countless collaborative projects on the Internet.

Thirty years later, the original Wiki Wiki Web is still running. Simplicity and openness remain enduring principles in collaborative technologies.

Microsoft Windows 95

On August 24, 1995, at midnight, lines stretched in front of CompUSA and Best Buy stores. Such a thing had never been seen for software. Microsoft orchestrated the launch of Windows 95 with unprecedented scale: 300 million dollars invested in a worldwide advertising campaign, the Rolling Stones and their "Start Me Up" as the soundtrack, Jay Leno hosting the event alongside Bill Gates. The Redmond company even produced an hour-long show with Jennifer Aniston and Matthew Perry, then at the height of their fame thanks to Friends. The result? Over 40 million copies sold in five weeks.

Behind this media hype lay a radical overhaul of the computing experience. Windows 95 definitively turned the page on MS-DOS and Windows 3.1. The desktop now displayed icons, a taskbar ran along the bottom of the screen, and a Start button concentrated program access. This interface transformed the computer into a tool that anyone could master, without sacrificing the advanced features that interested experienced users.

The system's 32-bit architecture required a compatible processor but unlocked new possibilities. Memory was managed better, resources were allocated more efficiently. Preemptive multitasking entered the scene for 32-bit applications, allowing multiple programs to run in parallel without the system bogging down. However, Microsoft had to solve a delicate equation: making old DOS and 16-bit Windows software coexist with new 32-bit programs. The Win32 API provided an answer by giving developers a consistent set of functions, compatible with both Windows NT and Windows CE. Programmers could thus port their applications from one platform to another without too much trouble.

Plug and Play technology aimed to simplify peripheral installation. In reality, its chaotic beginnings earned the system the ironic nickname "Plug

and Pray". Despite these initial hiccups, the principle of automatic hardware detection later became self-evident. File management also gained flexibility: long filenames, up to 250 characters, replaced the archaic 8-character constraint inherited from DOS. MacOS enjoyed this freedom, and Windows 95 finally caught up. The VFAT file system preserved compatibility with older applications, a compromise typical of Microsoft's approach.

The integrated network interface and native TCP/IP support arrived just as the Internet was really taking off. Microsoft launched simultaneously the Microsoft Network (MSN), a proprietary online service meant to compete with AOL, accessible through a monthly subscription. Internet Explorer was not part of the base system but was added in subsequent versions via the Microsoft Plus! pack. The "Briefcase" feature allowed file synchronization between different computers, foreshadowing the cloud services that would flourish a decade later. User profiles gave each member of a family or business the ability to customize their work environment.

The system included advanced network features: peer-to-peer file sharing, support for NetBEUI and IPX/SPX protocols, clients for NetWare and Microsoft networks. These characteristics facilitated its adoption in businesses, where it gradually supplanted Windows 3.1 and Windows for Workgroups. The competition hadn't given up. Mac OS captivated with its elegance and multimedia capabilities but remained confined to Apple hardware. IBM's OS/2 impressed technically but suffered from poor marketing. Linux, still an emerging free system, attracted technical users with its stability and modularity.

A typical computer of this era featured a 486 or Pentium processor, an IDE or SCSI hard drive, a double-speed CD-ROM drive, and a Sound Blaster sound card. These machines had to contend with Windows 95's requirements: 4 MB of RAM at minimum, 8 MB recommended, and substantial disk space. Microsoft evolved the system through several versions (OSR1, OSR2) that integrated FAT32, USB, and AGP. Windows 95 remained available until 2001, gradually giving ground to Windows 98 and then Windows ME.

The system's impact on the computer industry was considerable. It standardized the modern graphical interface and helped make the computer a familiar object in homes. This period marked the height of Microsoft's dominance in the consumer market. A redesigned interface, a modernized architecture, aggressive marketing: Windows 95 established a standard that influenced operating system development for the following twenty years. The system remains in memory as the one that democratized personal computing.

Microsoft DirectX

Windows 95 posed a significant problem for video game developers. Most titles still ran under DOS, which allowed applications to access hardware directly. With Windows and its secure memory management, graphics performance collapsed. Microsoft risked losing developers if nothing changed.

The solution came from a small company whose source code Microsoft acquired: RenderMorphics. This became the starting point for DirectX. The early versions impressed no one: bugs galore, programming interface of daunting complexity. The system relied on *execute buffers*, packages containing lists of vertices to display and their instructions. Developers hated this approach.

Everything changed with DirectX 5. The DrawPrimitive interface radically simplified programming without sacrificing performance. Microsoft had just created a credible option for developing games under Windows. The video game market was exploding, and the battle between DirectX and OpenGL began in earnest.

John Carmack, the lead architect at id Software, didn't mince words in December 1996. He judged DirectX "terribly flawed", an API that imposed absurd constraints on programmers without providing anything in return. Other prominent developers bombarded Microsoft with letters, all advocating the adoption of OpenGL.

Microsoft organized comparative demonstrations to prove DirectX's superiority. These tests were contested: the comparison methods seemed dubious. Debates often revolved around software rendering performance, an aspect that would quickly lose importance with the arrival of powerful and affordable graphics cards.

DirectX prevailed despite the controversies. The API integrated comprehensive tools for managing sound and input devices. The Apple market weighed little, which reduced interest in cross-platform compatibility. DirectX could also query graphics hardware capabilities, an appreciable asset.

The technical architecture relied on COM (Component Object Model), which allowed its use from C++, C# or Visual Basic. Pure C remained possible, at the cost of increased complexity. DirectX progressed alongside OpenGL in supporting custom *shaders*, giving programmers fine control over the graphics pipeline.

DirectX 8 abandoned DirectDraw, the 2D API that had long been a pillar of the system. Three years later, DirectX 10 arrived with Windows Vista after development conducted in collaboration with application creators and hardware manufacturers.

DirectX 10's major contribution was the *geometry shader*, a new programmable stage in the graphics pipeline. Positioned between vertex processing and rasterization, it could generate up to 1,024 primitives from a single input. The designers maintained certain stages in fixed functionality, preferring performance to total flexibility.

DirectX 10 standardized the processing of different programmable stages via a "common core". This virtual machine unified the management of input-output registers, temporary registers, and resource binding points. It used 32-bit assembly language, but Microsoft encouraged the use of the high-level language HLSL (High Level Shading Language) for programming shaders.

The technology has continued to evolve since. Microsoft wanted to keep Windows attractive to game creators, even if it meant weathering criticism about its business practices and lock-in strategy. DirectX has become an essential industry standard in video gaming.

A contested API at its beginnings transformed into a reference solution through well-conceived technical improvements and a keen understanding of developers' needs. Technical and strategic choices weigh heavily in technology evolution, intertwining with commercial stakes and community dynamics.

ICQ

In 1996, four Israeli developers tinkered with a messaging system that would change how people communicate online. Yair Goldfinger, Amnon Amir, Sefi Vigiser and Arik Vardi, along with Yossi Vardi, the latter's father, weren't trying to create such a tool. Their initial project simply aimed to make checking messages on beepers more convenient. ICQ was born almost by chance, as an auxiliary tool to facilitate their own collaborative work.

At that time, staying permanently connected was a luxury. Modems tied up the phone line, and maintaining an online presence on IRC for chatting wasn't always feasible. The team faced communication difficulties when working remotely. Hence the idea of ICQ, whose name says it all:

I Seek You. The software's primary function was to enable users to find each other online.

Two months were enough to develop the program. Each user received a unique identification number, the UIN, comparable to a phone number. Low numbers indicated early registration, creating a kind of social prestige among early adopters. You could see who was connected, exchange messages in real time, and search for other users through an integrated directory.

The software spread on its own, without a marketing campaign. Online gaming communities were the first to be won over. Players of MUDs, Meridian 59, or Diablo on Battle.net massively adopted ICQ to coordinate their games and stay in touch between sessions. This spontaneous adoption foreshadowed the role Discord would play much later in the gaming universe.

Success attracted America Online, which acquired Mirabilis in June 1998 for $407 million. Under AOL, ICQ continued its expansion while maintaining its identity and its Tel Aviv-based team. By 2001, the service had 100 million registered users, mostly young people between 13 and 29 who spent five hours a day connected.

Versions followed one another with their share of new features. ICQ integrated encryption starting with version 99b, while earlier versions lacked it. Voice calls followed, then video conferencing, file transfer, messages to mobile phones, and ICQphone for calling landlines. In 2013, ICQ became the first service to offer end-to-end encrypted video calls with the ZRTP protocol.

Social networks and mobile messaging reshuffled the deck. In 2010, AOL sold ICQ to Russian group Mail.ru for $188 million. The service was popular in Eastern Europe and Russia, with 42 million daily users. In 2021, Hong Kong experienced renewed interest in ICQ: downloads increased 35-fold in one week after controversies over WhatsApp's privacy practices.

ICQ adapted to the mobile era with applications for Android, iOS, Windows, and macOS. Artificial intelligence made its appearance to suggest quick replies and transcribe voice messages. During the COVID-19 pandemic, the service deployed video conferencing features for up to 30 participants, meeting remote work needs.

These efforts weren't enough. Against WhatsApp, Messenger, or Telegram, ICQ didn't reclaim its position. In May 2024, VK announced a definitive shutdown for June 26, inviting users to migrate to VK Messenger and VK WorkSpace. A pioneer disappeared.

ICQ laid the foundations for modern instant messaging: presence sta-

tuses, real-time messages, contact lists, user search. The service demonstrated the importance of asynchronous and personal communication on the Internet, before the advent of social networks and their messaging features. Its influence persists in the features of our current applications.

Through its innovative character, ICQ transformed the Internet into a space for personal and immediate communication. Romantic encounters, discussions between players, professional conversations, family exchanges: for nearly three decades, the service wove a web of human relationships throughout the digital world.

KDE

On October 14, 1996, a message appeared on Usenet. Matthias Ettrich announced the launch of a project called KDE, standing for Kool Desktop Environment. His ambition? To build a complete graphical desktop environment for UNIX and Linux. At that time, interfaces on these systems consisted mostly of rudimentary window managers like Feeble Virtual Window Manager for the X Window System. Users confronted with these spartan tools struggled to find the comfort of Windows or Mac desktops.

Ettrich made a technical bet: to rely on the Qt library. This choice immediately triggered controversy, as Qt's license clashed with some free software purists. But the project moved forward. A first beta version was released in 1997, followed by KDE 1.0 nine months later. A community of developers rallied around this initiative.

The architecture rested on several building blocks. KWin handled window display. Konqueror, which succeeded KFM, served both as a file manager and web browser. DCOP, later replaced by D-Bus, enabled applications to communicate with each other. The KHTML rendering engine deserves special mention: Apple would draw inspiration from it to create WebKit, which would power Safari and Chrome in their early days.

Each iteration, KDE 2 and KDE 3, enriched the functionality during the 2000s. In 2008, KDE 4.0 changed the game. The interface and technical architecture were completely overhauled. Plasma, the new desktop environment, transformed the user experience. Applications adopted Qt 4 and exploited the possibilities offered by this new version.

Organizationally, the project became more structured. KDE e.V. was established, a German legal association managing financial and legal matters. It collects donations, protects intellectual property, and organizes

Akademy, the annual conference that brings the community together. This professionalization did not diminish the project's collaborative spirit.

The following decade brought KDE Frameworks 5, which broke down libraries into reusable modules. Plasma 5 modernized the desktop while preserving its legendary flexibility. The project adapted to touchscreens with Plasma Mobile, proof that KDE knows how to keep pace with evolving usage patterns.

The development model is resolutely open. Decisions are made without a hierarchical pyramid, by consensus. Contributors from around the world participate: developers of course, but also translators, designers, and documentation writers. Google Summer of Code and other mentoring programs regularly attract new talent.

The applications cover all daily needs. Calligra for office work, Amarok for music, KMail for email, Kate for text editing... These programs rival proprietary solutions. Some KDE innovations, such as automatic detection of forgotten email attachments, have been adopted elsewhere.

KDE's influence extends beyond technology. The project proves that complex and innovative applications can be created as free software. It facilitates Linux adoption on personal computers by offering an interface accessible to Windows users. Distributions like openSUSE or Kubuntu have chosen it as their default environment, which speaks volumes about its maturity.

Thirty years after its birth, KDE stays the course: giving users back control of their digital environment, respecting their freedom and privacy. The project innovates in accessibility and security. Its longevity stems from its ability to attract new contributors without renouncing its values.

Cascading Style Sheets

In 1994, at CERN, the birthplace of the Web, Håkon Wium Lie identified a problem: the Web lacked a means to style documents. Having come from the MIT Media Laboratory, where he had worked on personalized newspaper presentations, he could see that this gap was holding back electronic publishing.

The idea of separating structure from presentation already existed. Tim Berners-Lee had designed his NeXT browser in 1990 with a rudimentary style sheet. But he had published nothing, believing that each browser should find its own display solution. Others were experimenting

on their own: Pei Wei with his ViolaWWW browser in 1992, which offered its own style language.

Then came NCSA Mosaic in 1993. This browser, which would popularize the Web, paradoxically represented a step backward: users could only modify a few colors and fonts. Web page creators complained. Marc Andreessen, one of Mosaic's programmers, initially refused any changes before reversing course when cofounding Netscape.

In 1994, on October 13, three days before the announcement of the Netscape browser, Håkon published the first version of Cascading HTML Style Sheets. Dave Raggett, principal architect of HTML 3.0, had pushed him to publish before the "Mosaic and the Web" conference in Chicago. Raggett had understood that HTML would never become a page description language.

Bert Bos joined Håkon. He was then developing Argo, a highly customizable browser with style sheets. Their two initial proposals differ from current CSS but already contain the original concepts. A distinctive feature of the language: it recognizes that the style of a web document cannot come solely from the author or the reader. Their preferences must be combined in a cascade, taking into account the capabilities of the display device and the browser.

The World Wide Web Consortium (W3C) became operational in 1995. A workshop on style sheets was held in Paris. Thomas Reardon from Microsoft committed to integrating CSS into Internet Explorer. At the end of 1995, the W3C established the HTML Editorial Review Board to validate future HTML specifications. The CSS specification became a work item toward a W3C Recommendation.

Internet Explorer 3 was released in August 1996. It was the first commercial browser to support CSS. Chris Wilson's team at Microsoft reliably implemented most of the color, background, font, and text properties. Netscape followed with Navigator 4.0, but its implementation remained limited. Opera entered the race in November 1998 with version 3.5, which offered more complete CSS1 support.

CSS got its own working group at the W3C in February 1997. Chris Lilley chaired it. The group developed CSS level 2, which became a recommendation in May 1998. Meetings were generally held by phone for one hour per week, with approximately four annual face-to-face meetings in different cities around the world.

Eric Meyer improved the testing process by developing a CSS1 test suite with the help of many volunteers. This suite significantly facilitated the developers' work. The Web Standards Project (WaSP) monitored browser compliance with W3C recommendations.

CSS developed with more than 60 modules defining different capabilities. The language adapted to new uses: creating electronic books in EPUB format, designing graphical user interfaces. Derived languages appeared. Qt Style Sheets for Qt widgets, JavaFX Style Sheets for JavaFX, MapCSS for describing map styles.

In 2008, web fonts underwent a major evolution. CSS level 2 integrated fonts into a web document, but only Internet Explorer implemented this feature. Microsoft and Monotype developed the proprietary Embedded OpenType format. The situation changed with the emergence of freely distributable fonts on the Web, with the WOFF format being the solution adopted by the community.

CSS has established itself as an essential Web standard: less than five percent of HTML pages do not use style sheets. Specialized conferences and numerous books testify to the vitality of this technology, which continues to evolve to meet the needs of the modern Web.

Flash

Jonathan Gay is a high school student programming his first graphics applications on an Apple II in 1985. His experimentation with the Pascal language leads him to collaborate with Charlie Jackson, founder of Silicon Beach Software. Together, they develop Airborne!, a game for Macintosh that combines animation and digital sound.

He continues by creating SmartSketch, a drawing software that works on both Macintosh and Windows. Users can draw on the screen with an electronic stylus. This innovation gives birth to FutureWave Software in the mid-1990s. When the Internet develops, the company transforms SmartSketch into a two-dimensional animation tool capable of displaying animated graphics on the Web. FutureSplash Animator is born in 1995.

Microsoft adopts the software for its MSN site. Success is immediate. In 1996, Macromedia acquires FutureWave and renames the program Flash. A period of intense innovation begins: audio, video, and interactivity transform the creation and distribution of content on the Internet.

The late 1990s and early 2000s mark Flash's heyday. The technology becomes the standard for online games, interactive websites, and animations. In 2000, ActionScript makes its appearance. This object-oriented programming language gives developers the ability to create sophisticated web applications. Millions of sites integrate Flash elements, from simple

animated buttons to complete immersive environments.

Technical advantages explain this success. The vector format guarantees consistent graphic quality regardless of display size. The plug-in, lightweight (less than 1 MB), installs easily. Distribution reaches 98% of computers connected to the Internet. The sandbox security model protects users by isolating Flash content, and third-party publishers can create compatible tools following the publication of the SWF format.

Adobe Systems acquires Macromedia in 2005 and continues development. But smartphones and tablets arrive. In 2010, Steve Jobs publishes a letter that sharply criticizes Flash: insufficient security, instability, excessive battery consumption. Apple refuses to integrate the technology into iOS. The decline begins.

HTML5 accelerates the movement. This new standard natively offers in web browsers what previously required Flash: animation, audio, video, advanced interactions. Developers migrate toward these open, standardized, and more efficient technologies.

Adobe adapts. The company redirects Flash toward creating HTML5 animations and renames it Adobe Animate. In 2017, Adobe announces the end of support for late 2020. Twenty-five years of a technology that shaped a generation's web experience thus come to an end.

Flash nevertheless leaves its mark on modern web development. The interactivity and user experience it popularized remain relevant. Fluid animation, audio-video synchronization, dynamic interfaces: these concepts are now integrated into Web standards. They testify to Flash's lasting impact on the evolution of the Internet.

This trajectory illustrates the rapid changes in digital technologies. It shows the importance of adapting to new uses and standards. The transition from Flash to HTML5 symbolizes the shift toward a more open, secure, and resource-efficient Web. A technology endures only if it responds to users' changing needs and the technical requirements of its era.

OpenBSD

The history of OpenBSD is rooted in the epic of UNIX and the Internet. When IBM and Digital dominated enterprise computing in the 1970s, the US Department of Defense was funding research on distributed networks. At the University of Berkeley, a group of developers coordinated

by researchers created BSD UNIX, which would become the reference implementation of the Internet's TCP/IP protocols.

In the early 1990s, the Berkeley Computer Science Research Group, which coordinated BSD development, was about to be dissolved. BSD had started as a collection of software for AT&T's UNIX and had evolved over the years into a complete operating system, and groups wanted to continue its development. Lynne and Bill Jolitz adapted the system to Intel x86 processors with 386BSD. Developers began sharing improvements in the form of "patchkits", giving rise to two distinct projects: FreeBSD, focused on optimizing for PC hardware, and NetBSD, aiming for maximum portability.

In October 1995, Theo de Raadt forked NetBSD to create OpenBSD. The first version was released in July 1996. This new distribution distinguished itself through its radical commitment to security and elegant design. The team launched a comprehensive source code audit, systematically searching for and fixing security vulnerabilities. This approach produced a system renowned for its exceptional reliability, with only two remotely exploitable vulnerabilities in the default installation since its inception.

OpenBSD innovated by integrating preventive security mechanisms. The system adopted the `W^X` (Write XOR Execute) principle, where memory cannot be simultaneously writable and executable. It introduced address space layout randomization (ASLR) to vary jump targets and gaps between memory regions at each execution. Unreadable and non-writable guard pages were placed at the end of allocated memory blocks to detect overflows. The developers implemented privilege separation, where daemons run with minimal rights in a restricted environment.

The project distinguished itself by publishing its source code in real time via anonymous CVS, an innovative practice at a time when most projects worked behind closed doors. This transparency allowed the community to continuously verify and improve the system's security. OpenBSD was a pioneer in the use of strong cryptography, becoming in 1997 the first free system to integrate IPSec by default. This decision led to complications with US restrictions on the export of cryptographic technologies, circumvented through project coordination from Canada.

Beyond its operating system, OpenBSD became the source of many widely used components. OpenSSH, created to replace proprietary secure connection solutions, established itself as the worldwide reference. The PF (Packet Filter) firewall, LibreSSL, the OpenSMTPd mail server, and other tools from the project were adopted by numerous systems. These software components are now found in most connected devices, from Cisco

routers to Apple products, including Android systems.

OpenBSD's philosophy prioritizes code correctness and security over immediate ease of use. The system activates only a minimum of services by default, requiring the administrator to explicitly configure desired functionalities. This approach, sometimes criticized for its rigidity, has proven effective in preventing numerous compromises. The developers strive to maintain compatibility with the original BSD model while strengthening security within the constraints of this framework.

The BSD license, more permissive than the GPL, allows the use of code in closed commercial products. This freedom has favored the adoption of OpenBSD as a foundation for various network security solutions. The system has established itself in critical infrastructures requiring maximum security: firewalls, edge-of-network servers, intrusion detection systems. While its use on workstations is limited by restricted support for consumer applications, OpenBSD continues to influence the entire industry through its security innovations.

PostgreSQL

In 1986, at the University of California, Berkeley, Professor Michael Stonebraker launched the POSTGRES project. DARPA, the Army Research Office, and the National Science Foundation funded this initiative, which followed INGRES, a relational system developed a few years earlier by the same team. The ambition was to push the boundaries of the relational model in a context where criticism was coming from all sides.

The relational model, introduced by Edgar F. Codd, sparked heated debates in the computer science community. Traditionalists insisted that this type of system could not function properly and that the proposed query languages remained too complex for users. The Berkeley team decided to silence the critics by creating a working prototype.

The first demonstration version appeared in 1987. The technical choices were bold: UNIX as the operating system and the C language for programming. These decisions proved judicious, influenced by exchanges with Ken Thompson, a Berkeley graduate. The team did not hesitate to rewrite entire system components to improve quality, applying an iterative development method that proved successful.

The project's early years saw intense research and development activity. In 1989, version 1 was distributed to a few external users. Version

2 was released in 1990 with a new rules system. Version 3, published in 1991, brought its share of improvements: support for multiple storage managers, query execution engine optimization, and a redesigned rules system.

POSTGRES found its place in varied applications. It appeared in financial data analysis, aircraft engine performance monitoring, an asteroid tracking database, medical information systems, and geographic applications. Universities adopted it as a teaching tool. In 1992, it became the primary data manager for the Sequoia 2000 scientific project.

The user community doubled in 1993. This growth generated a considerable support burden for the university team. In 1994, two students, Andrew Yu and Jolly Chen, added an SQL interpreter to the system. This evolution gave birth to Postgres95, a more compact and efficient version. The code base was reduced by 25% and performance improved by 30 to 50% on Wisconsin benchmark tests.

In 1996, the project took the name PostgreSQL. This new name emphasized continuity with POSTGRES and the integration of SQL functionalities. This new incarnation marked the beginning of a collaborative and open source approach. The system distinguished itself through its extensibility, which allowed adding new data types and operators without touching the core.

PostgreSQL's development rests on a minimalist philosophy. The system favors using external tools for functions such as connection management, replication, or backup. This approach promotes the creation of a rich ecosystem of complementary projects and adaptation to technological evolutions like SSD storage or cloud deployment.

The project's organizational structure reflects its collaborative nature. The core team operates with a minimal charter and no company is capable of employing the majority of its members. This distributed governance ensures that PostgreSQL remains under the control of its community of users and developers.

The emergence of the World Wide Web and mobile applications brought new constraints for traditional relational databases. PostgreSQL adapted by naturally integrating non-relational data types: complex documents, geographic data, GPS information, social network content. This ability to evolve without restructuring constitutes a first-rate technical advantage.

The 2000s saw the growing adoption of PostgreSQL by companies and organizations of all sizes. The system became a serious alternative to proprietary databases from Oracle, IBM, and Microsoft. Its open development model allows for continuous innovation and responsiveness to user needs that traditional vendors struggle to match.

PostgreSQL's longevity and success result from sound initial architectural choices and stable governance. The system maintains a balance between technical innovation and stability, while preserving its independence from commercial interests. This unique combination explains its current position as the most advanced open source database management system.

Contributions to PostgreSQL come from varied backgrounds: academic researchers, user companies, service providers, and independent developers. This diversity nurtures a dynamic of continuous improvement in code, documentation, and associated tools. The project maintains high quality standards while remaining accessible to new contributors.

PostgreSQL continues to evolve to meet modern requirements: large-scale performance, complex data analysis, artificial intelligence. Its extensible design allows the integration of these new functionalities without compromising its consistency or reliability. This adaptability, combined with its technical maturity, makes it a structural technology of contemporary computing capable of competing with historic proprietary solutions.

USB

In 1990, connecting a peripheral to a computer was an ordeal. The back of machines looked like a tangled mess of disparate and dusty connectors: serial and parallel ports for printers, PS/2 ports for keyboards and mice, 5-pin DIN connectors for older keyboards, not to mention SCSI ports and game ports for enthusiasts. Each accessory required its own type of connection, with its specific hardware characteristics and dedicated drivers. A real headache for users.

It was Intel that decided to bring order to this chaos by launching a project in 1992. The idea? Create a universal connector that would gradually replace all the others. Two years later, six companies joined forces to give birth to the Universal Serial Bus: Intel, Compaq, IBM, Microsoft, LSI, and Hewlett-Packard. Apple briefly joined the group before withdrawing to develop FireWire, its own technology. The ambition was clear: unify the ports, connectors, and electronic signals that linked computers and peripherals.

The first version was released in 1996, with a throughput of 1.56 Mbps. Though modest, the innovation lay mainly in the principle: a single type of cable to transfer both data and power. The designers wanted a compact and inexpensive connector, capable of acting as a translator between

the different ways peripherals communicated. Computers could thus manage several accessories simultaneously without them conflicting with each other.

Intel filed a patent for USB in 1997. But here's the thing: patents can kill a standard in its infancy. When a single company holds all the rights, it can block competitors or demand prohibitive royalties. Intel understood this well. The company created a patent pool, where rights are shared among several holders who split the royalties. This approach allowed manufacturers to freely produce USB-compatible devices without fear of lawsuits.

The promoters founded the USB Implementers Forum (USB-IF), which brought together more than 700 companies, to get the standard adopted by as many as possible, far beyond the circle of computer manufacturers. This diversity played a decisive role in USB's success.

Adoption was not immediate. IT companies began seriously integrating USB toward the end of the 1990s. The technology progressed rapidly: in 2000, USB 2.0 arrived to counter FireWire, which boasted speeds of 400 Mbps and allowed simultaneous bidirectional transfers. On paper, USB 2.0 reached 480 Mbps, but tests sometimes showed performance inferior to its rival. However, its broad compatibility and reasonable licensing fees limited FireWire's expansion. Apple finally integrated USB into its machines in 2003, before completely abandoning FireWire 400 in 2005.

In 2011, seven billion USB-equipped devices were manufactured each year worldwide. The standard continued to progress, reaching speeds of 40,000 Mbps in 2019. The USB port became the reference for just about everything: mice, keyboards, printers, external hard drives. The Micro-USB and Mini-USB variants further expanded its territory. In 2019, the global USB peripherals market was worth 31 billion dollars.

Standardization transformed the user experience. Any USB accessory works on any computer, among thousands of available options. Manufacturers can design their products knowing exactly how they will connect to machines, which reduces risks when developing new devices. This standardization has unleashed innovation: Wi-Fi adapters, optical drives, Ethernet ports, mobile network dongles-all have been able to develop thanks to this common foundation.

The environmental aspect deserves emphasis. By making accessories compatible with any new hardware, USB limits the quantity of electronic devices discarded. Less waste, fewer raw materials extracted, less CO^2 emissions related to the manufacture of proprietary cables and accessories.

In the Internet of Things universe, energy consumption is crucial. The USB protocol stipulates that in active mode, a peripheral has limited

current and must respond immediately to computer requests. If no communication occurs for a few milliseconds, the device enters suspension and its consumption drops. Recent innovations, such as low-power mode and crystal-free oscillators, have improved energy efficiency for battery-powered devices.

The patent pool has enabled manufacturers to converge on a common solution, even among competitors. The presence of alternative technologies, such as wireless connections or inductive charging, continues to push USB to evolve to remain relevant in an increasingly connected world.

VPN

The virtual circuits of the 1970s laid the groundwork for what would later become virtual private networks. These logical connections traversed networks to route long messages, creating a path between the source port and its destination. The concept worked well on highly meshed networks, but no one imagined then that it would become a cornerstone of communications security.

The expansion of the Internet in the 1990s changed everything. Companies found themselves facing a dilemma: how to connect their remote sites and mobile employees while preserving the confidentiality of their communications? Dedicated lines cost a fortune, and the Internet, while attractive for its ubiquity, exposed data to prying eyes. The solution came from encryption, an old technique brought up to date to create secure tunnels at the heart of the public network.

In 1996, Gurdeep Singh Pall and his team at Microsoft designed PPTP, the Point-to-Point Tunneling Protocol. PC Magazine awarded it the innovation of the year prize, and for good reason: it finally standardized what the industry had been seeking for years. PPTP carved tunnels through the Internet, secret passages where data could flow safely away from curious eyes. It was the first true industrial VPN solution.

Other protocols, like L2TP and IPSec, enriched the available toolkit. IPSec deserves closer attention: by working directly at the network layer level, it protected end-to-end communications with a robustness that made it an indispensable standard. Companies now had the tools they needed to build proper virtual private networks.

The 2000s saw VPNs grow in sophistication. Firewalls, strong authentication, fine-grained access management: solutions became more com-

prehensive. Remote work was gaining momentum, and with it the need to connect an ever-increasing number of remote users. Companies began abandoning their expensive dedicated connections in favor of VPN connections over the Internet. The savings were substantial, security remained solid.

SSL and TLS brought a welcome simplification. Gone were the complex client software deployments: a simple web browser was now enough to access corporate applications remotely. OpenVPN arrived in this context as a free and flexible alternative to proprietary solutions. The SSL/TLS technology it used appealed through its ease of use and compatibility with existing infrastructure.

Cloud computing and smartphones reshuffled the deck in the 2010s. VPN infrastructures had to adapt to unprecedented volumes of simultaneous connections. Mobile users multiplied, juggling between their different devices. Solutions integrated quality of service, performance optimization, everything needed to maintain user experience despite growing complexity.

Today, VPNs combine several technologies: IPSec for remote sites, SSL/TLS for web access, firewalls and strong authentication for security. They interconnect scattered offices, protect access to internal resources, and secure communications for mobile workers. This technological convergence meets current needs, but it wasn't obvious twenty years ago.

The history of VPNs ultimately reflects that of network usage patterns. Starting from rudimentary virtual circuits, they transformed into true digital fortresses. Mobility, cloud computing, cybersecurity: all challenges that VPN technology has successfully addressed. It continues to evolve, integrating the latest cryptographic advances to remain relevant.

Individuals have also embraced VPNs, going far beyond their initial professional scope. Protecting privacy on the Internet has become a major concern, and consumer VPN services have flourished. Some use them to bypass geographic restrictions for political reasons, others to mask their browsing activity. This democratization has changed the perception of the technology, once reserved for network experts.

The Internet of Things and 5G will generate new secure connectivity needs. Threats evolve and become more sophisticated, and VPNs must keep pace. Post-quantum cryptography looms on the horizon, promising new algorithms to resist tomorrow's computers. The VPN has not finished its evolution.

Hotmail

In 1996, checking your email required a certain amount of logistical planning. You had to install software on your computer, configure server settings, and above all, remain tied to that machine to access your messages. Jack Smith and Sabeer Bhatia, two former Apple employees who had passed through Stanford, found this situation absurd. They wanted to read their emails from anywhere, without running up against their company's strict firewalls. From this mundane frustration would emerge Hotmail, the first email service that could be used directly in a web browser.

The two partners pooled $4,000 in 1995 to build a prototype. The demonstration won over Draper Fisher Jurvetson, which injected $300,000 for 15% of the equity. But the money ran out quickly. A month before launch, the coffers were empty. Doug Carlisle threw them a lifeline: $100,000 in additional investment. Smith and Bhatia refused, fearing they would lose control of their creation. They chose to borrow instead.

On July 4, 1996, American Independence Day, Hotmail opened its doors to the public. The name played on the letters HTML, those four characters that structure web pages, by inserting them in capitals within "HoTMaiL". The service offered 2 megabytes of free storage. That was small, but no one complained at the time. Bhatia's real stroke of genius lay in a seemingly trivial detail: every message sent from Hotmail carried the signature "PS: I Love You" followed by a link to the service. This trick, which would later be called viral marketing, sent the numbers through the roof. One hundred thousand signups in less than a month, one million users in less than six months.

Microsoft observed this surge in power. By late 1997, Hotmail claimed 10 million users and captured a quarter of the webmail market. Bill Gates didn't let the opportunity slip. On December 30, 1997, the deal closed for $400 million. Bhatia hesitated at first, not thrilled by the monopolistic image that stuck to Microsoft, but ultimately acknowledged Gates's instinct.

Integration into the MSN galaxy accelerated growth. By early 1999, MSN Hotmail crossed 30 million users and registered 150,000 new signups every day. But this rapid expansion came with embarrassing setbacks. In 1999, anyone could log into any account by simply typing the password "eh". Microsoft spoke of an "unknown security issue" and dismissed the possibility of a deliberate backdoor. Two years later, another flaw: a simple URL manipulation was enough to read someone else's messages.

Google launched its Gmail service in 2004 with a gigabyte of storage, five hundred times more than Hotmail's 2 megabytes. Microsoft responded

by raising capacity to 250 megabytes and allowing 10-megabyte attachments. A complete overhaul of the service began, but the final version didn't emerge from beta until 2007 under the name Windows Live Hotmail. These delays left the field open to Gmail, while Hotmail began to look like a dinosaur.

Microsoft then multiplied adjustments: Bing integration, compatibility with Firefox and Chrome, quick filters, automatic inbox cleanup. In 2010, the Exchange ActiveSync protocol arrived, followed the next year by aliases and instant actions. Despite these efforts, the service carried a tarnished reputation. Savvy users shunned it, spammers made it their favorite playground.

July 2012 marked a fresh start with Outlook.com. Microsoft abandoned the Hotmail name and offered a streamlined interface that contrasted sharply with the old version. Success was immediate: more than 10 million voluntary signups in two weeks. The migration of Hotmail accounts was completed in May 2013. Microsoft then announced 400 million active accounts. Users could keep their @hotmail.com address or switch to @outlook.com.

In 2019, a hacker exploited a security flaw to use a support agent's credentials to access certain accounts. Dark mode also appeared that year, to save batteries and ease eyestrain.

Millions of @hotmail.com addresses remain in circulation, alongside @live.com, @msn.com, @passport.com, and @outlook.com. This story primarily tells that of a pioneering service, swallowed by a giant, which had to constantly reinvent itself to avoid disappearing into digital oblivion.

Internet Archive

Brewster Kahle created Internet Archive in 1996. He wanted to build a universal digital library that would preserve the history of the Internet. Not an archive locked away in a digital vault, but an open space where anyone could retrieve vanished web pages. The Wayback Machine, launched in October 2001, provides access 24 years later to over one trillion web pages and more than 99 petabytes of data. Staggering.

Kahle established simple rules from the start. Use only standard hardware. Ban commercial software. Keep the architecture simple enough that you don't need a PhD to maintain it. These pragmatic choices probably explain why the service has been running for over twenty years with fewer

than five people managing it. An almost incredible longevity in the web universe.

The infrastructure relies on distributed storage nodes and front-end web servers. Data is distributed across more than 2,500 nodes, representing over 6,000 hard drives. Total capacity approaches one petabyte. Each day, the system processes tens of millions of requests and transfers more than 40 terabytes of data.

Unlike conventional websites, Internet Archive doesn't use caching. This absence is surprising at first glance. But the designers quickly realized that archive queries are too dispersed in time and space. A traditional cache would serve no purpose. They preferred to directly optimize access to stored data.

The ARC format structures the storage of archived pages. Each file groups approximately 100 megabytes of unrelated pages, along with their headers. The organization makes storage efficient but requires sequential reading to retrieve a specific page. Each file exists on at least two different nodes, to avoid losing everything in case of failure.

To locate data, the system adopts an original method. No centralized index. Queries are sent via UDP to all storage nodes. Each node keeps the list of its files in memory and responds if it has what's being sought. This distributed approach withstands failures well and simplifies adding or removing nodes.

Statistics show that English overwhelms other languages in accessed content. European languages follow, far behind. More than 82% of human sessions arrive through external links. Wikipedia leads the sites that send traffic to Internet Archive. Most pages accessed no longer exist on the active web. Proof that the archive fulfills its role as memory.

Two types of users share access: humans and robots. Robotic sessions are ten times more numerous but generate as much data as human sessions. Robots never arrive with a referrer in their requests, whereas human users almost always arrive via links.

The web's growth imposes constant challenges. The index of archived URLs exceeds 2 terabytes. Initially, updating it was problematic. Disks overheated under intense activity. Moving to a better-cooled data center and an incremental update method solved the problem. Moreover, SSD drives open new possibilities. They outperform mechanical disks for random access, frequent in archive usage. These technologies could make cache systems efficient, previously impossible with traditional media.

Internet Archive collaborates with other institutions like Bibliotheca Alexandrina in Egypt or the European Archives. These collaborations serve to replicate data geographically. Availability improves, as does

preservation. The service also offers specialized collections: the Million Book Project, the Prelinger Archives. The mission expands beyond web archiving.

In 2024, Google Search facilitates access to the past more than ever. Archived versions of web pages are directly accessible with a simple link to the Internet Archive's Wayback Machine.

Internet Archive demonstrates that a minimalist approach can result in a sustainable service. Its architecture favors simplicity and robustness over sophistication. A large-scale archiving system is capable of functioning with a small team. This success now inspires other digital archiving projects worldwide.

Wi-Fi

When Norman Abramson launched the ALOHAnet project at the University of Hawaii in the 1970s, he probably had no idea he would transform our daily relationship with information. This experiment, which involved linking computers via radio waves across the Hawaiian islands, demonstrated that data could be transmitted without cables.

Things accelerated in 1985 when the Federal Communications Commission, under the impetus of Michael Marcus, authorized free use of certain frequency bands: 902-928 MHz, 2.4-2.4835 GHz, and 5.725-5.850 GHz. These ISM (Industrial, Scientific and Medical) bands, initially reserved for industrial and medical uses, became the playground for consumer wireless technologies. Other countries followed suit.

The first IEEE 802.11 standard appeared in 1997, with a modest speed of 2 Mbit/s on the 2.4 GHz band. But it was Apple that truly popularized the technology with the general public in 1999, by integrating Wi-Fi into its AirPort base station and iBook laptop. These devices used the IEEE 802.11b standard, which reached 11 Mbit/s. The name "Wi-Fi", created by a marketing agency, actually has no particular technical meaning, despite its phonetic similarity to "Hi-Fi".

The technical architecture of Wi-Fi draws directly from work on the ALOHA protocol. The CSMA/CA (Carrier Sense Multiple Access with Collision Avoidance) principle allows a radio channel to be shared among multiple devices without constant interference. Research by Kleinrock, Tobagi, and Lam at UCLA helped refine these medium access mechanisms, adapted to the specific constraints of radio communications.

Standards then followed in rapid succession. IEEE 802.11a introduced OFDM modulation and the 5 GHz band in 1999. In 2003, IEEE 802.11g brought 54 Mbit/s to the heavily congested 2.4 GHz band. The real breakthrough came in 2009 with IEEE 802.11n and MIMO technology, which multiplies antennas to reach 600 Mbit/s. IEEE 802.11ac crossed the symbolic gigabit-per-second threshold in 2013, with a theoretical throughput of 3.5 Gbit/s.

This performance race responds to growing connectivity needs. Early Wi-Fi networks struggled to load a simple web page, while today we stream 4K movies on multiple devices simultaneously. IEEE 802.11ax, branded as Wi-Fi 6 and published in 2021, pushes throughput to 9.6 Gbit/s while better managing dense environments like stadiums or airports. The standard incorporates sophisticated spectrum-sharing mechanisms and reduces device power consumption.

Wi-Fi has changed far more than just how we connect to the internet. The technology has redesigned workspaces, enabling professional nomadism. Coffee shops have become makeshift offices, airports have turned into meeting rooms. In our homes, Wi-Fi has enabled the explosion of connected devices, from thermostats to light bulbs to speakers. Schools and universities have rethought their teaching methods around this permanent connectivity.

Radio communications security was long an Achilles' heel, with early protections like WEP quickly becoming obsolete. WPA3 encryption now offers a satisfactory level of protection. Interference between devices, coexistence with other wireless technologies like Bluetooth, limited signal range: all problems that have found technical solutions over successive versions.

The future looks more ambitious. Wi-Fi 7 (IEEE 802.11be), finalized in 2024, targets 40 Gbit/s in theoretical throughput. These performance levels open prospects for virtual or augmented reality, which require minimal latency and considerable throughput. The integration of artificial intelligence in network management should improve dynamic spectrum allocation and user experience.

From the ALOHAnet experiment to Wi-Fi 6 networks, the journey illustrates computing's ability to transform a laboratory idea into everyday technology. Wi-Fi has become invisible precisely because it is omnipresent, present in our pockets, our homes, our cars. This discreteness paradoxically testifies to its success: the best technologies are those we no longer notice.

Google

The Internet entered a new dimension in the late 1990s. Web pages multiplied at an unimaginable pace, and search engines of the time struggled to keep up. AltaVista, Lycos, Excite: all faced the same problem. How could they sort through these millions of pages and separate the useful from the trivial?

Sergey Brin and Larry Page met in 1995 at Stanford. They were doctoral students, fairly brilliant, and began working on BackRub, a project that examined the links between web pages. The idea came to them quite naturally: an important page must be one that many other pages point to. The more references flowing in, the more legitimacy the page gains. It's simple, almost obvious in hindsight.

A year later, BackRub transformed. The project took on a new name, inspired by the mathematical term "googol" which denotes 10 to the power of 100. This reference reflected their ambition: to index a colossal mass of documents. The PageRank algorithm was born from this thinking, assigning a score to each page based on the quantity and quality of incoming links, creating a sort of spontaneous hierarchy of web content.

In September 1998, the two students founded Google Inc. in a garage in Menlo Park. Andy Bechtolsheim, cofounder of Sun Microsystems, advanced them $100,000. A few months earlier, they had tried to sell their technology to Excite for one million dollars. George Bell, the CEO, refused. This decision ranks among the most monumental mistakes in Internet history.

The technology made the difference. In 1999, Google was already processing 3.6 million queries daily. The interface was clean, the speed impressive, and above all the results were accurate. Internet users quickly realized this. The company developed efficient indexing methods and built a distributed infrastructure capable of absorbing unprecedented data volumes.

Google didn't just read page content. The algorithm examined their position in the web's network. It analyzed the text of links pointing to a page, making content indexing possible without having visited them. This approach worked remarkably well for images or PDFs.

The business model gradually took shape. In October 2000, the AdWords advertising system was launched, with advertisers buying keywords to appear in results. The innovation lay in the auction system and the

clear separation between sponsored links and organic results. User trust remained intact.

Growth accelerated at the turn of the 2000s. The number of indexed pages went from 1 billion in 2000 to over 4 billion in 2004. Google built its own data centers, equipped with custom-designed servers. This technical mastery maintained performance despite the traffic explosion.

The corporate culture fostered innovation. The motto "Don't be evil" set a course. Engineers had 20% of their time for personal projects. Gmail and Google News would emerge from this freedom. The company attracted the best talent by offering them a stimulating environment and generous conditions.

The 2004 IPO used an original auction system. The success exceeded expectations: the valuation reached $23 billion on the first day. Google's dominance in the digital economy was no longer in doubt.

Google transformed our relationship with knowledge. The search engine became a daily reflex for hundreds of millions of users. Our ways of searching and accessing information changed permanently. The influence extended beyond search: the techniques developed by the company, particularly in processing massive data sets, inspired countless innovations.

PageRank found applications in network analysis far beyond web search. Advances in distributed data processing, infrastructure management, and machine learning established benchmarks that still structure 21st-century computing.

VMware

IBM had already invented virtualization in the 1960s. The mainframes of that era allowed users to share the resources of a single physical machine across multiple virtual systems. But this technology remained confined to mainframes for decades, until a small team at Stanford University decided to transpose it to x86 processors.

In the mid-1990s, Professor Mendel Rosenblum and his students were attempting to build a supercomputer. To run different operating systems on their prototype, they experimented with virtualization. The results exceeded their expectations. Rosenblum then realized he had something bigger than a simple university project. He founded VMware in 1998 with his wife Diane Greene, Edouard Bugnion, Scott Devine, and Edward Wang.

The initial idea was simple: data centers were wasting their resources. Servers ran at 10 or 15% of their capacity because they were sized to handle peak activity. The rest of the time, processors and memory remained unused. Virtualization would allow multiple workloads to run on a single physical machine.

VMware Workstation 1.0 was released in 1999. It required a 266 MHz Pentium II and 64 MB of RAM. This first version supported MS-DOS 6, Windows 95/98/NT, Red Hat Linux 5.0, and a few other systems. The technical challenge was considerable: the x86 architecture had never been designed for virtualization. The engineers had to be clever.

Their solution combined two approaches. Direct execution for non-privileged application code, dynamic binary translation for system code. This hybrid method delivered decent performance while isolating virtual machines from each other. It was a feat: no one had managed to properly virtualize the x86 architecture until then.

ESX Server 1.5 arrived in 2002. This hypervisor could manage 64 virtual machines simultaneously with 3.6 GB of RAM each. Unlike Workstation, which installed on an existing operating system, ESX ran directly on bare metal. Performance was improved. That year, VMware secured its first patent (US 6397242) for its x86 virtualization techniques.

The following year marked the arrival of vMotion. This technology was a game changer: you could move a virtual machine from one physical server to another without service interruption. No need to stop applications for maintenance. VMware also launched vCenter Server, a centralized console to manage the entire virtual environment.

EMC Corporation acquired VMware in 2004 for 635 million dollars. The company opened offices in the United Kingdom, China, India, and Ireland. Three years later, EMC decided to take VMware public while retaining 90% of the capital. The stock was offered at 29 dollars. It closed its first day at 51 dollars, valuing the company at 19.1 billion dollars.

VMware vSphere was released in 2009. This platform integrated numerous advanced features for virtual resource management. Things accelerated in 2013 with NSX for network virtualization and vSAN for storage. Virtualization was no longer limited to servers: network infrastructure and storage also moved through the software layer.

Meanwhile, processors had evolved. AMD and Intel added hardware extensions dedicated to virtualization, named AMD-V and VT-x. VMware adapted its products to take advantage of them. These hardware features facilitated 64-bit system virtualization and boosted performance.

Dell acquired EMC in 2015 and thus inherited VMware. The company maintained its operational autonomy and continued its development. It

invested in environmental projects like Carbon Neutrality, aimed at reducing CO_2 emissions from data centers. In 2023, Broadcom finalized its acquisition of VMware.

One figure sums up VMware's impact beyond its own hypervisor: as early as 2009, the number of virtual servers exceeded that of physical servers. x86 virtualization had transformed the industry. It reduced infrastructure costs, optimized resource utilization, and brought flexibility to information systems. It also laid the foundation for cloud computing techniques: automation, resource elasticity, everything relied on these virtualization mechanisms.

By making virtualization accessible on the x86 architecture, VMware modernized infrastructure management. Its success pushed other players to develop their own solutions, like AWS Nitro, creating competition that benefits everyone.

Microsoft Windows 98

After the triumph of Windows 95, Microsoft dominated the personal computing market in 1998. The company had broken with IBM, now in decline in the consumer computer segment, which was abandoning OS/2 in favor of Windows and NT. The operating system that Bill Gates had nearly seen disappear on several occasions now ruled over the desktop computer universe.

Windows 98, codenamed Memphis, was released on June 25, 1998. The OSR2 version of Windows 95 had already integrated Internet Explorer 3 and then 4, the FAT32 file system, and USB support. This new iteration was meant to consolidate these advances and bring significant improvements. Jim Allchin, senior vice president of US business systems management, set the course: improve network functionality, simplify the Control Panel, and automate the system further. The team prioritized qualitative optimization over adding eye-catching features. The trade press did not hesitate to criticize this approach, lamenting the absence of spectacular innovations.

Development began in January 1997 with an initial developer version. The first alpha version (build 1387) was released on February 7, 1997. On October 3, beta 2.1 (build 1602) finally allowed upgrades from Windows 3.x. Release Candidate 0 arrived on February 16, 1998, followed by the final beta version on May 9. Technology enthusiasts thus had time to test the system before its official release.

On the technical front, Windows 98 introduced the Windows Driver Model (WDM), which gradually replaced the VxD model inherited from Windows 2.0 for 386. This evolution enabled the arrival of the ACPI (Advanced Configuration and Power Interface) interface. Without WDM, system hibernation would have been impossible, Plug and Play device performance would have plateaued, and audio processing would have suffered significant limitations. Hardware manufacturers adopted this new model gradually, but this modernization greatly facilitated later transitions between Windows versions.

Hardware support improved considerably. USB management matured with the addition of hubs, imaging, and audio devices. Compatibility with IDE and SCSI controllers expanded, and AGP support appeared. DirectX 5.2 enriched the system's multimedia capabilities, particularly for gaming. Multi-monitor management and DVD playback rounded out these new features.

Network functionality underwent a major overhaul. The integration of Winsock 2, SMB signing, DHCP optimization, and NDIS 5.0 modernized connectivity. Internet Connection Sharing simplified home networking. These technical improvements did not unfold without hiccups. During a public presentation at COMDEX in Chicago on April 20, 1998, Chris Capossela and Bill Gates suffered an embarrassing blue screen while attempting to demonstrate the system's Plug and Play capabilities.

Hardware requirements included, at minimum, a 486DX2 processor at 66 MHz, 16 MB of RAM, and 500 MB of available disk space. Official recommendations called for a Pentium with 24 MB of RAM, with some university IT departments suggesting 32 MB. Users discovered that by using the /nm parameter during installation, it was possible to bypass these prerequisites and install the system on an 80386 with only 4 MB of RAM. A minimal installation occupied only 120 MB of disk space. Times have certainly changed.

The first days of commercialization saw 530,000 copies sold in stores. This figure, lower than the record sales of Windows 95, can be explained by more modest promotion and less spectacular changes. Some retailers and customers nevertheless attempted to recreate the atmosphere of Windows 95 launches, with waiting lines and special midnight events.

Windows 98 SE (Second Edition), released on May 5, 1999, primarily brought bug fixes but included some improvements. Internet Explorer 5 replaced version 4, FireWire support appeared, and SBP-2 technology improved USB storage management. DirectX moved to version 6.1, while WDM support for audio and modems improved significantly. Support for the WinG API was dropped in favor of DirectX, and Windows Media

Player replaced RealPlayer 4.

The development organization at Microsoft was ahead of current industry software methods. One team focused on the current product while another prepared the next version. This separation maintained a clear direction while preserving compatibility with earlier systems. The gradual introduction of WDM illustrates this approach: old drivers remained usable while manufacturers adopted the new model.

Windows 98 marked a milestone in the evolution of operating systems, despite mixed reception from the technical press. Its legacy lies in its ability to have consolidated the innovations of its era while preparing for future developments. The system was gradually supplanted by Windows 2000 and then Windows XP, which offered increased stability and security thanks to their NT architecture. The end of official support by Microsoft in the mid-2000s marked the conclusion of this period.

XML

In 1996, the W3C entrusted Jon Bosak, an engineer at Sun Microsystems, with leading an ambitious working group. The mission was to design a markup language that would combine SGML's structural power with a simplicity of use that SGML had never achieved. HTML dominated the web, but its limitations were becoming glaring: designed for visual formatting, it proved inadequate for logically organizing information.

Two years later, XML was officially born as a W3C recommendation. The language introduced strict tag-based syntax, imposed the notion of well-formed documents, and gave developers the freedom to create their own tag vocabularies through DTDs. Each sector could now shape its own data structures with flexibility without waiting for an organization to standardize them.

The major innovation lay in the radical separation between content and presentation. An XML document could be transformed into a web page, a PDF, or a text file, depending on current needs. XSLT, standardized the following year, provided the transformation tools necessary for this modularity. Data acquired unprecedented autonomy, freed from their destination format.

Web services emerged in 2000, e-commerce took hold, and businesses discovered the value of exchanging information between disparate systems without resorting to haphazard conversions. XML naturally became es-

tablished. Its ecosystem grew richer: XPath let us navigate through documents, XQuery queried data, XML Schema gradually supplanted DTDs by providing more refined validation mechanisms.

This story has its roots much earlier, in 1969, in IBM's laboratories. Charles Goldfarb, Edward Mosher, and Raymond Lorie developed GML there, SGML's direct ancestor. Their insight: a document's structure must be independent of its content. This principle, groundbreaking at the time, still underpins our approach to managing digital information.

Databases adapted. Traditional relational systems integrated native XML document storage, while specialized solutions appeared to meet specific needs. Entire sectors built their standards on this format, the pivot of service-oriented architectures known as SOAs: digital publishing reorganized its production chains, finance structured its data exchanges.

XML's relative simplicity compared to SGML partly explains its success among developers. But it's primarily its self-descriptive nature that appeals: an XML document carries within itself the keys to its understanding. This characteristic ensures a certain longevity: files remain readable and usable years after their creation, with external documentation mostly remaining optional.

JSON's arrival in the mid-2000s somewhat disrupted the landscape. Lighter and more direct, it won over web development. Yet XML didn't yield ground. It retains its relevance where structural rigor matters, where formal validation is essential, where complex transformations are necessary. The two formats coexist, each excelling in its domain.

XML has transformed our approach to documentation, changed how organizations design their systems, and encouraged structured thinking about information. The hierarchical markup principles it popularized are found in much more recent formats. The semantic web and modern microservice architectures bear its imprint.

The W3C orchestrated this standardization process with a method that serves as a reference. XML and its satellite technologies today form a coherent set of specifications, the result of remarkable collective work. These standards continue to evolve, adapt to emerging needs, and testify to the vitality of a format approaching its thirtieth year without showing its age.

GNOME

In 1996, Miguel de Icaza embarked on an adventure that would transform the landscape of free systems: creating a complete graphical desktop environment for GNU/Linux. The GNOME project (GNU Network Object Model Environment) addressed a very real frustration. UNIX systems ran smoothly on servers, but their user interface remained rudimentary. How could they hope to attract the general public with such austere tools?

The situation became more complex with the emergence of KDE, which relied on Qt, a library whose license conflicted with free software principles. The community faced a dilemma: should they recreate Qt as free software or pursue something else? The GNOME team chose the latter. Emulating an existing API seemed like a dead end, as demonstrated by the difficulties of projects such as GNUstep, Wine, or LessTif.

The choice fell on Gtk+, a graphical library created for GIMP's needs, the image editing software developed by Peter Mattis and Spencer Kimball. Written in C but designed with objects in mind, Gtk+ provided a solid foundation. Components like Gdk Imlib for image management and VFS for file manipulation were built around it.

The project's beginnings reflected the spirit of free software: open discussions on mailing lists, source code shared via CVS, direct repository access for proven contributors. Version 0.0 was released in August 1997. March 1999 saw the arrival of version 1.0, immediately adopted by Red Hat Linux as its default environment.

GNOME was built in a modular fashion. Libraries like libgnome and libgnomeui coexisted with basic applications and development tools. This organization provided flexibility because everyone worked on their component without getting in each other's way. The interface followed strict style rules to ensure overall consistency.

From the outset, internationalization was among the priorities. The GNU `gettext` system translated interfaces into numerous languages. So GNOME became accessible to users worldwide, far beyond the English-speaking sphere.

The year 1999 saw the birth of two companies around the project: Eazel and Ximian, the latter founded by Miguel de Icaza. Eazel developed Nautilus, the file manager, but closed its doors in 2001. Ximian fared better by offering an enhanced version of GNOME and tools like Evolution, before Novell acquired it in 2003.

A milestone was reached in 2000 with the creation of the GNOME Foundation. This structure coordinated the efforts of various stakeholders

and preserved the initial objectives. The foundation combined a democratically elected board of directors and an advisory board that brought together companies and non-profit organizations.

That year, Sun Microsystems made a notable choice: GNOME replaced CDE on Solaris. The project gained recognition in the professional world. Sun went further by creating a laboratory dedicated to accessibility, opening the environment to people with disabilities.

The technical architecture relied on CORBA (Common Object Request Broker Architecture) to enable component communication. This technology allowed interaction between applications written in different languages. The developers created ORBit, a streamlined version of CORBA tailored for GNOME's needs.

The project encouraged the use of scripting languages like Scheme and Perl. This openness simplified application customization and automation. The GNOME card game used Scheme to define the rules for different solitaire variants.

Red Hat established itself as a major contributor, notably through its advanced research laboratory. The company implemented daily compilation that improved code quality by quickly detecting problems. Novell, Collabora, and Intel also contributed to the effort.

GNOME established a rigorous development cycle with major releases every six months. This sustained pace, innovative for the time, included phases of progressive freezes: first features, then the interface, and finally translations. This method stabilized releases without rushing.

The 2.0 version of 2002 consolidated the project's architecture. It improved component integration and further harmonized the interface. Subsequent versions continued this work of consistency while adding new capabilities.

GNOME demonstrated that a free project can bring together volunteers and companies around common objectives. Its balanced governance, blending community democracy and economic stakeholder participation, inspired other initiatives. The project proved that a free alternative to proprietary environments could achieve a professional level, both in quality and functionality.

Bluetooth

In Ericsson's laboratories in 1994, engineers set out to solve a simple problem: replacing the cables connecting electronic devices with an inexpensive, low-power radio link. This project would take on a surprising name, borrowed from Viking history. Harald Bluetooth, King of Denmark in the 10^{th} century, had unified Denmark and Norway. The designers saw it as a perfect symbol for a technology designed to connect different devices.

Four years later, in February 1998, five electronics giants—Ericsson, IBM, Intel, Nokia, and Toshiba—created the Bluetooth Special Interest Group. This association's mission was to define common technical specifications. The movement gained momentum: 3COM, Microsoft, Lucent, and Motorola joined the group, which already had more than 1,900 members by 2000.

The first commercial version was released in 1999. Devices from different manufacturers struggled to communicate with each other. Version 1.1, in 2002, corrected these teething problems and stabilized throughput at 1 Mbit/s. Two years later, version 2.0 tripled transmission speed thanks to Enhanced Data Rate.

The operation relies on piconets, these small networks where a master device coordinates up to seven active slaves. The 2.4 GHz frequency band serves as the operating space, with an interesting feature: the transmission frequency changes 1,600 times per second among 79 available channels. This frequency hopping limits interference.

Security was not overlooked. The designers included three levels of protection, from open mode to encrypted connection, with authentication and authorization mechanisms. But it was in 2010 that everything changed. Version 4.0 introduced Bluetooth Low Energy, a variant that consumes much less power. Smartwatches, health sensors, and home automation devices found their gateway to the wireless world.

Six years later, version 5.0 quadrupled range and doubled speed. The Internet of Things exploded, and Bluetooth adapted. Protocol layers stack up: the radio layer handles waves, the baseband controls packets, the link manager establishes connections. Above, L2CAP segments and distributes data. Protocols like RFCOMM also emulate serial ports to maintain compatibility with existing systems.

Profiles define uses: file transfer, synchronization, telephony, audio. The Generic Access Profile serves as the foundation for all others. This layered architecture enabled Bluetooth to go beyond its initial role as a

simple cable replacement. Forecasts predicted 5.4 billion devices shipped in 2023.

In 2017, the addition of mesh networking opened new possibilities. Extended networks became feasible, useful for connected lighting and building automation. In hospitals, sensors monitor patients remotely. Factories collect data to anticipate breakdowns. Bluetooth beacons guide customers through stores and revolutionize proximity marketing.

This technology illustrates how a simple radio link can evolve into a complex system. From point-to-point to mesh networks, from cable replacement to the backbone of IoT, Bluetooth has managed to transform itself without breaking compatibility with previous versions. Throughput increases, range extends, energy efficiency improves. And yet, a recent device can still communicate with an old peripheral from the 2000s.

MSN Messenger

When MSN Messenger arrived in 1999, instant messaging was nothing new. IRC had existed since 1988, born from the imagination of the Finnish developer Jarkko Oikarinen. This real-time chat system had already proven its usefulness in 1991 during events like the Gulf War in January or the attempted coup against Mikhail Gorbachev in August. Thousands of internet users from around the world gathered there to exchange information in real time, creating a primitive form of global social network.

Microsoft entered a particular context. Home computing was becoming mainstream, and Bill Gates' vision of a connected computer in every household, echoed by President Clinton, was beginning to materialize. Graphical interfaces had made machines accessible to novices. The ground was ready for consumer applications that no longer required advanced technical skills.

But MSN Messenger didn't simply replicate IRC. Where the latter favored public channels and group discussions, Microsoft bet on the intimacy of private conversations between authorized contacts. The inspiration came from ICQ, created in 1996 by four Israeli developers from Mirabilis. This clever little program with its clever name ("I seek you") had attracted 100,000 simultaneous users by June 1997. AOL purchased it in 1998, proof that the instant messaging market was attracting interest.

The technical architecture relied on a client-server model. Unlike some

competitors who opted for peer-to-peer, Microsoft centralized exchanges on its servers with proprietary protocols. This approach gave the company total control over its platform, but also created specific security vulnerabilities. Communications went through TCP port 5190, but resourceful users quickly learned to circumvent restrictions by redirecting traffic to other ports.

Features became richer with each version. Contact lists with presence indicators, instant text conversations, file transfers, customizable avatars, mood messages, and of course the memorable Nudges. The ability to modify one's status (online, busy, away, invisible) represented a notable social innovation. For the first time, users explicitly controlled their availability, something impossible with traditional telephones.

The arrival of mobile devices during the 2000s disrupted usage patterns. Versions for mobile phones appeared, but their adoption was mixed. The phones of that era suffered from crippling limitations: weak battery life, tiny screens, expensive connections. A study conducted in Sweden between 2007 and 2008 showed that users considered the mobile version as merely a supplement. They used it mainly to kill time, not really to communicate seriously.

Security posed problems from the start. Users exposed themselves to all sorts of threats: viruses, Trojan horses, conversation eavesdropping, identity theft. CERT documented attacks exploiting social engineering. Hackers played on users' trust to spread malware disguised as legitimate improvements. The fact that many teenagers used MSN Messenger without parental supervision worsened the problem.

In businesses, the tool sparked heated debates. Some organizations saw it as a way to streamline internal communication and informal collaboration. Others worried about the risks of confidential information leaks and decreased productivity. Many ended up deploying secured and controlled versions, when they didn't outright block access.

Interoperability remained a thorny issue. Users would have liked to chat with their contacts regardless of their messaging service. But Microsoft, AOL, and Yahoo maintained closed ecosystems despite repeated requests and pressure from the FCC. Solutions like Trillian allowed these barriers to be circumvented by aggregating multiple services, but without guaranteeing all native features.

The internet evolved. Social networks emerged and redefined online communication. Facebook offered its own integrated messaging. Modern smartphones made desktop applications obsolete. Microsoft eventually replaced MSN Messenger with Skype in 2013, marking the end of an era that had lasted fourteen years.

Contemporary messaging applications incorporate concepts invented or popularized by MSN Messenger: contact lists, presence indicators, instant conversations, file sharing, emojis. WhatsApp, Telegram, Signal, and Discord owe much to this pioneer.

MSN Messenger permanently altered our ways of communicating. By making instant messaging accessible to the general public, it transformed interpersonal relationships in the digital age. Its history reflects the technological and social transformations at the turn of the 20th and 21st centuries, a period when connected computing became an integral part of daily life.

RSS

Dave Winer launched scriptingNews in 1997 while working at UserLand. This format sought to simplify content sharing on the Internet. Two years later, Netscape adopted the idea and created RSS 0.90 (Rich Site Summary), an XML format with an RDF header that it immediately deployed on my.netscape.com.

UserLand responded with scriptingNews 2.0b1, which integrated RSS 0.90 features. Netscape countered with RSS 0.91, a streamlined version without RDF but retaining the essence of scriptingNews. UserLand then abandoned its own format to adopt RSS 0.91. Netscape subsequently discontinued development of the technology.

The year 2000 marked a turning point. A group led by Rael Dornfest at O'Reilly designed RSS 1.0, a complete overhaul based on RDF and XML namespaces. This version broke with the 0.9x lineage. Dave Winer continued on his own with RSS 0.92, adding optional elements to version 0.91.

In 2002, after leaving UserLand, Winer created RSS 2.0. This iteration enhanced RSS 0.92 while maintaining backward compatibility with previous versions. Harvard University published the specification under a Creative Commons license in 2003, opening the format to everyone.

RSS enabled websites to announce their updates in a standardized format. Users subscribed to these feeds through aggregators and received new content without visiting each site. Blogs greatly benefited from this automation.

Libraries and documentation centers quickly adopted the technology for their alerts and monitoring services. The format distributed scientific

journal summaries and new acquisitions. BioMed Central and Nature integrated it into their platforms.

The diversity of reading tools exploded. FeedReader and NetNewsWire coexisted with online solutions like Bloglines. Firefox and Opera added native feed management functions. Google Reader became a reference until its closure in 2013.

RSS's XML structure ensured interoperability. Each feed contained mandatory elements: title, link, description. Optional metadata enriched the whole. The hierarchy organized content by categories and accepted multimedia attachments.

Online media transformed how information was consumed thanks to RSS. Readers centralized tracking of their favorite sources and filtered according to their interests. This autonomy reduced dependence on traditional portals.

Digital marketing seized upon the format to distribute commercial offers and newsletters. Syndication reached audiences through aggregators. Google created AdSense to monetize RSS feeds, opening up advertising opportunities.

Social networks and Twitter gradually changed habits. RSS nevertheless remained useful for information professionals and users concerned with controlling their sources.

The format inspired other standards such as Atom, standardized in 2005 by the IETF. These complementary specifications enriched the web ecosystem by facilitating structured content aggregation.

RSS's distributed architecture influenced the social web. Automated publishing and subscriptions inspired social network notifications. The activity feed principle, central to modern interfaces, partly originated from there.

RSS's simplicity made it a pillar of the participatory web of the 2000s. The technology demonstrated the importance of open standards for service interoperability. Its legacy endures in current content distribution protocols.

In 2024, millions of sites and applications still use RSS. Feeds power monitoring systems, news aggregators and automation tools. This longevity testifies to the relevance of a decentralized model for information distribution.

SOAP

In late 1997, Microsoft began exploring an idea that would transform communication between applications: using XML for remote procedure calls over HTTP. The ambition was simple: enable machines to communicate across networks with standard data types, without the complications of proprietary protocols. DevelopMentor, accustomed to collaborating with Microsoft, and Userland, which saw the Web as a publishing platform, joined the venture. The name SOAP emerged in early 1998.

But things quickly became complicated. The DCOM team at Microsoft put up strong resistance. Rather than adopting this new approach, they preferred to leverage the company's dominant position to impose DCOM through HTTP *tunneling*. The in-house XML experts found the idea appealing but premature: they were waiting for the advanced features promised by XML Schema and namespaces. Faced with this deadlock, Userland took the initiative and published its own version of the specifications under the name XML-RPC during the summer of 1998.

In 1999, Microsoft made progress on XML Schema and integrated namespaces into its products. SOAP regained momentum, though the BizTalk team remained reluctant as their model was based on messaging, not remote procedure calls. On September 13, 1999, SOAP 0.9 was released for public review and submitted to the IETF. Three months later, SOAP 1.0 appeared with few changes.

In March 2000, the W3C announced it was considering activity around XML protocols. At the XTech conference, a lively session brought together several visionaries who debated future directions without reaching consensus. But May 8, 2000 marked a real turning point: SOAP 1.1 arrived at the W3C with IBM as co-author. This unexpected support changed everything. The new version proved much more modular and extensible, dispelling fears that SOAP would impose proprietary Microsoft technologies. IBM immediately published a Java implementation which it contributed to the Apache XML project for open source development. Skeptics began taking the protocol seriously. Sun expressed interest and worked to integrate web services into J2EE. Other vendors and open source projects followed suit.

In September 2000, the W3C formed a working group dedicated to the XML protocol, taking SOAP 1.1 as its starting point. After months of modifications, improvements, and difficult decisions about what to keep or abandon, SOAP 1.2 became an official recommendation in June 2003.

SOAP prevailed because it represented the best industrial compromise for standardizing XML-based multi-platform distributed computing. Its

simplicity was its major asset: historically, architectures that achieved mass adoption did so thanks to this quality.

The protocol defines the unit of communication through an envelope that frames all information. A message contains a body where arbitrary XML can be placed, accompanied by headers that carry data outside the main body. The processing model establishes precise rules for handling messages when extensions come into play. SOAP faults handle errors by identifying their source and cause, while allowing the exchange of diagnostic information between participants.

Extensibility works through SOAP headers that carry extension data with the message and can target specific nodes along its path. SOAP offers a flexible data representation mechanism: it accepts data already serialized in various formats (text, XML) and provides a convention for representing abstract structures like programming language types in XML.

Remote procedure calls and their responses "map" naturally to SOAP messages. This is a common type of interaction in distributed computing that corresponds well to procedural language constructs. The binding framework defines an architecture for building bindings that send and receive SOAP messages over arbitrary transports. This framework notably serves to move SOAP messages across HTTP, the ubiquitous Internet protocol.

SOAP remains widely used in enterprises for application and service integration, especially with legacy systems. The banking and financial sectors are among its loyal users. Google employs it for most of its applications, as do PayPal, Amazon, and eBay. The protocol maintains its relevance thanks to its technical robustness and its ability to handle complex operations that require maintaining conversational state and contextual information. Its evolution enabled the development of web services and the rise of distributed computing.

Transport Layer Security

The security of communications on the Internet today relies on a discrete yet essential technical foundation. In 1994, Netscape sought to protect exchanges on its Web browser and invented SSL (Secure Sockets Layer). The first public version, SSL 2.0, was released the following year. However, this initial attempt contained too many flaws to be truly reliable. SSL 3.0 arrived in 1996 to correct course.

Three years later, the IETF took over the project and renamed it TLS 1.0. This name change also marked a takeover by the international technical community. The protocol creates an encrypted tunnel between a client and a server, ensuring that no one can read or modify the data in transit.

The 2000s saw the emergence of players like Comodo, specializing in issuing digital certificates. Different formulas appeared: EV (Extended Validation) certificates in 2007, which require thorough verification, then DV (Domain Validation) certificates, simpler to obtain. TLS 1.1 was released in 2006 with some additional protections against certain attacks, without really convincing system administrators to migrate en masse.

In 2008, TLS 1.2 brought its share of innovations: extension support, enhanced security mechanisms. Yet adoption lagged. It would take several high-profile incidents for the sector to understand the urgency of modernizing its infrastructures.

Between 2011 and 2014, alerts multiplied. The attack against DigiNotar in 2011 revealed that a hacker had managed to penetrate this Dutch certificate authority and issue hundreds of fraudulent certificates. The BEAST attack exploited a vulnerability in TLS 1.0. The wake-up call was brutal.

In 2014 the Heartbleed flaw affected more than 300,000 public web servers. Three years after its discovery, nearly 180,000 devices remained vulnerable. POODLE in 2014, FREAK and DROWN in 2015-2016 confirmed that vigilance must be constant. Each incident accelerated the migration to TLS 1.2 and pushed for the development of a truly new version.

TLS 1.3 arrived in 2018 after several years of gestation. This version did not simply patch up the existing system: it thoroughly cleaned up the protocol. Obsolete or dangerous functionalities disappeared. The handshake, the initial exchange between client and server, went from two to just one round trip in most cases. The speed gain was immediately felt.

This version imposed Perfect Forward Secrecy, a property that prevents retrospective decryption of communications even if long-term keys fall into the wrong hands later. The number of cryptographic suites dropped from over 35 to just five. This major cleanup improved security but required adjustments in existing infrastructures.

The history of TLS reflects that of threats on the Internet. The protocol had to constantly adapt, juggling between innovation and compatibility with systems already deployed. Successive versions hardened cryptographic mechanisms and simplified their use. Sometimes, inertia slowed down developments. But security incidents regularly reminded us that we

could not rest on our laurels.

Now, TLS protects a large portion of web traffic. The protocol is no longer limited to banking transactions or sensitive data. It has become the standard for all online communication, especially since Google indicated in 2015 that encryption influenced indexing, even though this requirement is more a consequence of the PRISM affair and the need to raise the overall security level of the web. Through its modular architecture, new algorithms are added as cryptography advances.

The lessons from TLS development have spread to other security protocols. The importance of formal validation, the gradual transition to new versions, the delicate balance between security and performance have become guiding principles in the field. Accumulated experience shows that the security of digital communications requires continuous evolution.

Napster

Shawn Fanning was 18 years old when he created Napster in 1999. The story begins simply: this student wanted to share music files with his friends. He had no idea he was about to trigger one of the greatest revolutions in the history of music.

The technical principle was based on *peer-to-peer*. Users exchanged MP3 files directly between their computers, without a central server to store the data. Napster only maintained a directory of files available on connected machines. When someone searched for a song, the software established a direct connection with another member's computer that had the file.

In less than two years, more than 26 million users joined the service and exchanged over 80 million songs. This explosive growth was due to the simplicity of use and the fact that it was free. All you had to do was type in a song title to download it. The MP3 format, which compressed audio files without too much quality degradation, made these transfers fast with the Internet connections of that era.

The record companies quickly understood the threat. They had always controlled music distribution through the sale of physical media. Napster disrupted this model by allowing the free circulation of works on the Internet. The Recording Industry Association of America (RIAA) estimated its losses at 55 billion dollars over a decade.

The lawsuit began in December 1999. The RIAA was suing Napster for

contributory copyright infringement. The accusation was not about direct infringement, but about facilitating these violations by users. David Boies, Napster's lawyer, relied on the Audio Home Recording Act of 1992, which authorized private copies for non-commercial personal use.

The courts had to settle an unprecedented question: did file sharing constitute "fair use" under American law? Four criteria came into play: the purpose of the use, the nature of the work, the proportion used, and the impact on the market. In July 2000, Judge Marilyn Patel ordered the halt of distribution of protected files. An appeals court temporarily suspended this decision, but Napster shut down in 2001.

This closure stopped nothing. Other services took over with more decentralized architectures: Gnutella, Kazaa, then BitTorrent. Each learned from the legal flaws of the previous one to better protect themselves from prosecution. The music industry found itself facing a technical problem that it could not solve by legal means alone.

Apple seized the opportunity by launching iTunes in 2003. Steve Jobs understood that a legal alternative that was as simple as Napster had to be offered. The bet worked: people accepted paying if the service remained convenient. This approach showed that a new economic model was possible.

The story of Napster mainly reveals the inadequacy of copyright laws in the face of the possibilities opened up by the Internet. These laws had been designed for a world where copying a work required industrial means. Digital technology allowed anyone to reproduce and distribute content without any degradation. The legal frameworks of the 20^{th} century no longer held.

The service transformed music consumption habits. An entire generation became accustomed to free access and immediacy. This expectation did not disappear with the closure of Napster. It forced the industry to rethink its models, ultimately leading to streaming services like Spotify that attempt to reconcile free access and artist compensation.

The international dimension complicated matters. Napster reached users all over the world, but copyright laws remained national. How to apply the American legal framework to cross-border exchanges? This question remains relevant for all digital platforms.

The lawsuits created important case law. They showed that the Internet could not function as a lawless zone. Legislators around the world drew inspiration from them to develop their own regulatory frameworks. The Napster case established precedents that still influence debates on digital regulation. The balance between innovation, creators' rights, and user expectations remains fragile.

Year 2000 Bug

The story of the Year 2000 bug begins well before the years of panic that preceded the turn of the millennium. In the 1960s, when programmers worked with punch cards limited to 80 columns and every byte of memory cost a fortune, coding dates with two digits seemed a perfectly reasonable decision. Who would have imagined that these programs would still be running forty years later? This space-saving measure, trivial at the time, would transform into one of the greatest technical headaches of the late 20^{th} century.

The mechanism of the problem was disarmingly simple. Systems that stored only the last two digits of the year could not distinguish 1900 from 2000. Calculating duration between two dates, chronological sorting, validity checking: all operations that risked producing absurd results. The first signs appeared as early as the 1980s, well before the expression *Year 2000 bug* made newspaper headlines. In 1988, the British chain Marks & Spencer refused a delivery of canned goods whose system had interpreted the expiration date in 2000 as 1900, considering the products expired for nearly ninety years. The anecdote raised smiles, but it heralded far more serious troubles.

Awareness truly began around 1990. Experts started multiplying warnings: banking malfunctions, electrical grids shutting down, air traffic control paralyzed, medical equipment failing. In 1992, Mary Bandar, a 104-year-old woman, received an invitation to join a kindergarten class-her birth year "88" having been read as 1988 rather than 1888. The incident could have been amusing if it hadn't revealed the scale of the looming problem.

Governments finally reacted. The United Kingdom created "Taskforce 2000" in 1996, followed the next year by "Action 2000", endowed with an initial budget of one million pounds sterling that climbed to 17 million. The United Nations established in February 1999 the International Y2K Cooperation Center, funded by the World Bank. Companies launched massive compliance programs. The New York Stock Exchange devoted 30 million dollars to it over seven years.

But the problem wasn't confined to large mainframe systems. Programmable logic controllers, ubiquitous in industry for controlling machines and processes, often used two-digit dates. PCs presented their own complications, related to their real-time clock and BIOS. Certain versions, like Award v4.50 from 1994-1995, simply couldn't handle dates beyond

1999, just as Windows 95 was also incompatible with the transition to the year 2000.

Faced with this situation, a few technical strategies emerged. Expanding years to four digits represented the most solid solution, but it required thoroughly revising programs and databases. The so-called "windowing" technique preserved the two-digit format while adding interpretation logic: "00" to "19" corresponded to 2000-2019, "20" to "99" to 1920-1999. This method, less costly, had the flaw of simply postponing the problem to a later date.

Audits multiplied from 1997 onward. Firms began requiring their clients to certify the compliance of their critical systems to guarantee business continuity. Specialized companies developed automated tools to detect and correct problems in source code. COBOL programmers came out of retirement, demanding comfortable salaries to work on systems they had built decades earlier.

Tests revealed worrying flaws. The British Rapier missile system proved inoperable after 2000. A Swedish nuclear power plant shut down automatically during a millennium transition test. At Chrysler, a 1997 test paralyzed a plant's security system, blocking access and preventing payroll management.

The overall cost of corrections ranged between 300 and 500 billion dollars. The United States spent 34 billion, Italy 2.5 billion, Venezuela 100 million. Industrialized countries, whose economies relied more heavily on digital systems, invested proportionally more than others.

Ultimately, the transition to the year 2000 caused less chaos than feared. Incidents occurred: fifteen nuclear reactors shut down, card payment systems malfunctioned, power outages affected Hawaii. But the predicted catastrophe didn't happen, a direct result of efforts deployed during the preceding decade.

This experience left its mark. It demonstrated that software engineering principles like abstraction and encapsulation weren't just theoretical concepts. It highlighted the dangers of single points of failure and the value of loose coupling between systems. It proved that international mobilization in the face of an identified computer threat remained possible.

Paradoxically, the success of "Y2K" management hindered the adoption of certain lessons. The industry continued to favor speed to market over robustness. Supply chains tightened, reducing their ability to absorb shocks. The current cybersecurity crisis testifies to this persistent difficulty in designing computer systems that are both robust and secure. The Year 2000 bug was, in the end, a missed opportunity to truly learn from our mistakes.

This illustrates well two architectural principles to keep in mind in any design work, computer-related or not: “Today’s problems come from yesterday’s solutions” and “Cause and effect can occur at distant times and places”.

IX

2000

When humanity went online

The 21st century began in turmoil. The September 11, 2001 attacks shook our world to its very foundations. Overnight, the vision of digital technology was transformed. Governments toughened their approach to computer security, critical infrastructures were fortified, and communications were placed under surveillance. In Washington, the Patriot Act reshaped our relationship with digital privacy.

The Internet became democratized like never before. We witnessed the birth of what was called Web 2.0, a transformation where users became active participants. MySpace and Facebook created new virtual social territories. Twitter changed our relationship with news. Personal blogs flourished everywhere thanks to WordPress, giving voice to anyone who wished it. This ferment redefined our way of existing online.

The digital economy weathered a salutary storm. The bursting of the Internet bubble swept away the most fragile projects. The survivors had to prove their real value. Google established itself as the cartographer of this new information universe through its PageRank algorithm. Amazon revolutionized commerce by creating a completely reimagined shopping experience.

Mobility took center stage in our daily lives. The first smartphones

began to transform our habits. The iPhone in 2007 caused a total disruption. The way we touched and manipulated devices would never be the same again. Wireless networks multiplied, Wi-Fi became ubiquitous, and 3G gave us the Internet in our pockets.

E-commerce matured and payment systems became secure, inspiring confidence in even the most reluctant consumers. Data analysis enabled online retailers to know us sometimes better than we knew ourselves, suggesting products with unsettling accuracy. To meet this explosion in usage, data centers had to reinvent themselves, become more efficient, less energy-intensive.

A new sharing experience emerged with peer-to-peer technologies. Napster rattled the music industry, while BitTorrent transformed our sharing patterns. The major record labels initially resisted before adapting. Last.fm and Spotify invented new ways of listening to music, heralding the coming video streaming revolution.

In businesses, work was transformed. Broadband and collaborative tools made remote work possible. Organizations invested massively in their information systems, deploying ERP and CRM solutions to better manage their operations and customer relationships.

Open source gained legitimacy. Linux, once considered a fringe project, took its place in the server rooms of major corporations. Firefox proved that a free browser could hold its own against giants. Passionate communities formed around these projects, demonstrating the power of collective development.

The digital divide concerned governments worldwide. School equipment programs were launched. Internet access became a matter of social equality. In Asia, China and India built powerful technology industries in record time, transforming the global balance of the sector.

In datacenters, virtualization exploded and was a game-changer. A single physical server could host multiple virtual machines. This evolution gave rise to cloud computing. CIOs began to outsource their infrastructures, gaining agility.

Cybersecurity became a major issue. Cyberattacks multiplied and became sophisticated. Viruses evolved into complex ransomware and malware. Faced with this growing threat, security solutions had to constantly adapt. Organized crime moved online, targeting banking and personal data.

In schools, digital technology made its entrance. Interactive whiteboards replaced chalk. Virtual Learning Environments created new connections between teachers, students, and parents. Educational resources were enriched, diversified, and digitized.

Scientific research achieved decisive breakthroughs thanks to growing computing power. Supercomputers made precise climate simulations possible, advances in genomics, discoveries in fundamental physics. Scientists learned to work with distributed computing systems to solve increasingly complex problems.

Automation intensified across all sectors. Embedded systems became smarter, more communicative. Industrial robotics reached a milestone, making factories more productive but also raising important social questions.

Environmental awareness appeared in the IT world. The power consumption of datacenters became a concern. Green IT emerged as a necessity, no longer just a marketing argument.

New forms of collaboration emerged. Wikipedia, launched in 2001, showed the way toward truly collective intelligence. Wikis and other collaborative tools transformed the way teams worked together and shared their knowledge.

Our electronic devices converged into a connected ecosystem. Televisions, cameras, Hi-Fi systems: everything became "smart" and interconnected. This convergence required new standards so our equipment could communicate with each other.

This decade shaped our current digital world. It was not merely a technological evolution, but a transformation of our society. We changed the way we communicate, work, and entertain ourselves. The period 2000-2010 will remain in history as the time when humanity truly went online, when our world became irrevocably connected.

C#

Anders Hejlsberg arrived at Microsoft in 1996 with a solid reputation: he had developed the Turbo Pascal compiler and led the creation of Delphi at Borland. Microsoft then entrusted him with an ambitious mission: designing a new programming language for its future .NET platform. This language, initially called "Cool" during its conception, would become C#. The goal was clear: offer a modern alternative to Java, avoiding its pitfalls while adopting its best ideas.

C# distinguished itself from the start with its unified type system. All types, even basic ones like `int` or `double`, inherit from a single root type called `object`. This architecture simplifies data manipulation, regardless

of its nature. The language also incorporates automatic garbage collection, which reclaims space occupied by unused objects, and structured error handling through exceptions.

Each new version enriched the language with substantial features. Version 2.0 introduced generics, which strengthened code reusability and type safety. Anonymous delegates expanded possibilities in event-driven programming. With version 3.0 came LINQ (Language Integrated Query) features, which radically transformed how data is queried and manipulated.

Hejlsberg and his team paid particular attention to version compatibility. Existing applications had to continue functioning with new library versions. This concern shows through in several design choices: the `virtual` and `override` modifiers remain distinct, method overload resolution rules are clearly defined. This rigor guarantees program longevity.

Research at Microsoft played a decisive role in the language's evolution. Don Syme and Andrew Kennedy, researchers at Microsoft Research Cambridge, developed a prototype called Gyro to experiment with generics before their integration into C#. This collaboration between research and development illustrates well the pragmatic approach adopted: testing new ideas in exploratory projects before incorporating them into the production language.

C# gradually opened up to functional programming. Type inference, lambda expressions, all these concepts from languages like Haskell or ML were integrated over time. This evolution follows a general trend in the industry: programming paradigms blend together to offer developers more flexibility. The language nonetheless remains accessible to beginners, a characteristic intended from its conception.

The question of database integration comes up constantly. Developers must often juggle between C# and SQL, what Hejlsberg calls an impedance mismatch between these two worlds. LINQ addresses this problem by unifying the querying of various data sources directly in C# code, with consistent syntax.

Microsoft evolved C# and Visual Basic .NET in parallel, two languages that share many features while preserving their specificities. This strategy aimed to respond to the distinct preferences of their respective communities. The development team also maintains regular exchanges with researchers working on F#, the .NET platform's functional language, to explore new directions.

ECMA-334 and ISO/IEC 23270 standardization established technical standards guaranteeing implementation consistency. This standardization reinforced the portability of C# applications and their independence from

Microsoft.

C# resolutely adopts a component orientation. The language offers specific syntactic constructs for creating and using autonomous and self-documented software modules. This modular approach facilitates the development of complex applications and their long-term maintenance.

The C# developer community has considerably expanded, creating a rich ecosystem of libraries, tools, and best practices. Microsoft has managed to keep the language cutting-edge while preserving its stability and backward compatibility. The shift toward more declarative programming styles continues: developers describe their objectives rather than implementation details, leaving the compiler to optimize execution.

C# has adapted to the industry's changing needs without denying its founding principles of simplicity and productivity. Guided by a clear vision and nourished by research, its continued development makes it an indispensable tool for modern software development.

FLAC

Josh Coalson launched the development of FLAC in 2000. His ambition? To create an open format for lossless audio compression, in response to the proprietary solutions that dominated the market at the time. Music lovers and sound professionals alike were seeking alternatives to lossy formats such as MP3, and Coalson would meet this demand.

FLAC's lossless compression reduces an audio file to 50 or 60% of its original size without losing a single bit of sound information. A 300 MB Broadcast Wave Format file shrinks to a 100 MB file after compression. Once decoded, the result is strictly identical to the original in acoustic terms. This capability radically distinguishes FLAC from lossy formats that sacrifice information to save space.

The project joined the Xiph.Org Foundation in 2003. This nonprofit organization already hosted open multimedia formats such as Vorbis, Opus, and Theora. FLAC's integration into this ecosystem of open standards reinforced its legitimacy. Version 1.2.1 was released in 2007 and marked a form of maturity: the specifications have since undergone no major modifications. The reference tools continue to evolve, with version 1.3.2 released in January 2017.

FLAC's designers favored a pragmatic approach. The format consumes little memory during decoding and relies exclusively on integer operations.

These technical choices enable its deployment on systems with modest resources. The data stream is organized as metadata followed by audio frames. Each frame contains a header with a 14-bit synchronization code, subframes, padding bytes, and a frame footer.

The flexible format accepts up to 8 audio channels and samples encoded between 4 and 32 bits. For stereo, four encoding modes coexist: independent, left/side, right/side, and mid/side. The compression thus adapts to the nature of the processed sound signal.

The music industry adopted FLAC in successive stages. Initially confined to audiophile circles and alternative platforms, the format gained ground with the emergence of high-quality streaming. TIDAL and Deezer Elite launched their services in 2014, with Qobuz following suit. Online distribution evolved in parallel: 7digital and Bandcamp now offer FLAC downloads.

Native support by operating systems represented a decisive step. Android integrated it from version 3.1, and Windows 10 followed. Apple maintained its own strategy with ALAC, initially proprietary and then released as open source in 2011. Recent web browsers, Chrome 56 and Firefox 51, play FLAC files directly.

Digital preservation finds considerable advantages in FLAC. Each file embeds an MD5 fingerprint of the original audio data in its header to verify integrity. CRC checksums identify corrupted frames during streaming. These built-in validation mechanisms give FLAC an advantage over the WAV format, which requires external verification.

Heritage institutions have taken interest in the format's potential. The audio and video division of the U.S. Library of Congress considered its use for accessing sound recordings as early as 2005. The Internet Archive uses it in its Live Music Archives, where concerts and sessions are often available in both FLAC and WAV. The European Broadcasting Union uses it to distribute concert recordings via Musipop: original WAV files are converted to FLAC before satellite transmission.

Specialized tools accompanied the format's growth. VLC, Clementine, and Foobar2000 provide playback across different platforms. Conversion utilities like dbPowerAMP and FFmpeg facilitate exchanges with other formats. Mp3tag and Kid3 manage metadata.

The Internet Engineering Steering Group approved the creation of the CELLAR working group in 2016 to formalize FLAC specifications. This initiative aims to establish an official standard and strengthen the format's sustainability in the digital audio landscape.

An open project like FLAC can establish itself as a de facto standard in its field. The absence of patents or royalties on its use favors its adoption

by the music industry and cultural institutions. This format remains a relevant choice for digital archiving and high-quality music distribution.

JSON

Douglas Crockford was working at State Software and looking for a simple way to enable communication between a server and a web browser. He noticed that JavaScript had a convenient syntax for representing objects: key-value pairs within braces. The idea came to him to extract this notation from the language and turn it into a standalone data exchange format.

In 2001, Crockford formalized this subset of JavaScript under the name JSON, for JavaScript Object Notation. He published the specification on json.org and provided implementations in several languages. The format consisted of just a few rules: six data types were sufficient (strings, numbers, booleans, arrays, objects, and the null value) to represent complex structures. This brutal simplicity contrasted with the verbosity of XML, which was ubiquitous in data exchange at the time.

Official standardization came in 2006 with IETF RFC 4627. The document established the MIME type `application/json` and defined the precise syntax. Other versions followed: RFC 7158 in 2013, RFC 7159 in 2014, and RFC 8259 in 2017. Each revision clarified technical points and adjusted the specification to meet emerging needs in the field.

JSON became dominant because it addressed concrete needs. Its minimalist syntax remained human-readable while being easy for machines to parse. The absence of tags reduced the size of exchanged messages. And most importantly, it naturally meshed with JavaScript, the language that already dominated client-side web development. Developers had nothing new to learn because they were using structures they manipulated daily.

The web giants adopted it one after another. Yahoo! was among the first to integrate it into its services, followed by Google, Facebook, and Twitter. This massive adoption transformed JSON into a de facto standard for web APIs. REST architectures, which were becoming the norm for designing web services, aligned perfectly with this lightweight format.

The ecosystem around JSON developed at an astonishing pace. Libraries appeared in all major languages. Specialized variants emerged: GeoJSON for geographic data, JSON Schema for validating document

structure. NoSQL databases like MongoDB or CouchDB adopted it as their native storage format, extending its influence beyond simple data transport.

JavaScript frameworks like Angular or React manipulated JSON natively. Mobile development leveraged its lightweight nature to conserve bandwidth. JSON gradually became an invisible but ubiquitous component of web infrastructure, present in nearly every exchange between applications.

This simplicity also revealed its limitations. The inability to add comments hindered documentation. Number representation raised precision questions related to floating-point calculations. The absence of a standard format for dates forced developers to establish their own conventions. Variants like JSON5 or JSONC appeared to fill these gaps, at the risk of fragmenting the ecosystem.

A recent analysis of public JSON files shows that developers intensively exploit the format's capacity to represent nested structures. Objects contained in arrays themselves nested within other objects have become commonplace. This flexibility allows great expressiveness while maintaining a certain regularity that facilitates automated processing.

JSON has transformed development practices, and has fostered the growth of decoupled architectures where components communicate through well-defined interfaces. Its ease of manipulation has encouraged the adoption of functional and reactive paradigms. Development tools have integrated specific features for working with this format, from syntax highlighting to automatic validation.

The JSON specification is deliberately minimalist. Its designers chose to preserve the original simplicity rather than add features. Specific needs are covered by extensions and complementary tools that don't affect basic compatibility. This stability ensures the format's longevity in a constantly evolving technological environment.

The format was not imposed by a standardization body but naturally adopted because it met a real need. Its success stems from its alignment with existing practices and its ability to remain simple when so many other solutions sought complexity. JSON has become a pillar of modern web infrastructure, one of those elements we no longer notice because they seem so self-evident.

BitTorrent

By early 2000, the Internet was showing certain limitations. Transferring a large file was extremely challenging: email would saturate, FTP servers would collapse under the load. Bram Cohen had had enough of Internet bubble startups. These companies promised the moon, burned through investors' money, then disappeared before releasing anything at all. He decided to take matters into his own hands.

His professional experience had taught him one thing: fragmenting files for secure storage worked well. This idea kept nagging at him. What if this principle were applied to data sharing? Instead of a single server distributing a file to hundreds of users, why not split the file into pieces and let everyone share what they had locally?

This is how BitTorrent was born in 2001. The concept could be summed up briefly: a file is divided into segments. You download these segments from multiple sources simultaneously. Meanwhile, you're already sharing the pieces you've received with others. The more people involved, the better it works. Cohen named this reciprocity mechanism *tit-for-tat*, a name that captures the spirit well: give and take.

Summer 2001 saw the release of the first beta version. Cohen presented his protocol the following year at a conference. His initial ambition was to distribute Linux distributions. However, the technical architecture concealed genuine sophistication. The .torrent files contain only metadata: filename, size, segmentation, tracker address. This tracker coordinates exchanges between participants without ever storing a single byte of content.

Two algorithms made the difference. The first, called *rarest first*, prioritizes the least widespread segments in the network. The second, *choking*, regulates transfers by favoring those who actively share. These mechanisms ensure that a file remains available when thousands of users download it simultaneously.

Success materialized. By 2005, BitTorrent represented a substantial portion of global Internet traffic. Some used it for pirated content, as with all sharing protocols before it. Others saw an opportunity. Opera integrated BitTorrent into its browser that year and used it to distribute its updates. Blizzard Entertainment adopted it to distribute World of Warcraft patches. The Internet Archive followed in 2012 to make its digital collections accessible.

Regarding the protocol's internal mechanics, files are divided into 256 KB blocks. Each block receives a unique identifier via SHA-1 hashing that guarantees its integrity. The system maintains a dynamic list of connected

peers and continuously adjusts transfers according to each connection's performance. This decentralization ensures robustness and scalability.

Cohen had hit the mark: a peer-to-peer system could solve massive distribution problems. The bandwidth savings attracted businesses. Distributing a multi-gigabyte software update to millions of users cost a fortune with the traditional approach. BitTorrent changed the game.

The open source code fostered the emergence of a developer community. The protocol's relative simplicity accelerated its adoption. The incentive mechanisms for sharing guaranteed its practical effectiveness. These three elements explain why BitTorrent became so widely adopted.

More than twenty years later, the protocol is still running. Use cases have evolved, competing technologies have multiplied, but BitTorrent remains relevant for distributing large files. An idea born from frustration with technical limitations can durably transform Internet usage.

Apple Mac OS X

Apple went through a difficult period in the 1990s. Its operating system Mac OS, created in 1984 with the first Macintosh, lagged technically behind the competition. Originally designed for a computer with 128 KB of memory, it lacked preemptive multitasking and memory protection. MultiFinder, introduced in 1988, only provided a rudimentary form of cooperative multitasking. The architecture was aging poorly. By the mid-1990s, Apple relied on decade-old code, initially written for Motorola 68000 processors and adapted to PowerPC with varying degrees of success. Part of the kernel even ran in a 68K emulator, which only made performance worse.

Apple then launched the Copland project in 1994, with the ambition to create a modern system that would retain the Mac OS interface and compatibility. The company held numerous presentations for developers and published documentation, but Copland never achieved acceptable stability. The project was cancelled in 1996. Apple found itself in an urgent situation.

The solution came from elsewhere, through the acquisition of NeXT, the company Steve Jobs had founded after leaving Apple in 1985. NeXT had built NEXTSTEP, a remarkable system based on the Mach microkernel from Carnegie Mellon University. NEXTSTEP combined this modern kernel with a BSD layer and offered preemptive multitasking, memory

protection, support for multiple processor architectures, and an object-oriented development environment in Objective-C as early as 1989.

Transforming NEXTSTEP into Mac OS X required considerable work. Apple updated the Mach 2.5 code to version 3.0, updated the BSD portion with code from 4.4BSD and FreeBSD, added support for HFS file systems and Apple network protocols. The team developed I/O Kit, a new driver system that replaced the old DriverKit. This layer, written in a subset of C++, enabled an object-oriented approach for managing peripherals.

The Mac OS X architecture was based on modularity. The XNU kernel (*X is not UNIX*) combined three components: Mach managed system resources, BSD provided UNIX compatibility and network services, and I/O Kit handled drivers. This organization facilitated maintenance and evolution. Drivers loaded dynamically as KEXTs (*Kernel Extensions*), modules that could be added without recompiling the kernel.

For application transition, Mac OS X offered several environments. Cocoa, heir to NeXT's OpenStep libraries, created native applications. Carbon offered a modernized version of classic Mac OS APIs and facilitated porting existing software. A Classic environment ran legacy Mac OS applications without modification.

The innovations went beyond the kernel. The system introduced Mach-O, a universal binary format capable of containing code for different architectures in a single file. This feature simplified software distribution and prepared for future hardware transitions. The boot process relied on OpenFirmware for PowerPC, then on EFI for Intel processors, providing extended capabilities compared to traditional BIOS.

Memory management demonstrated the care given to performance. On x86 systems, Mac OS X adopted a 4/4 approach where user space and the kernel were not mapped simultaneously, unlike the classic 3/1 or 2/2 GB split. This design made better use of available memory.

The transition to Intel processors in 2005-2006 demonstrated the architecture's flexibility. Rosetta technology, based on Transitive's QuickTransit, executed PowerPC applications on the new x86 processors through dynamic recompilation. The 32-bit kernel continued to function on 64-bit machines in compatibility mode, preserving compatibility with existing extensions while providing access to 64-bit features.

Mac OS X's influence extended beyond desktop computers. The system served as the foundation for iOS for the iPhone and iPod touch, adapting the XNU kernel and certain components to the ARM architecture. This code reuse gave Apple technological consistency across its platforms.

The release of Darwin's source code, the core of Mac OS X, under the

Apple Public Source License in 1999 marked a turning point in Apple's strategy. This openness allowed developers to study the system's implementation and contribute to it. Projects like PureDarwin demonstrated that derivative systems could be created.

Mac OS X achieved a synthesis between UNIX heritage, NeXT innovations, and Apple's expertise in interfaces. The system enabled Apple to modernize its platform while preserving compatibility with existing applications. This solid technical foundation supported Apple's expansion into new markets and continues to evolve with user needs and technological advances.

SHA-256

The late 1970s marked the beginning of cryptographic hash functions. These algorithms, which transform any data into a fixed-size digital fingerprint, would become major building blocks of computer security.

In 1976, Whitfield Diffie and Martin Hellman published their groundbreaking paper on public-key cryptography. They explained that a one-way hash function is necessary to construct digital signatures. The first concrete work arrived shortly after: Michael Rabin proposed a design based on DES encryption that produced a 64-bit result. Gideon Yuval demonstrated that collisions could be found by exploiting the birthday paradox with a complexity of `2^(n/2)`. Ralph Merkle, for his part, established the basic requirements: resistance to collisions, second preimages, and preimages.

The 1980s saw numerous proposals emerge. The cryptographic community realized the importance of these primitives for securing digital communications. Ivan Damgård formalized the definition of collision resistance in 1987. Two years later, Moni Naor and Moti Yung introduced a variant called Universal One Way Hash Functions.

At the turn of the 1990s, Ronald Rivest created MD5, an evolution of MD4 optimized for software. This function quickly became successful: it proved to be approximately ten times faster than DES in software implementation. More importantly, MD5 escaped the export restrictions that weighed on encryption algorithms and could be used freely.

The National Security Agency then developed the SHA (Secure Hash Algorithm) family. NIST published SHA-0 in 1993. But the agency discovered a vulnerability two years later and released a corrected version

called SHA-1. In 2001, facing advances in cryptanalysis and the growing power of computers, the NSA designed the SHA-2 family. SHA-256 is part of it.

SHA-256 produces a 256-bit fingerprint and operates on 32-bit words. The algorithm processes messages in 512-bit blocks after specific padding. The process includes message expansion and iterative compression based on a Merkle-Damgård function. This construction ensures that the security of the compression function extends to that of the complete hash function.

The first cracks in MD5 appeared in 1992. Den Boer and Bosselaers found collisions for the compression function. In 1996, Hans Dobbertin discovered collisions for MD5 with a random initial value. These results did not yet really concern the community.

Everything changed in 2004. Xiaoyun Wang's team achieved a breakthrough by perfecting differential cryptanalysis. Their work enabled finding collisions for MD5 in milliseconds. The techniques developed by Wang significantly reduced SHA-1's security margin. The cryptographic community launched into a wave of research.

On December 31, 2008, Alexander Sotirov and his team made a major impact. They created a malicious CA certificate that exploited MD5 collisions. This attack theoretically allowed impersonating any website. The industry understood that MD5 had to be abandoned, despite its massive presence in existing systems.

NIST launched a competition in November 2007 to select SHA-3, a new standard hash algorithm. The objective: diversify available options and prepare for SHA-2's replacement if needed. The competition attracted 64 submissions, of which 51 were selected for the first round. In July 2009, 14 candidates advanced to the second round.

Meanwhile, SHA-256 held firm. The best known attacks only compromised a limited number of the algorithm's steps. SHA-256 resisted the techniques that had broken MD5 and weakened SHA-1. This robustness explains its growing adoption in numerous security protocols and applications.

Bitcoin's arrival in 2009 gave SHA-256 an unexpected dimension. The cryptocurrency's proof of work relies on finding partial collisions, exploiting the function's preimage resistance. This application demonstrated SHA-256's versatility.

SHA-256 is also found in the Internet of Things, distributed embedded systems, random number generation, and data encryption. This diversification testifies to its maturity and the trust it inspires.

This history shows that reliable alternatives must always be available when weaknesses appear in existing standards. It recalls the importance of maintaining standardized options. SHA-256's longevity proves that rigorous design anticipates developments in cryptanalysis and available computing power.

Microsoft .NET Framework

At Microsoft's Professional Developers Conference in 2000, the company unveiled .NET, a new *framework* for Windows that would redefine how software was built on its platform. This project, which had started under the rather uninspiring name of "Next Generation Windows Services", represented more than just an update to existing tools. Developers received the first beta versions of .NET 1.0 that same year.

Microsoft wanted to solve several problems at once. On one hand, Java was gaining ground and the Redmond company needed a credible response. On the other hand, Windows development remained fragmented across different technologies that didn't always communicate well with each other. The idea was to create something more coherent that would truly simplify developers' work. At the heart of this new platform was the Common Language Runtime, an execution environment that automatically handled previously complex tasks such as memory management and security aspects.

The first commercial release arrived in 2002 with Visual Studio .NET and framework 1.0. This version laid the foundations for an architecture that would endure through the following years: a common class library, a unified type system, and the ability to write code in multiple languages such as C# or Visual Basic .NET. Version 1.1 followed in 2003, bringing its share of performance improvements and new features.

With Visual Studio 2005 and framework 2.0, things took on another dimension. Generics made their appearance, allowing developers to write more robust and reusable code. This version consolidated .NET as an essential platform. The year 2008 marked a new turning point with multi-targeting, which gave the freedom to simultaneously target different framework versions-2.0, 3.0, and 3.5.

Versions then followed at a regular pace. Visual Studio 2010 integrated .NET 4.0, then the 2012 version welcomed 4.5. This progression allowed the platform to adapt to new software development requirements, particularly in web and cloud computing.

.NET's technical architecture rests on a well-designed stack of layers. At the base, the CLR executes code and automatically manages memory through its garbage collector. Above come the class libraries, which provide everything needed to build applications. Depending on requirements, the framework offers different environments: console applications for processing without an interface, Windows Forms for traditional desktop software, ASP.NET for the web.

.NET's compilation model stands out through an original approach. Source code doesn't transform directly into machine language. It first goes through an intermediate step, MSIL (Microsoft Intermediate Language). This strategy offers several benefits: code is portable across different architectures, it can be optimized at runtime, and programming languages are interoperable.

This interoperability actually constitutes one of .NET's strengths. Beyond C# and Visual Basic .NET, the platform welcomes F#, IronPython, and even IronRuby. The Common Type System defines how to declare and use data types, while the Common Language Specification establishes rules ensuring that all these languages can work together.

Anders Hejlsberg played a decisive role in this story. Chief architect of C#, he had previously worked at Borland on Turbo Pascal and Delphi. This experience informed the framework's design. Under his guidance, C# became .NET's flagship language, regularly integrating innovations such as LINQ, which allowed writing queries directly in code.

.NET's influence on the software industry was massive. The platform transformed how companies developed their Windows applications, giving them a truly productive and modern environment. The transition to the web became simpler with ASP.NET, which made creating solid web applications less challenging.

The framework introduced advanced concepts that few platforms offered at the time. Aspect-oriented programming, reflection, attributes: all tools that enabled the construction of more sophisticated applications. Automatic memory management freed programmers from tedious and error-prone tasks. Deployment became less problematic thanks to automatic installation of necessary components.

Microsoft progressively enriched .NET to keep up with technological developments. Cloud computing support, mobile development, universal Windows applications: the platform continually adapted to new paradigms and developers' expectations.

Creating .NET represented a colossal investment. Hundreds of engineers worked on it for years. This effort paid off: the framework became a de facto standard for Windows development, adopted by millions of de-

velopers worldwide. Its influence endures with .NET Core, its modern, cross-platform version.

Mozilla Firefox

In 1993, the Mosaic browser was created at the National Center for Supercomputing Applications. Its success was swift. Marc Andreessen, one of its creators, founded Netscape Communications Corporation the following year. The new browser, Netscape Navigator, captured the Web market.

Microsoft responded by developing Internet Explorer, which it integrated directly into Windows 95. The strategy worked. Netscape gradually lost ground. Facing this decline, the company made a bold decision in 1998: it released its browser's source code through the Mozilla project. The mozilla.org organization was created to coordinate this community development.

The situation became more complicated in 1999 when AOL acquired Netscape. The integration went poorly, and Mitchell Baker, a lawyer at Netscape and then at AOL, led mozilla.org with the rather original title of "Chief Lizard Wrangler". Her team continued development while maintaining autonomy from AOL.

After four years of work, Mozilla was released in 2002 in its first stable version. The browser was packed with features: tabs, pop-up blocking, download manager. But its overly cluttered interface put off mainstream users. A small team then decided to start over with a simpler foundation.

The new project went through several names: Phoenix, Firebird, before becoming Firefox to avoid trademark conflicts. Version 1.0 launched in November 2004. Success was immediate. Firefox introduced innovations that would become industry standards: natively integrated tabs, a modular extension system, a streamlined interface. The approach contrasted sharply with what existed at the time.

AOL discontinued Netscape Navigator in 2003. Mozilla transformed into an independent nonprofit foundation, with $2 million from AOL as seed funding. The Mozilla Foundation could thus develop Firefox freely, staying true to free software principles and an open Web.

The browser arrived at the right time. Internet Explorer had stagnated technically. Firefox offered a real alternative, more secure and better at respecting privacy. Its open development attracted a global community

of contributors. The foundation signed agreements with search engines, particularly Google, which generated revenue without compromising the project's independence.

From 2004 to 2008, Firefox steadily gained market share from Internet Explorer. It reached 20% of the global market in 2008. An open source project was directly competing with a dominant proprietary software. Firefox pushed Web standards and interoperability, where Microsoft had favored its closed ecosystem.

Google Chrome appeared in 2008. This browser, first based on WebKit then on Blink, gradually established itself. Firefox nonetheless retained its loyal users, attached to its vision of an open Web that respects individual freedoms.

The mobile explosion led Mozilla to launch Firefox OS in 2013. This mobile operating system, built on Web technologies, aimed to offer an open alternative to iOS and Android. Despite partnerships with manufacturers, Firefox OS never took off. The project ended in 2015.

In 2024, Firefox has approximately 200 million users. The browser continues to innovate in privacy protection, performance, and standards compliance. The Mozilla Foundation pursues its mission for an open and accessible Internet for all.

Firefox demonstrates that an open source project can disrupt a market dominated by commercial giants. The browser proved the importance of transparent governance and an engaged community. Mozilla's business model remains exceptional, however. The foundation derives substantial revenue from its partnerships with search engines without abandoning its nonprofit mission. This balance remains rare in the software world.

Apple Xserve

In May 2002, Apple returned to the server market with the Xserve, a computer designed for server racks. The company had not ventured into this territory since the crushing defeat of its Network Servers in the 1990s. At that time, Macs equipped many Fortune 500 companies, but they remained at the doorstep of their IT departments, never truly gaining entry.

The Xserve adopts a compact 1U format with a height of 4.4 cm. The first generation features PowerPC G4 processors at 1 GHz and DDR

memory, a first for Apple. The system architecture relies on four hot-swappable hard drive bays, each with its own Ultra ATA-100 bus. These proprietary modules come in 60 or 120 GB capacities, reaching a maximum total of 480 GB.

For connectivity, the device offers two Gigabit Ethernet ports, three FireWire ports, two USB ports, and one serial port for terminal connections. The front panel displays disk activity, processor load, and network status in color. Apple launches two versions: $2,999 for the single-processor model, $3,999 for the dual-processor model, including an unlimited Mac OS X Server license.

The machine evolves quickly. As early as 2004, the Xserve G5 takes over with PowerPC G5 processors reaching up to 2 GHz. This performance boost requires a complete chassis redesign to dissipate heat, which reduces the number of bays. This version introduces ECC memory, a novelty in Apple's catalog, and switches to SATA drives.

Apple completes its offering with the Xserve RAID, a 3U rack-mounted storage system. This device accommodates 14 modules of 180 GB each and communicates with the server via 2 Gb/s Fibre Channel links, achieving throughput of 400 MB/s. Its pricing challenges the competition: in 2004, the cost per gigabyte stands at $3.14, far below Dell ($9.05) and HP ($11.39).

The transition to Intel occurs in 2006. The Xserve adopts 64-bit Xeon processors reaching up to 3 GHz, multiplying the G5's performance by five. These processors generate less heat, which allows for the addition of redundant power supplies and increases storage capacity to 2.5 TB. Options multiply: dual-layer DVD burner, 32 GB of RAM, integrated RAID card.

The final iterations emerge between 2008 and 2009 with quad-core Xeon 5400 series processors. These machines reach new heights thanks to their refined design: independent 1,600 MHz front-side buses, 800 MHz DDR2 ECC memory, PCI Express 2.0 connectors. Storage breaks new ground with 15,000 RPM SAS drives offering 125 MB/s throughput and 3.5 ms access times.

Mac OS X Server constitutes the Xserve's major asset. This UNIX-certified system integrates file and printer sharing, web hosting, messaging, and media streaming. The 10.5 Leopard Server version enhances collaborative functions with iCal Server for calendars, Wiki Server for teamwork, and Podcast Producer for automatic multimedia content publishing.

Technical support adapts to enterprise needs: phone and email support available around the clock. Spare parts kits enable technicians to perform on-site interventions. The Server Monitor application continuously moni-

tors critical components: temperatures, fans, power supplies, hard drives, and network interfaces.

Apple discontinues the Xserve RAID in February 2008, favoring Promise Technology solutions certified for Xsan 2 instead. The Xserve server disappears in January 2011, but it remains a fine example of technical expertise in the enterprise server domain. The company then recommends using Mac mini or Mac Pro systems configured as servers. This withdrawal reflects a shift in Apple's priorities, turning more toward consumers and professionals than toward IT departments. For nine years, Xserve offered an elegant alternative to conventional solutions, combining performance, ease of use, and compact size.

Tor

In the 1990s, the U.S. Naval Research Laboratory was working on a problem that seemed quite removed from public concerns: how to protect the communications of its agents around the world? The solution they developed was called Onion Routing. No one imagined then that this military technology would become one of the most important tools for defending privacy on the Internet.

The technical principle is ingenious. Each message passes through several intermediate servers, encrypted in successive layers like the peels of an onion. At each step, a server removes one layer and forwards the packet to the next relay, never knowing either the origin or the final destination. An observer intercepting the traffic sees only one node among others, unable to trace the complete chain.

This era also saw the emergence of cypherpunks, those libertarian technologists convinced that encryption represented a bulwark against state surveillance. The meeting between these activists and military researchers at a university seminar created an unlikely alliance. The military needed a vast network of civilian users to drown their sensitive communications in abundant traffic. The cypherpunks wanted to democratize privacy protection technologies. Each party found what it needed.

In 2002, the protocol was reimplemented for the general public and released under a free license. This source code transparency mattered greatly: how could one trust a security tool funded by the U.S. State Department without being able to verify its operation? The community could now audit it freely.

The early days raised thorny technical questions. Should they add dummy data that masks genuine communications? Theoretical security would benefit, but connections would become unbearably slow. The developers made a choice: a network that's too slow would never attract enough users. Yet Tor's strength relies precisely on numbers: the more diverse the profiles, the better sensitive communications blend into the mass of daily traffic.

This philosophy guided the entire evolution of the project. Tor had to remain accessible to the greatest number while maintaining a high level of security. The gamble paid off. Journalists protecting their sources, human rights activists in authoritarian regimes, companies concerned about their trade secrets, ordinary citizens refusing advertising tracking: all found in this network an answer to their needs.

The network also makes it possible to host hidden services, recognizable by their ".onion" addresses. The physical location of these sites is masked, which raises controversies. Silk Road, that illegal marketplace shut down by the FBI, used Tor. But reducing the network to its criminal uses would be dishonest. Edward Snowden's revelations in 2013 showed that the NSA was failing to compromise Tor on a large scale, confirming its technical robustness against the world's most powerful adversary.

The architecture relies on thousands of volunteer servers scattered all around the world. Individuals, associations, sometimes universities, run these relays out of conviction: preserving a free Internet. This decentralized infrastructure constitutes a technical bulwark against mass surveillance. It also illustrates that an alternative governance of networks is possible, outside commercial or state logic.

Tor's major innovation concerns not so much the encryption of content as the protection of metadata. Who talks to whom, when, how many times? This information often reveals more than the content of the messages themselves. By tackling this structural problem, the network has inspired many other privacy protection projects.

Tor's sustainability rests on a fruitful paradox. The U.S. State Department continues to fund its development to promote freedom of expression in countries that censor the Internet. Activists sometimes use it to escape Western surveillance. This duality, far from weakening the project, strengthens its legitimacy and resilience. Serving state and citizen interests simultaneously is not a contradiction but a condition for survival.

Faced with the limitations of abstract mathematical models, the developers favored empirical observation of real threats and concrete uses. This pragmatic approach has produced an imperfect but effective system against mass surveillance. The initial technical challenges forged this

method: better a usable tool than a theoretically perfect but practically unusable solution.

The project shows that an alternative infrastructure can emerge from the margins of the Internet to become a pillar of online freedom of expression. Technical communities can create tools serving the public interest, outside the usual circuits. Onion routing born in a military laboratory has become a digital commons.

Apple iTunes

In 2000, the music industry faced an unprecedented crisis. Illegal MP3 file sharing on the Internet was exploding, major record labels clung to their old business models, and users were abandoning record stores. Steve Jobs, who had returned to Apple in 1997, saw an opportunity: to create a cohesive ecosystem around digital music.

The story truly begins in 2001 with the first iPod. This elegant portable player needed software so people could transfer their music to it. iTunes was born from this practical necessity: to offer Mac owners a simple way to convert their CDs into digital files and sync them with their device. Nothing new in itself, but a solution that actually worked.

Two years later, in 2003, everything changed. Apple launched the iTunes Store, an online shop where you could buy individual tracks for 99 cents. The concept was bold at the time. Record labels sold complete albums, not isolated songs. Jobs had to negotiate hard with the big five record companies. His argument? Ease of use and attractive pricing would turn people away from piracy. The label executives, skeptical but with their backs against the wall, eventually agreed.

The success exceeded all expectations. One million tracks sold in one week. Users appreciated being able to preview a song before buying, paying only for the songs they really wanted. The interface was intuitive, without unnecessary frills. iTunes introduced smart playlists, rankings, and personalized recommendations that changed how people discovered music.

In October 2003, Apple made a surprising choice: adapting iTunes for Windows. The Cupertino company abandoned its usual exclusivity and opened its doors to millions of new users. This strategic decision expanded the service's audience far beyond Mac enthusiasts. Anyone with a PC could now buy music online and sync their iPod.

Between 2004 and 2006, the music catalog constantly expanded. Deals with independent labels multiplied the available offerings. iTunes no longer just sold music: the platform welcomed podcasts, then TV series, and finally movies. The software gradually became a complete multimedia hub, far from its original purpose as a simple music library manager.

The App Store launched in 2008, designed for iPhone and iPod Touch applications. iTunes transformed into a digital crossroads where music, videos, applications, and digital books converged. The unified payment system and automatic synchronization between devices attracted more and more users. Apple had won its bet: creating an ecosystem that's hard to leave once you're in it.

The impact on the music industry is difficult to measure, but a few figures speak for themselves. In 2010, iTunes was the largest music seller in the United States, ahead of Walmart. Physical sales continued to collapse, certainly, but digital distribution revenues offset part of the losses. The service legitimized online music purchases and established a viable economic model for the digital era.

Technology evolved alongside usage patterns. Users accessed their library from any device thanks to cloud storage, introduced with iTunes Match in 2011. Streaming features gradually appeared, responding to new ways of consuming music. The AAC format chosen by Apple offered better sound quality than MP3 for a comparable file size. FairPlay digital rights management, less restrictive than competitors', allowed playback on multiple devices while reassuring record labels.

In 2019, Apple announced the end of iTunes. The software would be split into separate applications for music, podcasts, and video on macOS. iTunes proved that a polished interface, combined with an adapted business model, satisfies consumers without sacrificing content creators' interests. The platform redefined digital content distribution and inspired countless services that followed.

Skype

It's 1998. Niklas Zennström is working for Tele2, a Swedish telecom operator, when he pitches an Internet telephony project to management. The idea doesn't find takers at the time, but it keeps working on Zennström's mind. With the Dane Janus Friis, he first launches Kazaa, that *peer-to-peer* file-sharing software that will make headlines, before returning to his original project: Internet telephony.

In 2000, the Everyday.com portal gathers a team in which Zennström and Friis find three Estonian developers: Jaan Tallinn, Ahti Heinla, and Priit Kasesalu. Toivo Annus soon joins them. These developers come from Bluemoon, an Estonian company where they honed their technical skills creating video games during the Soviet era. The group forms the nucleus that will give birth to Skype.

Peer-to-peer technology, which the team perfected with Kazaa, becomes the heart of the system. Users communicate directly with each other, without going through central servers. Infrastructure costs plummet, communication quality soars. The name Skype comes from *Sky peer-to-peer*, first conceived as "Skyper", then shortened when they discover the domain name is unavailable.

Finding financing proves difficult. The legal controversies surrounding Kazaa have damaged the founders' reputation. They have to wait until July 2002 for the Draper Richards fund to agree to invest $250,000 for 5% of the capital. A modest valuation that will later prove to be a spectacular investment.

On August 29, 2003, Skype opens its doors to the public. About twenty people have developed this software that offers free computer-to-computer calls. The ease of use and sound quality immediately appeal. On the first day, 10,000 people download the application. A few months later, there are a million.

The company cultivates its own distinct style. The offices display no signage, a legacy of Kazaa's legal troubles that taught the founders discretion. In Tallinn, an inflatable pool sits in the meeting room. The team codes on the fly: as soon as a feature is designed, it's integrated into the software.

In 2004, Skype launches SkypeOut. The service opens up to calls to landlines and mobile phones at rock-bottom prices. Calls between Skype users remain free, but it's the beginning of monetization. Investors come running. The company raises $18 million from Index Ventures and Bessemer Venture Partners, among others.

The explosive growth attracts the giants. In September 2005, eBay acquires Skype for $2.6 billion. The Estonian developers become millionaires overnight, but the corporate culture begins to crack. Tensions rise between the Tallinn teams and those in London. The integration of a startup into a large company shows its limits, and relations deteriorate quickly. The founders retained intellectual property ownership of the peer-to-peer technology through their company Joltid, creating an inextricable situation. Microsoft finally acquires Skype in 2011 for $8.5 billion. A new chapter begins.

Under Microsoft's leadership, Skype evolves. Edward Snowden's revelations in 2013 show that the service, once renowned for its confidentiality, now cooperates with intelligence agencies. The rebel startup has become an institutional player in the digital world.

Skype's impact on telecommunications remains massive. The service made free international calls accessible to everyone, popularized video conferencing. In 2012, Skype handled 167 billion minutes of international calls, more than all traditional telecom operators combined.

The founders have taken different paths. Jaan Tallinn invests in projects related to humanity's survival; he now considers time more precious than money. The Estonian team, despite its fortune, maintains a certain modesty. No ostentatious spending, thoughtful investments.

Skype transformed an ambitious technical project into a universal communication tool. The service demonstrated the power of peer-to-peer and the importance of a simple interface. A technology can evolve, moving from disruptive innovation to institutionalization, while remaining useful to millions of people every day.

WordPress

In 2003, a software called b2/cafelog came to an abrupt end. Its creator abandoned the project, leaving behind open source code that two developers, Matt Mullenweg and Mike Little, decided to take over. Mullenweg was only 19 years old and his programming skills came from self-teaching, far from traditional university curricula. From this initiative was born WordPress, which would become one of the most widely used content management systems in the world.

The first versions of the software displayed two priorities: compliance with web standards and simple installation. The team developed the *Famous Five Minute Installation*, an installation process that required only five minutes where competitors often demanded more than half an hour. This philosophy of accessibility found its roots in the values of free software: offering everyone the freedoms to use, study, modify, and distribute the program. In contrast to proprietary software that locks down its code, WordPress opened itself completely.

Technical evolution followed a sustained pace. In 2004, version 1.2 introduced plugins, those extensions that allowed adding functionality without touching the system's core. The following year, themes and static

pages appeared in version 1.5, before 2.0 brought caching and a redesigned administration interface.

The year 2007 saw improvements in rapid succession. Version 2.1 integrated auto-save and spell checking. Widgets arrived with 2.2, facilitating customization of sidebars. Version 2.3 introduced tags and overhauled the taxonomy system, making content classification more flexible.

In 2008, the Happy Cog agency designed a new administration interface for version 2.5. The dashboard gained modular widgets and a shortcode API was born. A few months later, version 2.7 improved overall usability, added automatic updates, and simplified plugin installation. Threaded comment management enriched interactions between authors and readers.

Alongside technical development, Matt Mullenweg founded Automattic, the commercial company that hosts WordPress.com, the online version of the system. The organization adopted an unusual working model: teams were entirely distributed, with no central office. In 2014, Automattic exceeded one billion dollars in valuation with more than 300 employees spread across 37 countries.

WordPress's success rested on several pillars. The plugin system today counts more than 35,000 extensions, covering needs as varied as e-commerce, security, or performance optimization. The software's architecture appealed to developers who could rely on its components to create complex web applications, far beyond simple blogging.

The community organized around WordCamps, local conferences that spread worldwide. These gatherings created connections between users, developers, and businesses, nurturing a dynamic ecosystem where knowledge and best practices were exchanged.

Major organizations eventually adopted the platform. The New York Times, the Wall Street Journal, and the Washington Post used it for their websites. WordPress powered approximately 23% of global websites, proving its capacity to meet the most diverse requirements, from local artisans to international media.

Subsequent versions continued to enrich the system. Version 3.0 in 2010 brought custom post types and multi-site management. Responsive design integrated naturally, adapting to the smartphones and tablets flooding the market. The visual editor improved, emoji support arrived, and site icons integrated directly into the interface.

WordPress evolved into a complete application platform. Developers exploited its APIs to transform a simple blog into a booking system, an online store, or a learning portal.

The business model played on two fronts: the software remained free and open, but services like hosting, technical support, or premium features

generated revenue. This approach guaranteed the project's sustainability without sacrificing its accessibility.

Recruitment at Automattic reflected this particular culture. Candidates went through practical tests and trial periods, with actual skills counting more than diplomas. This method aligned with free software values, where effective contribution outweighed formal qualifications.

By democratizing website creation, WordPress enabled millions of people to exist and express themselves online. Its collaborative development model inspired other free projects, demonstrating that a global community could create and maintain large-scale professional tools.

Project management followed a pragmatic line. Technical decisions responded to users' real needs rather than passing trends. This stability reassured businesses choosing WordPress for their strategic projects, knowing the platform would evolve without abrupt disruption.

WordPress has become a universal web platform. This transformation was achieved while preserving the founding principles of simplicity and openness, while integrating the innovations necessary for its growth. The software managed to grow without denying its origins, remaining faithful to the spirit that drove its two creators in 2003.

Xen

In 1999, at the Computer Laboratory of the University of Cambridge, Dr. Ian Pratt and his team launched the XenoServers project. Their vision: build an infrastructure of distributed servers across the Internet where individuals or organizations, once authenticated, could purchase computing resources on demand. The idea was not new in principle, but its technical implementation promised to be ambitious. The name XenoServers refers to the Greek *xenos*, the stranger to whom one offers hospitality without granting blind trust.

The project's early steps focused on using Java virtual machines to execute client code. This solution ran into a prohibitive constraint that required rewriting all applications in Java. The team then changed course and aimed higher: virtualizing x86 hardware directly to run complete operating systems. At the time, virtualizing the x86 architecture was a technical challenge. Available approaches relied on instruction emulation and binary rewriting, methods that notably degraded performance.

Faced with these limitations, the researchers invented paravirtualization. The principle involves slightly modifying guest operating systems so they collaborate with a hypervisor, while allowing applications to run without modification. Keir Fraser, a doctoral student on the team, took charge of implementing this idea. He developed the core of the hypervisor and adapted Linux 2.2 to this new architecture. The hypervisor received the name Xen, which condenses that of the initial project.

One of Xen's technical bets lies in its ability to host virtual machines belonging to different clients on the same physical server. Isolation must be total: a client cannot access another's data or disrupt their operations, and each receives exactly the resources they pay for. The designers like to illustrate this requirement with a compelling example: simultaneously selling services to Coca-Cola and Pepsi while guaranteeing them absolute confidentiality.

The following years saw the project gain momentum. Laboratories like Intel Research and HP Labs provided financial and technical support. In 2003, the first free version of Xen was released under a dual license: GPLv2 for the hypervisor, BSD for components integrated into operating systems. This BSD license choice was not insignificant: it simplified porting to proprietary systems. Microsoft Research became interested in the project, with Paul Barham leading a prototype port of Windows XP.

The publication of the article "Xen and the Art of Virtualization" at the ACM SOSP conference marked the project's academic consecration. Internet service providers began deploying Xen commercially, exploiting its capabilities for virtualizing computing, networking, and storage.

Christian Limpach contributed in 2004 to porting NetBSD to Xen before joining the Cambridge team. Under Pratt's leadership, the open source community around Xen expanded. RedHat, SuSE, Sun, IBM, AMD: all adapted their operating systems or ported Xen to their hardware.

The researchers realized that extending the x86 architecture would make virtualization simpler and more efficient, while reducing the amount of privileged code required. A collaboration with Intel led to the development of Xen's HVM mode, which leverages Intel's VTx extensions. This mode avoids certain paravirtualized modifications to operating systems, while maintaining paravirtualization where it provides real gains. AMD-V support followed.

The year 2004 also saw the main contributors create XenSource in response to growing demands for consulting services. Funding an open source-based company in the United Kingdom and Europe remained a challenging path, as these business models were not well established there.

Ian Pratt reconnected with Simon Crosby, a former university colleague, and went to Silicon Valley where XenSource secured a first financing round of $8 million from KPCB and Sevin Rosen.

XenSource's first year focused on open source development. The initial strategy? Provide management tools and collaborate with Linux distributions. But when the latter announced their own solutions, this approach showed its limits. At the end of 2005, Peter Levine and Frank Artale joined the company through Accel Ventures. Their commercial vision reoriented XenSource toward direct confrontation with VMware. A strategic partnership with Microsoft in 2006 strengthened the company's credibility.

In 2007, Citrix acquired XenSource. Disappointed by VMware's entry into the desktop virtualization market, which directly competed with its application virtualization business, Citrix saw in this acquisition the opportunity to have a Microsoft-compatible platform.

Meanwhile, Amazon built its cloud infrastructure on Xen starting in late 2005. With millions of virtual machines deployed, Amazon is Xen's largest user, and probably the one deriving the greatest financial benefit from it. Amazon's ability to virtualize Windows rests on the adoption of Intel platforms equipped with virtualization extensions. Since then, at AWS, Xen has gradually given way to Nitro, the in-house solution.

In September 2022, following Citrix's acquisition by Vista Equity Partners and Evergreen Coast Capital, XenServer became an independent business unit within the Cloud Software Group. This reorganization opens a new chapter for a technology born in a British university laboratory.

Xen's success testifies to the role of open source in technological innovation and the capacity of academic projects to produce solutions that leave a lasting mark on the industry.

Apache Cassandra

Web giants hit a wall. Facebook, with hundreds of millions of users, must handle a seemingly simple feature: inbox search. But beneath this simplicity lies a formidable technical challenge. Every day, billions of writes flood into the systems, and growth shows no sign of slowing down. Traditional databases, designed in the 1970s, reveal their limitations when faced with these volumes.

Avinash Lakshman and Prashant Malik set out to create a new solution to address the problem. Lakshman doesn't come empty-handed; he

participated at Amazon in designing Dynamo, a distributed storage system that serves as a reference. Their insight is to combine two previously distinct approaches. On one side, the architectural principles of Dynamo that have proven themselves at Amazon. On the other, the data model of BigTable, Google's in-house solution. From this meeting, Cassandra is born, named after the prophetess of Greek mythology.

In July 2008, Facebook makes a decision that will change the game. Rather than jealously guarding its code, as Google and Amazon do by merely publishing papers describing their architectures, the social network releases Cassandra's entire source code under the Apache license. The gesture marks a break from the practices of technology companies. The following year, the project joins the Apache Foundation incubator, before achieving in February 2010 the coveted status of top-level project.

Cassandra's architecture breaks with established patterns. Where classic databases organize themselves according to a master-slave hierarchy, Cassandra adopts a model where all nodes play the same role. No conductor, no single point of failure. Data distributes automatically among participating machines, and the system scales horizontally without requiring an administrator to intervene. This total decentralization guarantees continuous availability, even when some nodes fail.

Data redundancy is embedded in the system's DNA. Cassandra automatically replicates information across multiple nodes. The administrator simply indicates the desired number of copies, the rest happens automatically. If a machine fails, the data remains accessible elsewhere. This approach, simple in principle, proves effective in practice.

Netflix offers a compelling example of what Cassandra makes possible. In 2011, the streaming service deploys a cluster of 288 instances in the cloud. The system handles 1.1 million writes per second coming directly from clients. With replication across three different availability zones, this figure climbs to 3.3 million writes per second. These performances would have been unthinkable a few years earlier with conventional technologies.

The arrival of the CQL language greatly facilitates adoption. Its syntax resembles that of SQL, known to millions of developers worldwide. A programmer familiar with relational databases can get to grips with Cassandra without relearning everything. Certainly, differences remain: no joins for example, which reflects the system's denormalized data model. But the entry barrier drops considerably.

The community grows rapidly. In 2012, over 1,000 production deployments are already running, from eBay to Disney and Netflix. Tools multiply, integrations with Hadoop, Spark, or Solr enrich the ecosystem. The big data world finds in Cassandra a reliable pillar.

DataStax emerges in 2010 and brings a commercial dimension to the project. The company hires the main contributors and offers an enterprise version with additional features, technical support, and administration tools like OpsCenter. The open source model finds there its viable economic counterpart.

The numbers speak for themselves. A comparative study presented at the Very Large Database conference in 2012 pits Cassandra against HBase. Read latency times prove up to 100 times faster, throughput eight times higher. These results don't come out of nowhere: they stem directly from the initial architectural choices.

Versions follow one another and bring their share of improvements. Version 1.0 in 2011 optimizes performance and compresses data. Version 2.0 in 2013 introduces lightweight transactions via the Paxos protocol. Version 3.0 in 2015 adds materialized views. Version 4.0 in 2021 deploys zero-copy streaming and supports Java 11. Each iteration pushes the system's capabilities a bit further.

Instagram, acquired by Meta, relies on Cassandra to manage data for over one billion active users each month. Use cases diversify: time series, real-time analysis, multimedia content management, e-commerce. Wherever volumes explode and unavailability costs dearly, Cassandra finds its place.

The impact extends beyond the purely technical dimension. Cassandra demonstrates that a database can combine high availability, linear scalability, and high performance. These three pillars respond exactly to the needs of contemporary web applications, where every minute of downtime counts in millions and where data never stops growing.

Cassandra finds favorable ground in hybrid and multi-cloud deployment. The system operates transparently across multiple data centers or different cloud providers. While keeping their data consistent, companies thus escape dependence on a single vendor.

Other projects draw inspiration from Cassandra. ScyllaDB incorporates its concepts in C++ to gain even more performance. The decentralized architecture and tunable consistency model now serve as a reference in the field of distributed storage.

Twenty years after its birth, Apache Cassandra remains a cornerstone of the NoSQL ecosystem. Its story tells of the maturation of distributed architectures and their capacity to adapt to the ever-increasing demands of modern applications. What began as an internal solution at Facebook has become a de facto standard for anyone who must manage massive data with strict availability guarantees.

Google Gmail

In 2001, the free email market was dominated by three established players: Microsoft Hotmail, Yahoo! Mail, and AOL. Each offered roughly the same thing-a few megabytes of storage space for messages. Users spent their time sorting through emails, regularly deleting old messages to avoid exceeding the imposed limit. This was the norm, and no one really complained, probably because the service was free.

At Google, an employee finally voiced her frustration. She was tired of this constant chore, those 4 megabytes that forced her to constantly archive, organize, and delete. Paul Buchheit, an engineer at the company, heard this complaint. At the time, Google only did web search. Developing an email service represented a complete pivot for the company, a departure from its usual territory.

Paul Buchheit wasn't attempting this for the first time. Before joining Google, he had worked on a prototype webmail that never saw the light of day. Armed with this failure, he decided this time to move quickly. In a single day, he created a first version by recycling existing code from Google Groups, which was used to search through Usenet archives. This first draft did only one thing: search through one's own emails. Nothing more, but it was a start.

The real technical breakthrough lay in the use of JavaScript and the XMLHttpRequest object. JavaScript had a bad reputation at Google, associated with invasive ads that polluted the web. However, this technology proved essential for building a responsive interface, without those page reloads that disrupted the user experience. The combination of JavaScript and XMLHttpRequest, which would later be called AJAX, made it possible to create something that truly resembled software installed on a computer.

Development proceeded step by step, with each new feature first tested by Google employees. Feedback came directly, improvements followed immediately. One of the notable ideas was organizing messages into conversations rather than isolated emails. This presentation adopted the logic of Usenet discussion groups, where following a thread of exchanges happened naturally.

The question of money arose. Marissa Mayer, who oversaw product development, leaned toward the classic model: limited free space, a paid option for those who wanted more. Buchheit proposed something else: displaying ads related to message content. The idea sparked debate about

privacy and confidentiality. But this approach gave birth to AdSense, which became one of Google's main revenue sources.

On April 1st, 2004, Gmail was announced with an incredible promise: 1 gigabyte of free storage. Two hundred and fifty times more than the competition. The announcement date and the scale of the offer sowed doubt. Google loved April Fools' pranks, and offering so much space seemed technically impossible, economically absurd. Many believed it was a joke.

The service launched by sending out invitations. Early users received a few invitations to distribute, creating a viral distribution system. This artificial scarcity allowed for gradual server scaling while building an image of an elitist, desirable service. Gmail invitations were even traded on eBay.

Gmail transformed Google. The company proved it could do more than web search, that it could create sophisticated web services. The techniques developed for Gmail, particularly that famous use of AJAX, became web development standards. Google published the Google Web Toolkit in 2005, a framework that allowed other developers to use these technologies.

The beta phase lasted five years. Gmail emerged from it in 2009, after progressively integrating new features: chat, spell checking, spam detection, connections with Calendar and Drive. The interface evolved but kept its original principles, that simplicity which was its strength.

The impact went far beyond email. Gmail redefined what could be expected from a web application. Microsoft and Yahoo! had to go back to the drawing board, drastically increasing their storage space. More broadly, the service showed that a web application could rival software installed on a hard drive.

Gmail reflects how Google approaches innovation. A trial-and-error approach, rapid iterations, attention to real needs. This method blends technical thinking with field observation. The service continues to evolve by integrating artificial intelligence to automatically sort messages, suggest responses, and detect phishing attempts.

Scala

Martin Odersky launched Scala's development in 2001 within the Programming Methods Laboratory at the École Polytechnique Fédérale de

Lausanne. His approach addressed a specific question: how to better support component-based software in programming languages?

Three years later, in January 2004, the first public version of Scala emerged on the Java Virtual Machine platform. A .NET version followed in June. The chosen name, "Scala", contracts *scalable language*: an extensible language that adapts to its users' needs. This notion of adaptability runs through the entire design: from small scripts to large systems, the tool must be able to keep pace.

Scala's originality lies in its fusion of object-oriented and functional programming under static typing. Scala unites these paradigms where other languages keep them separate. Its type system incorporates advanced concepts such as abstract types and path-dependent types, inherited from νObj calculus. Modular composition relies on *mixins* and traits. Views, in turn, offer modular component adaptation.

Compatibility with Java strongly shapes the design. Scala adopts a substantial portion of Java's syntax and type system. A Java developer finds familiar territory. Java libraries can be used directly in Scala, and vice versa. Scala classes inherit from Java classes and implement their interfaces. This interoperability enables the insertion of Scala code into existing Java projects without rewriting everything.

March 2006 marked the release of a second major version. The type system gained robustness, class composition mechanisms became more refined. The syntax remains deliberately conventional, yet conceals technical sophistication that allows the expression of complex concepts in few lines.

Scala's influences are manifold. On the object side, Smalltalk and Simula inspire the uniform object model. Universal nesting-this ability to nest almost all constructs within one another-comes from Algol, Simula, and then Beta. The functional approach recalls the ML family with SML, OCaml, or F#. Implicit parameters find their roots in Haskell's type classes, adapted here to the conventional object world. The concurrency library based on actors owes much to Erlang.

Rather than imposing a fixed set of constructs, Scala relies on extensibility. Programmers create their own abstractions. This philosophy evokes the bazaar rather than the cathedral, according to Eric Raymond's metaphor: the language grows through the addition of constructs invented by its users, not through modifications to its core.

Scala's technical contributions are notable in several areas. The uniform treatment of generic types and abstract types breaks new ground. Class composition via traits offers an original path. The extraction mechanism enables pattern matching independent of representation. These

innovations have been presented at various specialized conferences.

Industry has recognized Scala's value, especially for distributed applications and massive data processing. Apache Spark, written in Scala, demonstrates the language's capabilities on complex data processing systems. The concise syntax reduces code size compared to Java-up to 50% in common cases.

The Swiss National Fund, Microsoft Research, the MICS Research Competence Center, the European PalCom project, and the Hasler Foundation supported the development. The developer community actively participates with feedback and code contributions.

Scala's evolution reflects a pragmatic approach. There's no intent to build a perfect and rigid system. Adaptability and extensibility take precedence. This strategy gives developers the means to enrich the language according to their needs, creating a thriving ecosystem of libraries and frameworks.

In 2025, Scala continues its journey while maintaining its founding principles: the unification of object-oriented and functional programming, extensibility, and interoperability with established platforms. The language nourishes reflections on programming language design and software system architecture.

NGINX

Igor Sysoev had a problem. In 2002, this Russian engineer was working for Rambler, an internet portal that was seeing hundreds of thousands of simultaneous connections flooding in. The servers were struggling. Apache, the dominant solution of the time, was showing its flaws: each connection demanded its own system resources, creating a burden that became unsustainable when traffic surged. Sysoev was looking for something else, a different approach to handle this avalanche of requests.

He then set out to write a new web server, which he named NGINX (pronounced engine-x). Its architecture broke with conventions: instead of allocating a process or thread per connection, the software adopted an asynchronous, event-driven model. A master process supervised everything, a few worker processes shared the workload, and this handful of components was enough to handle thousands of requests in parallel, achieving spectacular resource efficiency.

In October 2004, Sysoev released version 0.1.0. He chose this date as

a reference to the launch of Sputnik, a nod to the first artificial satellite. The code was distributed under a BSD license, freely available. The first to try it were Estonian and Russian sites like rate.ee and zvuki.ru, which used NGINX to serve their MP3 files. The newcomer proved its worth.

The software gained ground, first in the Russian-speaking sphere, then beyond. System administrators discovered its talents: serving static content with remarkable speed, acting as a reverse proxy, distributing load across multiple servers, managing cache, and serving as a FastCGI gateway. NGINX proved versatile, reliable, fast. Its reputation crossed borders.

Seven years after the first public version, in 2011, Sysoev and Maxim Konovalov founded NGINX, Inc. The idea? Offer commercial support and advanced features while preserving the open source version. Investors reacted quickly: BV Capital and Runa Capital injected 3 million dollars as early as October 2011. Other rounds followed: 10 million in 2013, 20 million in 2014, 43 million in 2018. The company developed NGINX Plus, a paid offering that complemented the community version with extended monitoring and analytics capabilities.

F5 Networks acquired NGINX in March 2019 for 670 million dollars, a striking validation of the software's importance in contemporary web infrastructure. But the story took an unexpected turn in December of the same year: a search operation hit NGINX's Moscow offices. Rambler, Sysoev's former employer, claimed intellectual property rights over the code. The matter was resolved thanks to Sberbank's intervention, a Rambler shareholder, but the episode left its mark.

On the technical front, NGINX continued to evolve. It integrated HTTP/2, refined its compression and caching mechanisms, handled headers with flexibility. Shared memory was used to store cache, manage session persistence, control throughput. This efficiency proved appealing: in 2022, over 33% of websites worldwide ran on NGINX, surpassing Apache in various categories. Platforms like WordPress.com trusted it. The software adapted to modern architectures, including microservices and cloud.

January 2022: Sysoev announced his departure from NGINX and F5. He left behind a project that had reshaped the web's infrastructure. His latest creation, NGINX Unit, launched in 2017, targeted multilingual environments and microservices architectures. The legacy endures.

From Rambler to servers worldwide, the software grew as the web changed its face. Its success stems from clever architectural choices, a pragmatic approach to performance issues, and a successful balance between open source code and a viable business model.

OAuth

In 2006, Blaine Cook was working on Twitter's API and facing a problem that was becoming increasingly common among web developers. He was looking for a way to connect Twitter to Flickr, the photo-sharing service. The only available solution was to ask users to entrust their Flickr credentials to Twitter. A security aberration, but it was the norm at the time.

This technical impasse prompted Cook and other developers from major web platforms to come together. Google had AuthSub, Yahoo used BBAuth, AOL offered OpenAuth, and Flickr had its own API. Everyone was cobbling together their own solution independently. They needed to move beyond this fragmentation. In October 2007, their collaboration resulted in OAuth 1.0, an open protocol designed to solve this puzzle: how to authorize a third-party application to access a user's data without ever handling their password?

The mechanism relies on a simple yet effective principle. Imagine an online printing service that wants to access photos stored on another site. Instead of demanding the photo account credentials, it obtains a specific access token, limited to only the necessary operations. The user maintains control: they explicitly authorize access from a secure interface on the site hosting their photos. The printing service never knows the password.

The first security flaws emerged, and in June 2009, OAuth 1.0a corrected vulnerabilities that allowed identity spoofing. This revision clarified gray areas in the initial protocol and strengthened protections against attacks. The following year, in April 2010, the IETF published OAuth 1.0 as RFC 5849, conferring official technical specification status upon it.

The protocol defines three actors: the resource owner (the user), the client (the application requesting access), and the authorization server. The application first obtains temporary credentials. It then redirects the user to an authorization page where they review the requested permissions. After validation, the application receives a token it will use for future requests. At no point does it handle the user's credentials.

Security relies on safeguards. Each request must be signed with a secret key. The protocol incorporates unique tokens (nonces) and timestamps to block any malicious reuse attempts. The designers anticipated session hijacking, replay attacks, and spoofing.

Facebook, Google, Microsoft, and Twitter adopted OAuth. This massive adoption by web giants created an ecosystem of interconnected ap-

plications. Libraries appeared in all common programming languages, making integration more accessible.

But OAuth 1.0 showed its limitations. The implementation complexity, particularly for cryptographic request signing, discouraged developers. Native mobile applications, increasingly numerous, posed problems not originally anticipated. These constraints motivated a complete overhaul of the protocol with OAuth 2.0.

The OAuth protocol established the standard for delegated authorization on the web. It introduced concepts now central: explicit user consent, strict limitation of granted privileges, clear separation between authentication and authorization.

OAuth also played an educational role. Developers and users became aware of the issues surrounding personal data protection. The protocol demonstrated that it was possible to create interconnected services without sacrificing security. This awareness accompanied the evolution of privacy regulations.

Moreover, modern architectures bear the mark of OAuth. OpenID Connect builds on its foundations to add a standardized authentication layer. REST APIs and microservices adopt its architectural principles. Connected devices and decentralized identity systems explore new paths starting from the foundations laid by OAuth.

The community continues to adapt the protocol to new uses. Feedback from large-scale deployments feeds reflections on its future evolution. OAuth remains a pillar of web security, nearly twenty years after its creation.

Git

In 2005, Linus Torvalds found himself in an embarrassing situation. BitKeeper, the version control system the Linux community had been using for years, was no longer freely available to kernel developers. The story began with Andrew Tridgell, a respected developer, who decided to study BitKeeper's protocol through reverse engineering. Larry McVoy, BitKeeper's creator, considered this approach a violation of the terms of use and revoked the free license granted to Linux developers.

Torvalds attempted mediation for a few weeks, without result. Faced with this impasse, he had to find an alternative. He examined existing

systems, but none satisfied him. He who considered version control managers one of the least interesting areas of computing, just after databases, embarked on creating his own tool. His experience with BitKeeper had at least taught him one thing: the advantages of a distributed system where each developer has a local copy of the repository.

Torvalds then disappeared for a week. When he resurfaced, he presented the first version of Git. Initial development progressed at a remarkable pace. In ten days, Git was capable of managing its own source code. This speed did not come from a massive amount of code written, but from deep reflection on data organization. Torvalds had designed an architecture that favored conceptual simplicity and performance.

The system relied on several technical choices. Each developer had a complete copy of the repository, which eliminated dependency on a central server and solved write access issues. Git used SHA-1 cryptographic hashing to identify objects in the repository. This approach guaranteed code integrity and made any modification of the history impossible without detection.

Branch and merge management constituted one of Git's major innovations. These operations, known for being complex in previous systems, were simple and fast. Torvalds sometimes performed dozens of merges per day during intense periods of Linux kernel development. This fluidity transformed the way developers worked.

A few months after its creation, Torvalds entrusted Git's maintenance to Junio Hamano. He recognized in him a developer with excellent technical judgment. Under his direction, Git evolved to become more accessible while retaining its qualities of speed, reliability, and flexibility.

Git's adoption happened gradually. Google's Android project chose it as its version control system in 2009, which brought important validation from a large company. The creation of GitHub in 2008 by Tom Preston-Werner, Chris Wanstrath, and P.J. Hyett accelerated the tool's democratization. The platform offered a user-friendly web interface and social networking features for developers.

Microsoft, long reluctant about open source software, gradually changed its position. In 2012, the company began contributing to libgit2, a Git development library. The following year, it integrated Git support into Visual Studio and offered hosting via Azure DevOps. In 2017, Microsoft migrated Windows development to Git, creating the world's largest Git repository with 300 GB of source code.

Microsoft's acquisition of GitHub in 2018 for $7.5 billion marked a symbolic moment. This transaction illustrated Microsoft's transformation and confirmed Git's dominant position in the software industry. Stack

Overflow's 2018 statistics revealed that 88.4% of developers used Git, far ahead of Subversion (16.6%) or Mercurial (3.7%).

Git's design reflected the specific needs of Linux kernel development. The ability to handle a large number of simultaneous contributions, the speed of operations, robustness against errors: these characteristics, combined with its distributed nature, made it a tool suited to large-scale open source projects. Developers could work autonomously on their local copies, then synchronize their modifications with the rest of the team when they wished.

In 2024, Git remains the de facto standard for version control in the software industry. It has transformed the way developers collaborate and share their code. The principles of its distributed architecture have inspired many other tools and practices in the field of software development.

Redis

In 2009, Salvatore Sanfilippo, an Italian developer known as "antirez" pseudonym, ran into a frustrating problem. The web log analyzer he was developing for his startup was painfully slow with MySQL. Rather than patching together yet another optimization, he decided to create his own database. What started as a prototype in Tcl evolved into a C project called Redis, short for REmote DIctionary Server.

A few weeks after writing the first lines of code, Sanfilippo made a choice that would seal his project's future: he released it as open source. The decision proved wise. Developers embraced Redis en masse, drawn to its execution speed and apparent simplicity. The project achieved a success beyond its creator's expectations.

Redis breaks with traditional database conventions. Where most store their data on disk, Redis keeps everything in RAM. This architecture provides unparalleled access speed, but requires persistence mechanisms to guard against unexpected system shutdowns. Yet Redis isn't just fast: it natively handles varied data structures like lists, sets, hash tables, and sorted sets. A richness that distinguishes it from simple key-value stores.

VMware hired Sanfilippo in 2010 to work on Redis full-time while preserving the project's BSD license. Pieter Noordhuis, a leading contributor, joined the team shortly after. This professionalization accelerated development and Redis gradually established itself in modern architectures.

Tech giants recognized its value. GitHub and Instagram were among the notable early adopters. Twitter, Uber, Stripe, Slack, and Alibaba followed suit. This widespread adoption stems from the tool's versatility: Redis works equally well as a database, cache system, or message broker. Three roles for the price of one.

Sanfilippo designed Redis as a construction set. Rather than a monolithic product with fixed features, he created elementary building blocks that developers can assemble according to their needs. Lua scripting, the publish-subscribe system, and data streams were added progressively, following this modular philosophy. Redis resembles programmer's Lego, where everyone builds their own solution.

The creation of Redis Labs in 2015 (which became simply Redis) professionalized the ecosystem. This structure offers commercial services around the project while maintaining its open nature. After eleven years leading Redis, Sanfilippo stepped back in 2020 to focus on writing and his family. A withdrawal from daily operations, not a complete break.

Redis's technical architecture makes bold choices. Its single-threaded model avoids locking headaches and simplifies concurrency management. For data persistence, two mechanisms coexist: periodic snapshots and the append-only file (AOF). Master-slave replication distributes load and ensures availability. Pragmatic solutions that have proven their worth.

Redis introduces a distinctive way of thinking about data. Instead of limiting itself to classic key-value pairs, it directly manipulates complex structures through atomic commands. This approach reduces application code complexity and limits network exchanges between client and server. A significant time saver when working at scale.

The story could end there, but Sanfilippo returned in 2024 as Redis evangelist. His new mission: adapting the tool for the AI era. Besides the new license that is no longer open source by OSI standards, he's working on *vector sets*, an extension of sorted sets tailored for *embeddings* and language models. Redis evolves with its time.

YouTube

On February 14, 2005, three former PayPal employees registered the trademark, logo, and domain name for a project that would change the way we consume video on the Internet. Chad Hurley, Steve Chen, and Jawed Karim had the intuition that something was missing on the web:

a simple way to share videos. The idea emerged during a dinner in San Francisco, fueled by two events in 2004 that highlighted this technological gap. First, Janet Jackson's wardrobe malfunction at the Super Bowl, whose videos remained impossible to find online. Then, the Asian tsunami, whose images struggled to circulate on the web.

In May 2005, the site launched in beta version and immediately attracted 30,000 visitors per day. The first video uploaded gave no indication of the phenomenon to come. In this 19-second sequence titled "Me at the Zoo", Jawed Karim filmed the elephants at San Diego Zoo and commented on the size of their trunks. Nothing spectacular, yet this mundane fragment has now accumulated over 375 million views. Four months later, a Nike advertisement featuring Ronaldinho crossed the one-million-view threshold, revealing the platform's commercial potential.

The official launch took place on December 15, 2005, made possible by a $3.5 million investment from Sequoia Capital. This funding was invested in improving servers and increasing bandwidth, as growth was dizzying. In January 2006, the site recorded 25 million daily views. March saw the number of available videos exceed 25 million, with 20,000 new uploads each day. The following summer, the 100-million daily views milestone was reached.

This explosive expansion generated considerable costs in IT infrastructure and Internet connections. The copyright issue emerged when numerous media companies discovered their protected content in videos uploaded by users. In October 2006, Google made the decision and acquired YouTube for $1.65 billion in stock. Rather than merging the platform with Google Video, its own service that was struggling to gain traction, the Mountain View giant chose to maintain YouTube as an autonomous operation.

Google then negotiated agreements with entertainment companies to reduce legal risks related to copyright. These arrangements authorized the use of protected content and allowed users to incorporate certain copyrighted music into their videos. In 2008, a partnership with MGM went further by authorizing free streaming of films and series, funded by advertising.

Through the Partner Program in May 2007, content creators could now monetize their viral videos, transforming their passions into profitable activities. Some users generated six-figure revenues. Institutional adoption followed in 2009, when the U.S. Congress and the Vatican opened their official channels.

YouTube Live arrived in April 2011, streaming concerts, royal weddings and sporting events live. In December 2012, the "Gangnam Style"

music video became the first video to exceed one billion views. The event forced the platform to modify its counter to handle numbers beyond the limits of a 32-bit integer.

The business model diversified with paid services. Music Key appeared in 2014, replaced by YouTube Red in 2015, and YouTube Premium in 2018. These offerings provided ad-free viewing and additional features. In 2016, YouTube Go targeted emerging markets with an application optimized for limited connections.

Artificial intelligence entered the platform. In 2019, the Video Reach service used machine learning to optimize advertising campaigns. The following year, YouTube abandoned its manual categorization system in favor of more sophisticated classification algorithms.

In 2023, the platform claimed approximately 2.5 billion monthly active users, rising to second-largest social network globally behind Facebook. The YouTube Shorts format, a direct response to short videos popularized by TikTok, generated over 50 billion daily views. YouTube Premium reached 125 million subscribers in March 2025.

In twenty years, YouTube has revolutionized online video content consumption. The platform established a new democratized distribution model where anyone can become a creator. Its influence on digital culture, education, entertainment, and advertising continues to expand, making this service a pillar of the modern Internet.

Amazon Web Services

In 2003, Amazon was an e-commerce site growing rapidly. The servers struggled to keep up with the pace of orders, and every activity spike revealed the limitations of an infrastructure hastily built in 1994. Jeff Bezos and his teams sought ways to handle the load without endlessly multiplying hardware investments.

The company had launched a service meant to enable retailers like Marks & Spencer to build their own stores on the Amazon platform. Except no one had anticipated the architectural chaos awaiting developers. Code had piled up over the years without any real master plan. Extracting the software components cleanly to offer them to third parties was impossible. Everything had to be rebuilt from the ground up.

Amazon therefore began breaking down its infrastructure into distinct services, each accessible through well-documented programming in-

terfaces. This reorganization made the service functional, but it had an unexpected side effect: internal teams began working differently. Each service became autonomous, reusable, standardized. A new discipline took root in software development.

Meanwhile, Andy Jassy, Bezos's chief of staff, noticed a troubling paradox. Amazon was hiring engineers extensively, yet development speed stagnated. Digging deeper, he discovered that teams were losing three months setting up their technical environment before starting to code. Database, compute servers, storage: everyone was rebuilding the same foundations on their own. A tremendous waste of time and energy.

In summer 2003, Bezos gathered his executives at his home for a strategic retreat. What began as a routine session on Amazon's core competencies stretched well beyond the planned thirty minutes. Beyond e-commerce and logistics, one thing became clear: Amazon knew how to manage complex IT infrastructures. The constraint of thin margins had forced the company to optimize its data centers to excellence. This expertise was invaluable.

The idea then emerged to transform this infrastructure into an operating system for the Internet. All the components already existed at Amazon, but no one else had access to them. Developers worldwide were reinventing the same building blocks, losing the same three months, hitting the same problems. Amazon could sell them these ready-made components.

The vision took time to materialize. Amazon Simple Queue Service launched in 2006, closely followed by Simple Storage Service and Elastic Compute Cloud. These three services formed the foundation of AWS. Developers could now rent computing power, storage, message queues, without buying a single server. Everything billed based on usage, like water or electricity.

The business model disrupted established practices. Gone were the massive investments in machine rooms sitting idle half the time. Resources adjusted to demand, scaled up during peaks, scaled down during lulls. Technical complexity disappeared behind simple interfaces. Companies could finally focus on their business rather than managing servers.

AWS entered a virgin market. Surprisingly, competitors took years to react. Microsoft, IBM, Google watched the train pass before launching their own efforts. Too late: AWS had set the standards for *cloud computing* and captured over 30% of the market by 2016.

Revenue climbed to 10 billion dollars annually. From startups to giants like Netflix or Dropbox, everyone adopted Amazon services. An obsession with operational efficiency partly explained this success. Amazon built

massive data centers in strategic locations, where electricity and cooling were cheap. Economies of scale allowed it to regularly lower prices while remaining profitable.

AWS marked the transition from ownership computing to consumption computing. Like public utilities during the industrial revolution, the cloud became a commodity available on demand.

Hadoop

In 2006, Yahoo! faces a major challenge. Its WebMap infrastructure, the technology that builds the web graph to power its search engine, can no longer keep up. The graph contains over 100 billion nodes and 1 000 billion edges. The previous infrastructure, called Dreadnaught, has reached its limit at 800 machines. The architecture needs a complete overhaul to keep pace with the growth of the web. Yahoo! then chooses Apache Hadoop.

The story begins two years earlier. In 2004, researchers at Google publish two papers that will change everything. They describe their file system and the MapReduce architecture. These publications establish the technical foundation: a distributed file system capable of storing immense data volumes, combined with a model for processing them in parallel at large scale.

The first version of Hadoop emerges within the Apache Nutch project, before becoming independent in 2006. Four years later, Google obtains a patent on the MapReduce algorithm, but grants a license to the Apache Software Foundation. This decision paves the way for massive industry adoption.

The initial architecture rests on two pillars. HDFS (Hadoop Distributed File System) manages distributed data storage. MapReduce coordinates the execution of processing tasks. A central service, the JobTracker, oversees all operations, while TaskTrackers execute tasks on each cluster node.

This monolithic architecture quickly reveals its weaknesses. Developers attempt to repurpose MapReduce for uses it was never intended for. Some launch map-only tasks to run web servers or iterative calculations. These workarounds, despite their ingenuity, compromise the system's stability and efficiency.

The Apache community undertakes a major overhaul. YARN (Yet Another Resource Negotiator) emerges. This new version separates resource

management from the programming model. A central ResourceManager allocates cluster resources to applications. Each application has its own ApplicationMaster to coordinate its execution.

This architectural redesign transforms the ecosystem. New frameworks emerge: Apache Tez for executing task graphs, Spark for in-memory processing, Storm for real-time processing, Giraph for graph computing. Users are no longer constrained by MapReduce's limitations.

Hadoop adoption in the industry reaches considerable scale. Yahoo! deploys YARN across all its production clusters. The system processes approximately 500 000 jobs daily, representing over 230 years of computation. On a 2 500-machine cluster, the transition to YARN doubles average CPU utilization. The company runs twice as many tasks as before.

Performance improves. In 2013, the MapReduce implementation on YARN sets records in the Daytona and Indy GraySort benchmarks. The system sorts 1.42 terabytes of data per minute on a 2 100-node cluster. These results demonstrate its capability to efficiently process massive volumes.

Hadoop's architecture influences the design of modern distributed systems. Its ability to run on commodity hardware, its fault tolerance, and its simple programming model make it a reference point. The concept of data locality, where computation moves to the data rather than the reverse, becomes a defining principle of distributed computing.

Hadoop's success inspires the development of numerous competing platforms. Twitter creates Mesos, Facebook designs Corona, Google develops Omega. Each system brings its innovations, but all share Hadoop's architectural heritage: the separation between resource management and programming models.

Hadoop's evolution reflects the transformation of data processing needs. From a specialized tool for web indexing, it transforms into a generic platform supporting a multitude of applications. This transformation comes with continuous improvements in performance, flexibility, and ease of use.

By demonstrating the viability of large-scale distributed processing on commodity hardware, Hadoop democratizes massive data analysis. Companies of all sizes can now build high-performance data processing infrastructures without investing in expensive specialized hardware. The collaboration between companies, developers, and researchers within the Apache Foundation has enabled rapid advances and widespread knowledge sharing.

jQuery

In 2005, John Resig was a computer science student at the Rochester Institute of Technology. He spent his days coding websites and constantly ran into the same problem: making JavaScript work across different browsers was an uphill battle. Each browser had its own rules, its own quirks. You had to write multiple versions of the same code, test, fix, and start over. This situation exhausted developers and slowed everyone down.

He decided to create his own solution. The following year, at a technology event called BarCampNYC, he presented the first version of jQuery. His ambition could be summed up in one sentence: make JavaScript programming enjoyable. He drew inspiration from the work of Charles Willison, a British developer who had created `css_selector.js`, a library for selecting CSS elements. Resig took this idea and pushed it much further.

jQuery offered a simple interface for manipulating web pages. Developers could select HTML elements with complex expressions, modify them, and animate them. The selection engine worked remarkably well. Later, this engine was extracted into a separate library called Sizzle, as it represented such a significant technical advancement.

But jQuery's real strength didn't lie solely in its code. Resig had understood something that many other developers overlooked: good documentation was worth its weight in gold. Between 2006 and 2007, jQuery was the only open-source JavaScript library to offer complete and clear documentation. Other projects left developers to fend for themselves with the source code. This attention to detail changed everything.

Resig first hired a community manager. Not an additional developer, no: someone whose job was to answer questions and support users. He was thinking ahead and knew that a technical tool doesn't spread on its own.

Initially, jQuery remained a side project. Resig was working on other things. His hiring at Mozilla changed the game. The company allowed him to dedicate work time to jQuery. He could then build the necessary infrastructure and prepare the project to continue without him. He later left Mozilla for Khan Academy, but jQuery was already launched.

The library achieved tremendous success in the late 2000s. It made sophisticated techniques like animations and Ajax requests accessible. Developers adopted it massively. According to W3Techs, approximately 74% of websites used jQuery at its peak. A staggering figure.

The project grew, and so did its governance. A jQuery Board emerged in 2011, followed by a jQuery Foundation in 2012. Then came a merger

with the Dojo Foundation in 2015, creating the JS Foundation. In 2019, another merger with the Node.js Foundation: the OpenJS Foundation was born, and jQuery retained its status as a major project.

But the web was evolving. Browser vendors—Apple, Google, Microsoft, and Mozilla—began working together within the Web Hypertext Application Technology Working Group. Incompatibilities between browsers decreased. Browsers themselves integrated features that made jQuery less indispensable. The Fetch API replaced Ajax functions. The querySelector and querySelectorAll methods enabled element selection just as jQuery did. The classList interface simplified CSS class manipulation.

Browsers adopted automatic and continuous updates. Fixes and new features reached users automatically, without manual action. Developers could exploit new web features more quickly. They needed fewer compatibility solutions.

Internet Explorer gradually died. Microsoft discontinued support for versions 10 and earlier in 2016, keeping only IE 11. This disappearance freed developers from the constraints that had originally justified jQuery.

New frameworks emerged: React, Angular, Vue. They proposed a different approach, component-based, facilitating the construction of complex interfaces. They encouraged declarative programming where the developer describes the desired state rather than the steps to achieve it. jQuery, on the other hand, remained imperative.

Some major players abandoned jQuery. GitHub removed it from its interface. Bootstrap announced it would drop it in version 5, as jQuery constituted its main client-side JavaScript dependency.

Yet jQuery didn't disappear. For simple sites requiring little interactivity, the library remains a proven and easy-to-implement solution. Its imperative approach remains more accessible to beginners than modern declarative paradigms. In certain professional contexts where old versions of Internet Explorer still run, jQuery retains its usefulness. The library continues to evolve, maintained by a loyal community. Many developers appreciate its concise syntax and use it by preference, while native alternatives exist. The progression of web standards and the emergence of new approaches testify to the continuous evolution of technologies, a process in which jQuery has largely participated by establishing practices that endure.

PowerShell

In 2002, Jeffrey Snover wrote a document at Microsoft that would transform system administration on Windows. The "Monad Manifesto" presented an uncommon vision at the time: creating a shell that manipulates objects rather than text. The project's name referred to Leibniz and his concept of fundamental units that could, through aggregation, form more elaborate systems.

Windows administration was going through an identity crisis at the time. On one side, simple graphical tools for basic operations. On the other, complex programming languages for advanced automation. In between, a void. Administrators struggled to compose automation solutions without diving into heavy programming. Microsoft had spent years simplifying Windows for novices, paradoxically making certain tasks more difficult for experienced users who favored the efficiency of command lines.

Snover drew inspiration from UNIX shells while seeking to overcome their limitations. In these environments, commands produce unstructured text that subsequent commands must parse, a fragile and error-prone process. PowerShell's innovative idea lay in its object-oriented approach: the shell passes structured .NET objects between commands, eliminating this fragility.

The project nearly never came to be. Between 2004 and 2006, several major initiatives based on managed code at Microsoft experienced resounding failures, causing Windows Vista delays. Promoting a scripting language relying on .NET was risky. The Exchange Server team, the first to adopt PowerShell, developed an intermediate API layer that would allow them to switch to another solution if problems arose.

PowerShell finally arrived in 2006. It introduced "cmdlets", lightweight commands implemented as .NET classes. This architecture ensured syntactic and semantic consistency rarely found in traditional shells, where each command defines its own conventions. Version 2.0, delivered with Windows 7, enhanced the tool with remote script execution and background tasks.

The standardization of verbs in command names (Get-, Set-, New-) and the introduction of *providers* marked a breakthrough. The latter handle different data sources—file systems, Windows registry, Active Directory—with uniform syntax. Administrators no longer had to relearn a set of commands for each source, which significantly reduced the learning curve.

Adoption accelerated when Microsoft integrated PowerShell into its server products. Exchange Server 2007 made PowerShell mandatory for

certain tasks. This decision forced administrators to take the plunge, quickly recognizing the advantages in terms of efficiency and automation.

The shell's architecture broke with the traditional approach. Instead of creating standalone executables, developers write .NET classes inheriting from a common base class. This class provides parameter parsing, data validation, logging, and error handling out of the box. This code sharing reduces development costs and improves the user experience.

The Integrated Scripting Environment (ISE), added later, offered administrators a modern tool for writing and debugging scripts. Syntax highlighting, auto-completion, and a graphical debugger supported the transition from graphical interfaces to script-based automation.

An active community formed around PowerShell. Sysadmin shared scripts and best practices, while software vendors adopted the tool to administer their products. This adoption validated Snover's initial vision of a unified automation platform.

Security occupies a central place in PowerShell. The shell integrates code signing mechanisms and detailed activity logging. Organizations can control script execution and trace their use, meeting security and compliance requirements.

The impact on Windows administration proved considerable. PowerShell transformed interaction with the system, replacing repetitive manual actions with reproducible automated solutions. This evolution improved efficiency and reduced human errors in system operations.

In 2016, Microsoft made PowerShell open source under the MIT license and cross-platform. This decision reflected the company's evolution toward greater openness and acknowledged the reality of modern IT environments, where Windows coexists with Linux and macOS.

MongoDB

In 2007, three American *entrepreneurs*, Kevin Ryan, Dwight Merriman, and Eliot Horowitz, embarked on a platform-as-a-service project. They found that traditional relational databases struggled to meet the scalability and agility requirements of their infrastructure. This constraint led them to design their own solution.

The context of the era saw the myth of the universal database crumbling. IT architectures were diversifying, with everyone seeking answers to specific problems. NoSQL databases were emerging, as were time-series

databases. This fragmentation of the technological landscape forced a rethinking of tooling choices, as each type of system addressed particular needs.

MongoDB was built around a document-oriented model that uses a binary variant of JSON called BSON. This architecture allows for flexible data representation while maintaining high performance. The system finely manages cache and data locality, with a limit set at 16 MB per document to ensure optimal memory usage.

Technical evolution brought its share of innovations. WiredTiger, the storage engine later adopted, introduced ACID compliance in transactions. MongoDB thus managed to marry the flexibility of NoSQL with the transactional rigor of traditional relational systems.

Horizontal scaling ranks among MongoDB's major assets. Two sharding mechanisms coexist: range-based and hash-based. This architecture distributes data across multiple servers, making scaling transparent to applications. The query router exploits this geographic distribution to accelerate processing, so the code written by developers can remain identical regardless of the physical distribution of data.

Availability reaches 99.99% without human intervention thanks to automatic replication and self-healing capabilities. Replica sets automatically fail over during an outage, with a secondary node taking the place of the primary through a well-established election process.

Security, often the Achilles' heel of NoSQL databases, has benefited from continuous improvements. MongoDB now integrates encryption at rest via TDE, LDAP and Kerberos authentication, TLS/SSL encryption, as well as protections against SQL injection. These additions place it in a strong position against its NoSQL competitors.

Use cases emerge naturally. Real-time analytics applications on large volumes, such as those in financial markets or Twitter stream processing, find MongoDB a valuable ally. E-commerce leverages its behavioral analytics capabilities to refine customer experience. The Internet of Things represents another prime territory, as the flexibility of the data model effortlessly handles the heterogeneity of information collected by sensors.

Commercialization comes in two versions: enterprise and open source. This dual approach creates an active developer community while ensuring steady revenue. Integration with major cloud platforms like Amazon Web Services and Azure strengthens its presence in the modern database market.

But some limitations appear. MongoDB is poorly suited to highly relational data, environments requiring strict referential integrity, or operations involving multiple joins between documents. The absence of stored

procedures and triggers constrains business logic outside the database. Data warehouses and complex graph representations do not correspond to its strengths.

MongoDB's influence on the database ecosystem remains tangible. Its success validated the NoSQL approach for enterprise applications, and it shaped the design of modern data management systems. Its ability to process unstructured, semi-structured, and structured data within the same environment has redefined certain industry standards.

Ongoing development reflects the changing needs in data management. The progression toward distributed architectures, the growing importance of real-time analytics, and the emergence of new application models have charted its evolution. This continuous market adaptation explains its current position.

Apple iPhone

In January 2007, Steve Jobs took the stage at Macworld in San Francisco to present a device whose existence was known to almost no one. Two years of absolute secrecy preceded this moment. Only three AT&T executives had been allowed to see the iPhone before its public unveiling. Jobs had negotiated directly with the American carrier for an exclusive partnership that would redraw the balance of power in the telecommunications world.

The phone market in 2007 looked much like it had several years earlier. BlackBerry dominated with its physical keyboards, Palm offered its Treo devices, and users made do with complex interfaces that required real learning time. Mobile Internet remained a frustrating experience, messaging worked after a fashion, and the idea of a truly intuitive phone seemed out of reach.

The iPhone arrived with its 3.5-inch touchscreen and a simple idea: you touch directly what you want to do. No stylus, no keyboard hidden beneath the screen. Fingers are enough, and the surface responds to multiple simultaneous touch points. Apple patented this multipoint technology and placed it at the heart of a device that claims to be three things at once: a phone, an iPod, and a way to browse the Internet. This convergence had never truly worked before.

Jobs applied his usual obsessiveness to the iPhone. He scrutinized every screw, examined the case curves from every angle, and even had

the packaging redone because the adhesive strip wasn't exactly where it should be. At Apple, the product unboxing experience matters as much as the product itself. This obsession with detail reflected a particular vision of what a technological object should be: something that works before you've read the manual.

When the iPhone reached stores in June 2007, lines formed spontaneously. Apple and AT&T sold two versions: $499 for 4GB, $599 for 8GB. These prices frightened no one. The company modestly aimed for 1% of the global market by 2008, or 10 million units. It would far exceed this target.

The distribution broke with industry practices. The iPhone would not be sold at Best Buy or RadioShack. Apple imposed its own stores and AT&T's as the only points of sale. This restriction gave Apple complete control over the device's presentation and the purchasing experience. Traditional retailers would wait impatiently for months.

Technically, the iPhone accumulated innovations like the proximity sensor that turns off the screen when you bring the phone to your ear. An ambient light sensor adjusts brightness according to surrounding light. An accelerometer rotates the display when you turn the device. The operating system derives from OS X and handles multitasking with an efficiency that astonished observers. Apple built hardware and software simultaneously, making them communicate from the design stage. Its competitors separated these two worlds.

Success transformed Apple. The company dropped the word "Computer" from its corporate name in favor of simply Apple Inc. This change reflected the reality of a company that had long since stopped selling only computers. The iPhone accelerated this transformation and shifted the center of gravity toward mobile devices.

A culture of secrecy reigned within the company. Teams worked on project fragments without seeing the whole. Only a handful of managers knew the complete vision. This compartmentalization frustrated some engineers but ensured that nothing leaked before official announcements. Jobs directed everything from a highly centralized structure, refusing to create autonomous divisions that might dilute control.

The iPhone landed in the United Kingdom and France in late 2007, then in Asia the following year. In each country, Apple replicated the American model: a single carrier, an exclusive agreement, controlled distribution. These practices raised legal questions in Europe, where bundling a phone with a single carrier ran up against certain regulations. Apple negotiated case by case.

Competitors responded as best they could. Nokia launched its N95

with enhanced multimedia capabilities. Other manufacturers rushed toward touchscreens, without understanding that Apple's success stemmed as much from the software interface as from the hardware. The market shifted in a matter of months: what the iPhone offered became the new benchmark.

In 2008, Apple opened the App Store and changed the rules again. External developers could create applications and sell them directly to users. The iPhone thus gained thousands of functions that no one had imagined at the outset. An economic ecosystem emerged almost instantly, attracting independent programmers and large companies alike.

Harvard University estimated the value of free publicity generated by the iPhone launch at $400 million. Apple had spent almost nothing on traditional campaigns. The media did the work for free, relaying every announcement, every waiting line, every first impression. This marketing success rested on meticulous preparation and a keen sense of spectacle.

Apple's market value doubled in the twelve months that followed. Analysts understood that the iPhone was not just another product but the company's new pillar. Jobs had been right: a device that combines phone, media player, and Internet browser in a fluid interface can change usage patterns. It marked a breakthrough in the history of mobile computing and telecommunications.

Clojure

In 2005, Rich Hickey embarked on the Clojure adventure. He financed this venture with his retirement savings, taking a sabbatical year to design a language that would solve the problems he had been encountering daily for years. His ambition? To create a tool as acceptable in enterprise settings as Java or C#, but free from the unnecessary complexities that burden information system development.

Hickey was no novice. Since 1987, he had worked as a software architect, navigating between C++, Java, and C#. This long experience taught him one thing: traditional object-oriented languages impose a steep price. Imperative programming with mutable state demands enormous effort whenever you need to modify a program without breaking data consistency. These rigid systems resist change despite encapsulation and abstractions.

Alongside his work, Rich Hickey explored Common Lisp in his personal

projects. This practice opened his eyes: all this complexity he fought at the office wasn't inevitable. With Lisp's flexibility, you tackle a problem at exactly the right level, without superfluous layers. But Common Lisp remained confined to experimentation. Companies refused to deploy applications written in this language. Twice, Hickey had to rewrite in C++ systems he had developed in Common Lisp, or transform them into SQL stored procedures to satisfy his clients.

The idea then germinated to marry Lisp's power with Java's acceptability. Hickey chose the Java Virtual Machine as his foundation. This tactical decision immediately eliminated adoption barriers: Clojure is a first-class citizen of the Java ecosystem, with its native interoperability and ability to reuse existing libraries.

The first technical challenge was to make immutable data structures performant enough for practical use. During the winter of 2006-2007, Hickey implemented a persistent version of hash array mapped tries with a branching factor of 32. Java's array copy primitives proved efficient. This breakthrough delivered near-constant performance for common operations, finally making immutability viable in production.

In October 2007, the first alpha version of Clojure appeared on the jFli mailing list. The clojure.org site opened its doors, accompanied by a Google group. The reaction exceeded all predictions: 30,000 visits in three days. The eighteen months that followed saw the language expand considerably. The standard library quadrupled, growing from 100 to over 400 functions.

Clojure distinguishes itself through its state management. The language offers several reference types according to needs: refs use a transactional system inspired by databases, agents handle asynchronous updates, atoms offer simple atomic modifications. This approach contrasts with the uniform but limited model of traditional imperative languages.

May 2009 marked the release of Clojure 1.0, accompanied by the first book dedicated to the language. Version 1.1 arrived in December with chunked sequences and transients, bringing substantial performance gains. Version 1.2 in 2010 introduced protocols and user-defined types, completing the language's polymorphic arsenal.

The year 2011 opened a new chapter with ClojureScript. This implementation runs on JavaScript engines, extending Clojure to client-side web development. The project demonstrates the robustness of the language's abstractions: they work on a platform radically different from the JVM without conceptual modification.

In 2012, Hickey launched Datomic, a database that transposes Clojure's principles to storage: immutability, explicit time, functional seman-

tics. The EDN format (Extensible Data Notation) formalizes the serialization syntax used by Clojure, offering an alternative to traditional formats.

The community grew at its own pace. The Clojure/conj conferences began in 2010. ClojureBridge was born in 2014 to diversify the ecosystem through free workshops. The language found its place in various sectors: finance, climatology, retail, data analysis, publishing, healthcare, advertising, genomics.

Major players adopted it for critical systems. Walmart processes electronic receipts from its 5,000 stores with Clojure. Netflix analyzes 2,000 billion daily events thanks to the language. Nubank, a digital bank with over 12 million customers, builds its infrastructure on Clojure and Datomic.

The success rests on functional programming with native interoperability with the host platform, performant immutable structures, powerful concurrency abstractions, and generic data processing. These strengths make Clojure a tool suited to contemporary information systems. In 2025, development continues its measured course, improving tooling and developer experience rather than accumulating features.

F#

Microsoft sought to consolidate its .NET platform in the face of Java's rising dominance. The F# language emerged from a unique intersection: that between academic work on functional programming and the pragmatic needs of the software industry.

F#'s roots reach back to the 1970s, with Robin Milner's work on Meta Language. This language introduced type inference and pattern matching, concepts initially designed for manipulating mathematical proofs. The ML family, which includes Standard ML and OCaml, formed the theoretical foundation from which F# would inherit its core principles.

In 1997, Microsoft launched "Project 7", an ambitious initiative aimed at integrating fourteen languages, both academic and commercial, into the .NET ecosystem. Don Syme, a researcher at Microsoft Research in Cambridge, began developing F# in 2002. His goal? To marry the elegance of functional languages with the richness of the .NET platform.

The first public version arrived in 2005. F# distinguished itself through its ability to fuse functional and object-oriented programming. But above

all, it offered seamless interoperability with other .NET languages. Developers could use existing libraries without friction, a decisive advantage for adoption.

Thanks to the inferred type system inherited from ML, the compiler automatically deduces the types of expressions. Code gains conciseness without sacrificing safety. Active patterns facilitate extensible pattern matching, while computation expressions simplify the writing of asynchronous code. These mechanisms transform the manipulation of complex data into fluid operations.

Microsoft integrated F# into its official lineup in 2007. The language found its place in financial and scientific domains, where the rigor of typing and the conciseness of functional code meet critical requirements. F# 2.0 was released in 2010, the first version fully supported by Visual Studio.

In 2012, F# 3.0 introduced type providers. This mechanism automatically generates types corresponding to external data schemas, whether from SQL databases, web services, or structured files. With data integration made almost trivial, institutions like Credit Suisse and Morgan Stanley adopted F# for their analysis and trading systems, where every error can cost millions.

In 2010, Microsoft released F#'s source code under the Apache 2.0 license. This openness catalyzed the formation of a contributor community. The F# Software Foundation was established in 2014, providing an organizational framework for the language's development. F# gradually emerged from Microsoft's shadow to become a vibrant community project.

The language adapted to the changing landscape of computing. Projects like WebSharper and Fable enabled writing web applications in F#. Support for .NET Core opened the path to cross-platform development. F# was no longer confined to Windows; it ran on Linux and macOS, aligning with the cloud computing era.

C# borrowed several of F#'s features: pattern matching, non-nullable types, and certain functional constructs. The approach to asynchronous programming via computation expressions also inspired other languages. F# demonstrated that a functional language could coexist with object-oriented programming and even enrich it. It remains a niche language, certainly, but a niche where code quality and application safety matter above all else.

Groovy

In 2003, Java reigned over the computing world with its considerable library of components. Yet its rigid architecture discouraged those who simply wanted to script a task, prototype, or write minimal code. Ruby, Python, and Smalltalk seemed more agile, but their syntax bewildered programmers accustomed to Java. James Strachan then created Groovy: a language that would remain faithful to the Java spirit while eliminating its burdens.

The idea was simple: keep Java's appearance to avoid disorienting developers, but lighten the syntax and reduce constraints. Strachan wanted a "Python with Java flavor", where Groovy would serve as glue between existing Java libraries. The language had to integrate naturally into the JVM ecosystem without disrupting established habits.

March 2004 marked a decisive milestone: Groovy was submitted as Java Specification Request (JSR 241) and accepted by vote. This process required rigorous specification, a reference implementation, and compatibility tests. The workload proved overwhelming. Tensions emerged within the community over which directions to take. Strachan eventually withdrew, leaving Guillaume Laforge to take up the torch.

The JSR was never voted on in its final form, but this mandatory process forced the team to structure the language. Without this constraint, the features accumulated during initial development would likely have formed a disparate set rather than a coherent whole.

January 2007 saw the birth of version 1.0. It brought lexical closures, scripts, builders, interpolated strings (GStrings), and named parameters. The language natively integrated regular expressions, operator overloading, and literal syntax for lists and maps. The Groovy Development Kit enriched approximately 430 standard Java classes with new methods.

Two years later, Groovy exceeded its initial role as a scripting language. It was now used to develop complete applications. Its runtime metaprogramming capabilities, already robust, were supplemented by compile-time features. This evolution reduced code complexity without sacrificing performance.

In 2012, the language was gaining ground on projects traditionally reserved for Java. Users demanded more compile-time type checking and better performance. The team added a static nature to the language. Groovy didn't abandon its dynamism: it simply offered the choice between static and dynamic typing according to project needs.

AST (Abstract Syntax Tree) transformations, introduced with version 1.6, were game-changing. They allowed extending the language without

touching its grammar. This declarative approach, based on annotations, facilitated the application of Java design patterns. The @Immutable annotation, for example, automatically generated all the code necessary to create an immutable class conforming to best practices.

Version 2.4 brought Android support, which led to numerous optimizations in bytecode generation. The amount of code produced decreased, memory consumption of internal structures dropped, and performance improved. February 2020 saw the arrival of version 3.0 with its new parser based on Antlr4, offering better compatibility with modern Java syntax.

Groovy's organizational history resembles an obstacle course. First hosted on Codehaus, the project migrated to the Apache Software Foundation in 2015 after the platform's closure. G2One, SpringSource, VMware, Pivotal, and OCI sponsored its development in turn. Despite these changes, the open source community stayed the course and ensured the language's progression.

Groovy left its mark on other technologies. Kotlin drew inspiration from its lexical closures, builder concept, default *it* parameter for closures, templates and interpolated strings, Elvis operator, and safe navigation. Swift and C# adopted similar features like the safe navigation operator.

Groovy's philosophy rests on pragmatic extensibility. Rather than integrating every possible feature into the language core, it provides mechanisms for users to enrich the language according to their needs. AST transformations, type checking extensions, and domain-specific language (DSL) creation illustrate this approach.

The language remains alive and its community continues to innovate while preserving compatibility with Java, which remains at the heart of its design. Groovy maintains its balance between increased productivity for Java developers and dynamic features for specific use cases.

Android

In 2003, Andy Rubin founded Android Inc. in Palo Alto with the ambition of creating an operating system for mobile devices. The name suited him well: at Apple and later at MSN, his colleagues had nicknamed him "Android" because of his passion for robotics. A few years earlier, he had already founded Danger Inc., the company that gave birth to the Danger Hiptop, the phone that T-Mobile commercialized as the Sidekick in 2002.

Initially, the Android Inc. team focused on digital cameras. But that market collapsed while mobile phones were taking off. Rubin and his cofounders Rich Miner, Nick Sears, and Chris White decided to pivot toward a mobile operating system built on the Linux kernel. Money was desperately scarce. Steve Perlman, a friend and investor, even had to bring 10,000 dollars in cash to prevent bankruptcy.

Google acquired Android Inc. in 2005 for 50 million dollars. Larry Page and Sergey Brin were thinking ahead: they wanted to extend their empire beyond computer search. On July 11, 2005, the entire Android team joined the Mountain View offices.

At that time, the mobile landscape belonged to others. Nokia's Symbian dominated everything with 63.5% of the global market in 2007. Microsoft's Windows Mobile captured 12%, RIM's BlackBerry carved out 9.6%. And Apple arrived in January 2007 with the iPhone, redefining what a smartphone was. Facing these giants, Google chose a different path: offering an open and free system.

The Open Handset Alliance was born in 2007. This consortium brought together 34 companies, from phone manufacturers to operators to various technology companies. Their mission was to develop open standards for mobile devices. Android was this alliance's reference system, distributed as open source under the Apache license.

The first public version was released in 2008. The architecture relied on the Linux kernel, which managed the hardware. Above it ran Dalvik, a virtual machine specially designed by Google to optimize execution on mobile devices. Each application ran in its own instance of this virtual machine, so multiple programs could execute simultaneously.

Versions followed one another, named after desserts. Cupcake (1.5) arrived in 2009 with widgets and GPS navigation. Donut (1.6) improved search and accepted different screen resolutions. Eclair (2.0-2.1) managed different email accounts and refined the interface.

Froyo (2.2) in 2010 accelerated performance and strengthened security. Gingerbread (2.3) integrated NFC and better conserved battery. Honeycomb (3.0) in 2011 adapted Android for tablets. Ice Cream Sandwich (4.0) unified the experience between smartphones and tablets, with innovations like facial unlocking.

Jelly Bean (4.1-4.3) in 2012 made the interface smoother thanks to "Project Butter". KitKat (4.4) streamlined the system so it could run on modest machines. Lollipop (5.0) in 2014 introduced "Material Design", Google's new aesthetic. Marshmallow (6.0) rethought application permissions and extended battery life.

The business model contrasted sharply with competitors. Google of-

fered the system to manufacturers, making money through advertising and services like the Play Store. Apple kept its ecosystem closed, Microsoft charged licensing fees: Android took a third route.

Adoption exploded. In 2010, more than 100 million devices ran Android. In 2021, there were 3 billion worldwide. This dominance attracted the attention of competition authorities. The European Commission imposed a 4.34 billion euro fine on Google in 2018 for abuse of dominant position.

The system quickly exceeded its initial scope. It equipped tablets, smartwatches, televisions, cars. Android Wear handled connected objects, Android Auto handled automobiles, making this system a universal platform for mobile computing.

The technical architecture evolved with needs. ART progressively replaced Dalvik to improve performance and energy efficiency. The system integrated facial recognition, contactless payment, augmented reality. Security was continuously strengthened: data encryption, application isolation in sandboxes, monthly updates. The more transparent permission model gave users greater control.

Android transformed the mobile industry. The system democratized access to smartphones by allowing manufacturers to offer devices at all price points. It stimulated innovation through a platform open to developers. Its open-source model inspired other projects and influenced the evolution of mobile systems.

This success rests on a balance: the power of the Linux kernel, the efficiency of a virtual machine designed for mobile, an intuitive interface. Android satisfied manufacturers, developers, and users while adapting to technological changes and new use cases.

Google Chrome

In September 2008, Google launched the beta version of Chrome, its own web browser. At that time, Internet Explorer dominated the market and Firefox had established itself as the essential alternative. But web applications were growing in complexity, HTML5 was in preparation, and existing browsers were showing their limitations. Google observed that browsers handled these new applications poorly, regularly causing crashes that affected the entire browsing experience.

Chrome arrived with a multi-process architecture. Each tab runs in

its own process, isolated from the others. When a page crashes, only the affected tab closes. The browser's other components continue to function normally. This break from traditional monolithic browsers transformed the user experience.

Security occupied a central place from the design phase. Sandboxing isolates rendering processes by drastically limiting their privileges and access to the operating system. A compromised process cannot infect the rest of the system. Chrome relies on four Windows security mechanisms to strengthen this protection against malware.

The V8 JavaScript engine marked a major technical breakthrough. Developed by a Google team in Denmark, it compiles directly to machine code without going through an interpreter. Hidden classes optimize the processing of similar objects. The garbage collector is precise while remaining lightweight, making JavaScript execution much faster than on competing engines.

Chrome's interface is striking in its simplicity. The developers followed the principle of "simple interface, powerful core". They maximized page display space by reducing interface elements to the strict minimum. The address bar and search field merge into the omnibox. This minimalist approach would later inspire many other browsers.

Google simultaneously released Chromium, the open source version (and foundation) of Chrome. The majority of the source code was released under the BSD license, allowing developers to use it freely. This strategic choice promoted the adoption of Chrome technologies and stimulated innovation throughout the browser ecosystem.

Performance remains at the heart of concerns. In 2015, Chrome introduced a JavaScript bytecode cache. Frequently used scripts have their compiled code saved to disk. Pages load faster, even after a browser restart. This optimization illustrates the constant attention paid to execution speed.

Chrome integrates WebKit as its rendering engine, initially created by Apple. But Google retained only the WebCore component and developed its own adaptation layer. This customization gave it complete control over browser performance.

Success came quickly. One year after its launch, in August 2009, Chrome was the third most widely used browser in the world, surpassing Safari and Opera. In November 2010, it crossed the 10% market share threshold in the United States. The progression accelerated year after year.

In 2020, Microsoft abandoned EdgeHTML and rebuilt Edge based on Chromium. Brave, Opera, and many other browsers rely on this technol-

ogy. A rich ecosystem developed around the project.

The Chrome Web Store, launched in 2010, centralized extension distribution. This structured platform facilitated extension discovery and installation, an asset in Chrome's adoption, enriching the browser's functionality.

Automatic updates changed the way browsers are distributed. Chrome updates silently in the background, without user intervention. This approach improves security by ensuring everyone has the latest fixes. No more waiting for users to decide to manually update their browser.

Chrome evolves with the Web. The browser progressively integrates advanced technologies like the CSS Paint API for customizing graphic rendering, or CSS 3D transformations for sophisticated visual effects. These features meet the needs of modern web applications that now rival native applications.

Chrome's architecture inspires operating system design. Chrome OS, launched in 2011, extends the browser's principles to the scale of a complete OS. Google sees this as the future of computing, a web-centric world where the browser is the primary application execution platform. This vision has been validated by the explosion of SaaS solutions in enterprises.

The pursuit of speed never stops. Optimizations affect page loading, JavaScript execution, and image rendering. Chrome introduced resource preloading and data compression to further reduce loading times. Every millisecond counts.

In 2023, Chrome dominates the web browser market. Its influence on the evolution of web standards and browsing technologies is immense. Chrome's technical innovations have become widespread. The web has transformed into a true platform for sophisticated applications, and Chrome has played a decisive role in this transformation.

HTML5

At the turn of the 2010s, the Web was going through a period where smartphones were proliferating, social networks were exploding, and developers were running up against a frustrating limitation: how to write code that works everywhere without having to rewrite everything for each platform? HTML5 arrived in this context, not as a simple technical update, but as a redesign of the Web's architecture.

The story begins in 2004, when Ian Hickson, then at Apple, cofounded the WHATWG (Web Hypertext Application Technology Working Group). This group brought together engineers from Apple, Mozilla Foundation, and Opera Software, all convinced that the foundations of the Web needed rethinking. The W3C eventually incorporated their work into a broader vision of an open platform.

HTML5 wasn't just a new markup language. It was a coherent set that encompassed the Document Object Model, CSS style sheets, JavaScript, and a whole series of standardized interfaces for video, geolocation, local storage, and graphics. This comprehensive approach was a boon for developers.

Matthew McVickar, a web developer at Ocupop, expressed it clearly: HTML5 formalized practices that developers were already using, but in a makeshift way. With the JavaScript interface for geolocation, for example, mobile browsers accessed GPS without going through proprietary solutions. No more workarounds, it was time for standards.

CSS3 added its contribution by transforming how page appearance was managed. Developers now created visual effects directly in the browser, without going through Photoshop or other external tools. The time savings were considerable, the flexibility increased.

Local storage marked a break with traditional cookies. This function allowed the storage of large volumes of structured data on the client side, thus enabling Web applications to run without an internet connection. Graphics capabilities were also enhanced with native support for SVG vector graphics and the Canvas element, which opened the door to 2D and 3D animations via JavaScript.

Ian Hickson, who had meanwhile moved to Google, emphasized the technical rigor brought by HTML5. Previous versions of HTML left room for interpretation: two browsers could implement the same specifications differently while claiming to respect the standards. HTML5 established precise rules that guaranteed true compatibility.

But deployment faced obstacles. Video crystallized tensions: no single standard for compression, streaming, or digital rights management. In 2010, Apple refused Flash on the iPhone and iPad, preferring its own version of HTML5. Microsoft and Google developed their approaches, forcing developers to juggle multiple versions of their code.

Hui Zhang, a professor at Carnegie Mellon and cofounder of Conviva Inc., knew the problem well. For him, video remained the thorniest aspect of online content. Bringing together codecs, streaming protocols, and rights management required time and compromises between industry players.

The W3C planned in 2014 to finalize the HTML5 specification. Philippe Le Hégaret, head of the interaction domain, nevertheless tempered expectations: development continued. The organization regularly received requests to add features, such as voice-to-text conversion directly in the browser.

The WHATWG took a radical turn in 2011 by abandoning version numbering. HTML was a "living standard" in permanent development, a vision that fit the reality of a constantly evolving Web. Ian Hickson said it bluntly: HTML development would continue as long as the technology remained relevant.

This technology showed how standardization could stimulate innovation without sacrificing compatibility. It also illustrated the importance of collaboration between industry players in advancing the Web.

Memcached

In 2003, Brad Fitzpatrick faced a major challenge. LiveJournal, the blogging platform he had created, was literally exploding: over 2.5 million accounts and an infrastructure beginning to show its limits. Some 70 machines ran day and night, but the databases were buckling under the pressure of queries. The site had a particular feature that complicated everything: each piece of content had different security levels and appeared in multiple views. Generating static pages was impossible when the elements composing them each had their own lifecycle.

Fitzpatrick then observed a technical reality that would shape his entire approach. Processors were gaining speed year after year, while hard drives lagged behind. Why not leverage this computing power rather than exhaust resources with endless disk accesses? The idea emerged to use the RAM sitting idle in web servers, that memory just waiting to be put to work.

A first prototype was built in Perl. The trial proved disappointing: too slow, too memory-hungry. Fitzpatrick started over in C and built a single-process, single-thread daemon that relied on asynchronous I/O. To ensure his system would work everywhere, he integrated libevent, a library that automatically chooses the best file descriptor management strategy based on the execution environment.

The principle behind Memcached can be stated simply: transform available server memory into a vast shared pool. A distributed hash ta-

ble where data circulates between machines through consistent hashing. Nothing superfluous in the architecture: the server doesn't communicate with its peers, stores nothing on disk, offers three basic operations—set, get, delete—and that's it. This intentional austerity delivers formidable performance, with O(1) algorithms that respond instantly. The slab memory allocator prevents the fragmentation that plagued early versions using malloc.

Facebook adopted Memcached in 2007 and developed `mcrouter`, an enhanced variant with *sharding* features. Twitter followed with `twemcache`, tailored to its specific needs. These web giants validated Fitzpatrick's approach and made Memcached a de facto standard.

The figures are staggering. At Facebook, the system handles billions of requests per second and stores trillions of objects. The cache hit rate regularly reaches 92%: only a handful of requests actually need to query the database. The load is reduced accordingly. The system introduced "leases" to manage write conflicts and prevent hordes of clients from all attempting to regenerate an expired entry simultaneously.

The distributed architecture allows welcome flexibility. Servers can be added to or removed from the pool on the fly; the system adjusts seamlessly. This horizontal scaling capability quickly became common practice in modern web infrastructure design.

The community embraced the project. Client libraries sprouted for PHP, Python, Ruby, Java, C#, adding object serialization and transparent compression, making the tool more manageable in various contexts. The source code continues to evolve thanks to contributions from a thriving community that shows no signs of slowing down.

In 2024, Memcached still holds its ground in web infrastructures. The project maintains its original philosophy: stay simple, fast, reliable. This minimalist approach contrasts with more baroque alternatives like Redis, but perfectly meets pure caching needs. Memcached demonstrates that an elegant, stripped-down solution sometimes solves complex problems better than sophisticated machinery.

OpenStack

By the end of the 2000s, hardware and software resources were gradually shifting toward *cloud computing*, promising enhanced security and centralized management. In 2009, NASA was looking for a platform for

its internal web services. The market only offered proprietary solutions, locked in by a few industry giants.

The American space agency's developers believed that an open-source alternative would better serve a broad community: academic researchers, government institutions, and various businesses. Nothing like this existed. So they decided to create this solution themselves. The initiative attracted Rackspace, whose engineers were working on a similar project to support their hardware and services sales business.

In early 2010, Rackspace representatives contacted NASA. The two teams met in a Thai restaurant, and this modest collaboration gave birth to OpenStack. The official launch took place in July 2010, with the ambition of helping organizations deploy cloud services on standard hardware. Four months later, the first version, named Austin, was released. It provided cloud computing and object storage for Infrastructure as a Service.

The initial projects enabled control of resource pools: compute, storage, and networking, all through a centralized dashboard in data centers. Administrators had fine-grained control, while users obtained their resources instantly. The community then committed to releasing new versions every six months.

The OpenStack Foundation was established in 2012. This independent nonprofit organization had the mission of promoting the development and adoption of the software. The structure ensured neutral management of intellectual property and resource sharing, creating a level playing field for all participants.

The governance model rested on three pillars: a board of directors set the vision and governance, a technical committee managed releases and cross-project requirements, and a user committee gathered field feedback and advocated for their interests. The community operated according to four principles: open source code under the Apache 2.0 license, an open community producing an active ecosystem, transparent development with public code reviews, and open design with biannual summits accessible to all.

Contributors came from diverse backgrounds. Software engineers contributing to the code, writers providing documentation and translations, application developers creating solutions on the infrastructure, and operators and users deploying and managing their clouds. Working groups gathered requirements from specific segments such as telecommunications or the enterprise world. The commercial ecosystem included cloud providers and vendors adding their value to the base software.

Adoption accelerated across different sectors. BMW, Disney, and Walmart proved the solution's viability in production. PayPal adopted it

for the agility and availability demanded by its flagship products. Time Warner Cable relied on this flexibility to manage significant data volumes in its centers.

The technology expanded far beyond its initially intended uses. CERN employed it to analyze data from its Large Hadron Collider, managing over 115,000 compute cores. Adobe Digital Marketing used it to transform its virtualization environment into self-service computing. Australia's NeCTAR university adopted it for national collaborative research, supporting over 6,000 researchers.

In 2024, Rackspace reaffirmed its commitment with OpenStack Enterprise, a fully managed cloud solution for mission-critical workloads. The company had contributed over 5.6 million lines of code and ranked among the world's largest OpenStack cloud providers. Its expertise included over one billion server-hours of production experience.

The impact on the IT industry was considerable. The platform enabled organizations of all sizes to deploy their own cloud infrastructures, thus escaping the dominance of proprietary services. Its open-source nature fostered collaborative innovation and reduced infrastructure costs. The open APIs and support from virtually all major IT vendors gave users the freedom to move their workloads between private and public clouds according to their needs.

The benefits included accelerated time to market through a self-service portal, an API-based platform for developing cloud-native applications, and drastically reduced provisioning times. Interoperability and hybrid cloud capability allowed businesses to choose the best environment for their applications without depending on a single vendor.

Evolution continued with the integration of new technologies such as containers, virtual machines, and *bare metal* servers. The global community now counts over 56,000 individual members in more than 180 countries and over 600 companies. This growth testifies to the platform's continued relevance in a constantly changing technological landscape, where demand for flexible and open cloud solutions continues to grow.

Bitcoin

In October 2008, as the financial crisis shook the entire world and trust in banks collapsed, a mysterious document circulated on the internet. Its author, Satoshi Nakamoto, whose identity-whether a single person or a

group-would never be known, proposed nothing less than a revolution: creating a digital currency beyond the control of financial institutions. The document's title, "Bitcoin: A Peer-to-Peer Electronic Cash System", clearly announced its ambition. The goal was to build a payment system where transactions occur directly between users, without intermediaries.

Three months later, on January 3, 2009, Nakamoto mined the first block of what would be called the blockchain. This genesis block contained a message that spoke volumes about the project's motivations: "The Times 03/Jan/2009 Chancellor on brink of second bailout for banks". This reference to the British newspaper was no accident. It reminded everyone that banks were about to receive yet another bailout package, financed by public money. Bitcoin was thus born from protest, from a rejection of a banking system deemed to have failed.

The beginnings remained modest. The first subsequent block appeared only five days later, on January 8. This unusual delay still intrigues people today. Some see it as a testing period, others as a biblical reference to the six days of creation. In any case, the network started slowly. At the time, anyone could mine bitcoins with a home computer. The difficulty was set to the minimum, at 1, a trivial level compared to current standards that require industrial facilities.

Active until December 2010, Nakamoto disappeared without a trace. His voluntary departure left the community of developers to continue alone. This absence fuels speculation about his identity, but also ensures that no one controls the project. Bitcoin is truly a common good, with no leader or owner.

Exchange platforms emerged in 2011, allowing bitcoins to be bought and sold for traditional currencies. Mt. Gox established itself and soon handled 70% of global transactions. But in 2014, the hacking of this Japanese platform resulted in the loss of 850,000 bitcoins. The shock shook the ecosystem and brutally reminded everyone of the risks associated with these new assets.

The Bitcoin protocol relies on ingenious mechanisms. Proof of work requires miners to solve complex calculations to validate transactions. In return, they receive a reward in bitcoins. This reward, initially 50 units per block, halves approximately every four years. This process, called "halving", limits total issuance to 21 million bitcoins. This programmed scarcity echoes that of gold and contributes to the cryptocurrency's valuation.

Gradually, companies accepted Bitcoin as a means of payment. In 2012, a few pioneers took the plunge. Major players like Microsoft, Dell, or PayPal joined the movement. This adoption conferred new legitimacy

on what had initially been merely a cryptographic experiment.

Bitcoin's price went through spectacular cycles. In 2013, it climbed from less than $20 to over $1,000, before falling back. In 2017, another surge to nearly $20,000, followed by a collapse the following year. These extreme variations betrayed the speculative nature of the asset. No one really agrees on its actual value, if such a thing even exists.

Bitcoin's success inspired other projects. Hundreds of alternative cryptocurrencies, known as "altcoins", emerged with various technical proposals. The blockchain found unexpected applications: smart contracts, decentralized finance, product traceability. This technology, designed for a digital currency, extended far beyond its initial use.

Some states reacted by outright banning Bitcoin, while others attempted to regulate it. Concerns focused on money laundering, investor protection, and tax evasion. The environmental impact of mining, which consumes large amounts of electricity, also raised growing criticism. The debate remains heated between advocates of absolute financial freedom and those calling for strict regulation.

More than fifteen years after its creation, Bitcoin has established itself as an asset class in its own right, recognized by institutional investors. This shows how a radical idea, carried by a few lines of code, can shake the certainties of a millennia-old financial system. The technology continues to evolve: the Lightning Network attempts to solve transaction speed problems, while new improvements are regularly proposed.

This unprecedented experiment opened an immense field in cryptography and distributed systems. Bitcoin is an object of fascination, a symbol of protest that rejects intermediaries, central authorities, and controls. Ultimately, what is money? Who should control it? How should we organize trust in a digital world? These questions continue to fuel reflections on the future of our connected societies.

CouchDB

In 2005, Damien Katz left IBM with an idea in mind. His years working on Lotus Notes had taught him what a document database was, but also what it could become. At that time, the computing world revolved almost exclusively around relational databases. Yet something was off. Modern web applications and collaborative tools demanded something else, a flexibility that SQL couldn't quite deliver.

He started by writing CouchDB in C++. The early prototypes worked, but concurrency management tripped him up. He searched for a solution, browsed technical forums, and stumbled upon the blog "Lambda the Ultimate" where someone was discussing Erlang. This language, originally designed for telecommunications at Ericsson, had exactly what he needed. Fault tolerance, native distribution, concurrent processing: it was all there. Katz then made a radical decision. He threw away his C++ code and started over entirely in Erlang.

This technical choice shaped CouchDB. Erlang brought with it a philosophy: systems must survive failures, processes can fail without bringing down the whole system, data can be distributed naturally. These properties, designed for telephone exchanges that needed to run without interruption, fit perfectly with the needs of a modern database. The parallel between telecommunications and data storage wasn't obvious at first glance, but it proved relevant.

CouchDB broke with several established conventions. No rigid schema: JSON documents could evolve freely, transform as needed without requiring database migration. This freedom addressed the realities of web development where structures constantly change. The interface spoke HTTP, used GET to read, PUT to write, DELETE to remove. Nothing more, nothing less. A web developer found themselves on familiar ground.

JavaScript became the query language. Where other databases imposed their own dialect, CouchDB bet on a language every web developer knew. Views, those custom indexes that allow querying data, were written in JavaScript. MapReduce too.

Bidirectional replication stood among its most distinctive strengths. Two instances could synchronize, exchange their modifications after operating separately for days. For example, a smartphone loses its connection, continues working locally, then synchronizes everything once reconnected. Shortly after Apple's iPhone launch, this capability anticipated a world where mobile devices and intermittent connections would be the norm rather than the exception.

To handle concurrent writes, CouchDB adopted an optimistic approach. No locks that block, no forced waits. Each document has a revision number, somewhat like Git with its commits. When two modifications conflict, CouchDB keeps both versions and lets the application decide. This conflict management, inspired by version control systems, provided unusual flexibility for a database.

In 2011, CouchDB merged with Membase, a distributed cache technology with remarkable performance. From this union came Couchbase, which combined CouchDB's document richness with Membase's raw speed.

The hybrid creature inherited from both worlds: persistence and flexibility on one side, velocity on the other.

Technically, CouchDB rests on a few well-designed components. The storage engine writes to disk while maintaining indexes for fast access. The view system transforms documents into queryable indexes. The replication protocol ensures consistency between distant instances. These components interact by leveraging Erlang's strengths, which orchestrates everything with a certain elegance.

CouchDB participated in the emergence of the NoSQL movement, and proved that data storage could be thought of differently. Its influence can be seen in the architecture of many modern databases that adopted its ideas: schema-less documents, web-friendly APIs, intelligent replication. PouchDB, its JavaScript derivative for browsers, extended synchronization all the way to the web client.

Use cases multiplied where flexibility mattered. Mobile applications that needed to work offline, collaborative systems where users modified the same data, tools that needed to replicate between distant sites. CouchDB excelled in these situations where connection wasn't guaranteed, where structures evolved quickly, where data distribution was a necessity.

Node.js

In 2009, Ryan Dahl presented Node.js at a European JavaScript conference. This technology would be a game-changer by enabling JavaScript to run outside web browsers. Until then, the language had remained confined to the front-end since its creation by Brendan Eich in 1995 for Netscape. Originally, JavaScript was mainly used to animate web pages and manipulate the DOM. Netscape had attempted a server-side version with Netscape Enterprise Server, but the attempt had failed.

The name JavaScript reflects a marketing strategy: riding the wave of Java's popularity. Yet the two languages have nothing in common. JavaScript owes far more to Scheme and Self than to Java, from which it borrows only a few syntax elements. For years, serious programmers looked down on JavaScript, deeming it too slow and unsuitable for real applications. This reputation shattered with the arrival of more powerful engines, notably Google Chrome's V8. Through just-in-time compilation, code inlining, and dynamic optimization, JavaScript could rival C++ on certain benchmarks and outperform Python in most cases.

Node.js leverages the V8 engine to build a robust server platform. The architecture breaks with traditional approaches. Where Apache creates a new thread or process for each connection, Node.js relies on a single-threaded event loop. This strategy avoids the overhead of creating and managing threads, which becomes significant when the server is overwhelmed with requests.

Input-output management makes all the difference. Instead of blocking execution while waiting for an operation to complete, Node.js uses callbacks that handle results asynchronously. The server can thus juggle multiple requests while an operation runs in the background. Resources are better utilized, idle time disappears.

The community quickly embraced the project. In 2010, Isaac Schlueter launched NPM (Node Package Manager), the true cornerstone of the ecosystem. Developers can share and reuse code without friction, which accelerates adoption. The following year, a native Windows version emerged and considerably expanded the audience.

Joyent, a software and services company based in San Francisco, steered development for several years before the project joined the Node.js Foundation in 2015. During this transition to more open governance under a technical steering committee, Node.js became a collaborative project of the Linux Foundation. In 2019, the merger with the JS Foundation gave birth to the OpenJS Foundation, which better coordinates the JavaScript ecosystem.

The NPM registry now hosts over two million packages, making it the world's largest software repository. It contains frameworks like Express, development tools like browserify and gulp, and over 8,000 database clients, from redis to mongoose.

The single-threaded event loop might seem like a bottleneck on modern multi-core processors. Node.js circumvents this obstacle with the cluster module, which creates worker processes sharing the parent server's ports. A load-aware round-robin algorithm distributes requests among available cores.

Node.js's success popularized event-based asynchronous programming. This approach suits concurrency-intensive applications well, such as instant messaging or real-time services. The platform also fostered isomorphic JavaScript, where the same code runs on both client and server.

Node.js continues to evolve. Support for ECMAScript modules, performance improvements, and security enhancements are among current priorities. The platform remains relevant for microservices, serverless applications, development tools, and desktop applications via Electron. By offering a different way to build servers, Ryan Dahl redefined the construc-

tion of high-performance, scalable web applications.

Raspberry Pi

In 2006, Eben Upton and his colleagues at the Cambridge Computer Laboratory faced an unexpected problem: applicants to their program were becoming less and less proficient in computing. Worse still, their numbers were declining year after year. The diagnosis was clear. The computers of the 1980s, with their visible internals and direct programming, had disappeared in favor of sealed machines, carefully hidden behind plastic casings and polished interfaces. Computers no longer invited tinkering; they were simply used.

Upton dreamed of an affordable machine that would restore to young people the joy of experimentation he had known at their age. His initial prototype, named ABC Micro as a tribute to the BBC Micro, was limited to a board built around an Atmel microcontroller and a few memory chips. Early tests with children, however, revealed that this minimalist approach failed to spark the expected enthusiasm.

Also in 2006, Upton joined Broadcom. This opportunity gave him access to far more powerful components, notably the BCM2835 which integrated an ARM11 processor. This chip concentrated everything needed to run a real computer in a few square centimeters: HDMI output, 3D acceleration, video decoding, USB controller. The addition of a general-purpose ARM processor changed everything, particularly by running Linux on it.

The Raspberry Pi Foundation was established in 2009. Two years later, the final board took shape. Upton and his team envisioned limited production, a few hundred units per year at most. But the enthusiasm exceeded their wildest expectations. The foundation reoriented its strategy: it retained design, branding, and prototypes, but delegated manufacturing to RS Components and Premier Farnell.

In early 2012, the first 10,000 Raspberry Pis sold out within hours. The name circulated in technical circles and maker communities. The foundation seized this momentum to revive British electronics industry: after starting in China, production gradually migrated to Wales until achieving complete localization.

Users immediately diverted the board from its original educational purpose. The 32 input-output pins, which Upton had added almost as an afterthought, became the most prized feature. Raspberry Pi-powered

drones appeared, along with stratospheric balloons and home automation installations. One hobbyist transformed his microwave by adding a touchscreen, voice commands, a web interface, and a barcode reader connected to a recipe database.

These sophisticated projects, requiring advanced skills, paradoxically reinforced the educational dimension of the project. They showed young people what could be accomplished with programming knowledge. Workshops in schools confirmed this intuition: sometimes merely changing the color of a snake in a game was enough to make a student refuse to stop. This first sensation of control over the machine produced an effect comparable to what Upton had experienced in his youth.

The Raspberry Pi combines technical and economic characteristics that explain its success. Its 700 MHz ARM1176JZF-S processor, coupled with a VideoCore IV GPU, delivers remarkable multimedia performance for a device of this size. The SDRAM memory, increased from 256 MB on Model A to 512 MB on Model B, is sufficient to run Linux comfortably. The absence of a hard drive, replaced by a simple SD card, keeps the price low and simplifies operating system changes.

Hardware updates followed in succession without disruption. The Raspberry Pi 4 features a Broadcom BCM2711 quad-core 64-bit ARM Cortex-A72 at 1.5 GHz, up to 8 GB of LPDDR4 memory, Bluetooth 5.0, dual-band Wi-Fi, and USB 3.0 ports. These developments preserve backward compatibility while expanding the realm of possibilities.

The Raspberry Pi has proven that a market exists for accessible single-board computers. With 19 million units sold, it ranks third among general-purpose computers. It has revived interest in hands-on computer learning and inspired an entire generation of engineers and creators. Its story demonstrates how a local educational initiative, by hitting the mark, can trigger an international movement that redefines our relationship with technology.

X

2010

The Era of Constant Connectivity and Big Data

The decade of 2010-2020 marked a break in our relationship with the digital world. More than a simple technological turning point, these years witnessed the birth of a society where the digital merged with reality until their boundaries blurred. This globalization took concrete form in our daily lives. Populations, predominantly urban and mobile, demanded instant access to information and online services. This thirst for immediacy shaped the innovations of an entire era.

The early years of this decade revealed the unsuspected power of social networks, with Facebook and Twitter spearheading genuine revolutions. The Arab Spring of 2011 demonstrated how these platforms could bypass traditional media controlled by authoritarian regimes. A simple smartphone was now sufficient to broadcast images of demonstrations, coordinate gatherings, and give voice to those previously silenced. This media leap forward changed our perception of technology's role in social movements.

The widespread adoption of *smartphones*, coupled with the deployment of 4G networks, transformed every citizen into a content creator. We began producing and consuming astronomical quantities of data. This avalanche of information gave rise to the phenomenon we called Big Data.

Our traditional infrastructures buckled under the weight of these massive flows. New architectures and analytical methods had to be devised to make sense of this information deluge.

At the heart of this transformation, a handful of American companies built empires of unprecedented scale. Google, Apple, Facebook, Amazon, and Microsoft strengthened their grip on the global digital ecosystem. Their strategy relied on the methodical acquisition of any innovative startup that might threaten their dominance. This concentration of digital power raised legitimate concerns about user privacy and information control. Those who hold our data wield considerable power over our lives.

Facing this Western hegemony, China developed its own digital universe. Players like Alibaba, Tencent, and Baidu created a parallel ecosystem under the close supervision of the Chinese government and the Chinese Communist Party. Beijing implemented an ambitious national strategy aimed at dominating future technologies, particularly artificial intelligence and 5G networks. This rise to power transformed technological innovation into a genuine geopolitical issue. The Sino-American rivalry shifted from the oceans to laboratories and data centers.

The year 2013 marked an earthquake in our relationship with digital security. Edward Snowden revealed the staggering extent of surveillance conducted by American intelligence services. The world discovered that our communications were systematically intercepted and analyzed. This shock triggered global awareness about the fragility of our privacy in the digital age. Europe responded by drafting the General Data Protection Regulation, which has since become a global reference. Cybersecurity, previously a marginal technical concern, emerged as a strategic priority for states and organizations.

Social networks gradually colonized our information space. Their influence on public opinion formation became evident during the 2016 U.S. presidential election with the Cambridge Analytica scandal, which revealed how Facebook had enabled the exploitation of millions of users' data for political purposes. These platforms showed their vulnerability to manipulation attempts. Even more concerning, their recommendation algorithms trap us in "filter bubbles", comfortable but distorted information universes. This fragmentation of public debate raises questions about how our democracies function in the digital age.

Cloud computing transformed the IT infrastructures of businesses and individuals alike. Why invest in expensive and underutilized servers when you can rent computing power on demand? This flexibility accelerated the digital transformation of traditional organizations. But this migration to the cloud also created new dependencies. The question of digi-

tal sovereignty arose again with particular urgency, especially in Europe where the dominance of American providers sparked legitimate concerns.

Our daily lives became populated with connected devices. The Internet of Things wove an invisible yet omnipresent network around us: smartwatches, voice-activated speakers, automatic thermostats. These devices constantly capture information about our habits, movements, and preferences. They introduced new ways of interacting with machines. Voice and gesture gradually replaced keyboard and mouse. This constant collection of personal data fuels increasingly sophisticated artificial intelligence systems.

Bitcoin, born in the shadow of the 2008 financial crisis, truly emerged during this decade. This digital currency, independent of central banks, challenged our financial systems. Financial technologies transformed entire sectors: mobile payments, peer-to-peer lending, automated investment management. Traditional institutions had to adapt to these nimble new competitors. Regulators, often caught off guard by galloping innovation, oscillated between mistrust and fascination for these technologies that promised to democratize access to financial services.

The stunning progress in artificial intelligence and robotics accelerated the automation of numerous tasks. Jobs once thought reserved for humans found themselves threatened by learning machines. This evolution raised questions about the future of work, preparing young generations for a world where skills would become obsolete quickly, and ensuring a just transition for those whose jobs were disappearing. These concerns took on growing political dimensions as digital inequalities widened between different population groups.

The platform economy disrupted entire sectors. Uber, Airbnb, Deliveroo—these names became emblematic of a new economic model based on direct connections between supply and demand. This transformation created hybrid forms of employment, neither traditional employment nor conventional entrepreneurship. While these platforms opened new opportunities and streamlined exchanges, they also generated considerable social tensions. Job insecurity, circumvention of regulations, tax evasion—criticisms multiplied, forcing legislators to rethink legal frameworks overtaken by innovation.

Awareness of the digital world's environmental impact gradually took hold. The carbon footprint of data centers, Bitcoin mining's energy consumption, and planned obsolescence of electronic devices became realities that began influencing the technological choices of organizations concerned with their ecological responsibility. Initiatives for a more sustainable digital world emerged, seeking to reconcile innovation and sustainability. This

reflection on the environmental ethics of digital technologies would become central in the following years.

The end of the decade experienced brutal and unexpected acceleration. The 2020 global pandemic precipitated the digital transformation of entire sectors. Overnight, remote work became widespread, education moved online, and e-commerce became vital. This health crisis served as a full-scale test for our digital infrastructures, revealing both their resilience and their limits. It also highlighted the urgency of bridging the digital divide. Access to quality connectivity and digital skills emerged as essential conditions for participation in social and economic life.

At the end of these ten years, the digital world no longer appeared as a distinct sector but as the fabric of our social organization. The innovations of this period responded to our thirst for connection and immediacy. But they also generated new ethical, social, and environmental problems that would continue to shape the evolution of computing. The 2010s will be remembered as the period when our world definitively shifted into the digital age, with its dazzling promises and troubling shadows. Computing now shapes our perception of the world and our social interactions.

Elasticsearch

The story of Elasticsearch begins as a love story. In 2004, Shay Banon, a young developer, moved to London to support his wife, who dreamed of becoming a chef. Without employment, he decided to put this time to good use by exploring emerging technologies. His project: to create an application where his wife could store and retrieve all the culinary knowledge acquired during her training.

At the heart of this application, Banon envisioned a simple search box. One would only need to enter a concept, a thought, or an ingredient to immediately explore gastronomic possibilities. This vision led him to Apache Lucene, the reference library for search in Java. His immersion in Lucene gave birth to Compass, a project aimed at simplifying the use of Lucene in conventional Java applications. Conceptually, Compass presented itself as “Hibernate for Lucene”, facilitating the mapping of the domain model to the search index.

The project captivated Banon, who ended up dedicating more time to it than to the initial culinary application. A few months after its creation, he open-sourced Compass. The success was immediate. Developers appreciated the simplicity with which they could now index their domain

models in Lucene and perform searches. This freedom gave them access to unexpected and innovative uses.

In 2010, drawing on this experience and user feedback, Banon launched Elasticsearch. This new creation maintained Compass's philosophy of simplicity while addressing modern needs for distribution and scalability. He left his position at Gigaspaces to devote himself entirely to Elasticsearch development for nearly two years.

The open-source community embraced the project. Developers enriched the features, created an administration console, and developed graphical analysis tools. These collective contributions generated considerable momentum. In 2012, Elasticsearch attracted the attention of chief technology officers from numerous companies who praised its qualities. With 200,000 monthly downloads, an unprecedented figure for enterprise software, Elasticsearch established itself as an essential solution.

The founding team formed around Banon with Steven Schuurman, Uri Boness, and Simon Willnauer, all recognized experts in the search field. Together, they created a company to support the project's development. The company attracted the best talent from the Lucene community and became the preferred destination for open source in the search domain.

Elasticsearch's architecture is based on an innovative approach. The system uses JSON and a RESTful interface, making it accessible from any programming language. Its distributed nature gives it the capacity to process large volumes of data by automatically distributing the load across multiple nodes. Schema flexibility allows the storage of complex documents without prior structure definition.

Over the years, the ecosystem expanded. Logstash came to facilitate data ingestion, while Kibana offered advanced visualization capabilities. These tools form the ELK stack (Elasticsearch, Logstash, Kibana), which became a reference for log analysis and infrastructure monitoring. In the 2020s, lighter alternatives such as Fluentd or FluentBit often replaced Logstash for data ingestion.

Elasticsearch use cases multiplied: website search, user interaction analysis, anomaly detection in security systems. Organizations like CERN, Wikimedia, Foursquare, and GitHub adopted it to manage their search and data analysis needs.

In 2018, the company went public, confirming the project's maturity. The following year marked a significant change. Elastic, the company behind Elasticsearch, modified its license. In January 2021, it abandoned the Apache 2.0 license for a dual proprietary license, including the Server Side Public License (SSPL). This decision prompted a reaction from Amazon Web Services, which created a fork of the project named OpenSearch,

maintaining the open-source tradition under the Apache 2.0 license.

Elasticsearch's impact on the software industry extends beyond its initial objective as a search engine. The technology has redefined how organizations manage and explore their data. Its distributed architecture and API-oriented approach have influenced the design of numerous modern systems. The ease of use, combined with powerful features, gives developers the ability to integrate sophisticated search capabilities without deep expertise in information retrieval. Elasticsearch's widespread adoption illustrates the growing importance of real-time data analysis in contemporary information systems.

A personal project sometimes evolves into a major infrastructure, used by thousands of organizations around the world.

HEIF

The JPEG format has reigned over the digital imaging world since 1992. Mobile devices, computers, servers-all use this format that has established itself as a de facto standard. But this hegemony conceals technical weaknesses that have been revealed over time: no support for high color depth, no high dynamic range management, no transparency. And then there are those characteristic visual artifacts, that compression which shows its limits in the face of current needs.

In 2013, the technical team at Nokia Technologies began considering an alternative. They defined the requirements, identified use cases. The specification development took a year and a half, until the summer of 2015. But the project's fate shifted with Microsoft's acquisition of Nokia. The company transferred its rights and information to Apple, which took over.

HEIF (High Efficiency Image File Format) breaks with traditional image format design. Where JPEG stores a single image, HEIF can contain multiple images, entire sequences, and their metadata. The structure relies on HEVC (H.265) video coding technologies, whose advanced compression techniques are applied to still images. It's a paradigm shift.

In 2017, Apple took the plunge. The iPhone 7, the sixth generation iPad, Macs running OS X High Sierra adopted the format. The company introduced the .heic extension for HEVC-encoded images, reserving .heif for files using other codecs. Millions of users discovered the format without really noticing.

The gains were immediate: a 50% reduction in file size compared to JPEG, with equivalent or superior visual quality. The format handles burst capture, simultaneous recording of videos and still images, animations, focal and exposure images in a single file. Editing operations on predefined derived images are made possible.

HEIF's architecture inherits from the ISO/IEC 14496-12 base format. The box system enables data nesting, creating a hierarchy of relationships. The first box, named "ftyp", contains general encoding metadata. A "meta" box holds the remaining information. The format integrates image tiling, which allows parallel decoding or partial loading of regions of interest in high-resolution images.

Apple adapted the specifications in its own way. The company systematically uses 512×512 pixel tiles, generating "infe" entries for each tile. These tiles are assembled into a single image via a "grid"-type derived image. Apple exploits auxiliary depth maps, accompanied by XMP metadata.

Format adoption is gradually expanding. Android has integrated it since version 9 (Pie), Samsung offers it on certain models like the Galaxy S10. Windows 10 introduced it in version 1809, but specific extensions must be installed from the Microsoft Store. The transition requires technical and practical adjustments.

For mobile photography, the impact is considerable. Users keep more photos on their devices without sacrificing visual quality. Non-destructive editing is more accessible, metadata management more flexible. Application developers have new possibilities at their disposal.

On the digital forensics side, typically for criminal investigations, HEIF complicates the work. With its structure's ability to conceal data in different boxes, to manipulate relationships between images, investigators must develop new methodologies, account for multiple data layers, variable implementations across manufacturers.

The ISO/IEC 23008-12 standard now defines the format's technical specifications. This standardization guarantees interoperability between different implementations. Successive updates have introduced support for predictive image coding, bursts, region annotation. Development continues.

The HEIF format adapts to the performance constraints of modern devices. But the transition from JPEG is gradual. Compatibility issues slow adoption, though the technical advantages are undeniable. Change takes time, and the outcome remains uncertain.

Rust

In 2006, Graydon Hoare was working at Mozilla when he launched a personal project in systems programming. This project, named Rust in reference to rust fungi and the English word *robust*, remained discreet until 2010. That year, Mozilla recognized the language's potential and decided to support its development.

The first pre-alpha version of the compiler was released in January 2012. From the start, Rust tackled a contradiction that had plagued developers for decades: how to achieve fine-grained control over hardware resources while guaranteeing memory safety? Existing languages forced a choice between performance and safety. Rust refused this compromise.

Rust's type system relies on two original concepts: *ownership* and *borrowing*. The principle is simple: only one entity can own a resource at any given time. This rule, verified at compile time, eliminates an entire category of bugs in one fell swoop. Concurrency problems, memory leaks, and invalid accesses disappear before the program executes.

Microsoft revealed a striking figure: 70% of security vulnerabilities fixed in its products originated from memory management errors. Mozilla observed the same thing with Firefox. These statistics explain why Rust found its audience. The language has actually earned the title of "most loved language" in the Stack Overflow survey every year since 2016.

The ecosystem has been structured around Cargo, the official package manager. This tool radically simplifies dependency management and project building. The community has developed thousands of libraries, called *crates*, which cover just about every imaginable need. This richness has facilitated the language's adoption in industry.

Dropbox was among the first to bet on Rust for its critical infrastructure. Facebook, Amazon, and Cloudflare followed. Microsoft is exploring its use to strengthen the security of its system components. Mozilla has rewritten certain parts of Firefox in Rust, demonstrating that the language could be integrated into monumental projects already in production.

Concurrency management constitutes another strength of Rust. The compiler detects data races before the first execution. It's impossible to launch a program where multiple threads access shared data simultaneously without appropriate synchronization. This static verification saves hours of nightmarish debugging.

The language evolves with remarkable discipline. The designers introduce advanced features without ever breaking compatibility with existing

code. The development process relies on specialized working groups that debate each proposal at length. This governance produces carefully considered decisions.

Rust has established itself in various domains. It can be found in operating systems, web browsers, development tools, and distributed applications. The Internet of Things is adopting the language for its security guarantees and efficiency on limited hardware. Critical embedded systems benefit from a language that combines C's performance with protections absent from traditional low-level languages.

The Rust language proves that reconciling security and performance can be done without sacrificing productivity. Other language projects are inspired by its innovations. Secure development practices are evolving under its influence. Companies are offering training, consulting, and specialized services, signaling the commercial maturity achieved.

Rust's sophisticated type system represents a conceptual breakthrough. It demonstrates that a compiler can verify complex properties about code and guarantee their enforcement at runtime. This approach opens new perspectives in language design. Security is no longer a contradictory objective to performance.

New features appear regularly, always respecting the founding principles. Industrial adoption is accelerating. The community remains dynamic, and the language is becoming a lasting part of the computing landscape.

For decades, developers accepted memory bugs and associated security flaws as inevitable. But Rust has shattered this fatalism. It provides concrete solutions to age-old problems while maintaining the performance of established languages. This dual technical and community victory shows that development tools can be rethought to meet contemporary requirements for security and quality.

ChromeOS

In 2009, Google launched development of an operating system that broke with established conventions: ChromeOS, designed to function solely with web applications. The context explains this bold move. Netbooks, those small, affordable laptops oriented toward the Internet, were attracting more and more users. The company chose an unusual method to understand their needs: rather than commissioning traditional studies, engi-

neers distributed 200 ChromeOS machines to Google employees. Matthew Papakipos, who was leading the project technically, installed three devices at home. He observed that sessions were brief: a quick search, a short message sent, then the device was closed.

On July 7, 2009, Google officially announced ChromeOS. In November, the source code became accessible under the name Chromium OS. This duality persists: ChromeOS refers to the commercial product distributed only on specific hardware through partners, while Chromium OS remains the open source project that any developer can compile and modify. The streamlined interface draws inspiration from the Chrome browser, reflecting the system's primary purpose: accessing the web, quickly and simply.

On November 19, 2009, Sundar Pichai, Google's vice president overseeing Chrome, presented a preliminary version at a press conference. The demonstration showed a desktop that resembled the Chrome browser, enhanced with application tabs taking up less space and that could be pinned for easier access. Boot time then reached seven seconds. The teams set about reducing it further. Chris Kenyon, vice president of OEM services at Canonical, which publishes Ubuntu, announced a technical contribution under contract. Google wanted to leverage existing components from the open source community rather than reinventing everything.

An interface adapted to touch tablets was presented in January 2010. The design incorporated a virtual keyboard and larger icons, placed above the tabs. In March 2010, Google envisioned two versions: a consumer edition and an enterprise version.

The Cr-48 prototype was unveiled in December 2010. This test machine, produced in 60,000 units, introduced a curious innovation: the Caps Lock key disappeared in favor of a dedicated search key. Criticism arose over the limitations in offline mode. Google responded by demonstrating a version of applications working without connection, such as Google Docs, and announced a 3G plan including 100 MB of free monthly data through Verizon.

ChromeOS architecture is deployed on three levels: firmware, system services, and the browser with its window manager. The firmware accelerates boot time by avoiding detection of obsolete peripherals. Security constitutes a constant concern, with verification at each step of the boot process and integrated restoration capabilities.

Print management illustrates Google's approach. The Google Cloud Print service allows printing from any device on any printer, escaping the usual complexity of print drivers. The system integrates a media player capable of playing MP3 and JPEG files offline.

Early users of the Cr-48 delivered mixed feedback. Kurt Bakke from

Conceivably Tech highlighted its rapid adoption as a family device for quick searches and social media browsing. But performance with Flash disappointed, and the absence of offline features drew criticism.

Hardware integration was achieved through partnerships with Acer, Adobe, Asus, Intel, Samsung, and Dell. In July 2010, Google's CEO, Eric Schmidt, dismissed the idea of a Google-branded ChromeOS netbook. He preferred to rely on the experience of PC manufacturers accustomed to working with Microsoft.

ChromeOS's business model differs from traditional systems. Schmidt acknowledged that software compatibility would be more limited than Windows, but the free license allowed manufacturers to offer competitive devices, between 300*and*400. Glyn Moody of *Linux Journal* even anticipated free distribution of hardware to develop Google's advertising model.

ChromeOS adoption accelerated from 2015 onward, especially in American schools. IDC reported 4.6 million Chromebooks sold in 2014, twice as many as in 2013. Gartner projected 14.4 million units in 2017. The gradual integration of Android applications and the ability to run Windows applications through virtualization solutions expanded the system's capabilities.

The performance of Chromebooks based on Intel processors demonstrated the importance of hardware optimization. Comparative tests revealed gains of 50% and battery life 58% superior to competing ARM solutions. These results testified to engineers' continuous investment in improving system efficiency.

ChromeOS offers a web-centric approach that leverages cloud computing. Its simplicity, security, and automatic update model influence the design of modern operating systems today. This operating system is far from having replaced existing systems, but it has charted a different path, proving that a lightweight and connected alternative could find its place in a market dominated by established giants.

NVMe

SSD drives began to spread in the early 2010s, but encountered an unexpected problem. These new storage media exceeded the performance of mechanical hard drives, yet they remained constrained by communication protocols that were not designed for them. SATA and SAS interfaces, conceived for spinning disk drives, imposed their limitations on memories

capable of much better performance.

A working group bringing together major industry players decided in 2011 to tackle the problem head-on. Rather than trying to adapt existing solutions, these engineers started from scratch. They published the first specification of NVMe (Non-Volatile Memory Express), version 1.0. The guiding principle was simple: create a native interface for non-volatile memories that directly exploits the PCI Express bus and communicates naturally with modern processors.

The proposal disrupted established practices. Where SATA manages only 32 commands in a single queue, and SAS 256, NVMe creates up to 64,000 queues, each capable of containing 64,000 commands. This multi-queue architecture aligns with the multi-core processors that now equip machines. Each application can have its own queues, without interfering with others.

Version 1.1 arrived in October 2012 with its share of improvements. The protocol integrated multi-path access management and resource sharing, essential functions in professional environments. Autonomous power management mechanisms also made their appearance, anticipating the needs of laptops and mobile devices.

In 2014, the creation of NVM Express, Inc. structured the standard's governance. This organization brought together over a hundred companies and accelerated development. Version 1.2 was released in November with notable improvements: advanced namespace management, memory buffer optimization, and the ability to update firmware without interruption.

Physical formats multiplied to adapt to different uses. PCIe expansion cards equipped servers and workstations. M.2 modules found their place in laptops and compact machines. U.2 drives, in 2.5-inch format, installed in traditional storage bays. This diversity facilitated NVMe adoption across all market segments.

The protocol integrated security mechanisms compliant with Trusted Computing Group specifications. Data encryption at rest and secure erasure addressed growing confidentiality requirements. These functions are essential in a context where data protection has become a priority.

The year 2015 brought the NVMe-MI (Management Interface) specification. This extension standardized out-of-band device management, unifying configuration, monitoring, and maintenance commands. System administrators gained simplicity and consistency in managing their storage infrastructures.

The turning point of 2016 saw the birth of NVMe over Fabrics. This evolution extended the protocol to storage networks, whether using Ethernet, Fibre Channel, or InfiniBand. NVMe thus moved beyond direct

connections to integrate into complex network architectures, without losing its advantages in terms of performance and latency.

Version 1.3, published the same year, enriched the standard with functions intended for mobile devices. Power management optimization and integration of mechanisms adapted to embedded systems made NVMe the first truly universal storage interface. From phone to data center, a single protocol now covers all needs.

The numbers speak for themselves. NVMe cuts in half the number of processor instructions required for input-output processing compared to SATA or SAS. This efficiency translates into reduced latency and higher throughput. NVMe devices exceed one million operations per second, where previous interfaces plateaued well below.

Software support followed. Windows, Linux, and MacOS integrated native drivers. Virtualization environments like VMware ESXi adopted the protocol, enabling its use in virtualized infrastructures that form the core of modern data centers.

NVMe's success stems from a comprehensive approach. By completely rethinking the interface between processors and non-volatile memories, the protocol eliminates bottlenecks inherited from old architectures. This redesign fully exploits the performance of new memory technologies, without compromise linked to obsolete constraints.

Within a few years, NVMe established itself as an essential standard, supported by a rich ecosystem of products and solutions. Its continuous evolution guarantees its relevance for future storage technologies, confirming its central place in the architecture of contemporary computer systems.

Dart

In October 2011, during the GOTO conference in Aarhus, Denmark, Lars Bak and Kasper Lund introduced a new programming language developed at Google to the public: Dart. The stated ambition was clear: to offer a credible alternative to JavaScript for web development, a language that would escape the flaws and limitations that developers criticized in the venerable JavaScript.

The two Google engineers designed Dart with one idea in mind: to unify software development across different platforms without sacrificing performance. The language's syntax deliberately borrows from established

languages like Java, C, and JavaScript, to avoid disorienting developers who would make the transition. This calculated familiarity aims to facilitate adoption: no need to relearn everything from scratch.

On November 14, 2013, Google released version 1.0 of Dart. This was the first stable release, the one that had to convince the community that Dart could indeed replace JavaScript in web applications. The following year, in July 2014, Ecma International validated the language during its 107$^{\text{th}}$ General Assembly and created the ECMA-408 standard. This official recognition conferred on Dart a legitimacy that a simple corporate project would not have had.

But the beginnings were not as glorious as hoped. Google harbored the ambition to integrate a Dart virtual machine directly into Chrome to boost application performance. The idea caused concern. Web developers feared a fragmentation of the web, where each browser would impose its own virtual machine. Faced with this resistance, Google backed down. In 2015, with version 1.9, the project for native browser integration was abandoned.

The story of Dart could have ended there, relegated to yet another unremarkable web language. But in 2017, Google launched Flutter, a framework for creating user interfaces that work on mobile, web, and desktop from the same source code. And Flutter chose Dart as its primary language. This alliance gave new life to the language, which finally found its true purpose: to serve as the foundation for cross-platform development.

August 2018 marked a turning point with the arrival of Dart 2.0. This version restructured the language, notably through the introduction of a much stricter type system. The code gained in safety and readability, two essential qualities for complex applications. The compiler was also overhauled to produce performant native code, enabling Flutter applications to achieve performance comparable to solutions developed specifically for iOS or Android.

Dart offers two distinct compilation modes. Ahead-of-Time (AOT) compilation transforms code into optimized JavaScript for the web, or into native machine code for mobile applications. Just-in-Time (JIT) compilation facilitates more fluid development through hot code reloading: you modify the code, and the application updates instantly without a complete restart. This flexibility considerably accelerates the development cycle.

The language adopts a resolutely object-oriented approach: everything is an object, including numbers and functions. Classes, interfaces, mixins, generics... Dart incorporates the familiar concepts of modern languages. Automatic memory management frees the developer from the tedious tasks of manual resource allocation and deallocation.

Flutter's success pulled Dart upward. Version 2.7 introduced extension methods, which allow adding new functionality to existing types without modifying their source code. This technique proved valuable for enriching user interfaces developed with Flutter.

The ecosystem gradually expanded. The Dart SDK integrates a dart2js compiler to generate JavaScript, a static analyzer that detects errors before execution, a code formatter, and a package manager. These tools support the developer throughout the entire creation process.

Flutter's approach distinguishes Dart from other cross-platform solutions. Instead of compiling to native iOS or Android code, Flutter directly draws each pixel on screen through its own rendering engine. This strategy guarantees an identical user experience across all platforms, without depending on the native graphical components of each system.

Dart's virtual machine was designed to make the most of the language. The garbage collector manages memory efficiently, while the compiler produces code optimized for each target platform, enabling Dart applications to achieve high performance while maintaining a single codebase.

Companies of all sizes use Dart and Flutter to develop their applications. The ability to reuse code between mobile, web, and desktop represents considerable savings in time and money. Write once, deploy everywhere, with performance delivered.

Dart's future remains closely tied to that of Flutter. Google continues to invest in both technologies, regularly releasing updates that enrich functionality and improve performance. The language evolves to meet the needs of modern development, with a constant objective: to make developers more productive.

Started as an alternative to JavaScript, Dart ultimately found its way in cross-platform development. Its association with Flutter gave it a second life and a true playground.

Kotlin

In 2010, JetBrains, a Prague-based company specializing in development tools, embarked on creating a new programming language. The objective was to design an alternative to Java that would be more concise and safer, while maintaining compatibility with the Java Virtual Machine. This compatibility is no trivial detail, as it ensures developers can gradually adopt the new language without rewriting their existing code.

The project emerged from the shadows in July 2011 under the name "Project Kotlin". A few months later, in February 2012, JetBrains opened the source code under the Apache 2.0 license. This choice of openness was not insignificant: it reflected a desire to involve the developer community from the project's earliest stages. Four years later, on February 15, 2016, version 1.0 arrived. This first stable version marked Kotlin's true launch, as it began to attract attention with its streamlined syntax and pragmatic approach. One figure immediately stood out: Kotlin reduces the number of lines of code needed by an average of 40% compared to Java for accomplishing the same task.

The year 2017 changed everything. In May, Google announced its official support for Android development, making Kotlin a credible alternative to Java. This recognition propelled the language to the heart of mobile development. But Kotlin didn't stop there. In April, it reached a new milestone by becoming multiplatform through Kotlin/Native and stabilized support for JavaScript. Six months later, version 1.2 introduced the ability to share code between backend, web frontend, and Android mobile applications. This capability addressed a growing need: reusing code across different platforms without rewriting everything.

In October 2018, Google and JetBrains jointly created the Kotlin Foundation. This structure guarantees the language's independence and ensures transparent governance. A year later, in 2019, Google went further by declaring Kotlin as the preferred language for Android. Concretely, this means that new Android features and APIs arrive first in Kotlin. The language received the "Breakout Project of the Year" award at the O'Reilly Open Source Software convention, recognition of its meteoric rise.

What appeals in Kotlin is first its approach to handling nullability. The type system directly integrates this management, drastically reducing errors related to null references that plague Java developers. Lambda expressions and extension functions enrich the possibilities for functional programming. As for data classes, they simplify the creation of objects intended for storing information.

The language evolved rapidly. Version 1.1 brought coroutines that facilitate asynchronous programming. Version 1.3 introduced contracts and *inline* classes, improving code performance and safety. The compiler transforms source code into JVM *bytecode*, making its execution possible on all platforms supporting Java.

The numbers speak for themselves. In 2020, more than 4.7 million developers used Kotlin, with 65% in production. For 56% of them, it had become their primary language. Giants like Google, Uber, Atlassian, Pinterest, and Square integrated it into their production code. Kotlin is

the 13th most popular programming language, surpassing Scala and Swift. GitHub's 2018 report already ranked it as the fastest-growing language. Stack Overflow places it fourth in the ranking of professional developers' favorite languages.

The community grew. In 2020, more than 100 people worked on the language at JetBrains, supported by over 350 independent contributors participating in its evolution. This collective enthusiasm fuels the ongoing development of the language and its ecosystem.

Kotlin reduces the boilerplate code that weighs down Java, prevents common errors, and simplifies complex tasks. Its multiplatform approach meets current software development expectations.

WebRTC

WebRTC is a technology born from the convergence of web browsers and real-time communication. Its roots run deep through several decades of Internet protocol evolution.

Back to 1996. The publication of the Real-time Transport Protocol (RTP) in RFC 1889 laid the foundations for what would become WebRTC. Ron Frederick, one of the protocol's authors, had begun experimenting with the Sun VideoPix video capture card as early as October 1992. His goal was to create an IP multicast-based network videoconferencing tool. The project was called "nv" (Network Video tool) and was inspired by "vat", an audio conferencing tool developed by LBL that used a lightweight session protocol to manage participants.

Video compression represented a considerable technical challenge. Frederick wanted to achieve an acceptable stream within bandwidth limited to 128 kbps, the capacity of a standard ISDN line. This required a compression factor of approximately 20 for the targeted image size and frame rate. Frederick succeeded in developing an innovative technique that became the subject of a patent (US5485212A) on software video compression for teleconferencing.

The first version of nv was released in November 1992. It broadcast certain IETF sessions worldwide. The impact was immediate: approximately 200 subnets in 15 countries could receive this broadcast, and between 50 and 100 people used nv during this period. The success was confirmed at various international workshops, notably the Australian NetWorkshop and the MultiG workshop on distributed virtual realities in Sweden.

Frederick released the source code publicly in February 1993. He then introduced a new wavelet-based compression system and added color video support. The network protocol used by nv and other Internet conferencing tools served as the basis for the RTP protocol that the IETF standardized. RTP became an essential standard, first published in RFCs 1889-1890, then revised in RFCs 3550-3551.

The evolution of nv continued with ports to different hardware platforms and video capture devices. NASA chose this tool to broadcast live coverage of space shuttle missions. In 1994, Frederick enhanced nv with compression algorithms developed by others, including hardware systems such as the CellB format from the SunVideo capture card.

This early history of real-time Internet communications foreshadows WebRTC. The challenges encountered by pioneers like Frederick established the foundations of modern protocols. IP multicast enabled efficient data distribution to multiple recipients, a different approach from the unicast architecture later adopted by WebRTC.

The shift from multicast to unicast can be explained by practical reasons. The complexity of implementing multicast in existing networks led to favoring simpler-to-deploy solutions. This evolution required the introduction of new architectures, such as SFUs (Selective Forwarding Units), to manage media stream distribution among participants.

WebRTC's history took a turn with the arrival of Serge Lachapelle. His passion for communication software dates back to the 1990s, when he developed a project to join video calls directly from a browser. This experience led him to co-found Marratech, a company specializing in group videoconferencing. Google acquired Marratech in 2007 and integrated valuable expertise into its team.

The development of Gmail's voice and video messaging represented a significant intermediate step. This achievement involved the complex integration of different components: audio under GIPs license, video from Vidyo, and networking via libjingle. Justin Uberti took on the technical challenge of making these systems with disparate APIs work together, which required extensive knowledge in networking, cryptography, and media.

The launch of the Chrome project at Google opened a new chapter. The ambition to enhance browser capabilities, combined with a desire to abandon the NPAPI API for security reasons, created a favorable context for WebRTC's emergence. Chrome's sandbox architecture, which isolates potentially dangerous operations in separate processes, influenced WebRTC's secure design.

WebRTC's birth addressed three main objectives. First, simplifying

the development of real-time communication experiences. Second, promoting open and unhindered human communication. Finally, ensuring security by default. To realize this vision, Google acquired and open-sourced several key technologies, notably those from On2 for video and Global IP Solutions for real-time communications.

WebRTC's standardization represented an unprecedented effort. It required collaboration between two standardization bodies (IETF and W3C) and hundreds of international experts. Harald Alvestrand, with his IETF experience, initiated the process. An informal meeting in Maastricht in 2010 brought together developers from Google, Cisco, Ericsson, Skype, Mozilla, and Linden Labs. The specification's foundations were laid.

WebRTC built upon the legacy of existing technologies. It avoided reinventing already proven solutions. The decision not to standardize signaling, handled by SIP and other protocols, illustrates this pragmatic approach. This strategy concentrated efforts on innovation rather than redefining established concepts.

WebRTC's use in cloud gaming demonstrates its versatility. Advances in artificial intelligence improve communication quality, notably through background noise suppression. This expansion of use cases testifies to the robustness and flexibility of WebRTC's architecture. International collaboration and commitment to open standards have created a platform that continues to innovate and adapt to new use cases.

WebSocket

In the early 2000s, web developers faced a recurring problem. The applications they were building-instant messaging systems, online games, collaborative dashboards-required constant exchanges between browsers and servers. However, the Web as it existed at the time couldn't really handle this type of dialogue. HTTP, the protocol governing Internet communications, operated on a rigid model: the client requests, the server responds. No room for fluid two-way conversation.

Engineers found workarounds. The most common, "HTTP polling", consisted of bombarding the server with regular requests to check if it had anything new to say. A variant, "long polling", kept the connection open until information arrived. These makeshift solutions worked, certainly, but generated considerable network traffic. Each HTTP request carried bulky headers-hundreds of bytes each time-sometimes to transport just a few bytes of useful information.

RFC 6455, published by the IETF in December 2011, put an end to this era of improvisation. WebSocket proposed a radically different approach. The protocol starts with a standard HTTP handshake, then switches to a permanent connection. Once established, this channel sends messages in both directions with minimal overhead-a few control bytes instead of the hundreds previously required. The protocol handles text and binary data indifferently, and integrates connection monitoring mechanisms via "ping" and "pong" messages.

Browsers adopted this innovation. Chrome, Firefox, and Safari deployed experimental implementations as early as 2010. Internet Explorer joined the movement in 2012. This rapid convergence among vendors testified to a genuine market need. Developers could finally create responsive web applications without resorting to complex and fragile workarounds.

The numbers speak for themselves. On a trading site displaying real-time stock prices, WebSocket reduces traffic by 417 bytes per update per client. With 4,835 simultaneous visitors, the bandwidth savings climb to 16 megabits per second. The impact is also measured in adoption statistics: 1.6% of the top million sites in 2018, 6.3% in 2020. This growth testifies to a technology that has reached maturity.

The ecosystem structured itself around the protocol. Socket.io and SockJS offer high-level APIs that simplify its use. Chat services, collaborative platforms, and financial applications made it their workhorse. Web analytics tools use it to collect real-time data on user behavior. WebSocket finds its place in the Internet of Things and microservices architectures, where its lightness and bidirectionality offer tangible advantages.

The technology doesn't escape misuse. Some developers neglect security best practices-Origin header verification, systematic TLS encryption. Malicious actors misuse the protocol for tracking or distributing unwanted software. These shadow areas remind us that no technical innovation is neutral.

HTTP/2, standardized in 2015, introduced the Server Push feature. Servers can now send resources to clients without waiting for a request. This capability doesn't replace WebSocket, however: it doesn't create a true bidirectional channel. Google implicitly acknowledged this by dropping support for Server Push in Chrome in 2020, confirming WebSocket's unique position in the web landscape.

Competing technologies emerged, such as Server-Sent Events. WebSocket nonetheless retains its relevance. The stabilization of its adoption rate since 2019 suggests it has found its place in the palette of available tools. Modern frameworks integrate it by default, developers master it, use cases multiply.

The protocol didn't revolutionize anything: it simply made possible what previously required acrobatics. This conceptual simplicity, combined with measurable operational efficiency, explains its success.

Go

In 2007, at Google, engineers faced a problem that had become commonplace in the company. Existing languages no longer met the needs of modern applications: intensive networking, multicore processors, distributed environments. Rob Pike, one of the designers of the future Go language, summed up this technical impasse with some frustration. The industry had numerous tools, certainly, but none truly fit the bill.

Cloud computing had just emerged from infancy, service-oriented architecture was finding its footing in large organizations, and continuous integration was beginning to transform development practices. The computing landscape already resembled today's. Go was not born in a laboratory cut off from the real world; it emerged at the heart of the constraints faced by development teams.

One striking detail in this story: among Go's creators is Ken Thompson, who had designed UNIX and the C language decades earlier. This lineage is far from anecdotal. It traces a direct line between two eras of computing, with Go embodying a contemporary reinterpretation of the principles that guided C's creation.

Java, which appeared in 1991, had established itself through its portability. But its proprietary status until 2006 created complications: the 2012 Oracle lawsuit against Google attests to this. As for C, born in 1972 to rewrite UNIX, it had spread for lack of alternatives; it was one of the few fast and cross-platform languages accessible to early programmers. Go, on the other hand, was designed from the outset for current needs.

The language stands out through its thoughtful design. Compilation times remain short, machine code is generated directly, strong typing imposes a discipline that makes programs readable and reliable. Comprehensive and free documentation accompanies the whole. In a development team, often large and with varied skills, these characteristics take on another dimension: collaboration becomes more fluid, build times are reduced, existing code is more easily deciphered.

With automatic memory management, developers no longer need to explicitly program memory allocation and deallocation, unlike in C or

C++. This automation improves productivity, certainly, but above all it strengthens security by reducing vulnerabilities related to memory handling errors.

Native concurrency support represents perhaps the most significant innovation. Applications can execute multiple tasks simultaneously rather than sequentially, thus exploiting multicore processors. For applications distributed across multiple servers or microservice architectures, this functionality proves invaluable.

Deploying Go applications remains simple. Unlike Java, which requires local interpreters or virtual machines, Go programs compile into standalone binaries. The Get tool downloads and installs an application via a single command. This approach perfectly matches DevOps practices that prioritize consistency in development pipelines and simplicity of updates.

In the field of data science, Go demonstrates its versatility. It easily integrates algorithms written in Python into microservice applications. The resulting performance often eliminates the need for optimization in C or C++, low-level languages known for their complexity.

The developer community around Go has continued to grow. In 2016, more than 700 programmers contributed to its development. The TIOBE programming language index illustrates this rise: Go moved from 55^{th} to 10^{th} place between 2016 and 2017.

Many open source platforms have adopted Go. Cloud Foundry chose it for components like `gorouter`, which manages connectivity for microservice applications. Docker, which transformed application deployment, is written entirely in this language. Docker developers opted for Go in 2013, attracted by its ease of installation, testing, and adoption. InfluxDB, a high-performance storage system, leverages the language's efficiency and reliability.

Leading technology companies have integrated Go into their infrastructures. Dropbox migrated a significant portion of its infrastructure from Python to Go to improve its scalability. Netflix uses Go to create microservices requiring performance and scalability. Uber developed its real-time geolocation service in Go, which displays the best performance among all its services.

Go's success is explained by its alignment with modern software development needs. Its design takes into account cloud environments, distributed processing, and microservice architectures. Ease of use, combined with high performance and facilitated maintenance, makes it a relevant choice for companies modernizing their applications.

Go's emergence marks the evolution of programming languages toward

solutions adapted to cloud computing and distributed architectures. Contemporary software development needs have led to rethinking programming tools, creating a language that meets the requirements of current computing while maintaining an elegant and pragmatic approach.

Julia

For the past twenty years, researchers using scientific computing have lived with a constraint that slows down their work: they must choose between a language that's easy to use for testing ideas and a language that runs fast for producing results. When prototyping, they use Python or MATLAB. But as soon as serious calculations are needed, they start over in C++ or Fortran. This constant back-and-forth between two languages wastes time and complicates projects unnecessarily.

This is the problem that Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral Shah decided to tackle in 2009 at MIT. They wanted to create a language that would combine the simplicity of dynamic languages with the speed of compiled languages. A bold bet, since these two qualities seem inherently contradictory.

The first version of Julia was released in 2013. From the beginning, the language targeted scientific computing with an architecture designed for operations on multidimensional arrays, in the manner of Fortran. The engine relies on LLVM, which generates machine code adapted to different hardware architectures. Julia compiles code on the fly, as it executes. This *just-in-time* compilation reconciles two worlds previously thought separate: the responsiveness of an interpreted language and the speed of a compiled program.

Julia's originality lies primarily in its multiple dispatch system. Unlike traditional object-oriented languages where a method depends on the type of a single object, Julia selects the right function based on the types of all arguments. This mechanism makes code much more flexible and facilitates the composition of libraries that weren't designed together. The language also integrates powerful macros, native parallel computing, and seamless interaction with existing C or Fortran code.

The community took shape. In 2014, the first JuliaCon took place, quickly becoming an annual event. Three years later, the registry already contained over 1,000 packages. Major companies began taking interest: Aviva and the Federal Reserve Bank of New York in finance, Pfizer, Moderna, and AstraZeneca in pharmaceuticals, ASML for semiconductors.

The performance is impressive. Benchmarks show that certain mathematical functions written in pure Julia run faster than their optimized equivalents in C or Fortran. The implementation of the inverse error function beats MATLAB's by a factor of 3 to 4, and SciPy's by 2 to 3. This speed comes from metaprogramming and code generation techniques that Julia exploits naturally, where a low-level language would require contortions.

The name of the Jupyter environment recalls the connection with Julia (along with Python and R). These notebooks, which mix code, results, graphics, and text, fit well with the needs of teaching and reproducible research. MIT and other universities now use it for their numerical computing courses.

Julia runs on various architectures: ARM processors and GPUs. The integrated package manager simplifies dependency management and ensures environment reproducibility. Tools automate the creation of bridges to C++ libraries, providing access to a considerable software heritage.

In 2022, Julia tackled large-scale scientific projects. The Climate Modeling Alliance uses it for climate modeling, the Celeste project for astronomical image processing, and high-energy physics teams for data analysis. These applications demonstrate that the language performs well on complex and massive calculations without sacrificing code readability.

The development follows a classic open-source approach, with code on GitHub. Between 2009 and 2017, over 40,000 commits accumulated. Contributions go through strict review that maintains quality. Comprehensive documentation and online tutorials help newcomers get into the language without too much difficulty.

The initial design choices proved sound for following the evolution of scientific computing. The team continues to improve compilation times, automatically optimize code, and support new architectures. The common thread remains the same: a single language that combines ease of use and performance.

Julia shows that programming languages can be rethought to transform scientific practices. By elegantly solving the two-language problem, it changed what we expect from a computational tool: productivity without compromise on speed.

TypeScript

When Microsoft launched TypeScript in 2010, the web development landscape was undergoing rapid transformation. JavaScript, once relegated to simple browser interactions, had become an essential language. Execution engines were gaining performance, single-page web applications were proliferating, and Node.js was paving the way for server-side JavaScript. This expansion brought with it a problem: maintaining large JavaScript codebases was a real headache.

Anders Hejlsberg, the architect behind C#, led the project with a clear ambition. The goal was not to replace JavaScript, but to enhance it. TypeScript presents itself as a syntactic superset of JavaScript: any valid JavaScript code remains valid TypeScript code. The core idea was to add an optional static type system. Developers can annotate their code, and these annotations disappear at compilation. The generated JavaScript retains no trace of the typing.

This technical decision reflects a keen understanding of real needs. The type system adopts a structural approach rather than a nominal one, which better suits JavaScript developers' habits. In this ecosystem, objects often emerge directly, without going through formal classes. TypeScript embraces this practice instead of fighting it.

One design choice deserves attention. TypeScript's creators deliberately chose not to guarantee complete type safety. They preferred to facilitate gradual adoption and maintain compatibility with existing libraries. This flexibility allows teams to introduce TypeScript into their projects without rewriting everything at once.

Steve Ballmer, Microsoft's former CEO, had compared open source licenses to "a cancer that attaches itself to everything it touches." He denounced the lack of accountability in free software. Yet the TypeScript team knew they would never win over the JavaScript community without embracing open source. In 2012, Microsoft released TypeScript under the Apache-2.0 license, first on CodePlex, then on GitHub in 2014.

Adoption grew gradually. In 2025, the GitHub repository has garnered over 107,000 stars and ranks among the platform's twenty most popular projects. The NPM package records over 100 million weekly downloads. More than 61,000 other packages depend on it. These figures reflect established trust.

The language introduces structural concepts: interfaces, classes, modules. Its type inference system reduces required annotations while preserving the benefits of static typing. Development tools are enriched with autocompletion and code navigation. The compiler, written in TypeScript,

demonstrates the language's capacity to handle complex projects. It analyzes source code, verifies types, and generates standard JavaScript while preserving program semantics.

The community has built a rich ecosystem around TypeScript. The DefinitelyTyped repository gathers thousands of type definition files for traditional JavaScript libraries. These declarations enable the use of existing libraries with the advantages of static typing. Successive versions bring advanced features: generic types, conditional types, utility types. These additions allow the expression of complex constraints and reinforce code safety. A new version is released every two months, ensuring constant evolution.

TypeScript has established more rigorous practices in the JavaScript ecosystem. Certain concepts have enriched JavaScript itself, as evidenced by new ECMAScript versions. This success testifies to Microsoft's transformation toward a less closed approach to software development.

Docker

About fifteen years ago, a young startup named dotCloud made a decision that would upend the IT industry. By releasing the code of its containerization engine to the public, the company offered Docker to the world. The way we build, distribute, and run applications would never be the same again. Yet the history of containers doesn't begin with Docker, far from it.

Let's go back to 1979. UNIX introduced `chroot`, a mechanism that allowed changing the root directory of a process. A modest building block that laid the groundwork for an idea: isolating application storage from one another. FreeBSD took a leap forward in 2000 with its *jails*, providing true system-level virtualization. Linux-VServer followed suit the next year, followed by OpenVZ in 2005. All these technologies explored a similar path: partitioning system resources to compartmentalize applications.

But none really took off. Too complex, too rigid, they remained confined to niche uses. LXC arrived in 2008 and for the first time, there was a complete container manager that relied on the standard Linux kernel, without hacks or risky modifications. Lightweight virtualization was finally accessible, at least in theory.

dotCloud used LXC to run its cloud platform. Its engineers had cobbled together an in-house tool in Python, called "dc", which was used to

manipulate LXC containers. Create, start, stop, expose network ports: the basic functions were there. Except the architecture was problematic. Concurrent access to container metadata created lock conflicts. The system held together, but fragility lurked.

In 2013, Solomon Hykes and his team took the plunge: they released their engine under the name Docker. Docker didn't just make containers usable, it made them simple. The standardized image format packages an application with everything it needs. Read-only layered stacks allow sharing common parts between images. The Docker Hub registry made sharing images a breeze.

The architecture evolved quickly. Docker broke free from LXC as early as 2014 by developing `libcontainer`, its own library for communicating with the Linux kernel. No external dependencies, more control, more stability. The technical foundations rested on `cgroups` that limit resources, `namespaces` that isolate processes, and storage drivers like AUFS or OverlayFS that manage image layers with finesse.

The modular design took shape. The `dockerd` daemon exposes the API and orchestrates everything. `containerd` handles container execution. `runc` standardizes their creation. This architecture built from blocks allows replacing what doesn't fit and adapting the tool to specific needs.

Docker's success raised a question: how to guarantee interoperability? The Linux Foundation launched the Open Container Initiative in 2015 to establish open standards. Docker contributed by providing the specifications of its image format and `runc` as a reference. That same year saw the birth of the Cloud Native Computing Foundation, which would host Kubernetes, Prometheus, and `containerd`. The ecosystem structured itself, standards took hold.

In April 2015, Docker raised $95 million in a Series D round led by Insight Venture Partners. Prestigious names joined the venture: Goldman Sachs, Coatue, Northern Trust. Six months later, an additional $18 million came in. The valuation exceeded one billion dollars. Docker became a "unicorn", all the heavyweights of Silicon Valley were betting on this technology that promised to revolutionize software deployment. The company went through successive funding rounds until reaching nearly $280 million raised. In 2017, a new round brought in $75 million more, pushing the valuation to $1.3 billion.

Docker disrupted practices. A containerized application behaves the same way everywhere: on the developer's machine, on test servers, in production. Resources consumed remain reasonable, far from the heaviness of traditional virtual machines. Server density improved as a natural consequence.

The company also developed Docker Swarm, its native orchestrator. The idea seemed sound: managing entire fleets of containers distributed across multiple machines while maintaining the simplicity of the Docker API. A few commands were enough to deploy a cluster. No steep learning curve, just a natural extension of what developers already knew. In 2016, Docker integrated Swarm directly into the engine with "Swarm Mode". It was in the box, ready to use.

Except Google had released Kubernetes in 2014. The fruit of a decade of experience with Borg, its internal orchestration system, Kubernetes grew at breakneck speed. The open source community was on fire. The three major cloud providers adopted Kubernetes: Google of course, but also Amazon with EKS and Microsoft with AKS. VMware and Red Hat rallied to it. By 2017, the battle was already lost for Docker Swarm. Docker's leadership acknowledged this by announcing that the platform would support Kubernetes. A barely disguised admission of defeat.

Tensions mounted internally. Three CEOs in a single year. The company was flying blind. It had two distinct activities that operated according to incompatible logics: on one side Docker Desktop and Docker Hub, beloved by developers but difficult to monetize; on the other Docker Enterprise, a suite of tools for large enterprises with integrated orchestration, lucrative contracts, but growing complexity.

In November 2019, Mirantis, a cloud specialist looking for direction after the waning of OpenStack, purchased Docker's entire enterprise business. The transaction included Docker Enterprise Engine, Docker Trusted Registry, Docker Unified Control Plane, the CLI, and 750 large enterprise customers including a third of the Fortune 100. Three hundred employees joined Mirantis. The amount wasn't disclosed, but it was paltry compared to the $280 million raised. Observers spoke of a fire sale.

The same day, Docker announced $35 million in funding from Benchmark Capital and Insight Partners. Scott Johnston, product director since 2014, became the third CEO of the year. The company restructured, refocused on developers, keeping Docker Desktop and Docker Hub. The team went from several hundred people to about sixty. Many saw it as the end of an era. Commentators declared Docker dead.

Docker and Mirantis agreed to collaborate on open source development. Mirantis committed to supporting Docker Swarm for at least two years, but the message was clear: Kubernetes would be the main orchestrator. The acquisition of Docker Enterprise assets allowed Mirantis to position itself against giants like VMware and Red Hat. Two years after the acquisition, Mirantis announced annual revenue exceeding $100 million, half of which came from assets purchased from Docker. The graft

had taken.

For its part, Docker rose from the ashes. Focused on its core business, the company found a growth trajectory again. Annual recurring revenue climbed from $6 million in 2020 to $50 million in 2021, then to over $135 million in 2023. In March 2022, Docker raised $105 million in a Series C round led by Bain Capital Ventures, with a valuation of $2.1 billion. The workforce doubled to reach 150 people. In 2021, Docker Business was born, an offering for large enterprises, and Docker Desktop's license terms changed. The refocusing bet paid off.

What's striking about Docker's story is less the pure technical innovation than the democratization of an idea. Containers existed, but nobody really used them. Docker made them accessible, almost obvious. This simplification paved the way for microservices architectures, accelerated the DevOps movement, and transformed how we think about software deployment.

Components separated, giving users the choice of elements that suited them. New tools constantly emerge to optimize image building, strengthen security, and orchestrate containers. Docker's legacy lies in the open standards it established, in the foundations it laid for the industry, and in its ability to reinvent itself after nearly disappearing. Docker Inc. still exists, smaller but viable, while Docker technology permeates the entire modern cloud ecosystem.

SHA-3

In the late 1970s, cryptographic hash functions began to take shape. These algorithms transform any data into a fixed-size digital fingerprint, used to verify data integrity and create digital signatures. In 1976, Diffie and Hellman published their seminal paper on public-key cryptography; they identified the need for a one-way hash function to construct digital signature schemes.

Between 1978 and 1979, Rabin, Yuval, and Merkle laid the theoretical foundations. Rabin proposed a design based on DES encryption with a 64-bit output. Yuval demonstrated that collisions for an n-bit hash function could be found in $2^{n/2}$ time using the birthday paradox. Merkle introduced the requirements for collision resistance, preimage resistance, and second preimage resistance, which remain the fundamental properties expected of a cryptographic hash function.

The 1990s saw the emergence of MD4 and MD5, designed by Ron Rivest, which became widely used standards. SHA-1, published by the NSA in 1995, established itself as MD5's successor with a 160-bit output. SHA-2 arrived in 2001 with variants producing digests from 224 to 512 bits. These algorithms are based on the Merkle-Damgård construction: the input message is divided into fixed-size blocks to which a compression function is iteratively applied.

In 2004, Xiaoyun Wang's team revolutionized the field by refining differential cryptanalysis to the point of making collision discovery for MD5 trivial. This work also significantly reduced SHA-1's security. In 2008, a team managed to create a rogue CA certificate by exploiting MD5 weaknesses. The cryptographic community grew concerned about the robustness of existing hash functions. SHA-2 uses construction principles similar to SHA-1, raising questions about its future resistance.

NIST launched a public competition in 2007 to select a new algorithm that would become SHA-3. The specifications required that candidates support outputs of 224, 256, 384, and 512 bits to ensure compatibility with SHA-2. Submissions had to be royalty-free and usable worldwide without restrictions.

64 candidates were submitted in 2008, with 51 selected for the first round. By mid-2009, approximately half of the algorithms had been broken. This carnage demonstrated the difficulty of designing a secure and efficient hash function. In July 2009, NIST selected 14 candidates for the second round: Blake, Blue Midnight Wish, CubeHash, ECHO, Fugue, Grøstl, Hamsi, JH, Keccak, Luffa, Shabal, SHAvite-3, SIMD, and Skein.

The Keccak algorithm, developed by Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche, stood out for its innovative architecture based on the sponge construction. Unlike the Merkle-Damgård construction, this approach absorbs the message block by block with a portion called the "rate" of 1,152 bits, while the remaining 448 bits serve as a security parameter. The internal state forms a $5 \times 5 \times 64$-bit cube totaling 1,600 bits.

In October 2012, NIST designated Keccak as the winner of the SHA-3 competition. The decision was based on its excellent hardware performance, comfortable security margin, and architecture distinct from other SHA algorithms. Standardization was completed in August 2015 with the publication of FIPS 202. SHA-3 includes four conventional hash functions (SHA3-224, SHA3-256, SHA3-384, and SHA3-512) as well as two extendable-output functions SHAKE128 and SHAKE256.

The first implementations of SHA-3 revealed interesting characteristics. On FPGA, the algorithm requires more hardware resources than

SHA-1 but offers better cryptographic resistance. Performance tests indicate that SHA-3 is approximately 5.5 times slower than SHA-1 due to its increased complexity. For password cracking by brute force, a 6-character password requires a few minutes with SHA-1 versus over an hour with SHA-3.

The introduction of SHA-3 marks an evolution in the history of cryptographic hash functions. Its sponge construction breaks with the traditional approach and inspires new modes of use. The extendable-output functions SHAKE bring innovation: the digest size can be adapted to specific application needs. SHA-3 complements the SHA-2 family by providing architectural diversity that strengthens overall resilience against future cryptanalytic advances.

To date, SHA-3 has not suffered any significant attack undermining its security. Research continues to explore the possibilities offered by the sponge construction, particularly for designing authenticated cryptographic primitives. The success of the SHA-3 competition demonstrates the effectiveness of an open and collaborative approach to developing robust cryptographic standards. This methodology now serves as a model for other standardization initiatives in cryptography.

Spark

In 2009, a group of researchers at the University of California, Berkeley began developing Spark. Their goal was to create a distributed computing system capable of surpassing MapReduce's limitations. The problem had become evident: modern data processing applications needed to constantly reuse the same datasets across multiple parallel operations, whether for machine learning or interactive analysis.

The project's central idea lies in an abstraction called RDD, for *Resilient Distributed Datasets*. These are collections of objects distributed across multiple machines, accessible only in read mode, but capable of rebuilding themselves if a partition were to disappear. This approach provides fault tolerance without duplicating data, elegantly solving reliability issues in distributed environments.

The first public version arrived in 2010. Spark stood out for its ability to keep data in memory between operations, whereas traditional systems systematically rewrote everything to disk. For iterative algorithms, those that process the same data over and over, the performance gain is spectacular.

The choice of Scala as the programming language proved decisive. This statically typed language, which runs on the Java virtual machine, offers expressiveness that simplifies writing distributed applications. Developers manipulate RDDs with functional constructs, making the code more readable.

The project joined the Apache Software Foundation ecosystem in 2013, thereby strengthening its adoption. Features multiplied and SparkSQL emerged to execute relational queries on distributed data. Spark Streaming arrived to process real-time streams.

The system's modular design allows progressive addition of specialized libraries. MLlib offers machine learning algorithms adapted to distributed environments. GraphX provides tools for processing large-scale graphs. These extensions enrich the ecosystem while preserving interface consistency.

The architecture integrates an optimizer named Catalyst, introduced in 2014. This component analyzes and improves SQL queries as well as data transformations. It exploits Scala's characteristics to generate high-performance code, merge operations, and reduce transfers between cluster nodes.

The unified programming interface represents a breakthrough. Developers combine different types of processing in the same application: SQL, streaming, machine learning. No more juggling between different specialized systems. This approach streamlines data pipeline complexity.

The community experienced exceptional growth. In 2015, more than 1,000 contributors participated in development, making Spark one of the most dynamic open source initiatives in data processing. Companies like Databricks, founded by the original creators, built commercial services on this technology.

The performance is impressive. For iterative algorithms, Spark can be up to 100 times faster than Hadoop MapReduce thanks to keeping data in memory. In the domain of SQL queries, it rivals specialized systems like Impala, while providing greater flexibility.

Banks use it for risk analysis, e-commerce sites for their recommendation systems, research centers to process their experimental data. This diversity of uses demonstrates the system's versatility.

In 2016, Spark became the Apache Foundation's most active project. Some organizations deployed clusters of more than 8,000 nodes. The introduction of DataFrames, inspired by R and Python, simplified usage for data scientists manipulating tabular data.

Spark's design influences the evolution of distributed systems. The programming model based on functional transformations and efficient

memory management inspires numerous subsequent projects. Its unified approach for different types of processing is an industry reference.

Development continues by integrating advances in optimization and accessibility. The project maintains its orientation toward ease of use while improving its technical capabilities. Abundant documentation and learning tools train a new generation of developers in distributed processing principles.

By offering a solution to the limitations of existing approaches, while maintaining a balance between power and ease of use, this project transforms how organizations process their data at scale. Its continued evolution reflects the ever-changing nature of the distributed data processing field.

Kubernetes

In 2006, Google faced a difficult equation. The company offered Gmail, YouTube, and its search engine for free to millions of users. How could they manage a colossal infrastructure without costs spiraling out of control? Engineers then developed Borg, an internal system capable of running thousands of applications across their fleet of machines. The name paid homage to an alien race from Star Trek, creatures connected by a collective consciousness. The analogy fit the project well. Borg optimized the use of hardware resources by orchestrating the execution of applications in containers, a technology that was only beginning to emerge. Google kept this treasure secret for years, aware of the technical advantage it represented.

In 2013, three engineers from the company – Craig McLuckie, Joe Beda, and Brendan Burns – had an insight. Docker had just made containers popular among developers, but the technology remained limited. It could only manage containers on a single machine. The experience gained with Borg opened the way to something more ambitious. Why not create a system capable of orchestrating containers at the scale of an entire data center?

They named their idea “Project Seven”, a reference to the character Seven of Nine from Star Trek, a former Borg drone. This nod established a link to the original system while marking a break toward greater openness. For that was where the real gamble lay. Making this project open source represented a risk. Urs Hölzle, Google’s chief technical officer, saw this container management expertise as a competitive advantage.

But the company understood that widespread adoption required an active community of external developers.

The project ultimately took the name Kubernetes, from the ancient Greek kubernētēs, which designates the pilot or helmsman of a ship. This choice continued the maritime theme of Docker. The system was to steer containers, as a helmsman guides his ship. The logo displays seven spokes, a discreet trace of the initial name "Project Seven".

The designers built Kubernetes drawing on lessons learned from Borg and its successor Omega. The architecture is based on clusters, with worker nodes on one side executing containerized applications, and control plane nodes on the other managing orchestration. The system uses labels to organize and manage containers flexibly, abandoning Borg's rigid sequential numbering.

Version 1.0 was unveiled in 2015 at the OSCON conference. From launch, Microsoft, Red Hat, and IBM provided their support. Google entrusted the project to the Cloud Native Computing Foundation (CNCF), a nonprofit organization hosted by the Linux Foundation. This transfer guaranteed the project's neutrality and collaborative development.

Kubernetes' technical capabilities address the requirements of modern applications. Automatic container replication, load balancing, service discovery, application health monitoring, automatic repair in case of failure. The system pools multiple machines to efficiently distribute workload.

Kubernetes' approach to APIs marks a notable evolution from Borg. Each object has three fundamental fields: metadata, desired state specification, and current status. This standardization simplified the development of tools and extensions by the community.

The management of "pods" brings a new dimension. A pod can contain multiple containers sharing resources. This design helps break down complex applications into smaller, better-isolated components without losing coordination between them.

Kubernetes' rise is part of a broader transformation in software development. The system accompanied the growth of cloud computing and microservices architectures. Companies can now deploy their applications consistently, on their own servers as well as on AWS, Microsoft Azure, or Google Cloud.

The impact on the computer industry proved massive. In 2020, Kubernetes ranked among the ten most popular open source projects on GitHub. According to the CNCF, it became the second largest open source project after Linux. 71% of Fortune 100 companies use it.

This story shows how a technology developed to solve Google's internal problems transformed into an industry standard through open source. It

underscores the role of developer communities in large-scale technology adoption. Kubernetes has transformed how companies develop, deploy, and manage their applications, giving them the agility needed to innovate quickly without sacrificing service stability.

Swift

In 2014, the mobile application market was experiencing spectacular growth. Mobile device sales were approaching one billion units, double that of personal computers. Apple seized this moment to announce, in June, the release of Swift, a new programming language blending imperative, object-oriented, and functional approaches.

Since 1996, the company had relied on Objective-C to develop its applications. But the language had aged. Swift arrived as an answer to this need for renewal: simplifying code without sacrificing power, making programming more fluid, more intuitive. The designers sought to create something expressive, where interaction with the code feels almost natural.

The success was immediate. In less than a year and a half, Swift joined the club of the twenty most used languages in the world. This rise can be explained first by iOS's immense installed base: over 700 million devices sold. Apple's dominance in its segment and the announcement of Objective-C's gradual replacement did the rest. Developers had little choice, but the language proved appealing enough to win them over.

Swift brought its share of innovations. Optional types, for example, represented a real advance. This feature, common in functional languages but rare elsewhere, forces developers to explicitly handle null values. Fewer errors, greater clarity. Version 2.0 introduced a new system for error handling with do-try-catch, abandoning the old NSError approach inherited from Objective-C.

Experienced developers adapted without too much difficulty. Some characteristics recalled Objective-C, like those named parameters that make code readable almost like an English sentence. Yet an analysis of questions asked on StackOverflow revealed gray areas. Nearly 17.5% of queries concerned fundamental concepts: type manipulation, array handling. The language wasn't as simple as it appeared.

Development practices showed interesting patterns. Optional chaining became popular. This elegant technique avoids dereferencing errors while keeping code compact. Instead of stacking conditions, operations

are chained with a question mark. The program becomes safer and more pleasant to read.

The early years were not free from criticism. The compiler, in particular, was frustrating. It lacked stability, its error messages remained obscure. Xcode, the development environment, had bugs that slowed down teams. These technical problems contrasted with the promises of an improved development experience.

Integration with Objective-C was a major challenge. Apple had provided mechanisms to make both languages coexist within the same application. This compatibility was vital: impossible to rewrite everything at once, and existing libraries were too numerous to do without. The transition happened in stages, project by project.

Faced with the language's limitations, developers devised workarounds. The error handling system, for instance, didn't always suit asynchronous code. Some then adopted result enumerations, circumventing the restrictions. This creativity testified to a community taking ownership of the tool, shaping it to their needs.

The App Store generated considerable revenue. In January 2015, developers had earned 25 billion dollars, more than on other platforms. This financial windfall attracted both companies and independents. Swift was a profitable investment, not just a matter of technical preference.

The language evolved based on feedback. Each new version corrected flaws, optimized features. Documentation expanded, tutorials multiplied, and learning became less arduous for newcomers. Apple listened, adjusted, improved.

Swift represented a turning point in Apple's strategy. The company didn't just follow trends; it forged a modern language integrating current best practices. This approach preserved the ecosystem's stability while propelling it toward the future. No abrupt break, but a controlled transformation.

Introducing a new programming technology is delicate. One must innovate without losing what's been gained, attract without imposing, renew without breaking. Swift succeeded, establishing itself as an essential player in mobile development. The following years would confirm this position, making the language far more than a simple replacement for Objective-C.

Kafka

In 2008, at LinkedIn, Jay Kreps faced a challenge that no one had really anticipated. The professional social network was generating millions of messages each day, and traditional systems struggled to keep pace. Classic transactional data-profiles, user relationships-remained manageable. But it was primarily the mass of behavioral information that posed problems: profile views, keyword searches, various interactions. These digital traces, essential for understanding member expectations, literally overwhelmed the existing infrastructure.

Kreps then decided to develop a new solution. He partnered in 2010 with Neha Narkhede and Jun Rao to finalize what would become Apache Kafka. The name references writer Franz Kafka, a nod to the software's primary purpose: efficiently managing message stream writing. The team designed a distributed platform capable of broadcasting events in real time, with an architecture designed to handle considerable volumes.

The idea that makes all the difference lies in a counter-intuitive approach to storage. Where classic solutions keep messages in RAM, Kafka writes them directly to hard disk, sequentially. This method takes advantage of the natural optimization of linear input-output operations on hard drives, delivering impressive throughput without sacrificing reliability.

LinkedIn released Kafka as open source in 2011, under the aegis of the Apache Foundation. Silicon Valley giants took interest: Twitter, Netflix, and Yahoo adopted the technology, soon followed by more traditional companies like Goldman Sachs, Walmart, and Cerner. At LinkedIn, the numbers are staggering: over 800 billion messages processed daily, representing more than 175 terabytes of data flowing flawlessly.

In 2014, the three creators founded Confluent. Their objective: develop the ecosystem around Kafka and offer enterprise solutions based on this technology. The impact was immediate: the adoption rate multiplied sevenfold in the eleven months following the company's launch.

The architecture rests on simple but effective concepts. Messages are organized into "topics", categories that structure data flows. Each topic divides into partitions, replicated across different servers to guarantee availability and fault tolerance. This organization distributes the load and scales the system horizontally, without theoretical limit.

Development continued with major features. Kafka Connect, available since version 0.9, simplifies integration with other systems through a standardized framework for connectors. Kafka Streams offers a stream processing library directly integrated into applications, without additional infrastructure to deploy.

In 2022, KRaft (Kafka Raft) was released, a native consensus mechanism that replaces the historical dependency on ZooKeeper. The architecture becomes simpler and more reliable. This year also saw the introduction of tiered storage: part of the data can now be offloaded to object storage systems like S3, significantly reducing operating costs.

Kafka has transformed how companies think about their real-time data processing architectures. The model has inspired numerous alternatives: RedPanda offers an implementation compatible with the Kafka protocol, while WarpStream develops a radically different architecture favoring S3 storage.

In 2024, Kafka has established itself as a mature technology. Its adoption continues to grow, driven by the constant increase in real-time processing needs and the rise of the Internet of Things. The community that formed around the project regularly contributes to its improvement, ensuring continuous evolution.

The success stems from a solid technical architecture, a precise response to a real need, and a dynamic community supported by a stable company. What started as an internal project at LinkedIn has become a cornerstone of modern architectures.

GraphQL

In 2012, Facebook's teams hit a technical obstacle threatening the performance of their mobile applications. The News Feed API, designed according to the REST standards of the time, multiplied round trips between client and server. Each screen required multiple requests to different endpoints, and this fragmentation slowed down the user experience. On mobile networks, where every millisecond counts, the situation was critical. GraphQL was born as a pragmatic response to a real problem.

The initial idea rests on a simple observation: why couldn't the client ask for exactly what it needs, no more and no less, all at once? REST architectures impose a rigid structure where each *endpoint* returns a predefined set of data. The client often retrieves too much useless information, or not enough, forcing it to chain calls. Facebook's team reversed the logic: instead of the server dictating what it sends, the client specifies what it wants to receive.

For three years, GraphQL remained confined to Facebook's servers. Engineers refined the concept, tested different approaches, corrected er-

rors. The language took shape around a schema describing relationships between data as a graph. The syntax drew inspiration from JSON, familiar to web developers. In 2015, Facebook decided to publish the specification and a reference implementation. The community then discovered a radically different way of querying APIs.

The schema constitutes GraphQL's cornerstone. It defines the available object types, their fields, and how they connect to each other. With this explicit contract between client and server, frontend developers know exactly what data they can request, backend developers understand what they must provide. The strong type system catches errors before queries execute, a safety net appreciated in production.

GitHub was among the first major adopters. The company rebuilt its public API with GraphQL, followed by Pinterest and Coursera. Twitter took a progressive path: TweetDeck and Twitter Lite served as testing grounds before deployment on Android and iOS applications. Each organization adapted the technology to its constraints, forged its own practices, contributed new tools.

Twitter's case illustrates the necessary adjustments well. The infrastructure relies on microservices communicating via Thrift, a serialization system developed by Facebook. Integrating GraphQL required partial restructuring: a frontend layer intercepts GraphQL queries, validates them, handles authentication. Backend services preserve their business logic but expose their capabilities in a new way. This hybrid architecture preserves existing investments while exploiting GraphQL's advantages.

Coursera encountered other difficulties. The first migration attempt failed due to synchronization problems between the GraphQL schema and actual data. The team found a clever solution: middleware translates existing REST APIs into GraphQL on the fly. Clients benefit from the new interface while the backend evolves gradually. This pragmatic approach demonstrates that there isn't just one right way to adopt GraphQL.

The technical benefits didn't take long. Gone was the under-fetching that forced multiple requests. Gone was the over-fetching that wasted bandwidth transferring superfluous information. A single well-formulated query retrieves exactly what's necessary, involving multiple related resources if needed. Mobile applications breathe easier, web pages load faster.

The introspection type system opened unexpected possibilities. Development tools query the schema to provide autocompletion, dynamic documentation, real-time validation. IDEs transform into efficient assistants that guide the developer instead of leaving them to navigate through obsolete PDF documentation.

Gradually, the community enriched the language. To reuse query fragments, fragments emerged as an elegant solution. Directives offer a flexible mechanism to adapt behavior without modifying the schema. Subscriptions respond to the growing need for real-time updates, transforming GraphQL from a request-response system into a bidirectional communication channel.

Security represents a particular challenge. Query flexibility is a double-edged sword: a malicious client could formulate extremely complex requests to overload the server. The community developed safeguards: query depth limitation, complexity analysis before execution, granular permission control. These mechanisms allow exploiting GraphQL's power without exposing infrastructure to abuse.

Gartner predicts that more than 60% of enterprises will use GraphQL in production by 2027, compared to less than 30% in 2024. This progression testifies to the maturity achieved by the technology. Apollo, Hasura, StepZen and other players offer solutions that simplify the deployment and management of GraphQL servers. The ecosystem expands, best practices crystallize, pitfalls are better identified.

Beyond simply replacing REST, GraphQL modifies the way of thinking about client-server interactions. The client no longer endures backend decisions, it expresses its needs. This inversion changes the dynamic between frontend and backend teams, encourages better collaboration, clarifies responsibilities. The shared schema is the common language that aligns everyone's efforts.

Facebook wasn't trying to revolutionize the web, just accelerate its mobile application. The solution found proved applicable far beyond the initial context. By publishing the specification and animating an active community, Facebook transformed an internal solution into an industry standard. GraphQL thus joins the list of technologies born from a particular need that ended up redefining the practices of an entire profession.

TensorFlow

Google launched the Brain project in 2011, an initiative designed to explore deep neural networks in an industrial setting. The objective was straightforward: build practical artificial intelligence systems, initially for research, then for the company's products. This work led to DistBelief, an early machine learning system that provided researchers with a testing ground to experiment with complex neural architectures.

From 2011 to 2015, DistBelief served as the foundation for numerous breakthroughs. Unsupervised learning, language representation, image classification, object detection, video and speech recognition: application domains multiplied. More than fifty teams at Google and other Alphabet subsidiaries deployed neural networks based on this technology. The services that benefited covered a broad spectrum: search engine, online advertising, speech recognition, Google Photos, Maps, StreetView, Translate, YouTube.

Building on this experience, Google designed TensorFlow as DistBelief's successor. The platform was released in November 2015 under the Apache 2.0 license with a completely redesigned architecture that was a game-changer in the design of learning systems. Its flexibility enabled the expression of highly varied algorithms as computation graphs, all with enhanced performance for both training and model deployment.

TensorFlow's distinguishing feature lies in its ability to operate on heterogeneous distributed systems. A single model runs seamlessly on a mobile phone or in a computing center equipped with thousands of graphics processors. This adaptability drastically simplifies the practical use of learning systems by eliminating the need to create specialized versions for each environment.

It relies on a representation of computations as directed graphs. Nodes represent mathematical operations, edges carry tensors, these typed multidimensional arrays. This abstraction naturally suits learning algorithms, which constantly manipulate matrices and vectors. The system automatically manages the distribution of computations across available processing units.

The architecture is built around several components. The system's core defines basic operations: elementary mathematical computations, array manipulation, linear algebra, building blocks for neural networks. These operations benefit from optimized implementations according to processor types. Variables maintain the model's state from one execution to another. Queues ensure synchronization and efficient data transfer.

Developers access TensorFlow through Python and C++ interfaces. They build their models by assembling basic operations into a computation graph. The system then handles distributed execution, memory management, and optimizations. Tools like TensorBoard help understand and debug models by visualizing their structure and the evolution of learning metrics.

TensorFlow integrates symbolic gradient computation. This functionality automates a delicate task: calculating the derivatives required by optimization algorithms such as stochastic gradient descent. The system

analyzes the computation graph and automatically generates the operations corresponding to partial derivatives, applying the chain rule of differentiation.

Numerous optimizations improve performance and resource utilization. Common subexpression elimination removes redundant computations. Judicious placement of operations on different processors limits communications. Compression of data exchanged between machines reduces the required bandwidth. These optimizations are critical when training large-scale models.

Deployment in Google products demonstrates TensorFlow's effectiveness. The speech recognition system's error rate plummeted from 13.25% to 2.5% over five years. Image classification performance on ImageNet jumped from 63% to 91% accuracy. These dramatic leaps resulted from the combination of more sophisticated architectures and the ability to efficiently process enormous volumes of data.

Google simultaneously developed a range of specialized processors: the Tensor Processing Units (TPUs). These integrated circuits, tailored for reduced-precision linear algebra operations, offer substantial gains in performance and energy efficiency compared to conventional CPUs and GPUs. Successive generations of TPUs form massively parallel systems, reaching several exaflops for machine learning.

The open-source release created an active community. Researchers and developers adopted TensorFlow in highly diverse fields: computer vision, natural language processing, robotics, information retrieval, geographic information extraction, drug discovery. Contributors continuously enriched the platform.

The flexible architecture and high performance made TensorFlow a reference for research and deployment of complex models. The availability of mature tools and comprehensive documentation democratized access to deep learning techniques for a broad audience of developers.

TensorFlow's evolution reflects the rapid progress of artificial intelligence. From early research prototypes to large-scale production systems, the platform has accompanied the development of increasingly sophisticated applications. Its distributed architecture and optimizations fully leverage modern computing resources to train ever more powerful models.

Over the years, TensorFlow has been enriched with various extensions. Support for new model types, performance improvements, addition of development and debugging tools have progressively strengthened its capabilities. The platform continues to evolve to meet the growing demands of machine learning and artificial intelligence.

Accelerated Mobile Pages

In October 2015, Google launched Accelerated Mobile Pages, a technology designed to speed up mobile browsing. The findings are unequivocal: 53% of internet users abandon a page that takes longer than three seconds to load. Faced with this problem, Google sought a radical solution.

The project emerged from discussions between Google and European publishers as part of the Digital News Initiative. The initial idea targeted news sites, the first to be affected by reader frustration on smartphones. The technical solution rests on three pillars: a streamlined version of HTML, a JavaScript library that optimizes resource loading, and a caching system to store content. But this architecture imposes severe constraints. Style sheets cannot exceed 50 KB, custom JavaScript is prohibited, and images must use specific tags such as `<amp-img>` or `<amp-video>`.

The initial results exceeded expectations. For example, the Washington Post recorded 23% more users returning within a week of their first visit. In Europe, the Plista platform observed an average 220% increase in click-through rates on AMP content, with peaks of 600% for certain publishers. These figures attracted attention. Pinterest, Reddit, Bing, and eBay adopted the technology. Google integrated it into its News section in February 2016, then extended it to all mobile results six months later. A small lightning bolt then appeared to indicate AMP pages in search results.

On the technical front, the performance was spectacular. Pages loaded in under one second and consumed ten times less data than their conventional equivalents. The AMP cache automatically verified page compliance and adapted images to mobile screens. This infrastructure achieved speed levels unattainable by traditional websites.

AMP usage quickly expanded beyond media. In 2018, the technology moved into e-commerce, interactive emails, and narrative content creation with AMP Stories. WordPress, which powers more than a third of the web, offered an official extension to simplify its deployment.

But AMP's success came with sharp criticism. Despite its open-source code, many criticized Google for tightly controlling this technology. The fact that AMP pages were served from Google's servers raised questions: who really owns the content? The preference given to AMP pages in search results fueled accusations of anti-competitive practices. These criticisms led Google to announce, in 2018, the standardization of AMP principles through the Web Platform Incubator Community Group. The objective

was to allow any site to achieve instant loading without going through AMP, by simply meeting certain performance criteria with more open standards such as Web Packaging and Feature Policy.

In 2020, Google continued developing AMP while gradually opening up the format. AMP E-mail brought interactivity to electronic messages, while AMP Stories created immersive visual content. The project now includes more than 600 million pages across 232 countries and 104 languages.

The technology has undeniably accelerated smartphone browsing, but it has also reinforced centralization around major platforms. Its evolution toward more open standards reflects the search for a balance between performance and decentralization. Beyond technical aspects, with AMP, speed has become a determining factor in the mobile experience.

Natural Language Processing

Since the 1950s, researchers have been trying to teach computers to understand our language. Natural language processing, more commonly referred to as NLP in specialized circles, has long stumbled over the same obstacles: how to capture the subtleties, double meanings, and changing context of a conversation? Early machines worked with fixed dictionaries and rigid grammatical rules. Then came statistical approaches in the 2000s, which brought their share of improvements without truly solving the fundamental problem.

The year 2017 marks a turning point. On the Mountain View campus, Ashish Vaswani and Jakob Uszkoreit are discussing machine translation in a hallway. A seemingly mundane conversation that will lead to something unexpected. Their team consists of eight Google researchers, including Illia Polosukhin, a science fiction enthusiast. It's precisely a film, *Arrival*, that gives him an idea. In this story, aliens communicate with symbols that express entire concepts at once, without following our usual sequential logic. Why couldn't machines do the same with sentences?

This reflection leads to the concept of self-attention: instead of analyzing words one after another like reading a book, the system grasps the sentence as a whole. All words interact with each other simultaneously. Noam Shazeer, who has been working at Google since 2000 and created the famous "Did you mean?" function, hears about the project in the offices. He joins the venture. The first trials on English-German translations yield encouraging results.

The resulting Transformer architecture is faster and more accurate than previous recurrent neural networks, and it's not limited to text. The model processes images, computer code, DNA sequences. This versatility surprises its creators.

The following year, another team from Google AI publishes BERT. The acronym conceals a simple idea: reading in both directions. The context of a word depends on what comes before, but also on what follows. BERT performs on eleven different language processing tasks, breaking previous records.

Curiously, all eight researchers behind Transformers leave Google in the following years. Each founds their own company, exploits the technology in their own direction. Aidan Gomez launches Cohere for businesses, Jakob Uszkoreit creates Inceptive and applies Transformers to vaccines, Noam Shazeer develops Character.ai where users design their own conversational agents. Vaswani and Parmar establish Essential.ai. We find here a classic pattern: large organizations struggle to transform their discoveries into commercial products. Researchers leave to create elsewhere.

This architectural foundation has since fueled all modern language models. The ability to process enormous volumes of data, to weave complex links between elements of a sequence, makes Transformers an indispensable tool. Applications multiply: more natural translation, coherent text generation, document analysis, high-performing question-answering systems. Machines are approaching the nuances of human language, better understanding the relationships between ideas.

From word-by-word processing of the early days to contemporary attention systems, each step has brought computers closer to our linguistic complexity. Transformers are not just a technical feat. They serve as a foundation for a cascade of innovations in various fields: medicine, scientific research, education. The scientific community constantly develops new variants.

As these models gain sophistication, they alter our relationship with technology. The boundary between human communication and machine interaction becomes blurrier. Future developments promise new possibilities while raising questions of ethics and responsibility. How far will this capacity of systems to understand and produce natural language go? The answer is being written day by day, in laboratories as in our daily uses.

WebAssembly

In 1995, JavaScript emerged as a simple scripting language. No one imagined it would become the engine for sophisticated Web applications. Originally, it was mainly used to validate forms or animate a few page elements. The heavy lifting was done on servers, with browsers only handling display and basic interactions.

Everything changed with the arrival of high-performance engines like V8. JavaScript began handling programs of several thousand lines, applications that would have been unthinkable just a few years earlier. The language found itself being used far beyond what it had been designed for.

This transformation pushed Web stakeholders to seek alternative paths. Microsoft tried its hand with ActiveX, which relied on signing x86 binaries, an approach based on trust rather than genuine technical guarantees. Google then developed Native Client, which introduced sandboxing for machine code on the Web. Performance approached native levels, but developers had to follow strict constraints in their code, such as using binary masks before each memory access.

Emscripten then opened up a new possibility: compiling C or C++ to JavaScript. The system created a runtime environment in JavaScript and translated the code to a specialized subset, which gave birth to asm.js. Low-level code could finally be executed in a browser, but JavaScript remained a constraining intermediary.

It was in 2015 that the story took an unexpected turn. Engineers from Google Chrome, Mozilla Firefox, Microsoft Edge, and Apple Safari sat down at a table to work together. These historical competitors, who had engaged in "browser wars" for years, decided to collaborate. Their goal? Create a portable bytecode format they called WebAssembly.

The project had clear ambitions. It needed a compact representation, fast validation, efficient compilation. And above all, safe execution without performance penalty. The designers made a bold choice: formalize the semantics from the start. WebAssembly thus became the first industrial language designed with formal semantics from its inception.

The resulting architecture has some distinctive features. WebAssembly relies on a stack machine to represent computations. But thanks to its type system, compilers can analyze data flow between instructions without having to physically materialize this stack. Control flow is structured: no wild `goto`, but classic constructions via nested blocks.

Memory works like a large array of bytes, separated from code and the execution stack. A failing or compromised program can only corrupt its

own data. This isolation ensures security: the code executes in a sandbox that prevents it from accessing the host system without authorization.

The binary format was designed for the network. Instructions fit in one byte, integers use the LEB128 format. Browsers can start compilation as soon as the first bytes are received, without waiting for the complete module download.

In 2017, the four major browsers integrated WebAssembly support. Each team chose its own implementation strategy. V8 and SpiderMonkey repurposed their JIT compilers to compile modules before instantiation. JavaScriptCore followed a similar path. Chakra preferred to translate on the fly to an internal bytecode, then compile the most frequently used functions.

WebAssembly often reaches 80 to 90% of native code performance, while maintaining the Web's portability and security. The generated code is compact: 62.5% of the size of asm.js on average, 85.3% of that of native x86-64.

WebAssembly's influence extends beyond the browser. Its design makes it usable in other contexts. Standalone implementations exist that transform WebAssembly into a portable format for native applications. This echoes Java's original ambition of "write once, run anywhere".

Development continues. Zero-cost exceptions have arrived, as have threads and SIMD instructions. Integration with browser garbage collectors is underway, which will facilitate the compilation of languages like Java, C#, Swift, or OCaml to WebAssembly.

Beyond the technical aspects, WebAssembly testifies to the achievement of collaboration in the Web ecosystem. The project shows that a rigorous, almost academic approach can coexist with the realities of industrial development. This technology has redefined what can be run in a browser, without sacrificing the security that lies at the heart of the Web.

WebAuthn

Since the first computer systems emerged, the question of user authentication has been a constant concern. Passwords quickly became the obvious answer: a character string known to the user, simple to implement, easy to understand. Except that this simplicity poorly conceals its weaknesses. Passwords are stolen, guessed, intercepted. In 2024, more than 80% of security breaches still exploit weak or stolen passwords.

Over time, other methods have emerged. Two-factor verification, digital certificates, biometrics. Each contributes its part, but they sometimes remain cumbersome to deploy or lack true uniformity. This is where the W3C and the FIDO Alliance decided to change the game with Web Authentication, more simply called WebAuthn.

Google, Mozilla, Microsoft, and Yubico rolled up their sleeves to develop this specification published in 2019. The idea breaks with what had been done for decades. Instead of sharing a secret between the user and the server, WebAuthn relies on asymmetric cryptography. When you register with a service, a key pair is created. The private key stays securely on your device, while the public key goes to the server with a randomly generated identifier.

In this architecture, the public key stored on the server side is useless for hacking your account. Attackers are increasingly losing interest in databases. Authentication is tied to the exact domain of the site: a key created for example.com will never work on evil-example.com. Phishing attempts lose much of their effectiveness.

WebAuthn rests on three pillars. Robustness first: authentication relies, when possible, on a hardware security module that keeps private keys and performs cryptographic calculations. Scope second: each key pair is only valid for a specific origin, like the cookies we know well. Attestation finally: authenticators can provide a certificate that proves to the server that the public key comes from a trusted source.

Browsers quickly adopted this technology. Chrome, Firefox, Edge, Safari: all support the standard. Developers therefore have the ability to integrate it uniformly. Modern operating systems followed with their own authentication systems: Windows Hello at Microsoft, Touch ID at Apple.

The process itself unfolds in several stages. During registration, the server transmits data that links the user to their credentials, with information about the organization and a random string to counter replay attacks. The browser uses the Web Authentication API to request the creation of a new key pair.

The response contains the authenticator identifier, client data in JSON, an attestation object. The server verifies all this according to a 19-step procedure described in the specification. Once validated, it stores the public key and identifier in its database.

For subsequent logins, you prove that you possess the private key by creating an assertion. The server sends a new random string, the authenticator generates a digital signature that the server verifies with the public key it has kept. No more risk of password theft, and the user experience is improved.

WebAuthn is part of FIDO2, a set of technologies that aims to generalize passwordless authentication. The need is felt: in 2024, only 28% of users enable two-factor authentication on their online accounts.

The specification continues to evolve with new features and implementations that are refined. Developers find code examples and libraries in different languages, notably Python and Go, to integrate WebAuthn into their applications.

The history of WebAuthn shows how authentication methods on the Internet are transforming. By replacing passwords with cryptographic keys, this standard meets current security requirements while simplifying users' lives. Major web players are increasingly adopting it, and one can bet that in the future, authentication will become both more secure and more convenient.

This evolution is part of a broader movement where asymmetric cryptography is gradually replacing the shared secrets of the past. WebAuthn represents an important milestone in the history of computer security, providing a standardized solution to this authentication problem that has pursued us since the beginning.

XI

2020

The Digital Era Confronts Global Upheavals

Covid-19 struck the world without warning at the beginning of the 2020s, upending our most deeply rooted habits. Overnight, our screens became our only windows to the world. We witnessed an accelerated mutation of society: conference rooms gave way to video calls, classrooms to online learning platforms, cafés to virtual discussions. This brutal shift toward all-digital revealed both our technological dependence and our capacity for adaptation.

Digital infrastructures, once invisible to the general public, suddenly found themselves in the spotlight. The question of who controlled these vital resources became a political debate. Europe became acutely aware of its fragility in the face of American cloud giants: Amazon Web Services, Microsoft Azure, Google Cloud. The old continent responded by launching GAIA-X, an attempt to counter this hegemony. An ambitious project, perhaps belated, but symbolic of a will for strategic autonomy.

On the global chessboard, the Sino-American technological war took a dramatic turn. Washington wielded semiconductors as a weapon to slow China's ascent, banning the export of advanced chips to Beijing. In response, China redoubled its efforts to create its own production chain.

The public appearance of ChatGPT in late 2022 captured the collective imagination. Artificial intelligence and language models demonstrated

they could write, code, invent, and converse. This wave sparked both fascination and concern. Parliaments worldwide took up the issue, with Europe leading the charge through its AI Act. The question was no longer whether AI would transform our professions, but how quickly and to what extent.

Cybersecurity moved from the status of a technical concern to that of an existential issue. Ransomware attacks targeted hospitals, administrations, and vital industries. Russia's invasion of Ukraine added a new dimension to the conflict: the war was also being fought in cyberspace, each ground offensive accompanied by its digital counterpart. This reality forced states and companies to strengthen their invisible defenses.

The cryptocurrency roller coaster captivated media attention. The spectacular collapse of FTX in 2022 served as a reminder of this young sector's volatility. Regulators, long hesitant, began to establish guardrails. Central banks, initially skeptical, worked on their own digital versions of currency. Beyond speculation, blockchain found its place in industrial traceability and document certification.

The digital environmental footprint came to light. The servers that power our connected lives consume the energy equivalent of entire countries. Cryptocurrency mining was singled out for its excessive environmental impact. In response to this awareness, tech companies multiplied their green promises: data centers powered by renewable energy, more durable hardware, less resource-intensive software.

Our homes filled with connected devices: voice assistants in constant listening mode, watches tracking our vital signs, cameras monitoring our comings and goings. This technological intrusion raised questions about our privacy. European GDPR became a global reference, inspiring similar legislation around the world, from Brazil to California.

The social media galaxy went through a turbulent period. Elon Musk's acquisition of Twitter transformed the platform into a controversial laboratory of unlimited free speech. TikTok conquered global youth, under the worried eye of Western governments concerned about the Chinese influence it carries. New networks, more intimate, emerged in reaction to the omniscient giants.

5G wove its global web, despite controversies surrounding Chinese equipment makers, Huawei foremost among them. This technology enabled unprecedented industrial applications like hyper-connected factories, autonomous and communicating vehicles, and remote surgery. Meanwhile, laboratories worked on 6G, promising dizzying speeds and infinitesimal latency.

Space, the new digital frontier, transformed into a playground for vi-

sionary billionaires. Low Earth orbit satellite constellations, like SpaceX's Starlink, began weaving a global internet network, bringing connectivity to the most isolated regions. This space democratization nonetheless ran up against astronomers' concerns, worried about seeing the night sky dotted with artificial points of light.

Virtual and augmented reality sought their path beyond the simple gadget. The concept of the "metaverse" resurrected by Mark Zuckerberg, crystallized both hope and skepticism. While the general public remained lukewarm, these technologies found their utility in professional training, industrial maintenance, and architectural design. Digital immersion progressed step by step, far from futuristic promises.

Quantum computing reached important milestones. Experimental processors achieved stability unthinkable just a few years earlier. While the universal quantum computer remains a distant horizon, specific applications emerged in molecular simulation and logistics optimization. This disruptive technology sparked strategic interest, each power fearing to fall behind in this twenty-first-century race.

Robots left factory cages to work alongside humans. These "cobots", more flexible and intelligent than their predecessors, adapted to complex and variable tasks. Post-pandemic labor shortages accelerated this trend. In warehouses, hospitals, and fields, these new machines changed the nature of work, raising as much hope as apprehension.

The State digitized at a forced march. Administrations offered more personalized and smoother online services. Cities equipped themselves with sensors to optimize traffic, lighting, and waste collection. Digital identity advanced, between administrative convenience and fears of authoritarianism. The path toward digital citizenship took shape, winding and fraught with obstacles.

Education retained lasting traces of the Covid period. Classrooms durably integrated educational software. Online courses gained pedagogical maturity. Artificial intelligence personalized learning paths. This accelerated digitization nonetheless widened inequalities, between connected and disconnected students, between innovative and neglected institutions.

Medicine embraced digital tools. Connected medical devices enabled remote monitoring of chronic patients. Diagnostic assistance algorithms proved themselves in detecting cancers and rare diseases. Teleconsultation entered common practice. This digital breakthrough confronted questions of confidentiality and care's humanity, while opening new therapeutic perspectives.

A collective awareness of digital ethical issues gradually developed. Algorithmic bias, organized disinformation, the attention economy, wide-

spread surveillance: these subjects left specialized circles to enter public debate. Tech-ethics established itself as a discipline influencing digital service design.

The first part of this 2020s decade will remain the one where digital became both our greatest strength and our major vulnerability. Cascading crises revealed its central place in organizing our societies. More than a simple economic sector, computing asserted itself as the nervous system of our civilization, carrying our collective hopes as much as our anxieties. This awareness calls for renewed governance, where innovation rhymes with responsibility, where technical progress harmonizes with social justice and environmental sustainability.

Apple M1

When Apple unveiled the M1 chip in November 2020, few observers grasped the magnitude of the impending disruption. Yet this idea had its roots in an old frustration of Steve Jobs: dependence on processor suppliers. Since the Apple I and its modest MOS Technology 6502, the company had been subject to the technical choices and schedules imposed by others. This situation hardly suited a company that cultivated absolute control over its products.

The first attempts at emancipation date back to the 1990s with the Newton, that ahead-of-its-time tablet that was already collaborating with ARM. The Newton's commercial failure masked valuable learning: Apple discovered the subtleties of processor design. The iPod then marked an intermediate step with its Portplayer PP5502 system on chip, equipped with two ARM cores. This architecture appeared in the original iPhone, with Samsung still manufacturing the components.

In 2008, Apple acquired P.A. Semiconductor for $278 million, an acquisition that went relatively unnoticed at the time but proved decisive. This Texan company brought the missing expertise to design in-house processors. The first fruit of this union was born two years later: the A4 that powered the first-generation iPad, then the iPhone 4.

Samsung manufactured these A4 chips, but Apple switched to Taiwan Semiconductor Manufacturing Company (TSMC). This migration revealed a carefully considered strategy: mastering the design while relying on Taiwanese manufacturing expertise. The subsequent A-series processors confirmed the relevance of this approach. Each generation ex-

ceeded expectations, delivering remarkable performance with controlled power consumption.

The secret lies in optimization. Unlike general-purpose processors from Intel or AMD that must satisfy a thousand different use cases, Apple's chips focus on a closed ecosystem. This specialization pays off: an iPhone with less RAM than a rival Android smartphone often displays superior performance. The harmony between silicon and operating system makes the difference.

This success in mobile naturally pushed Apple toward computers. In June 2020, at WWDC, Tim Cook announced the Mac transition to ARM architecture. Six months later, the first MacBook Air and 13-inch MacBook Pro with the M1 chip hit the market.

The M1 marked a technological breakthrough. Etched in 5-nanometer technology at TSMC, it brings together 16 billion transistors on a tiny surface. Its architecture breaks conventions: CPU, GPU, Neural Engine, and unified memory coexist on the same substrate. This integration eliminates traditional bottlenecks between separate components. Data flows faster, latency decreases, energy efficiency improves.

The M1's ARM architecture favors simplicity. Its RISC (Reduced Instruction Set Computing) instructions contrast with the growing complexity of Intel's x86 processors. This minimalist philosophy, inherited from mobile processors, adapts perfectly to the needs of modern computing. Eight computing cores share the tasks: four optimized for pure performance, four others for energy efficiency. This intelligent distribution automatically modulates consumption according to workload.

The first tests astounded the industry. An M1 MacBook Air outperformed a 16-inch MacBook Pro equipped with an Intel Core i9 processor in many benchmarks, all while operating fanless. Battery life doubled, sometimes tripled compared to previous models. These spectacular gains reshuffled the laptop market.

Apple didn't stop there. The M1 Pro, with its 33.7 billion transistors and ten computing cores, targets creative professionals. The M1 Max pushes the exercise to the extreme with 57 billion transistors and a 32-core graphics processor that rivals dedicated graphics cards. These variants established Apple as a key player in high-performance processors.

This rise in power disrupted the computing ecosystem. Intel, accustomed to dictating its terms for decades, discovered that a competitor could design more efficient processors by starting from a different architecture. Apple's vertical approach, which controls the entire chain from silicon to applications, demonstrated its superiority over the traditional model of assembling generic components.

The shockwave extended beyond Apple. Qualcomm accelerated the development of ARM processors for Windows laptops. Microsoft adapted its operating system to better exploit this architecture. AMD and Intel rethought their strategies in the face of this new competitive landscape.

Beyond raw performance, the M1 embodies a different vision of computing. It favors the harmonious integration of components rather than the race for specifications. This philosophy extends to the entire Apple ecosystem, where each element is designed in symbiosis with the others.

Patient strategy can revolutionize an entire sector. Starting from legitimate frustration, Apple methodically built its expertise until it surpassed the historical leaders. This transformation anticipates the future of an industry where the boundary between hardware and software is fading in favor of a comprehensive approach to innovation.

Zig

In 2015, Andrew Kelley embarked on the ambitious endeavor of creating a new systems programming language. At that time, developers had a substantial arsenal at their disposal with C, C++, Rust, and Go. Yet Kelley identified a recurring problem: each language carried its own limitations that complicated the writing of truly reliable software.

His reasoning began with a striking comparison. In aviation or the elevator industry, safety systems are layered to make accidents nearly impossible. Software, however, still suffers from a reputation for fragility and unpredictability. Kelley wanted to change this with Zig.

The new language adopted a radical philosophy: do less, but better. Zig eliminates C's preprocessor, a mechanism deemed too complex and error-prone. This decision might appear regressive, but Kelley introduced other mechanisms that solve the same problems more elegantly.

Memory management in Zig breaks with current trends. No garbage collector here, unlike Java or Python. The unpredictable program interruptions to free memory disappear. Zig relies on a system of "allocators" that gives developers fine-grained control over memory, without the burden of traditional C. Developers know exactly when and how their program uses memory.

Errors, every programmer's nightmare, receive special treatment. Gone are exceptions that can pop up anywhere in the code. Zig integrates errors

directly into function return types. The compiler literally forces developers to handle every possible error. This constraint, burdensome at first, significantly improves the reliability of the final code.

The Zig compiler conceals the ability to execute code during compilation, which opens up metaprogramming possibilities without resorting to C++'s complex macros. Many errors are detectable by developers before execution and performance can be optimized upfront.

Interoperability with C represents a strategic asset for Zig. The language doesn't merely use existing C libraries; it functions itself as a high-performance C compiler. This duality facilitates gradual adoption in existing projects. Developers can start by using Zig as a simple C compiler, then progressively migrate to its advanced features.

The integrated build system replaces tools like Make or CMake with a unified solution. No need to juggle different systems depending on the platform: Zig compiles identically everywhere. This standardization drastically simplifies life for developers working across multiple systems.

Several significant projects have adopted Zig. Bun.js, that Node.js alternative that's been making waves, is developed with this language. Its creator, Jarred Sumner, explains this choice by Zig's ease of learning compared to C++ or Rust, while maintaining modern features and excellent development safety.

The Zig community is growing around carefully crafted documentation and an ecosystem expanding on GitHub. The Zig Software Foundation, with Loris Cro as VP of community, structures the language's development and promotion. This organization lends legitimacy to the project against established giants.

Zig's different compilation modes deserve closer examination. In debug mode, the compiler automatically detects many common errors, helping developers identify problems. In release mode, these checks vanish to unleash the full power of the processor. This flexibility addresses the contradictory needs of development: safety during creation, performance in production.

The path to version 1.0 is still long. This deliberately distant milestone reflects the creators' caution, preferring to stabilize the language rather than rush a release. Despite this development status, Zig runs in production in certain projects, proof of its technical maturity.

Zig doesn't claim to replace C overnight. The ambition is more subtle: to propose a modern evolution that corrects C's historical flaws without losing its qualities. This progressive approach makes Zig adoption gradual for developers, starting with using it as a C compiler before exploring its innovations.

The Zig adventure illustrates a constant quest in computing: finding the perfect balance between simplicity, safety, and performance. By eliminating accidental complexity while preserving control over hardware, this language charts an original path in the crowded landscape of systems development tools.

Mojo

In 2022, Chris Lattner left Apple after designing Swift and founded Modular with Tim Davis, a former Google employee. They wanted to create a language that would finally reconcile Python with performance. A common ambition given the numerous attempts, but their approach differs from previous solutions.

Artificial intelligence is literally exploding. GPUs are running at full capacity, TPUs (Tensor Processing Units) are proliferating, and yet developers remain stuck between two worlds: on one side Python, readable but sometimes desperately slow, on the other C++ or Rust, fast but with a complexity that discourages many. This divide comes at a high cost. Research teams write their prototypes in Python, then production teams completely rewrite them in another language. A considerable waste of time and energy.

Mojo was born from this frustration. But unlike past attempts, Lattner and Davis are not trying to replace Python. They want to extend it, elevate it. The gamble is bold: preserve Python's familiar syntax while integrating advanced concepts borrowed from Rust for memory management and LLVM for compilation.

The language incorporates sophisticated optimization tools from its conception. Tiling optimization, for example, automatically reorganizes calculations to best exploit processor caches. The auto-tuning module adjusts execution parameters according to available hardware. These technical innovations remain transparent to the programmer who writes seemingly simple code.

Compatibility with Python is the trump card. Libraries like NumPy or Matplotlib work without modification. This interoperability avoids starting from scratch, a classic pitfall for new languages. A developer can migrate gradually, replacing only critical parts with optimized Mojo.

The numbers speak for themselves. Some benchmarks show accelerations of 68,000 times compared to standard Python. These spectacular

performances are explained by the use of MLIR, a technology developed by Google to optimize code on different types of processors. Where Python interprets each instruction, Mojo compiles and optimizes everything.

Modular accompanies the language with a complete ecosystem. Basalt for machine learning, Endia for scientific computing, Lightbug HTTP for the web. These libraries, entirely written in Mojo, demonstrate the language's capabilities while serving as examples for developers.

On May 2, 2023, the Mojo Playground opened its doors. This online platform immediately attracted attention: 120,000 registrations in a few months, a community of 19,000 members on Discord and GitHub. The enthusiasm surprised the creators, while developers shared their experiments, created libraries, and proposed improvements.

Mojo's technical architecture breaks with certain Python conventions. Inferred static typing improves performance without making writing more cumbersome. Value semantics, where functions receive copies rather than references, avoids many classic bugs. These design choices reflect Lattner's experience with Swift and his understanding of modern development pitfalls.

Tooling follows the language's development. A Visual Studio Code extension, a Jupyter kernel for notebooks, advanced debugging features. This software suite facilitates adoption by teams already familiar with the Python ecosystem.

The impact on the industry is beginning to take shape. Research teams are using Mojo for their artificial intelligence projects, eliminating the gap between prototype and production. Optimized algorithms reduce the energy consumption of data centers, an issue that has become critical with the explosion of language models.

The community is producing its first remarkable projects. Maxim Zaks implements sophisticated data structures, others develop libraries for matrix computation. This creativity demonstrates rapid appropriation of the language by its users.

Modular continues development with ambition. The stated goal: make Mojo a complete superset of Python. The gradual opening of the source code should accelerate this evolution. Support for macOS and Windows is expanding, broadening the base of potential users.

In 2024, Mojo continues its growth. Performance improves, new features appear, the ecosystem grows richer. The democratization of artificial intelligence and the explosion of high-performance computing needs create favorable ground for this type of innovation.

Solutions often emerge from the creative combination of existing approaches rather than conceptual revolutions. Lattner and Davis were able

to identify a widely shared frustration and propose a pragmatic response. Their success is due as much to technical quality as to a fine understanding of modern developers' needs.

WebGPU

The University of Illinois at Urbana-Champaign faced an unexpected challenge in 2013. Between 2002 and 2011, massive online courses were experiencing explosive growth, digital education in higher education had climbed 17.3% annually, increasing student enrollment from 1.6 to 6.7 million. But how could GPU programming be taught remotely when most learners lacked access to these specialized processors?

Traditional solutions quickly revealed their limitations. Lending computers equipped with GPUs? Impractical at scale. Reserving computer labs? The idea seemed outdated given the scope of MOOCs. As for traditional computing clusters, they accumulated disadvantages: prohibitive costs, complex maintenance, and a learning curve that discouraged beginners.

WebGPU was born from a simple yet bold insight: transform any web browser into a gateway for GPU programming. The idea revolutionized the pedagogical approach at the time. No more laborious installations or specific configurations – an internet connection was all that was needed to program in CUDA, OpenCL, or OpenACC.

The architecture envisioned by the Illinois team consisted of three complementary elements. On one side, web servers hosted a streamlined user interface. In the center, a database orchestrated the management of work submitted by students. Finally, GPU-equipped compute nodes executed code under secure conditions. This technical triangulation concealed genuine sophistication: each code fragment was first analyzed to detect forbidden system calls, then confined to an environment with drastically limited permissions.

The large-scale test came with the "Heterogeneous Parallel Programming" course on Coursera. Over 100,000 participants enrolled, validating the initial intuition while also revealing the specifics of massive education. For while attendance was impressive, it came with a well-known phenomenon: 85% of enrollees dropped out along the way. WebGPU therefore had to manage dramatic fluctuations, with activity spikes concentrated on assignment due dates.

The technical team constantly adjusted the number of available GPUs, relying on the flexibility of Amazon AWS cloud services. This continuous juggling built valuable expertise in elastic management of computing resources. The experience accumulated enough feedback to envision a major overhaul.

WebGPU 2.0 emerged in 2015 with enhanced ambitions. The user interface migrated to OpenEdX, simplifying the integration of educational content. But the most striking innovation lay in adopting Docker to isolate execution environments. This technology, still young at the time, brought unprecedented granularity in resource management and facilitated the deployment of custom configurations.

Docker containers dynamically associated with physical GPUs according to the specific needs of each practical assignment. This architectural flexibility finely adapted learning environments while preserving the overall system's security. The University of Illinois deployed this new version for its traditional courses ECE 408 and ECE 598HK.

Success extended beyond Illinois borders. North Carolina State University adopted WebGPU, followed by the University of Tennessee. The PUMPS summer school in Barcelona integrated the platform into its intensive training sessions. This geographical spread confirmed the approach's relevance beyond the initial context.

WebGPU's deliberately simplified interface, limited to six main operations, illustrated a particular pedagogical philosophy. Unlike traditional development environments that multiply functionalities, this deliberate restriction guided learning by avoiding dispersion. Experience showed that a constrained framework sometimes better fostered understanding than total freedom.

Some experiments nevertheless failed. Peer evaluation, attempted during the first sessions, had to be abandoned due to implementation difficulties. These failures fed reflection on the limits of automation in technical education.

The design team meticulously documented their work in academic publications. This transparency contributed to advancing knowledge about online educational platforms. Issues of scalability, security, and automated assessment found in WebGPU a testing ground rich in lessons.

The platform's history revealed the transformations of higher education in the digital age. WebGPU addressed the specific requirements of parallel programming while adapting to the constraints of massive training. Its development testified to the growing importance of cloud infrastructure in modern education.

In 2024, as artificial intelligence stimulated unprecedented demand for

GPU training, WebGPU's legacy persisted. Current platforms drew inspiration from its innovations in elastic resource management and automated assessment. This story demonstrated how a technical solution born from a specific pedagogical need could generate lasting advances in digital education.

WebGPU's modular design remained relevant. Its founding principles – accessibility, scalability, security – continued to inspire creators of new learning platforms. The experience gained with this system enriched our understanding of issues related to democratizing advanced computing technologies, revealing that a pedagogical innovation could sometimes anticipate the technological developments it accompanied.

Epilogue

This book ends where perhaps the greatest technological rupture in our history begins. The technologies recounted here already belong to the past. Some survive, others have disappeared. But what lies ahead surpasses in scope everything we have known since the invention of the transistor.

Quantum computing is no longer a distant promise. Google demonstrated in 2024 that one of its quantum processors was 13,000 times faster than the most powerful supercomputer in the world. IBM plans to deliver systems capable of executing 15,000 quantum operations by 2028. Public investments reached 10 billion dollars in 2025, compared to 1.8 billion the previous year. These figures reflect a brutal acceleration. The error correction threshold, which has long blocked any practical application, is beginning to be crossed. The simulation of complex molecules for drug discovery, the optimization of industrial processes, or the creation of new materials are becoming feasible in the short term. But quantum computing threatens current encryption systems, which will collapse like houses of cards when faced with these machines. The race has already begun to develop post-quantum algorithms before the first truly operational computers leave the laboratories.

Artificial intelligence has crossed in just a few years the wall that separated experimentation from mass usage. It appears everywhere, and it is indeed the legacy of the hardware, systems, languages, and software presented in this book (but not only) that made its development and execution possible. In recommendation systems, voice assistants, text and image generation, autonomous driving. But this omnipresence reveals flaws. AI models can be manipulated, biased, hijacked. Their energy consumption is exploding. A single latest-generation language model requires

as much electricity as a small city during its training. Data centers are multiplying to absorb this demand. And no one really knows where this trajectory is leading us. Ethical questions are piling up faster than answers. Who controls these systems? Who is responsible when they make mistakes? How can we ensure they don't reproduce the discrimination present in their training data? Europe is attempting to regulate with the AI Act, but technology advances faster than legislative texts, which are very static.

Cyberattacks have changed scale. In 2024, the average cost of a data breach reached 4.88 million dollars, up 10% over one year. Ransomware accounts for 59% of attacks suffered by organizations. The average ransom paid exceeds 2 million dollars. Hospitals, critical infrastructure, and transportation systems have become prime targets. In 2024, the attack against Change Healthcare disrupted medical claims processing for more than 100 million Americans. Snowflake saw data from over 100 of its clients exfiltrated, including AT&T, Ticketmaster, and Santander. Unpatched vulnerabilities remain the main entry point: 20% of breaches begin with the exploitation of a known but unpatched vulnerability. The average time to identify and contain a breach is 241 days. More than nine months during which attackers can move freely through systems. Faced with this, the means remain inadequate. Companies recruit security teams they struggle to train and retain. AI-powered defense automation is beginning to show its effectiveness, with a 108-day reduction in detection time for those who use it. But AI also serves attackers, who automate reconnaissance, exploitation, and exfiltration. The battle now takes place at machine speed.

Brain-computer interfaces are leaving the laboratories. Neuralink implanted its first volunteers in 2024. A quadriplegic patient now controls a computer cursor by thought to play chess. Other companies like Synchron or Precision Neuroscience have less invasive approaches. In 2025, about twenty clinical trials are underway. Medical applications initially target paralyzed people, those with ALS or other neurodegenerative diseases. But the stated ambition is to restore vision in the blind with visual prosthetics, enhance cognitive abilities, merge human and artificial intelligence. These projects raise dizzying questions. Where does the human begin, where does the machine end when a chip reads and modifies brain activity? The risks of misuse are immense. Who guarantees that a state or company won't seek to monitor or manipulate thoughts through these devices? Ethical safeguards exist on paper. Their application in a context of fierce international competition remains to be proven.

The user experience is shifting toward new paradigms. Augmented reality glasses are emerging as the next major screen after the smartphone.

Meta, Apple, Google, Xreal, all are investing massively. Ray-Ban Meta glasses already integrate AI to identify objects in real time. The 2025 models display information directly in the field of vision with a 70-degree angle. The market is expected to grow from 678,600 units in 2023 to 13 million in 2030. These devices change how we interact with information. No need to take out your phone: directions float on the sidewalk, translations appear as overlays, notifications are superimposed on the real world. But this fusion of physical and digital raises concerns. Surveillance becomes permanent with always-on cameras. Privacy disappears in the face of incessant data collection. Tablets failed to replace the PC because they didn't offer enough power for complex tasks. AR glasses could succeed where they failed by freeing the hands while providing access to unlimited virtual workspace. Will the laptop and smartphone survive this transition? Hard to say. Some predict that within five years, AR glasses will replace the smartphone as the primary device. Others estimate it will take another decade to overcome technical and cultural obstacles.

Will we all wear a chip under our skin or in our brain connected to the internet via Wi-Fi? The idea is no longer science fiction. RFID implants are already used for contactless payments or access to secure buildings. Thousands of people have taken the plunge. But widespread brain implants remain a distant horizon. The technical obstacles are immense. Long-term biocompatibility is not guaranteed. Risks of infection or rejection exist. And above all, social acceptability remains very low. Few people would agree today to have their skull drilled to improve their cognitive performance. Glasses or augmented contact lenses represent a less invasive and probably more acceptable path. Several companies are working on lenses capable of displaying information or measuring physiological parameters. But component miniaturization and energy management still pose problems. We will likely have to wait until the 2030s to see these devices leave the prototype stage.

Corporate IT infrastructures are becoming more complex. Hybrid cloud, microservices architectures, containers, *edge computing* create environments of unprecedented sophistication. Managing these systems requires expensive skills. AI-powered automation promises to simplify this management, but it introduces new dependencies. What happens when a critical system relies on an AI model whose internal functioning no one truly understands? Failures become more difficult to diagnose. Regulations are tightening. GDPR in Europe, data protection laws in California and elsewhere impose increasing constraints. Companies must trace the origin of data, document their processing, guarantee their security. Non-compliance fines run into hundreds of millions. This regulatory pressure pushes toward more transparency, but it also slows innovation. Finding

the balance between user protection and economic actors' agility remains an unresolved challenge.

The return of geopolitical conflicts accelerates the militarization of computing. AI embedded in drones, ground robots, or autonomous weapon systems is becoming an operational reality. Boston Dynamics may promise not to weaponize its robots, but others have no such scruples. The Spot robot is being tested by several armies for reconnaissance missions in hostile zones. Variants equipped with CBRN sensors detect chemical, biological, or radiological threats. Armed prototypes are already circulating in certain countries. China, the United States, Russia, and Israel are all developing AI-driven weapon systems. The decision to fire can already be made by an algorithm in certain configurations. International treaties struggle to keep up. The ban on lethal autonomous weapons has been discussed for years without result. Each state fears falling behind its adversaries. This race exhausts resources, but no one dares to slow down. Computing is becoming a sovereignty issue on par with energy or access to raw materials. Semiconductors, data, and AI algorithms are now strategic weapons. Sanctions and export restrictions are multiplying. The fragmentation of the digital world into rival geopolitical blocs is intensifying.

What technologies will need to be invented to deal with all this? Post-quantum encryption is under development, but its large-scale deployment will take years. Zero-trust architectures, where no actor is considered trustworthy by default, are being deployed slowly. Confidential computing, which protects data even during processing, is progressing. But each new layer of security adds complexity and slows systems. Finding the right balance between protection and performance remains more of an art than a science. Formal verification technologies, which mathematically prove that a program does exactly what is expected of it, could become more widespread. Today confined to critical systems like aeronautics or nuclear power, they could extend to other domains if the tools become more accessible. Quantum computing itself could provide solutions in cryptography or simulation of complex systems. But it won't be a magic wand. Each advance opens new problems.

The history of computing has taught us one thing: ruptures are never the ones we expect. No one predicted the scale of the web in the 1990s. No one anticipated the explosion of smartphones after 2007. No one really knows which technology will disrupt the next decade. The bets are open. But one certainty remains: computing will continue to reinvent itself, to surprise us, to force us to rethink our ways of working, communicating, and thinking. This book tells a piece of a story that never ends. Every period is a new beginning. The technologies we will use in ten years may already exist, somewhere in a laboratory, in the form of a prototype that

no one yet takes seriously. Or perhaps they don't exist yet, waiting for a brilliant idea to cross the mind of a researcher or engineer somewhere in the world.

But what freedoms are we willing to give up for more security or comfort?

Bibliography

[1] Olivier Cogis, Jérôme Palaysi, and Richard Terrat. Les machines : architecture des ordinateurs - d'une introduction historique à la définition d'une machine virtuelle universelle -. working paper or preprint, January 2020.

[2] Lloyd Strickland and Harry R. Lewis. *Leibniz on Binary: The Invention of Computer Arithmetic*. The MIT Press, Cambridge, MA, USA, 2022.

[3] A. Glaser. *History of Binary and Other Nondecimal Numeration*. Tomash, 1981.

[4] Kenneth C. Smith. A multiple valued logic: A tutorial and appreciation. *Computer*, 21(4):17–27, April 1988.

[5] Saul Engelbourg. Herman Hollerith: Forgotten Giant of Information Processing. By Geoffrey D. Austrian. (New York: Columbia University Press, 1982. xvii + 418 pp. Illustrations, notes, selected bibliography, and index. $19.95.). *Journal of American History*, 69(4):1008–1009, 03 1983.

[6] Lars Heide. *Punched-cards Systems and the Early Information Explosion, 1880-1945*. Studies in Industry and Society. Johns Hopkins University Press, United States, 2009.

[7] Maurice V. Wilkes. Charles babbage—the great uncle of computing? *Commun. ACM*, 35(3):15–ff., March 1992.

[8] George Boole. *The mathematical analysis of logic: being an essay towards a calculus of deductive reasoning*. Macmillan, Barclay, & Macmillan, Cambridge, 1847.

[9] Janet Heine Barnett. Origins of boolean algebra in the logic of classes: George boole, john venn and c. s. peirce. 2011.

[10] Seidner Florian. History of the boolean algebra. 2012.

[11] Michael Schroeder. A brief history of the notation of boole's algebra. 1997.

[12] George Strawn and George Strawn. Masterminds of punched card data processing: Herman hollerith and john billings. *IT Professional*, 25(6):90–93, November 2023.

[13] William F. Luebbert and Geoffrey Austrian. *Hollerith, Herman*, page 785–786. John Wiley and Sons Ltd., GBR, 2003.

[14] Zachary S. Bischof, Romain Fontugne, and Fabián E. Bustamante. Untangling the world-wide mesh of undersea cables. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*, HotNets '18, page 78–84, New York, NY, USA, 2018. Association for Computing Machinery.

[15] Maurice V. Wilkes. *Differential analyzer*, page 537–539. John Wiley and Sons Ltd., GBR, 2003.

[16] Larry Owens. *Vannevar Bush and the differential analyzer: the text and context of an early computer*, page 3–38. Academic Press Professional, Inc., USA, 1991.

[17] V. Bush. The differential analyzer. a new machine for solving differential equations. *Journal of the Franklin Institute*, 212(4):447–488, 1931.

[18] Vannevar Bush. As we may think. *Interactions*, 3(2):35–46, March 1996.

[19] Frank B. Cannonito. The gödel incompleteness theorem and intelligent machines. In *Proceedings of the May 1-3, 1962, Spring Joint Computer Conference*, AIEE-IRE '62 (Spring), page 71–77, New York, NY, USA, 1962. Association for Computing Machinery.

[20] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1936.

[21] Charles Petzold. *The Annotated Turing: A Guided Tour Through Alan Turing's Historic Paper on Computability and the Turing Machine*. Wiley Publishing, 2008.

[22] Arthur Evans. The lambda calculus and its relation to programming languages. In *Proceedings of the ACM Annual Conference - Volume 2*, ACM '72, page 714–716, New York, NY, USA, 1972. Association for Computing Machinery.

[23] Claude E. Shannon and Warren Weaver. *A Mathematical Theory of Communication*. University of Illinois Press, USA, 1963.

[24] Henry S. Tropp. *Stibitz, George Robert*, page 1687–1689. John Wiley and Sons Ltd., GBR, 2003.

[25] Raúl Rojas. The Z1: architecture and algorithms of konrad zuse's first computer. *CoRR*, abs/1406.1886, 2014.

[26] AARON STRAUSS. Eniac: The first computer. *The Mathematics Teacher*, 69(1):66–72, 1976.

[27] Scott McCartney, J. Ross Macdonald, and Harvey G. Cragon. ENIAC: The Triumphs and Tragedies of the World's First Computer. *Physics Today*, 53(7):58–59, 07 2000.

[28] M. Riordan and L. Hoddeson. *Crystal Fire: The Birth of the Information Age*. The Sloan technology series. Norton, 1997.

[29] W. Shockley. *Electrons and Holes in Semiconductors: With Applications to Transistor Electronics*. Bell Telephone Laboratories series. Van Nostrand, 1950.

[30] John Gilbey. Biography: The abc of computing. *Nature*, 468:760–761, 2010.

[31] Kiril Boyanov. John vincent atanasoff: the inventor of the first electronic digital computing. 2003.

[32] Tara H. Abraham. (physio)logical circuits: The intellectual origins of the mcculloch–pitts neural networks. *Journal of the History of the Behavioral Sciences*, 38(1):3–25, 2002.

[33] Jack Cowan. Neural networks: The early days. In D. Touretzky, editor, *Advances in Neural Information Processing Systems*, volume 2. Morgan-Kaufmann, 1989.

[34] Thomas Haigh and Mark Priestley. Colossus and programmability. *IEEE Ann. Hist. Comput.*, 40(4):5–27, 2018.

[35] B. Randell. *Colossus: Godfather of the Computer*, pages 349–354. Springer Berlin Heidelberg, Berlin, Heidelberg, 1982.

[36] Christian Robert and George Casella. A short history of markov chain monte carlo: Subjective recollections from incomplete data. *Statistical Science*, 26(1), February 2011.

[37] Christian Robert and George Casella. A History of Markov Chain Monte Carlo–Subjective Recollections from Incomplete Data–. working paper or preprint, August 2008.

[38] F.C. Williams and T. Kilburn. A storage system for use with binary-digital computing machines. *Proceedings of the IEE - Part II: Power Engineering*, 96:183–200, 1949.

[39] Andrew V. Haeff. The memory tube and its application to electronic computation. 3(24):281–286, October 1948.

[40] Mathieu Triclot. Norbert Wiener's politics and the history of cybernetics. In *2nd International Conference of the European Society for the History of Science: "The Global and the local : The history of science and the cultural integration of Europe"*, pages 763–766, Cracovie, Poland, September 2006.

[41] N. Wiener and R.R. Kline. *Norbert Wiener-A Life in Cybernetics: Ex-Prodigy: My Childhood and Youth and I Am a Mathematician: The Later Life of a Prodigy.* MIT Press, 2018.

[42] David Sharp. Manchester baby simulator user guide. 2001.

[43] S. H. Lavington. *The Manchester Mark 1.* Springer Berlin Heidelberg, Manchester, England, 1975.

[44] Claude Elwood Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27:379–423, 1948.

[45] James V. Stone. Information theory: A tutorial introduction. *CoRR*, abs/1802.05968, 2018.

[46] William F. Schmitt. The UNIVAC Short Code. *Annals of the History of Computing*, 10(1):7–18, 1988.

[47] M V Wilkes and W Renwick. The edsac - an electronic calculating machine. *Journal of Scientific Instruments*, 26(12):385, dec 1949.

[48] Maxime Lamothe, Yann-Gaël Guéhéneuc, and Weiyi Shang. A systematic review of api evolution literature. *ACM Comput. Surv.*, 54(8), October 2021.

[49] Maurice Vincent Wilkes, David J. Wheeler, Stanley Gill, and Fernando J. Corbató. The preparation of programs for an electronic digital computer. 1958.

[50] Ira W. Cotton and Frank S. Greatorex. Data structures and techniques for remote computer graphics. In *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*, AFIPS '68 (Fall, part I), page 533–544, New York, NY, USA, 1968. Association for Computing Machinery.

[51] C. J. Date and E. F. Codd. The relational and network approaches: Comparison of the application programming interfaces. In *Proceedings of the 1974 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control: Data Models: Data-Structure-Set versus Relational*, SIGFIDET '74, page 83–113, New York, NY, USA, 1975. Association for Computing Machinery.

[52] Roy Thomas Fielding and Richard N. Taylor. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000. AAI9980887.

[53] R. R. Everett. The whirlwind i computer. *Electrical Engineering*, 71(8):681–686, 1952.

[54] F. E. Hamilton and E. C. Kubie. The ibm magnetic drum calculator type 650. *J. ACM*, 1(1):13–20, January 1954.

[55] D. D. Majumder. A magnetic drum digital storage system. *IETE Journal of Research*, 7, 1961.

[56] Ravi Nair. Evolution of memory architecture. *Proceedings of the IEEE*, 103(8):1331–1345, 2015.

[57] Lee Revens. The first 25 year: Acm 1947-1962. *Commun. ACM*, 30(10):860–865, October 1987.

[58] Joel E. Cohen. *A Model of Simple Competition*. Harvard University Press, Cambridge, MA and London, England, 1966.

[59] R. W. Hamming. Error detecting and error correcting codes. *Bell System Technical Journal*, 29(2):147–160, 1950.

[60] P. Kuttner. The rope memory: a permanent storage device. In *Proceedings of the November 12-14, 1963, Fall Joint Computer Conference*, AFIPS '63 (Fall), page 45–57, New York, NY, USA, 1963. Association for Computing Machinery.

[61] Lars Heide. Making Business of a Revolutionary New Technology: The Eckert-Mauchly Company, 1945–1951. In John Impagliazzo, Per Lundin, and Benkt Wangler, editors, *3rd History of Nordic Computing (HiNC)*, volume AICT-350 of *History of Nordic Computing 3*, pages 207–214, Stockholm, Sweden, October 2010. Springer. Part 6: The Making of a Nordic Computing Industry.

[62] Nancy Stern. The binac: A case study in the history of technology. *IEEE Ann. Hist. Comput.*, 1(1):9–20, July 1979.

[63] Kazuo Goda and Masaru Kitsuregawa. The history of storage systems. *Proceedings of the IEEE*, 100(Special Centennial Issue):1433–1440, 2012.

[64] *The Electrical World*. Number vol. 11 à 12. W.J. Johnston, 1888.

[65] Frank Land. Remembering LEO. In Arthur Tatnall, editor, *Reflections on the History of Computing : Preserving Memories and Sharing Stories*, volume AICT-387 of *IFIP Advances in Information and Communication Technology (SURVEY)*, pages 22–42. Springer, 2012.

[66] David T. Caminer, John Aris, Peter Hermon, and Frank Land. LEO: the incredible story of the world's first business computer. *J. Database Manag.*, 10(1):44, 1999.

[67] Kurt W. Beyer. *Grace Hopper and the Invention of the Information Age*. The MIT Press, 2009.

[68] Harvard University. *A Manual of Operation for the Automatic Sequence Controlled Calculator*. MIT Press, Cambridge, MA, USA, 1985.

[69] C. J. Bashe, W. Buchholz, G. V. Hawkins, J. J. Ingram, and N. Rochester. The architecture of ibm's early computers. *IBM Journal of Research and Development*, 25(5):363–376, 1981.

[70] Stephen Kaisler. *First Generation Mainframes: The IBM 700 Series*. Cambridge Scholars Publishing, March 2018.

[71] Donald E. Knuth. The ibm 650: An appreciation from the field. *Annals of the History of Computing*, 8(1):50–55, 1986.

[72] Donald E. Knuth. *The Art of Computer Programming Volumes 1-3 Boxed Set*. Addison-Wesley Longman Publishing Co., Inc., USA, 2nd edition, 1998.

[73] Smruti R. Sarangi. *Basic Computer Architecture*. White Falcon Publishing, 1st edition edition.

[74] J. W. Backus. *The Fortran Automatic Coding System for the IBM 704 EDPM*. January 1956.

[75] Albert Hoagland. History of magnetic disk storage based on perpendicular magnetic recording. *Magnetics, IEEE Transactions on*, 39:1871 – 1875, 08 2003.

[76] Marvin L. Minsky, Nathaniel Rochester, Claude E. Shannon, and John McCarthy. A proposal for the dartmouth summer research project on artificial intelligence, august 31, 1955. *AI Magazine*, 27(4):12, 2006.

[77] John A. McKenzie. *TX-O Computer History*. MIT, October 1974.

[78] V. A. Newell and F. M. Tonge. *An Introduction to Information Processing Language V*. RAND Corporation, Santa Monica, CA, 1960.

[79] Phillippe Dreyfuss. System design of the gamma 60. In *Proceedings of the May 6-8, 1958, Western Joint Computer Conference: Contrasts in Computers*, IRE-ACM-AIEE '58 (Western), page 130–133, New York, NY, USA, 1958. Association for Computing Machinery.

[80] R. J. Creasy. The origin of the vm/370 time-sharing system. *IBM J. Res. Dev.*, 25(5):483–490, September 1981.

[81] John Backus. *The history of Fortran I, II, and III*, page 25–74. Association for Computing Machinery, New York, NY, USA, 1978.

[82] Peter Naur. Successes and failures of the algol effort. *ALGOL Bull.*, (28):58–62, July 1968.

[83] John McCarthy. *History of LISP*, page 173–185. Association for Computing Machinery, New York, NY, USA, 1978.

[84] J.S. Kilby. Invention of the integrated circuit. *IEEE Transactions on Electron Devices*, 23(7):648–654, 1976.

[85] Stuart Bennett. Control and the digital computer: The early years. *IFAC Proceedings Volumes*, 35(1):237–242, 2002. 15th IFAC World Congress.

[86] Charlie Bachman, Dan Fogal, Jack Jones, Dick Kerr, Chuck Greenberger, David Black, Mary Hollis, Greg Dillon, Jim Sweeney, Steve Wright, Bob Bemer, Grace Murray Hopper, Bill Randall, Brian Reynolds, Danny, Bill Olle, John Young, Warren Simmons, Bob Grace, Howard Bramberg, Marty Greenville, Stan Epwood, Anaheim Bushed, Tax Metaxides, Bill McGee, John Gosden, Marty Greenfield, Jonas Raven, George Mann, Goodrich Hubert, Ron Hamm, Jim Manner, Herb Beta, Bill Keating, Peg Harper, Dick Schubert, and Herb Manative. Conference on data systems languages (codasyl) interview: May 27-28, 1969. In *Computer Oral History Collection*, page 26–es, USA, 1999. Smithsonian Institution Press.

[87] Gio Wiederhold and John McCarthy. Arthur samuel: Pioneer in machine learning. *IBM Journal of Research and Development*, 36(3):329–331, 1992.

[88] John W. Backus. The syntax and semantics of the proposed international algebraic language of the zurich ACM-GAMM conference. In *Information Processing, Proceedings of the 1st International Conference on Information Processing, UNESCO, Paris 15-20 June 1959*, pages 125–131. UNESCO (Paris), 1959.

[89] Robert Garner and Rick Dill. The legendary ibm 1401 data processing system. *IEEE Solid-State Circuits Magazine*, 2(1):28–39, 2010.

[90] American Standards Association. Sectional Committee on Computers, X3 Information Processing, and Business Equipment Manufacturers Association (U.S.). *American Standard Code for Information Interchange: Sponsor: Business Equipment Manufacturers Association. Approved June 17, 1963.* American Standards Association Incorporated, 1963.

[91] Chuck Boyer. The 360 revolution. *IBM*, 2004.

[92] Craig Partridge. The technical development of internet e-mail. *IEEE Annals of the History of Computing*, 30(2):3–29, 2008.

[93] B.W. Kernighan. *Unix: A History and a Memoir*. Independently Published, 2020.

[94] Robert Sedgewick. The analysis of quicksort programs. *Acta Inf.*, 7(4):327–355, December 1977.

[95] Laila Khreisat. Quicksort a historical perspective and empirical study. *International Journal of Computer Science and Network Security*, December 2007.

[96] Roger L. Wainwright. A class of sorting algorithms based on quicksort. *Commun. ACM*, 28(4):396–402, April 1985.

[97] Jared Reyes. A system of communication between two computers using novel frequency shift keying techniques. *University of South Dakota*, 2024.

[98] Stewart Brand. SPACEWAR! - by Stewart Brand - Fanatic Life and Symbolic Death Among the Computer Bums. *Rolling stone*, 1972.

[99] Michel Élie and Philippe Picard. Unix et les systèmes ouverts dans Bull, avant l'Internet. *Cahiers d'histoire du Cnam*, vol.07 - 08(2):pp. 163–171, 2017.

[100] Ralph E. Griswold and Madge T. Griswold. *History of the Icon programming language*, page 599–624. Association for Computing Machinery, New York, NY, USA, 1996.

[101] Ralph E. Griswold. *A history of the SNOBOL programming languages*, page 601–645. Association for Computing Machinery, New York, NY, USA, 1978.

[102] Kenneth E. Iverson. *A programming language*. John Wiley & Sons, Inc., 1962.

[103] Ole-Johan Dahl and Kristen Nygaard. Simula: an algol-based simulation language. *Commun. ACM*, 9(9):671–678, September 1966.

[104] Max V. Mathews, Joan E. Miller, F. R. Moore, John R. Pierce, and J. C. Risset. *The Technology of Computer Music*. The MIT Press, 1969.

[105] M. V. Mathews. The digital computer as a musical instrument. *Science*, 142(3592):553–557, 1963.

[106] Ivan E. Sutherland. Sketchpad: a man-machine graphical communication system. In *Proceedings of the May 21-23, 1963, Spring Joint Computer Conference*, AFIPS '63 (Spring), page 329–346, New York, NY, USA, 1963. Association for Computing Machinery.

[107] George Radin. The early history and characteristics of pl/i. *SIGPLAN Not.*, 13(8):227–241, August 1978.

[108] American National Standards Institute. *American National Standard for information systems: programming languages, full BASIC: ANSI X3.113-1987*. Number 68-2 in FIPS PUB. American National Standards Institute, 1430 Broadway, New York, NY 10018, USA, 1987.

[109] Allison Randal. The ideal versus the real: Revisiting the history of virtual machines and containers. *ACM Comput. Surv.*, 53(1), February 2020.

[110] James E. Thornton. The cdc 6600 project. *Annals of the History of Computing*, 2(4):338–348, 1980.

[111] Elliott I. Organick. *The multics system: an examination of its structure*. MIT Press, Cambridge, MA, USA, 1972.

[112] J.W. Cooley, P.A.W. Lewis, and P.D. Welch. Historical notes on the fast fourier transform. *Proceedings of the IEEE*, 55(10):1675–1677, 1967.

[113] Cynthia Solomon, Brian Harvey, Ken Kahn, Henry Lieberman, Mark L. Miller, Margaret Minsky, Artemis Papert, and Brian Silverman. History of logo. *Proc. ACM Program. Lang.*, 4(HOPL), June 2020.

[114] Barbara Klein, Richard Alan Long, Kenneth Ray Blackman, Diane Lynne Goff, Stephen Paul Nathan, Moira McFadden Lanyi, Margaret M. Wilson, John Butterweck, and Sandra L. Sherrill. *An Introduction to IMS: Your Complete Guide to IBM Information Management System*. IBM Press, 2nd edition, 2012.

[115] Joseph Weizenbaum. Eliza — a computer program for the study of natural language communication between man and machine. *Commun. ACM*, 9(1):36–45, January 1966.

[116] L. H. Seawright and J. A. Kelch. An introduction to CP-67/CMS. *IBM Data Processing Division*, September 1968.

[117] V. Rajaraman. *Fundamentals of Computers*. Prentice Hall India Pvt., Limited, 2004.

[118] Manabu Hirano and Ryotaro Kobayashi. Ransmap: Open dataset of ransomware storage and memory access patterns for creating deep learning based ransomware detectors. *Computers & Security*, 150:104202, 2025.

[119] Douglas Engelbart. Interview, 2003.

[120] Douglas C. Engelbart and William K. English. A research center for augmenting human intellect. In *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*, AFIPS '68 (Fall, part I), page 395–410, New York, NY, USA, 1968. Association for Computing Machinery.

[121] Vint Cerf. How the internet came to be. January 1993.

[122] Steven Pemberton. The B programming language and environment. *CWI Newsletter*, (3):2–14, June 1984.

[123] Gerald. Luecke and Jack P. Mize. *Semiconductor memory design and application*. Texas Instruments Electronics Series. McGraw-Hill, New York, 1973.

[124] Steve Crocker. Host Software. RFC 1, April 1969.

[125] F. Baccelli, B. Blaszczyszyn, and P. Muhlethaler. An aloha protocol for multihop mobile wireless networks. *IEEE Transactions on Information Theory*, 52(2):421–436, 2006.

[126] Norman Abramson. The aloha system: another alternative for computer communications. In *Proceedings of the November 17-19, 1970, Fall Joint Computer Conference*, AFIPS '70 (Fall), page 281–285, New York, NY, USA, 1970. Association for Computing Machinery.

[127] Leo B. Brodie. *Thinking FORTH: a language and philosophy for solving problems*. Prentice-Hall, Inc., USA, 1984.

[128] Niklaus Wirth. 50 years of pascal. *Commun. ACM*, 64(3):39–41, February 2021.

[129] Gordon Bell and William D. Strecker. Computer structures: What have we learned from the pdp-11? *SIGARCH Comput. Archit. News*, 4(4):1–14, January 1976.

[130] Digital Equipment Corporation. *PDP-11/40 Processor Handbook*. Digital Equipment Corporation, 1972.

[131] Douglas Comer. The ubiquitous b-tree. *ACM Comput. Surv.*, 11(2):121–137, 1979.

[132] R. Bayer and E. McCreight. Organization and maintenance of large ordered indices. In *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, SIGFIDET '70, page 107–141, New York, NY, USA, 1970. Association for Computing Machinery.

[133] James Adkinson, Warren Dalziel, James Porter, Yan Rosenshteyn, and Herbert Thompson. International Business Machines corporation (IBM) 8 inch floppy disk drives oral history panel. *Computer History Museum*, page 38, 2005.

[134] B.W. Kernighan. *A Tutorial Introduction to the UNIX Text Editor [Ed]*. 1997.

[135] Dogan Ibrahim. *Microcontroller Projects in C for the 8051*. Newnes, London, England, April 2000.

[136] *Intel 8008 8-Bit Parallel Central Processing Unit*. Intel, 3065 Bowers Avenue, Santa Clara, California 95051, April 1972.

[137] Michael Betker, John Fernando, and Shaun Whalen. The history of the microprocessor. *Bell Labs Technical Journal*, 2:29 – 56, 09 2002.

[138] Federico Faggin, Marcian E. Hoff Jr., Stanley Mazor, and Masatoshi Shima. The history of the 4004. *IEEE Micro*, 16(6):10–20, December 1996.

[139] File Transfer Protocol. RFC 959, October 1985.

[140] Dennis M. Ritchie. *The development of the C programming language*, page 671–698. Association for Computing Machinery, New York, NY, USA, 1996.

[141] Alain Colmerauer and Philippe Roussel. *The birth of Prolog*, page 331–367. Association for Computing Machinery, New York, NY, USA, 1992.

[142] Alan C. Kay. *The early history of Smalltalk*, page 511–598. Association for Computing Machinery, New York, NY, USA, 1996.

[143] L. H. Seawright and R. A. MacKinnon. Vm/370: a study of multiplicity and usefulness. *IBM Syst. J.*, 18(1):4–17, March 1979.

[144] IEEE standard for ethernet. *IEEE Std 802.3-2022 (Revision of IEEE Std 802.3-2018)*, pages 1–7025, 2022.

[145] Brad A. Myers. A brief history of human-computer interaction technology. *Interactions*, 5(2):44–54, March 1998.

[146] D. Cooke, J. Urban, and S. Hamilton. Unix and beyond: An interview with ken thompson. *Computer*, 32(5):58–64, 1999.

[147] Diomidis Spinellis. A repository of unix history and evolution. *Empirical Software Engineering*, 22(3):1372–1404, Jun 2017.

[148] V. Cerf and R. Kahn. A protocol for packet network intercommunication. *IEEE Transactions on Communications*, 22(5):637–648, 1974.

[149] B. Cosell and D. Walden. Developing telnet's negotiated options. *IEEE Annals of the History of Computing*, 25(2):80–82, 2003.

[150] Donald D. Chamberlin, Morton M. Astrahan, Michael W. Blasgen, James N. Gray, W. Frank King, Bruce G. Lindsay, Raymond Lorie, James W. Mehl, Thomas G. Price, Franco Putzolu, Patricia Griffiths Selinger, Mario Schkolnick, Donald R. Slutz, Irving L. Traiger, Bradford W. Wade, and Robert A. Yost. A history and evaluation of system r. *Commun. ACM*, 24(10):632–646, October 1981.

[151] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.

[152] Gary Kildall. CP/M: A family of 8 and 16-bit operating systems. *BYTE*, June 1981.

[153] Donald D. Chamberlin and Raymond F. Boyce. Sequel: A structured english query language. In *Proceedings of the 1974 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, SIGFIDET '74, page 249–264, New York, NY, USA, 1974. Association for Computing Machinery.

[154] Stanley Mazor. Intel 8080 cpu chip development. *IEEE Ann. Hist. Comput.*, 29(2):70–73, April 2007.

[155] P J Horne. Implementation of an INTEL 8080 microprocessing development system using existing minicomputers. *IEEE Trans. Nucl. Sci.*, 24(3):1772–3, 1977.

[156] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

[157] H. Edward Roberts and William Yates. Altair 8800 The most powerful minicomputer project ever presented. *Popular Electronics*, pages 33–38, January 1975.

[158] Forrest M. Mims III. The tenth anniversary of the altair 8800. *Computers & Electronics*, pages 58–82, January 1985.

[159] Richard M. Russell. The cray-1 computer system. *Commun. ACM*, 21(1):63–72, January 1978.

[160] CERN Bulletin Issue No. 17/1988. April 1988.

[161] Valérie Schafer. X.25, une norme pour le transport de données. *Encyclopédie d'histoire numérique de l'Europe*, May 2021.

[162] Joseph Amankwah-Amoah. Competing technologies, competing forces: The rise and fall of the floppy disk, 1971–2010. *Technological Forecasting and Social Change*, 107:121–129, May 2016.

[163] D. A. Nowitz and M. E. Lesk. A dial-up network of UNIX systems. August 1978.

[164] Peter H. Salus. Distributing the news. *Usenix.org*, 40(4), August 2015.

[165] Mike Willegal. Steve Wozniak and the Apple I. December 2019.

[166] Apple Computer Company. Apple I Operation Manual. 1976.

[167] Jim Joyce. Interview with Bill Joy. *Unix Review*, August 1984.

[168] Christopher Espinosa. *APPLE II reference manual.* Apple Computer, 1979.

[169] PET User Group. *PET User Notes*, 2(1), 1979.

[170] Digital. 1977–1997... and beyond, Nothing Stops It! 1997.

[171] Steven Levy. *Crypto: How the Code Rebels Beat the Government–Saving Privacy in the Digital Age.* Penguin USA, 2001.

[172] Lewis Thompson. A short, BSD-specific, UNIX history. November 2003.

[173] Yukiharu SAMBE. Brief history of arcade video game display technologies. *Synthesiology*, 6(2):93–102, 2013.

[174] Barbara Beeton, Karl Berry, and David Walden. Tex: A branch of desktop publishing evolution, part 1. *IEEE Annals of the History of Computing*, 40:78–93, 07 2018.

[175] Barbara Beeton, Karl Berry, and David Walden. Tex: A branch in desktop publishing evolution, part 2. *IEEE Annals of the History of Computing*, 41(2):29–41, 2019.

[176] Tim Bergin. The origins of word processing software for personal computers: 1976-1985. *Annals of the History of Computing, IEEE*, 28:32–47, 11 2006.

[177] Jeffrey R. Yost. An interview with Seymour Rubinstein OH 391. *Charles Babbage Institute, Center for the History of Information Processing*, May 2004.

[178] John R. Hudson. Some notes on dBASE. June 2023.

[179] Jean Ichbiah. Ada: past, present, future. *Commun. ACM*, 27(10):990–997, October 1984.

[180] Michael Hauben and Ronda Hauben. *Netizens: on the history and impact of Usenet and the Internet*. IEEE Computer Society Press, Washington, DC, USA, 1997.

[181] Steven Bellovin. Netnews: The origin story. *IEEE Annals of the History of Computing*, PP:1–13, January 2024.

[182] Lazar Raković, Marton Sakal, and Veselin Pavlićević. Spreadsheets -how it started. *International Scientific Journal of Management Information Systems*, 9:9–14, December 2014.

[183] Martin Campbell-Kelly and Paul Ceruzzi. An interview with dan bricklin and bob frankston OH 402. *Charles Babbage Institute, Center for the History of Information Processing*, May 2004.

[184] Barry B. Brey. *The Intel microprocessors (4th ed.): 8086/8088, 80186/80188, 80286, 80386, 80486, Pentium, and Pentium Pro processor: architecture, programming, and interfacing.* Prentice-Hall, Inc., USA, 1997.

[185] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the smalltalk-80 system. In *Proceedings of the*

11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL '84, page 297–302, New York, NY, USA, 1984. Association for Computing Machinery.

[186] Nuno Garcia, Fabio Gil, Barbara Matos, Coulibaly Yahaya, Nuno Pombo, and Rossitza Goleva. Keyed user datagram protocol: Concepts and operation of an almost reliable connectionless transport protocol. *IEEE Access*, PP:1–1, February 2019.

[187] Gregg Williams. The Commodore VIC-20 Microcomputer: A low-cost, high-performance consumer computer. *BYTE*, pages 46–64, May 1981.

[188] Microsoft Corporation. *Microsoft XENIX Operating System OEM Directory*. Microsoft Corporation, Bellevue, Washington, May 1983. Document référence OEM00918.

[189] Microsoft Corporation. *Microsoft XENIX Operating System Version 3.0 Release Summary*. Microsoft Corporation, Bellevue, Washington, 1983.

[190] Robert B. Greenberg. The unix operating system and the xenix standard operating environment. *BYTE*, 6(6):248–264, June 1981.

[191] W. E. Pete Peterson. *Almost perfect: How a Bunch of Regular Guys Built WordPerfect Corporation*. Prima Lifestyles, USA, November 1993.

[192] Caralie Heinrichs and Emilie Vandal. Digital preservation guide: 3.5-inch floppy disks. *Preservation guide*, December 2019.

[193] Internet Control Message Protocol. RFC 792, September 1981.

[194] Sol Libes. The origins of MS-DOS. *Micro/Systems Journal*, July 1986.

[195] Tim Paterson. An inside look at MS-DOS. *BYTE*, June 1983.

[196] Review BBC Micro. *Practical Computing*, January 1982.

[197] The IBM Personal Computer. *BYTE*, 7(1), January 1982.

[198] T.B. Henderson. *The Osborne Portable Computer*. Que on systems. Que Corporation, 1982.

[199] Mukesh Sahu and Jeebananda Panda. A time efficient approach to data compression for lzw algorithm. In *2023 Annual International Conference on Emerging Research Areas: International Conference on Intelligent Systems (AICERA/ICIS)*, pages 1–3, 2023.

[200] David E. Weisberg. *The Engineering Design Revolution - CAD History*. 2008.

[201] Devon Shapiro, Merritt Patridge, Erin McCafferty, Hillary Herstin, Smita Gupta, and Crystal Leveillee. *The Invention of Compact Discs*. EIS Main Project, November 2012.

[202] Al Hartmann. iAPX286 compiler writer's guide. *Intel Corporation*, May 1983.

[203] Roberto Dillon. *Ready: A Commodore 64 Retrospective*. Springer, January 2015.

[204] Nick Montfort, Patsy Baudoin, John Bell, Ian Bogost, Jeremy Douglass, Mark C. Marino, Michael Mateas, Casey Reas, Mark Sample, and Noah Vawter. *10 PRINT CHR$(205.5+RND(1)); : GOTO 10*. The MIT Press, 11 2012.

[205] Rodney Dale. *The Sinclair Story*. Duckworth, 1985.

[206] Jean-Noël Lafargue. Archéologie d'un carton de l'Informatique pour tous. *Cahiers d'histoire du Cnam*, vol.15(1):pp. 97–131, 2021.

[207] Brigitte Morin. La saga de Thomson. *Science & Vie Micro*, (61), May 1989.

[208] Simson Garfinkel and Eugene H. Spafford. In memoriam: Charles m. geschke (1939–2021). *Commun. ACM*, 64(7):22, June 2021.

[209] Bill Calkins. *Solaris 8: System Administrator Certification: Exams 310-011 and 310-012*. Que Corp., USA, 2001.

[210] *Soviet bloc computers direct descendant of western technologies*. CIA, June 1989.

[211] Jules H. Gilder. The Coleco Adam: system review. *BYTE*, pages 206–220, April 1984.

[212] William Arnold. *COMPAQ Portable Computer: Use, Applications and Basic*. Harcourt Brace College Publishers, 1984.

[213] Sam Williams. *Free as in freedom.* O'Reilly, Beijing [u.a.], 1. ed. edition, 2002.

[214] Bjarne Stroustrup. *A history of C++: 1979–1991.* AT&T Bell Laboratories, 1996.

[215] Neil Smyth. *Objective-C 2.0 Essentials.* 3rd edition, 2012.

[216] P. Mockapetris and K. J. Dunlap. Development of the domain name system. In *Symposium Proceedings on Communications Architectures and Protocols*, SIGCOMM '88, page 123–133, New York, NY, USA, 1988. Association for Computing Machinery.

[217] P. Mockapetris and K. J. Dunlap. Development of the domain name system. *SIGCOMM Comput. Commun. Rev.*, 18(4):123–133, August 1988.

[218] Cisco. *Internetworking Technologies Handbook, Fourth Edition.* Cisco Press, 2003.

[219] Jonathan Brossard. SMB : Sharing more than your files... In *Blackhat USA*, Las Vegas, United States, July 2015. Blackhat.

[220] William Aspray. Oral History of Mitch Kapor. *Computer History Museum*, November 2004.

[221] Grady Booch. Oral History of Charles Simonyi. *Computer History Museum*, February 2008.

[222] O.W. Linzmayer. *Apple Confidential 2.0: The Definitive History of the World's Most Colorful Company.* No Starch Press Series. No Starch Press, 2004.

[223] Bruce Daniels. The architecture of the lisa- personal computer. *Proceedings of the IEEE*, 72:331 – 341, April 1984.

[224] Jonathan Pearce and Graham Bland. *MSX an introduction.* Century Communications Ltd, London, 1984.

[225] Jonathan Haworth. *Oric - The story so far.* Cambridge, November 1992.

[226] Paul Weissmann. *OpenPA, The book of PA-RISC.* Bonn, Germany, March 2024.

[227] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces.* CreateSpace Independent Publishing Platform, North Charleston, SC, USA, 2018.

[228] Post Office Protocol: Version 3: Extended service offerings. RFC 1082, November 1988.

[229] Microsoft Corporation. Post Office Protocol Version 3 (POP3) Extensions. June 2016.

[230] Ivor Spital. Sir Alan calls it a day at Amstrad. *The Amstrad Times*, June 2008.

[231] Fred Guterl. Design case history: Apple's macintosh: A small team of little-known designers, challenged to produce a low-cost, exceptionally easy-to-use personal computer, turns out a technical milestone. *IEEE Spectrum*, 21:34–43, 1984.

[232] David Kummer and Tom Moorman. Case study: Pcjr: Misreading the market. *IEEE Spectrum*, 24(5):30–31, 1987.

[233] Stefan Monnier and Michael Sperber. Evolution of emacs lisp. *Proc. ACM Program. Lang.*, 4(HOPL), June 2020.

[234] Patrick Davison. Because of the pixels: On the history, form, and influence of ms paint. *Journal of Visual Culture*, 13:275–297, 12 2014.

[235] Brigitte Morin. La saga de Thomson. *Science & Vie Micro*, 61, May 1989.

[236] Robert W. Scheifler and Jim Gettys. The X Window System. *ACM Trans. Graph.*, 5(2):79–109, April 1986.

[237] Hamira Shaikh, Mosina Shaikh, Shahid Shaikh, Aatif Qureshi, Farman Khan, Viren Patel, Shakila Shaikh, and Shiburaj Pappu. Closer look at the new processor: 80386. *International Journal of Computer Sciences and Engineering*, 7:48–51, 10 2019.

[238] D.L. Mills. Internet time synchronization: the network time protocol. *IEEE Transactions on Communications*, 39(10):1482–1493, 1991.

[239] P. Sestoft. *Spreadsheet Technology.* IT University technical report series. IT University of Copenhagen, 2011.

[240] Lykourgos G. Zacharogeorgas. *Microsoft Windows: The evolution of a revolutionary product.* 2005.

[241] John Cook, Martin Croft, Christina Erskine, Alan Fitch, Mike Gerrard, John Lettice, and Max Phillips. *The Complete Guide to Amstrad PCW 8256 & 8512*, volume 1. Roger Munford, 1986.

[242] Commodore Amiga Inc. *Amiga Hardware Reference Manual*. Addison-Wesley Publishing Company, 1989.

[243] Suzanne Crocker. Oral History of Paul Brainerd. *Computer History Museum*, May 2006.

[244] IBM. *Additional aix security tools on ibm eserver pseries, ibm rs/6000, and sp/cluster*. IBM Corp., USA, 2000.

[245] Bertrand Meyer. *Eiffel: the language*. Prentice-Hall, Inc., USA, 1992.

[246] Joe Armstrong. A history of erlang. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, page 6–1–6–26, New York, NY, USA, 2007. Association for Computing Machinery.

[247] D. L. Mills and H. Braun. The nsfnet backbone network. In *Proceedings of the ACM Workshop on Frontiers in Computer Communications Technology*, SIGCOMM '87, page 191–196, New York, NY, USA, 1987. Association for Computing Machinery.

[248] D. L. Mills and H. Braun. The nsfnet backbone network. *SIGCOMM Comput. Commun. Rev.*, 17(5):191–196, August 1987.

[249] Robert Elliott and Mallikarjun Chadalapaka. Small Computer Systems Interface (SCSI) Command Ordering Considerations with iSCSI. RFC 3783, May 2004.

[250] Chet Ramey. Bash – The GNU shell. *The Linux Journal*, August 1994.

[251] Mike Battilana. The GIF Controversy: A Software Developer's Perspective. January 1995.

[252] Michael Stevenson. Having it both ways: Larry Wall, Perl and the technology and culture of the early web. *Internet Histories*, 2(3-4):264–280, 2018.

[253] David Ungar and Randall B. Smith. Self: The power of simplicity. In *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*, OOPSLA '87, page 227–242, New York, NY, USA, 1987. Association for Computing Machinery.

[254] David Ungar and Randall B. Smith. Self: The power of simplicity. *SIGPLAN Not.*, 22(12):227–242, December 1987.

[255] John Ousterhout. History of tcl. Tcl Website.

[256] Acorn Archimedes. *Personal Computer World*, pages 98–104, August 1987.

[257] Archie RISC OS. *Personal Computer World*, pages 152–155, January 1989.

[258] Benoît Georges. 1984-2024: la folle épopée du Mac d'Apple. *Les Echos*, January 2024.

[259] Jimmy Maher. *The Future Was Here: The Commodore Amiga*. The MIT Press, 2012.

[260] R. L. Cook, F. L. Rawson, III, J. A. Tunkel, and R. L. Williams. Writing an operating system/2 application. *IBM Systems Journal*, 27(2):134–157, 1988.

[261] Unicode Consortium, editor. *The Unicode Standard, Version 4.0*. Addison-Wesley Longman Publishing Co., Inc., USA, 2003.

[262] Jacques Morleghem. *Windows 2 et 386 facile*. 1989.

[263] David J. Murray. Frame technology and framemaker. *IEEE Ann. Hist. Comput.*, 41(3):62–74, July 2019.

[264] Q.G. Schmierer and A.H. Wottreng. Ibm as/400 processor architecture and design methodology. In *[1991 Proceedings] IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 440–443, 1991.

[265] Stijn Peeters. *The IRC Paradigm: An Alternative Approach for Alternative Social Networks*. May 2013.

[266] Daniel Stenberg. History of IRC.

[267] ibiblio. Gulf War IRC logs, January 1991.

[268] 2meta.com. Historic Chat Logs, January 1991.

[269] David Bozak. Soviet Coup Archive, August 1991.

[270] Rene Brogatzki. SNMP - Simple Network Management Protocol. Seminar Innovative Internettechnologien und Mobilkommunikation SS2009.

[271] PKWARE Inc. Technical Note – ZIP File Capacities.

[272] Barry B. Brey. *The Intel 32-Bit Microprocessors: 80386, 80486, and Pentium Microprocessors*. Prentice Hall PTR, USA, 1st edition, 1994.

[273] Tim Berners-Lee. Information management: A proposal. May 1989.

[274] Dr.S.Narayanan, Dr.C.Sabarigirinathan, Dr.K.Vinayagavel, and Dr.D.Deepiha. MS Office – Computer Application. *International Journal of Scientific Engineering and Applied Science*, 4, September 2018.

[275] Apple Computer Inc. Apple Computer, Inc. agrees to acquire NeXT Software Inc. *Press release*, December 1996.

[276] Jeffe Schewe. 10 years of Photoshop. *PEI Mag*, February 2000.

[277] IBM. *User's reference for PS/1 computer*. International Business Machines Corporation, April 1990.

[278] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of haskell: being lazy with class. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, page 12–1–12–55, New York, NY, USA, 2007. Association for Computing Machinery.

[279] Michael Stevenson and Anne Helmond. The historical trajectories of algorithmic techniques: an interview with bernhard rieder. *Internet Histories*, 4:1–10, February 2020.

[280] Mark P. McCahill and Farhad X. Anklesaria. *Evolution of Internet Gopher*, pages 235–246. Springer Berlin Heidelberg, Berlin, Heidelberg, 1996.

[281] W. J. Buchanan. *PCMCIA (PC Card)*, pages 563–570. Springer US, Boston, MA, 2004.

[282] Rassol Raissi. *The Theory Behind Mp3*. December 2002.

[283] Simon Whibley. WAV Format Preservation Assessment. *British Library*, January 2016.

[284] Charles Severance. Guido van rossum: The early years of python. *Computer*, 48(2):7–9, 2015.

[285] S.N. Bokhari. The linux operating system. *Computer*, 28(8):74–79, 1995.

[286] D. Clark. Visual basic and the future of web-application development. *Computer*, 34(8):16–18, 2001.

[287] E. McLellan. The alpha axp architecture and 21064 processor. *IEEE Micro*, 13(3):36–47, 1993.

[288] R. Neelamani, R. de Queiroz, Zhigang Fan, S. Dash, and R.G. Baraniuk. Jpeg compression history estimation for color images. *IEEE Transactions on Image Processing*, 15(6):1365–1378, 2006.

[289] Site web IBM. IBM ThinkPad history.

[290] Dave Shreiner and The Khronos OpenGL ARB Working Group. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Versions 3.0 and 3.1.* Addison-Wesley Professional, 7th edition, 2009.

[291] Adobe Systems. *Portable Document Format Reference Manual.* APL Series. Addison-Wesley Publishing Company, 1993.

[292] Sugandha Sharma, Krishna Nand Patel, and Aashish Siddhath Jha. Cryptography using blowfish algorithm. In *2021 3rd International Conference on Advances in Computing, Communication Control and Networking (ICAC3N)*, pages 1375–1377, 2021.

[293] Chenyuan Kou and F. Springsteel. The security mechanism in the world wide web (www) and the common gateway interface (cgi). example of central police university entrance examination system. In *Proceedings IEEE 31st Annual 1997 International Carnahan Conference on Security Technology*, pages 114–119, 1997.

[294] Annelise Anderson. *FreeBSD: an open-source operating system for your personal computer.* The Bit Tree Press, Portola Valley, CA, USA, 2nd edition, 2001.

[295] B.B. Brey. *The Intel Microprocessors: 8086/8088, 80186/80188, 80286, 80386, 80486, Pentium, Pentium Pro Processor, Pentium II, Pentium III, Pentium 4, and Core2 with 64-bit Extensions : Architecture, Programming, and Interfacing.* Pearson Prentice Hall, 2009.

[296] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. The evolution of lua. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, page 2–1–2–26, New York, NY, USA, 2007. Association for Computing Machinery.

[297] John M. Chambers. S, r, and data science. *Proc. ACM Program. Lang.*, 4(HOPL), June 2020.

[298] Timothy Rooney. *Dynamic Host Configuration Protocol (DHCP)*, pages 53–68. Wiley, 2010.

[299] R.J. Vetter, C. Spell, and C. Ward. Mosaic and the world wide web. *Computer*, 27(10):49–57, 1994.

[300] Stephen A. Thomas. *SSL and TLS Essentials: Securing the Web with CD-ROM*. John Wiley & Sons, Inc., USA, 2000.

[301] Robert Cowart. *Windows NT 3.51, with CD-ROM (Unleashed)*. Macmillan Publishing Co., Inc., USA, 3rd edition, 1996.

[302] Jacob R. Lorch and Alan Jay Smith. Apple macintosh's energy consumption. *IEEE Micro*, 18(6):54–63, November 1998.

[303] H. Iwasaki. Cd-rewritable and future disc technology. In *1997 Optical Data Storage Topical Meeting ODS Conference Digest*, pages 9–10, 1997.

[304] David E. Kalish. IBM makes a fashion statement with new aptiva s series. *The Washington Post*, September 1996.

[305] J. Charles. Open source: Netscape pops the hood [in the news]. *IEEE Software*, 15(4):79–82, 1998.

[306] Oreste Signore. W3c technologies: a key for interoperability. 01 2003.

[307] Karen Angel. *Inside Yahoo!: Reinvention and the Road Ahead*. John Wiley & Sons, 2002.

[308] Bernard J. Jansen, Amanda Spink, and Jan Pedersen. A temporal comparison of Altavista Web Searching: Research Articles. *J. Am. Soc. Inf. Sci. Technol.*, 56(6):559–570, April 2005.

[309] Charles Severance. The apache software foundation: Brian behlendorf. *Computer*, 45(10):8–9, 2012.

[310] Martin C. Brown and Cherie Plumlee. *Beos: Porting UNIX Applications*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1998.

[311] Y. Tsunoda, R. Imura, M. Takahashi, and T. Arai. Present and future of dvd technology. In *1996. Digest of Technical Papers., International Conference on Consumer Electronics*, pages 356–, 1996.

[312] Zeqian Shen and Neel Sundaresan. ebay: an e-commerce marketplace as a complex network. In *Proceedings of the Fourth ACM International Conference on Web Search and Data Mining*, WSDM '11, page 655–664, New York, NY, USA, 2011. Association for Computing Machinery.

[313] John Nickolls and William J. Dally. The gpu computing era. *IEEE Micro*, 30(2):56–69, 2010.

[314] Prashanta Das and Ganesh Deka. *History and Evolution of GPU Architecture*, pages 109–135. 01 2016.

[315] A. Van Hoff. The case for java as a programming language. *IEEE Internet Computing*, 1(1):51–56, 1997.

[316] Allen Wirfs-Brock and Brendan Eich. Javascript: the first 20 years. *Proc. ACM Program. Lang.*, 4(HOPL), June 2020.

[317] Mark Hills. Evolution of dynamic feature usage in php. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 525–529, 2015.

[318] Robert J. Sheehan. Teaching operating systems with ruby. In *Proceedings of the 12th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, ITiCSE '07, page 38–42, New York, NY, USA, 2007. Association for Computing Machinery.

[319] Robert J. Sheehan. Teaching operating systems with ruby. *SIGCSE Bull.*, 39(3):38–42, June 2007.

[320] Daniel J. Barrett, Richard E. Silverman, and Robert G. Byrnes. *SSH, the Secure Shell: The Definitive Guide*. O'Reilly Media, Inc., 2005.

[321] Heike Jurzik. Surfing the shell. *Linux Magazine*, (54):84–87, May 2005.

[322] Sasha Pachev. *Understanding MySQL Internals*. O'Reilly Media, Inc., 2007.

[323] Dries Buytaert. The history of MySQL AB. *Planet MySQL*, March 2010.

[324] Elliot Jaffe and Scott Kirkpatrick. Architecture of the internet archive. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, SYSTOR '09, New York, NY, USA, 2009. Association for Computing Machinery.

[325] Jessah Foulk. Oral history interview with Rob Glaser, August 13, 2002 – Museum of History & Industry. *Speaking of Seattle Oral History Collection*, August 2002.

[326] R.P. Garg. An overview of sun ultrasparc/sup tm/ based high performance computing systems. In *Proceedings 15th International Parallel and Distributed Processing Symposium. IPDPS 2001*, pages 1 pp.–, 2001.

[327] David Walden. Ward Cunningham . *IEEE Annals of the History of Computing*, 33(04):62–67, October 2011.

[328] Bradley Bargen and Peter Donnelly. *Inside DirectX: in-depth techniques for developing high-performance multimedia applications*. Microsoft Press, USA, 1998.

[329] ICQ. *ICQ 99 User's Guide: Official Edition (with CD-ROM)*. ICQ Incorporated, 1999.

[330] Lydia Pintscher. *20 Years of KDE: Past, Present and Future*. August 2016.

[331] Robert Reinhardt, Joey Lott, and Peter Santangeli. *Macromedia Flash MX Actionscript Bible*. John Wiley & Sons, Inc., USA, 1st edition, 2002.

[332] Michael Stonebraker and Greg Kemnitz. The postgres next generation database management system. *Commun. ACM*, 34(10):78–92, October 1991.

[333] Dawoud Shenouda Dawoud and Peter Dawoud. *7 Universal Serial Bus (USB)*, pages 245–282. 2020.

[334] Sanjay Hardikar, Sameer Rajan, and Saraswati Parmar. Virtual private network: A study of its various aspects. In *2023*

International Conference on IoT, Communication and Automation Technology (ICICAT), pages 1–7, 2023.

[335] Robert Garner and Marc Weber. Interview of the ALOHAnet Principal Designers. *Computer History Museum*, February 2023.

[336] Jiyu He. The google pagerank algorithm. In *Proceedings of the 2nd International Academic Conference on Blockchain, Information Technology and Smart Finance (ICBIS 2023)*, pages 417–422. Atlantis Press, 2023.

[337] Jean-Pierre Teyssier, Fabien de Groote, Markus Mayer, Peter Schlemko, Andrej Rumantsev, Viktar Khutko, Roland Gesche, and Ralf Doerner. The xml file format as a general solution for measurement data storage and exchange. In *2007 69th ARFTG Conference*, pages 1–5, 2007.

[338] Daniel M. German. The gnome project: a case study of open source, global software development. *Software Process: Improvement and Practice*, 8(4):201–215, 2003.

[339] Jaap Haartsen, Mahmoud Naghshineh, Jon Inouye, Olaf J. Joeressen, and Warren Allen. Bluetooth: vision, goals, and architecture. *SIGMOBILE Mob. Comput. Commun. Rev.*, 2(4):38–45, October 1998.

[340] Wen-Chih Chang, Te-Hua Wang, and Yen-Ching Huang. Exploring the distribution and relationship between msn and the physical world. In *The 6th International Conference on Networked Computing and Advanced Information Management*, pages 716–721, 2010.

[341] Konstantinos Barlas, Eleni Berki, Iulia Adomnita, Thrushna Nalam, Golnaz S. Nejad, and Jari Veijalainen. Formal specification of open standards and the case of rss v2.0. In *Proceedings of the 18th Panhellenic Conference on Informatics*, PCI '14, page 1–6, New York, NY, USA, 2014. Association for Computing Machinery.

[342] Kenn Scribner, Kennard Scribner, and Mark C. Stiver. *Understanding Soap: Simple Object Access Protocol*. Sams, USA, 2000.

[343] T. Dierks and E. Rescorla. Rfc 5246: The transport layer security (tls) protocol version 1.2. *RFC Editor*, 2008.

[344] Bengt Carlsson and Rune Gustavsson. The rise and fall of napster - an evolutionary approach. In *Proceedings of the 6th International Computer Science Conference on Active Media Technology*, AMT '01, page 347–354, Berlin, Heidelberg, 2001. Springer-Verlag.

[345] Rob Thomsett. The year 2000 bug: A forgotten lesson. *IEEE Softw.*, 15(4):91–93,95, July 1998.

[346] Lutz Prechelt. The surprising dynamics of a simple year 2000 bug. *SIGSOFT Softw. Eng. Notes*, 24(3):56–57, May 1999.

[347] Anders Hejlsberg, Mads Torgersen, Scott Wiltamuth, and Peter Golde. *C# Programming Language*. Addison-Wesley Professional, 4th edition, 2010.

[348] M.Q.C. van Beurden and A. Weaver. RFC 9639: Free lossless audio codec (FLAC). Technical report, USA, 2024.

[349] Felipe Pezoa, Juan L. Reutter, Fernando Suarez, Martín Ugarte, and Domagoj Vrgoč. Foundations of json schema. In *Proceedings of the 25th International Conference on World Wide Web*, WWW '16, page 263–273, Republic and Canton of Geneva, CHE, 2016. International World Wide Web Conferences Steering Committee.

[350] Johan Pouwelse, Paweł Garbacki, Dick Epema, and Henk Sips. The bittorrent p2p file-sharing system: measurements and analysis. In *Proceedings of the 4th International Conference on Peer-to-Peer Systems*, IPTPS'05, page 205–216, Berlin, Heidelberg, 2005. Springer-Verlag.

[351] Amit Singh. *Mac OS X Internals*. Addison-Wesley Professional, 2006.

[352] Andrew W. Appel. Verification of a cryptographic primitive: Sha-256. *ACM Trans. Program. Lang. Syst.*, 37(2), April 2015.

[353] Kristina Chodorow. *MongoDB: The Definitive Guide*. O'Reilly Media, Inc., 2013.

[354] Jason Swarts. The collaborative construction of "fact" on wikipedia. In *Proceedings of the 27th ACM International Conference on Design of Communication*, SIGDOC '09, page 281–288, New York, NY, USA, 2009. Association for Computing Machinery.

[355] Thuan L. Thai and Hoang Lam. *.NET Framework Essentials (2nd Edition)*. O' Reilly & Associates, Inc., 2002.

[356] Charles Severance. Mitchell baker: The mozilla foundation. *Computer*, 46(2):7–9, February 2013.

[357] Kelley Misata. The tor project: An inside view. *XRDS*, 20(1):45–47, September 2013.

[358] Contel Bradford. *The iPod & iTunes Handbook: The Complete Guide to the Portable Multimedia Revolution*. Atlantic Publishing Company (FL), 2008.

[359] Nils Hammar. *Skype – Reasons for Growth*. Thèse de doctorat, Stockholm School of Economics, 2007.

[360] Tim Ferris. Matt Mullenweg on Characteristics and Practices of Successful Entrepreneurs, June 2018.

[361] Yuriko Isoda and Noriko Taji. Linkedin: The Growth Path of the World Largest Professional Network. *The Research Institute for Innovation Management, HOSEI UNIVERSITY*, February 2017.

[362] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, page 164–177, New York, NY, USA, 2003. Association for Computing Machinery.

[363] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37(5):164–177, October 2003.

[364] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[365] Xuan Zhao, Niloufar Salehi, Sasha Naranjit, Sara Alwaalan, Stephen Voida, and Dan Cosley. The many faces of facebook: experiencing social media as performance, exhibition, and personal archive. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '13, page 1–10, New York, NY, USA, 2013. Association for Computing Machinery.

[366] Martin Odersky, Philippe Altherr, Vincent Cremet, Iulian Dragos, Gilles Dubochet, Burak Emir, Sean McDirmid, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Lex Spoon, Erik Stenman, and Matthias Zenger. An overview of the scala programming language (2. edition). 03 2015.

[367] Raghu Garud and Arvind Karunakaran. Process-based ideology of participative experimentation to foster identity-challenging innovations: The case of gmail and adsense. *Strategic Organization*, 16, 06 2017.

[368] Will Reese. Nginx: the high-performance web server and reverse proxy. *Linux J.*, 2008(173), September 2008.

[369] Daniel Fett, Ralf Küsters, and Guido Schmitz. A comprehensive formal security analysis of oauth 2.0. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 1204–1215, New York, NY, USA, 2016. Association for Computing Machinery.

[370] Santiago Perez De Rosso and Daniel Jackson. Purposes, concepts, misfits, and a redesign of git. *SIGPLAN Not.*, 51(10):292–310, October 2016.

[371] Santiago Perez De Rosso and Daniel Jackson. Purposes, concepts, misfits, and a redesign of git. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, page 292–310, New York, NY, USA, 2016. Association for Computing Machinery.

[372] Vinoo Das. *Learning Redis*. Packt Publishing, 2015.

[373] Gloria Chatzopoulou, Cheng Sheng, and Michalis Faloutsos. A first step towards understanding popularity in youtube. In *2010 INFOCOM IEEE Conference on Computer Communications Workshops*, pages 1–6, 2010.

[374] J. Yates Monteith, John Mcgregor, and John Ingram. Hadoop and its evolving ecosystem. volume 987, June 2013.

[375] Andy Oakley and Jeffrey Snover. *Monad: Introducing the New MSH Command Shell and Language for Windows*. O'Reilly Media, Inc., 2005.

[376] Daylin Vliert. Apple iphone: A market case study. *MacEwan University Student eJournal*, 5, 04 2021.

[377] Rich Hickey. A history of clojure. *Proc. ACM Program. Lang.*, 4(HOPL), June 2020.

[378] Don Syme. The early history of f#. *Proc. ACM Program. Lang.*, 4(HOPL), June 2020.

[379] Paul King. A history of the groovy programming language. *Proc. ACM Program. Lang.*, 4(HOPL), June 2020.

[380] Muhammad Haris, Basit Jadoon, and Farhan Khan. Evolution of android operating system: A review. *Asia Pacific Journal of Contemporary Education and Communication Technology*, November 2018.

[381] Md Tajmilur Rahman, Peter C. Rigby, and Emad Shihab. The modular and feature toggle architectures of google chrome. *Empirical Softw. Engg.*, 24(2):826–853, April 2019.

[382] Idilio Drago, Marco Mellia, Maurizio M. Munafo, Anna Sperotto, Ramin Sadre, and Aiko Pras. Inside dropbox: understanding personal cloud storage services. In *Proceedings of the 2012 Internet Measurement Conference*, IMC '12, page 481–494, New York, NY, USA, 2012. Association for Computing Machinery.

[383] Gary Anthes. Html5 leads a web revolution. *Commun. ACM*, 55(7):16–17, July 2012.

[384] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling memcache at facebook. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, page 385–398, USA, 2013. USENIX Association.

[385] Jose Teixeira, Gregorio Robles, and Jesús M. González-Barahona. Lessons learned from applying social network analysis on an industrial free/libre/open source software ecosystem. *CoRR*, abs/1507.04587, 2015.

[386] Aviv Zohar. Bitcoin: under the hood. *Commun. ACM*, 58(9):104–113, August 2015.

[387] J. Chris Anderson, Jan Lehnardt, and Noah Slater. *CouchDB: The Definitive Guide Time to Relax*. O'Reilly Media, Inc., 1st edition, 2010.

[388] Seikyung Jung. Web development with node.js. *J. Comput. Sci. Coll.*, 33(6):154–156, June 2018.

[389] Eben Upton and Gareth Halfacree. *Raspberry Pi User Guide*. Wiley Publishing, 3rd edition, 2014.

[390] Clinton Gormley and Zachary Tong. *Elasticsearch: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2015.

[391] Miska Hannuksela, Jani Lainema, and Vinod Malamal Vadakital. The high efficiency image file format standard [standards in a nutshell]. *Signal Processing Magazine, IEEE*, 32:150–156, 07 2015.

[392] William Bugden and Ayman Alahmar. Rust: The programming language for safety and performance, 2022.

[393] Gunjan Bansal, Madhulika Sharma, Sarita Yadav, and Usha Yadav. Google Chrome Operating System. *International Journal of Innovative Research in Technology*, 2(5), October 2015.

[394] Gabriel Haas and Viktor Leis. What modern nvme storage can do, and how to exploit it: High-performance i/o for high-performance storage engines. *Proc. VLDB Endow.*, 16(9):2090–2102, May 2023.

[395] Gilad Bracha. *The Dart Programming Language*. Addison-Wesley Professional, 1st edition, 2015.

[396] Bruno Gois Mateus and Matias Martinez. On the adoption, usage and evolution of kotlin features in android development. In *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, ESEM '20, New York, NY, USA, 2020. Association for Computing Machinery.

[397] Sean Dubois, Claes Morgen, and Alex Zhukov. *WebRTC for the curious*.

[398] Paul Murley, Zane Ma, Joshua Mason, Michael Bailey, and Amin Kharraz. Websocket adoption and the landscape of the real-time web. In *Proceedings of the Web Conference 2021*, WWW '21, page 1192–1203, New York, NY, USA, 2021. Association for Computing Machinery.

[399] K. L. Clark and F. G. McCabe. Go!—a multi-paradigm programming language for implementing multi-threaded agents. *Annals of Mathematics and Artificial Intelligence*, 41(2–4):171–206, August 2004.

[400] Valentin Churavy, William F Godoy, Carsten Bauer, Hendrik Ranocha, Michael Schlottke-Lakemper, Ludovic Räss, Johannes Blaschke, Mosè Giordano, Erik Schnetter, Samuel Omlin, Jeffrey S. Vetter, and Alan Edelman. Bridging hpc communities through the julia programming language, 2022.

[401] Philip Wadler and Gavin Bierman. Typescript: The Next Generation. September 2013.

[402] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239), March 2014.

[403] Ron Miller. Mirantis acquires docker enterprise, 11 2019.

[404] Adrian Ionel. What we announced today and why it matters, 11 2019.

[405] Ron Miller. After selling enterprise biz, docker lands $35m investment and new ceo, 11 2019.

[406] Docker Inc. Docker restructures and secures $35 million to advance developer workflows for modern applications, 11 2019.

[407] Ron Miller. Mirantis on run rate over $100m two years after buying docker enterprise assets, 2 2022.

[408] Ron Miller. As docker gains momentum, it hauls in $105m series c on $2b valuation, 3 2022.

[409] Frederic Lardinois. Docker raises $95m series d round for its container platform, 4 2015.

[410] Docker Inc. Docker secures $95m in series d funding to meet the unrivaled demand for its open source platform, 4 2015.

[411] Mike Wheatley. Container shakeup: Docker sells enterprise business to mirantis, appoints new ceo, 11 2019.

[412] Mike Melanson. Mirantis acquires docker enterprise, 11 2019.

[413] Bart Preneel. The first 30 years of cryptographic hash functions and the nist sha-3 competition. In *Proceedings of the 2010 International Conference on Topics in Cryptology*, CT-RSA'10, page 1–14, Berlin, Heidelberg, 2010. Springer-Verlag.

[414] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache spark: a unified engine for big data processing. *Commun. ACM*, 59(11):56–65, October 2016.

[415] James G. Shanahan and Laing Dai. Large scale distributed data science using apache spark. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '15, page 2323–2324, New York, NY, USA, 2015. Association for Computing Machinery.

[416] Nishant Garg. *Apache Kafka.* Packt Publishing, 2013.

[417] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes: Lessons learned from three container-management systems over a decade. *Queue*, 14(1):70–93, January 2016.

[418] Marcel Rebouças, Gustavo Pinto, Felipe Ebert, Weslley Torres, Alexander Serebrenik, and Fernando Castor. An empirical study on the usage of the swift programming language. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 634–638, 2016.

[419] Vlatko Spasev, Ivica Dimitrovski, and Ivan Kitanovski. An overview of graphql: Core features and architecture. *ICT Innovations 2020, Web Proceedings ISSN null*, pages 110–123, 2020.

[420] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: a system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 265–283, USA, 2016. USENIX Association.

[421] Byungjin Jun, Fabián E. Bustamante, Sung Yoon Whang, and Zachary S. Bischof. Amp up your mobile web experience: Characterizing the impact of google's accelerated mobile project. In *The 25th Annual International Conference on Mobile Computing and Networking*, MobiCom '19, New York, NY, USA, 2019. Association for Computing Machinery.

[422] Roberto Theron, Alicia Garcia-Holgado, and Samuel Marcos-Pablos. An experience with microsoft teams to improve the interaction with the students. In *2021 XI International Conference on Virtual Campus (JICV)*, pages 1–3, 2021.

[423] Richard E. Turner. An introduction to transformers, 2024.

[424] Tianyang Lin, Yuxin Wang, Xiangyang Liu, and Xipeng Qiu. A survey of transformers, 2021.

[425] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, page 185–200, New York, NY, USA, 2017. Association for Computing Machinery.

[426] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webassembly. *SIGPLAN Not.*, 52(6):185–200, June 2017.

[427] Anders Møller. Technical perspective: Webassembly: a quiet revolution of the web. *Commun. ACM*, 61(12):106, November 2018.

[428] Jeff Hodges, Giridhar Mandyam, and Michael B. Jones. Registries for Web Authentication (WebAuthn). RFC 8809, August 2020.

[429] Abdul Dakkak, Carl Pearson, and Wen-Mei Hwu. Webgpu: A scalable online development platform for gpu programming courses. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 942–949, 2016.

License

Attribution-ShareAlike 4.0 International

==========

Creative Commons Corporation ("Creative Commons") is not a law firm and does not provide legal services or legal advice. Distribution of Creative Commons public licenses does not create a lawyer-client or other relationship. Creative Commons makes its licenses and related information available on an "as-is" basis. Creative Commons gives no warranties regarding its licenses, any material licensed under their terms and conditions, or any related information. Creative Commons disclaims all liability for damages resulting from their use to the fullest extent possible.

Creative Commons public licenses provide a standard set of terms and conditions that creators and other rights holders may use to share original works of authorship and other material subject to copyright and certain other rights specified in the public license below. The following considerations are for informational purposes only, are not exhaustive, and do not form part of

our licenses.

Our public licenses are intended for use by those authorized to give the public permission to use material in ways otherwise restricted by copyright and certain other rights. Our licenses are irrevocable. Licensors should read and understand the terms and conditions of the license they choose before applying it. Licensors should also secure all rights necessary before applying our licenses so that the public can reuse the material as expected. Licensors should clearly mark any material not subject to the license. This includes other CC-licensed material, or material used under an exception or limitation to copyright.

By using one of our public licenses, a licensor grants the public permission to use the licensed material under specified terms and conditions. If the licensor's permission is not necessary for any reason--for example, because of any applicable exception or limitation to copyright--then that use is not regulated by the license. Our licenses grant only permissions under copyright and certain other rights that a licensor has authority to grant. Use of the licensed material may still be restricted for other reasons, including because others have copyright or other rights in the material. A licensor may make special requests, such as asking that all changes be marked or described. Although not required by our licenses, you are encouraged to respect those requests where reasonable.

==========

Creative Commons Attribution-ShareAlike 4.0 International Public License

By exercising the Licensed Rights (defined below), You accept and agree to be bound by the terms and conditions of this Creative Commons

Attribution-ShareAlike 4.0 International Public License ("Public License"). To the extent this Public License may be interpreted as a contract, You are granted the Licensed Rights in consideration of Your acceptance of these terms and conditions, and the Licensor grants You such rights in consideration of benefits the Licensor receives from making the Licensed Material available under these terms and conditions.

Section 1 -- Definitions.

a. Adapted Material means material subject to Copyright and Similar Rights that is derived from or based upon the Licensed Material and in which the Licensed Material is translated, altered, arranged, transformed, or otherwise modified in a manner requiring permission under the Copyright and Similar Rights held by the Licensor. For purposes of this Public License, where the Licensed Material is a musical work, performance, or sound recording, Adapted Material is always produced where the Licensed Material is synched in timed relation with a moving image.

b. Adapter's License means the license You apply to Your Copyright and Similar Rights in Your contributions to Adapted Material in accordance with the terms and conditions of this Public License.

c. BY-SA Compatible License means a license listed at creativecommons.org/compatiblelicenses, approved by Creative Commons as essentially the equivalent of this Public License.

d. Copyright and Similar Rights means copyright and/or similar rights closely related to copyright including, without limitation,

performance, broadcast, sound recording, and Sui Generis Database Rights, without regard to how the rights are labeled or categorized. For purposes of this Public License, the rights specified in Section 2(b)(1)-(2) are not Copyright and Similar Rights.

e. Effective Technological Measures means those measures that, in the absence of proper authority, may not be circumvented under laws fulfilling obligations under Article 11 of the WIPO Copyright Treaty adopted on December 20, 1996, and/or similar international agreements.

f. Exceptions and Limitations means fair use, fair dealing, and/or any other exception or limitation to Copyright and Similar Rights that applies to Your use of the Licensed Material.

g. License Elements means the license attributes listed in the name of a Creative Commons Public License. The License Elements of this Public License are Attribution and ShareAlike.

h. Licensed Material means the artistic or literary work, database, or other material to which the Licensor applied this Public License.

i. Licensed Rights means the rights granted to You subject to the terms and conditions of this Public License, which are limited to all Copyright and Similar Rights that apply to Your use of the Licensed Material and that the Licensor has authority to license.

j. Licensor means the individual(s) or entity(ies) granting rights under this Public License.

k. Share means to provide material to the public

by any means or process that requires permission under the Licensed Rights, such as reproduction, public display, public performance, distribution, dissemination, communication, or importation, and to make material available to the public including in ways that members of the public may access the material from a place and at a time individually chosen by them.

l. Sui Generis Database Rights means rights other than copyright resulting from Directive 96/9/EC of the European Parliament and of the Council of 11 March 1996 on the legal protection of databases, as amended and/or succeeded, as well as other essentially equivalent rights anywhere in the world.

m. You means the individual or entity exercising the Licensed Rights under this Public License. Your has a corresponding meaning.

Section 2 -- Scope.

a. License grant.

1. Subject to the terms and conditions of this Public License, the Licensor hereby grants You a worldwide, royalty-free, non-sublicensable, non-exclusive, irrevocable license to exercise the Licensed Rights in the Licensed Material to:

a. reproduce and Share the Licensed Material, in whole or in part; and

b. produce, reproduce, and Share Adapted Material.

2. Exceptions and Limitations. For the

avoidance of doubt, where Exceptions and Limitations apply to Your use, this Public License does not apply, and You do not need to comply with its terms and conditions.

3. Term. The term of this Public License is specified in Section 6(a).

4. Media and formats; technical modifications allowed. The Licensor authorizes You to exercise the Licensed Rights in all media and formats whether now known or hereafter created, and to make technical modifications necessary to do so. The Licensor waives and/or agrees not to assert any right or authority to forbid You from making technical modifications necessary to exercise the Licensed Rights, including technical modifications necessary to circumvent Effective Technological Measures. For purposes of this Public License, simply making modifications authorized by this Section 2(a)(4) never produces Adapted Material.

5. Downstream recipients.

    a. Offer from the Licensor -- Licensed Material. Every recipient of the Licensed Material automatically receives an offer from the Licensor to exercise the Licensed Rights under the terms and conditions of this Public License.

    b. Additional offer from the Licensor -- Adapted Material. Every recipient of Adapted Material from You automatically receives an offer from the Licensor to exercise the Licensed Rights in the Adapted Material under

the conditions of the Adapter's License You apply.

c. No downstream restrictions. You may not offer or impose any additional or different terms or conditions on, or apply any Effective Technological Measures to, the Licensed Material if doing so restricts exercise of the Licensed Rights by any recipient of the Licensed Material.

6. No endorsement. Nothing in this Public License constitutes or may be construed as permission to assert or imply that You are, or that Your use of the Licensed Material is, connected with, or sponsored, endorsed, or granted official status by, the Licensor or others designated to receive attribution as provided in Section 3(a)(1)(A)(i).

b. Other rights.

1. Moral rights, such as the right of integrity, are not licensed under this Public License, nor are publicity, privacy, and/or other similar personality rights; however, to the extent possible, the Licensor waives and/or agrees not to assert any such rights held by the Licensor to the limited extent necessary to allow You to exercise the Licensed Rights, but not otherwise.

2. Patent and trademark rights are not licensed under this Public License.

3. To the extent possible, the Licensor waives any right to collect royalties from You for the exercise of the Licensed Rights, whether directly or through a

```
          collecting society under any voluntary or
          waivable statutory or compulsory licensing
          scheme. In all other cases the Licensor
          expressly reserves any right to collect
          such royalties.


Section 3 -- License Conditions.

Your exercise of the Licensed Rights is expressly
made subject to the following conditions.

  a. Attribution.

       1. If You Share the Licensed Material
          (including in modified form), You must:

            a. retain the following if it is
               supplied by the Licensor with the
               Licensed Material:

                 i. identification of the creator(s)
                    of the Licensed Material and any
                    others designated to receive
                    attribution, in any reasonable
                    manner requested by the Licensor
                    (including by pseudonym if
                    designated);

                ii. a copyright notice;

               iii. a notice that refers to this
                    Public License;

                iv. a notice that refers to the
                    disclaimer of warranties;

                 v. a URI or hyperlink to the
                    Licensed Material to the extent
                    reasonably practicable;

            b. indicate if You modified the Licensed
```

Material and retain an indication of any previous modifications; and

c. indicate the Licensed Material is licensed under this Public License, and include the text of, or the URI or hyperlink to, this Public License.

2. You may satisfy the conditions in Section 3(a)(1) in any reasonable manner based on the medium, means, and context in which You Share the Licensed Material. For example, it may be reasonable to satisfy the conditions by providing a URI or hyperlink to a resource that includes the required information.

3. If requested by the Licensor, You must remove any of the information required by Section 3(a)(1)(A) to the extent reasonably practicable.

b. ShareAlike.

In addition to the conditions in Section 3(a), if You Share Adapted Material You produce, the following conditions also apply.

1. The Adapter's License You apply must be a Creative Commons license with the same License Elements, this version or later, or a BY-SA Compatible License.

2. You must include the text of, or the URI or hyperlink to, the Adapter's License You apply. You may satisfy this condition in any reasonable manner based on the medium, means, and context in which You Share Adapted Material.

3. You may not offer or impose any additional or different terms or conditions on, or

apply any Effective Technological Measures to, Adapted Material that restrict exercise of the rights granted under the Adapter's License You apply.

Section 4 -- Sui Generis Database Rights.

Where the Licensed Rights include Sui Generis Database Rights that apply to Your use of the Licensed Material:

a. for the avoidance of doubt, Section 2(a)(1) grants You the right to extract, reuse, reproduce, and Share all or a substantial portion of the contents of the database;

b. if You include all or a substantial portion of the database contents in a database in which You have Sui Generis Database Rights, then the database in which You have Sui Generis Database Rights (but not its individual contents) is Adapted Material, including for purposes of Section 3(b); and

c. You must comply with the conditions in Section 3(a) if You Share all or a substantial portion of the contents of the database.

For the avoidance of doubt, this Section 4 supplements and does not replace Your obligations under this Public License where the Licensed Rights include other Copyright and Similar Rights.

Section 5 -- Disclaimer of Warranties and Limitation of Liability.

a. UNLESS OTHERWISE SEPARATELY UNDERTAKEN BY THE LICENSOR, TO THE EXTENT POSSIBLE, THE LICENSOR OFFERS THE LICENSED MATERIAL AS-IS AND AS-AVAILABLE, AND MAKES NO REPRESENTATIONS OR

WARRANTIES OF ANY KIND CONCERNING THE LICENSED MATERIAL, WHETHER EXPRESS, IMPLIED, STATUTORY, OR OTHER. THIS INCLUDES, WITHOUT LIMITATION, WARRANTIES OF TITLE, MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, ABSENCE OF LATENT OR OTHER DEFECTS, ACCURACY, OR THE PRESENCE OR ABSENCE OF ERRORS, WHETHER OR NOT KNOWN OR DISCOVERABLE. WHERE DISCLAIMERS OF WARRANTIES ARE NOT ALLOWED IN FULL OR IN PART, THIS DISCLAIMER MAY NOT APPLY TO YOU.

b. TO THE EXTENT POSSIBLE, IN NO EVENT WILL THE LICENSOR BE LIABLE TO YOU ON ANY LEGAL THEORY (INCLUDING, WITHOUT LIMITATION, NEGLIGENCE) OR OTHERWISE FOR ANY DIRECT, SPECIAL, INDIRECT, INCIDENTAL, CONSEQUENTIAL, PUNITIVE, EXEMPLARY, OR OTHER LOSSES, COSTS, EXPENSES, OR DAMAGES ARISING OUT OF THIS PUBLIC LICENSE OR USE OF THE LICENSED MATERIAL, EVEN IF THE LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH LOSSES, COSTS, EXPENSES, OR DAMAGES. WHERE A LIMITATION OF LIABILITY IS NOT ALLOWED IN FULL OR IN PART, THIS LIMITATION MAY NOT APPLY TO YOU.

c. The disclaimer of warranties and limitation of liability provided above shall be interpreted in a manner that, to the extent possible, most closely approximates an absolute disclaimer and waiver of all liability.

Section 6 -- Term and Termination.

a. This Public License applies for the term of the Copyright and Similar Rights licensed here. However, if You fail to comply with this Public License, then Your rights under this Public License terminate automatically.

b. Where Your right to use the Licensed Material

has terminated under Section 6(a), it reinstates:

1. automatically as of the date the violation is cured, provided it is cured within 30 days of Your discovery of the violation; or

2. upon express reinstatement by the Licensor.

For the avoidance of doubt, this Section 6(b) does not affect any right the Licensor may have to seek remedies for Your violations of this Public License.

c. For the avoidance of doubt, the Licensor may also offer the Licensed Material under separate terms or conditions or stop distributing the Licensed Material at any time; however, doing so will not terminate this Public License.

d. Sections 1, 5, 6, 7, and 8 survive termination of this Public License.

Section 7 -- Other Terms and Conditions.

a. The Licensor shall not be bound by any additional or different terms or conditions communicated by You unless expressly agreed.

b. Any arrangements, understandings, or agreements regarding the Licensed Material not stated herein are separate from and independent of the terms and conditions of this Public License.

Section 8 -- Interpretation.

a. For the avoidance of doubt, this Public License does not, and shall not be interpreted to, reduce, limit, restrict, or impose conditions on any use of the Licensed Material that could lawfully be made without permission under this Public License.

b. To the extent possible, if any provision of this Public License is deemed unenforceable, it shall be automatically reformed to the minimum extent necessary to make it enforceable. If the provision cannot be reformed, it shall be severed from this Public License without affecting the enforceability of the remaining terms and conditions.

c. No term or condition of this Public License will be waived and no failure to comply consented to unless expressly agreed to by the Licensor.

d. Nothing in this Public License constitutes or may be interpreted as a limitation upon, or waiver of, any privileges and immunities that apply to the Licensor or You, including from the legal processes of any jurisdiction or authority.

==========

Creative Commons is not a party to its public licenses. Notwithstanding, Creative Commons may elect to apply one of its public licenses to material it publishes and in those instances will be considered the "Licensor." The text of the Creative Commons public licenses is dedicated to the public domain under the CC0 Public Domain Dedication. Except for the limited purpose of indicating that material is shared under a Creative Commons public license or as otherwise permitted by the Creative Commons policies

published at creativecommons.org/policies,
Creative Commons does not authorize the use of the
trademark "Creative Commons" or any other
trademark or logo of Creative Commons without its
prior written consent including, without
limitation, in connection with any unauthorized
modifications to any of its public licenses or any
other arrangements, understandings, or agreements
concerning use of licensed material. For the
avoidance of doubt, this paragraph does not form
part of the public licenses.

Creative Commons may be contacted at
creativecommons.org.