The Evolution of Software Engineering from FORTRAN to LLMs

Pardis Noorzad

February 20, 2026

Abstract

This article examines seven decades of software engineering evolution, from FORTRAN to LLMs. We trace major milestones in four eras (foundations, Internet and Web, cloud and infrastructure, AI coding) and use this history to analyze where today's AI coding tools fit. We discuss what changed when new paradigms arrived, what stayed the same, and what economic patterns of previous breakthroughs imply for AI-assisted development. The article includes an in-depth look at AI coding milestones: transformers, in-context learning, Copilot and Codex, RLHF, RAG, agentic interfaces, extended reasoning, and code benchmarks.

Introduction

My first exposure to AI, save a few books from Scholastic, was through a number of electives in college. The understanding at the time was that if we were to achieve the promises of AI, we would not merely write ordinary software but rather software that would generate other software. Of course, code generation was already underway in many forms. Compilers turned high-level code into machine code. Parser generators like yacc turned a grammar into a parser. But that was all purpose-built and deterministic. AI code generation is different.

The discourse about AI today is almost ritualistic. "It's the largest technological leap we've seen." "It's bigger than the internet." "It's as big as smartphones." "SaaS is so over." This article examines seven decades of software engineering evolution, from FORTRAN to LLMs. We draw on this history to analyze where today's AI tools fit. What changed when new paradigms arrived? What stayed the same? And what can the economic patterns of previous breakthroughs reveal about this one?

The article is organized into four eras, each built from milestones presented in roughly chronological order. We begin in 1957.

Foundations. This era established the core abstractions that programming would build on for decades.

1957. FORTRAN eliminates the need for scientists to understand computer hardware

Problem. In the mid-1950s, scientific computing required programming in assembly language. The IBM 704 had 36-bit words and three 15-bit index registers. A programmer writing code to solve differential equations needed to understand both the mathematical method and the hardware details. These included which registers to use, instruction timing, and minimizing the instruction count in inner loops.

This dual expertise created a bottleneck. Universities employed small numbers of programmers who understood both scientific problems and machine architecture. A physicist at Los Alamos might wait weeks for a programmer to translate equations into code [1]. The programmer might not understand the scientific context, causing errors.

In addition, programs were non-portable. Code for the IBM 704 would not run on UNIVAC. Each new machine required complete reimplementation.

Solution. In 1956, John Backus was able to convince IBM executives to fund FORTRAN. He estimated that in 1954, more than half of operating costs were programming costs, despite computers being enormously expensive. He argued that automating translation would reduce programming costs. The first FORTRAN compiler shipped in 1957 [2].

FORTRAN (Formula Translation) let scientists write mathematical expressions in notation close to standard syntax. The statement `Y = A * X + B` directly expressed the computation without registers or memory addresses. A recurrence like the Fibonacci sequence could be written in a handful of lines:

```
INTEGER I, N, F0, F1, TMP
F0 = 0
F1 = 1
READ *, N
DO 10 I = 1, N
TMP = F1
F1 = F0 + F1
F0 = TMP
10 CONTINUE
PRINT *, F1
```

The compiler performed register allocation, instruction selection, and optimization. Many believed compilers could never match skilled assembly programmers. But Backus demonstrated the compiler often generated faster code than hand-written assembly by performing tedious optimizations systematically.

Within five years, most scientific computing moved from assembly to FORTRAN. Scientists became programmers. Computational fluid dynamics, molecular modeling, weather forecasting, and financial modeling all benefited. FORTRAN did not merely accelerate existing work. It made feasible work that had been almost impossible to undertake.

1968. Structured programming makes programs comprehensible by constraining control flow

Problem. By 1968, program size had outpaced human comprehension. FORTRAN, COBOL, and assembly relied heavily on GOTO statements that could transfer control to any labeled statement. A program might contain hundreds of GOTOs jumping to labels scattered throughout thousands of lines.

GOTO statements made local reasoning impossible. Understanding what a program did at any point required tracing all possible execution paths from anywhere. A label on line 500 might be reached by GOTOs from lines 100, 250, 780, and 1200. The number of paths grew combinatorially with program size.

Dijkstra called this "spaghetti code" where control flow wove like tangled strands [3]. By

the late 1960s, commercial systems exceeded 50,000 lines and operating systems approached 100,000 lines. NATO convened a conference in 1968 to address "the software crisis" [4]. Programs had become too complex to understand.

Solution. Dijkstra's "Go To Statement Considered Harmful" argued that GOTOs should be eliminated entirely [3]. He proposed restricting control flow to three constructs. These were sequential execution, conditional execution (if-then-else), and iteration (while loops).

Böhm and Jacopini proved these three constructs were sufficient to express any algorithm expressible with GOTOs [5]. The restriction did not reduce expressive power. Floyd and Hoare had shown that structured constructs admitted formal reasoning (preconditions, postconditions), whereas arbitrary GOTOs did not [6] [7]. Niklaus Wirth designed Pascal [8] to enforce structured programming through syntax. The language had no GOTO statement. Programs could be understood by reading top to bottom, following nested control flow.

The practical effect was that software could grow. Before structured programming, systems above a certain size simply could not be understood or maintained. After it, teams could build operating systems, banking platforms, and airline reservation systems. The improvement in maintainability was consequential. Programs hundreds of thousands of lines long became possible.

1970. Relational databases separate logical data organization from physical storage implementation

Problem. Through the 1960s, programs stored data in flat files with application-specific formats. Each program defined its own file structure and wrote custom parsing code. This worked for isolated applications but created problems as organizations accumulated data and needed to share it.

The fundamental issue was tight coupling. Every program accessing a customer file needed to understand the exact byte layout, and adding a new field required modifying every program that touched that data, even those not using the new field.

IBM's Information Management System (IMS), introduced in 1966, provided hierarchical organization. But accessing data required manual navigation. To find all orders for a customer, a program traversed pointers to child records. There was no declarative way to express access patterns. Different applications wrote redundant filtering logic. When business rules changed, organizations faced updating inconsistent code across dozens of programs.

Solution. Codd's 1970 paper "A Relational Model of Data for Large Shared Data Banks" [9] proposed organizing data as mathematical relations, that is, tables with rows and columns. Each table represented an entity type, each row an instance, each column an attribute.

Codd grounded the model in set theory and predicate logic. Relational algebra and calculus were equivalent, so systems could accept declarative queries and automatically generate efficient procedural execution plans.

Later, relational databases made ACID transactions [10] (atomicity, consistency, isolation, durability) standard. Applications could focus on business logic rather than implementing concurrency control and crash recovery.

The crucial innovation was separating logical organization from physical storage. Users worked with tables conceptually. The database system decided how to store them, what indexes to maintain, and how to organize bytes. Changing storage layout did not require

modifying applications. That data independence made possible a single, shared source of truth. Multiple applications and users could read and update the same data with consistent results. These are the systems we take for granted today, from banking and reservations to inventory and ERP, where many programs depend on the same records.

1971. Unix establishes the operating system as a portable hardware abstraction layer

Problem. Before Unix, operating systems were tightly coupled to their hardware. Software written for an IBM mainframe could not run on a DEC minicomputer. The problem was not just different instruction sets. The ways programs interacted with the system (file I/O, process creation, inter-process communication), together with device drivers, memory management, and scheduling, were all machine-specific. Moving software to new hardware meant rewriting it for a new operating environment, not just recompiling.

Solution. Thompson and Ritchie built Unix at Bell Labs between 1969 and 1971 [11]. Unix exposed a single interface instead of vendor-specific system calls. Programs ran as processes. The same read and write operations applied to files on disk, terminals, and devices. "Everything is a file" meant that one abstraction covered all I/O. The kernel implemented the interface in privileged mode and mediated access to hardware. Programs invoked it through system calls, so the kernel hid the details of any particular device.

That interface had two consequences. Software written to it could run on any machine running Unix, so organizations could change hardware without abandoning software. The abstraction also allowed an interpreted shell. Thompson added one that read command sequences from the terminal or from scripts and executed them. The shell turns command strings into system calls the same as any other process. Programmers could orchestrate tools with a short script instead of compiled software. For example, the following runs two compressions at once. A trailing `&` runs a command in the background so the next one starts right away. The shell's `wait` pauses until all background commands finish.

```
gzip file1.log & gzip file2.log & wait
```

Software portability and orchestration efficiency gave organizations incentive to adopt Unix and to port the kernel to new architectures. Linux, BSD, and macOS implemented the same interface and became the principal Unix-like systems. The shell established scripted tool chains as the standard approach to orchestration. Unix-like systems dominate servers, mobile devices, and cloud infrastructure.

1973. C made systems software like Unix portable across different computer architectures

Problem. Unix had made application software portable across machines that ran Unix. However, systems software such as kernels, device drivers, and system utilities was not portable. It was written in assembly for performance via control over memory layout, interrupts, and registers. No high-level language had shown it could match assembly for that workload. Different machines (IBM, DEC, CDC, and others) came with different instruction sets and architectures. Assembly code for one did not run on another. Porting a Unix kernel or systems stack meant a full rewrite in that machine's assembly. Thus portability and performance seemed to be in conflict.

Solution. Dennis Ritchie developed C between 1969 and 1973 at Bell Labs to achieve portability without sacrificing performance [12]. C provided pointers and low-level operations while abstracting machine-specific details. Data types such as `int` and `char` had no fixed size, so compilers could map them to each architecture. Programmers confined machine-dependent code to a small amount of assembly or conditional code. The rest was portable C. The same source could be compiled for different CPUs with minimal changes.

A program written to the Unix interface could run on any machine that ran Unix. But porting Unix to a new machine meant rewriting the kernel and utilities in that machine's assembly. C provided a portable language for systems programming. In 1973 Thompson and Ritchie rewrote Unix in C [11]. Porting the Unix kernel to a new architecture was as easy as recompiling the C source and adapting a small amount of assembly, not rewriting the entire system.

Unix had made programs portable across machines that ran Unix. C made Unix portable across architectures, and made systems software written in C portable by recompilation. C performance stayed close to assembly. It became and remains the standard language for operating systems, databases, and network stacks.

1970s–1980s. Object-oriented programming enforces encapsulation to manage complexity

Problem. By the late 1970s, software systems had grown to hundreds of thousands of lines. Procedural programs organized code as functions operating on global or parameter-passed data. That structure worked for small programs but failed at scale. The central issue was that data structures lived as global variables or parameters, and any function could read or modify their internals. Verifying that an invariant held, such as that account balances never went negative, required checking every function that touched the relevant data. Changing the representation of a type, such as dates, forced updates across every function that used it. With all functions in a single namespace, naming conflicts and unintended coupling were common. Architectural boundaries existed only by convention. Programmers under pressure could bypass them, and large systems tended to degrade.

Solution. Object-oriented languages such as C++ [13], Smalltalk, and Java addressed the encapsulation problem by making boundaries enforceable in the type system. Classes bundled data with the operations that could act on that data. Callers could use only the exposed methods. Invariants could be enforced in one place instead of by auditing every function. Inheritance and polymorphism supported reuse and abstraction without breaking encapsulation. Design patterns [14] codified recurring designs. Before OOP, keeping a large system coherent depended on every programmer respecting boundaries by discipline. After, the language enforced those boundaries. Teams could own classes, change internals without breaking callers, and build systems that could grow to millions of lines without the same collapse into unmaintainability. OOP became and remains the dominant basis for enterprise and systems software.

Internet and Web. This era saw the Internet become a universal substrate and the Web the primary way software reached users.

1983. TCP/IP makes the Internet a universal network layer

Problem. Through the 1970s, computer networks had proliferated in isolation. ARPANET used its Network Control Protocol (NCP). The Xerox PARC Ethernet had different conventions. Packet radio networks, satellite networks, and local area networks each had distinct protocols for addressing, routing, and reliability. Interconnecting them required understanding each network's quirks. A program written for one could not simply talk to another. Programmers building distributed systems had to implement compatibility layers or choose a single network and accept its limitations.

Solution. Vint Cerf and Bob Kahn had laid the theoretical foundation in 1974 with "A Protocol for Packet Network Intercommunication" [15], which described how to interconnect dissimilar networks through gateways. The protocol split into two layers. IP (Internet Protocol) handled addressing and routing packets across networks, and TCP (Transmission Control Protocol) handled reliable, ordered delivery on top. The design was deliberately minimal. Networks kept their internal structure, and the Internet layer handled only what was necessary to pass packets between them.

On January 1, 1983, ARPANET completed the transition from NCP to TCP/IP [16]. Every host on the network switched to the new protocol. The "flag day" created a single, interoperable network.

The abstraction was profound. Programmers no longer needed to understand packet switching, routing algorithms, or the differences between Ethernet and satellite links. They wrote to sockets, a simple API for sending and receiving byte streams, and the network handled the rest. TCP guaranteed delivery and ordering. IP handled addressing across any connected network. Applications could be built once and run anywhere the Internet reached.

File transfer (FTP/SFTP), email (SMTP), naming (DNS), the Web (HTTP), and every subsequent Internet application built on this foundation. TCP/IP eliminated the need to understand the network layer.

1989–1993. The World Wide Web enables universal software distribution through browsers

Problem. In the 1980s, getting software to people was a logistics problem. Applications like WordPerfect and Lotus 1-2-3 were sold in boxes of floppy disks, each compiled for a specific operating system. A program for MS-DOS would not run on Mac OS or Unix. Updates required mailing new physical media to every user. Business applications accessed by multiple users followed a client-server model that required installing and maintaining software on every client machine independently. Distribution was slow, updates were painful, and every new operating system meant recompiling and repackaging from scratch.

Solution. Tim Berners-Lee proposed the Web at CERN in 1989 [17] as a way to share documents and files across the Internet. Its original motivation had nothing to do with software delivery. CERN's scientific documentation was scattered across hundreds of incompatible computers. Researchers spent significant time just locating information that existed somewhere on the network. Berners-Lee wanted to link documents through hypertext so people could navigate between them without knowing where they were physically stored. By 1990, he had built the first HTTP server, the first browser, and defined HTML and URLs. Hostnames in URLs were resolved by DNS, the same naming layer the rest of the Internet already used. CERN released the protocol royalty-free in 1993 [18].

The insight that made the Web transformative was universality. Any computer with a browser could access any server, regardless of operating system. Marc Andreessen and Eric Bina built Mosaic [19] at the National Center for Supercomputing Applications, the first graphical browser. As adoption grew, programmers saw the implication. Applications could be hosted on a server rather than shipped on floppy disks. A single deployment to the server made the new version available to every user, eliminating the need to mail updated media to customers. Initially that meant downloading software from the Web. Netscape Navigator was distributed that way, as were Winamp, RealPlayer, and countless early desktop applications. The pattern persists today. Zoom, VS Code, and most desktop and mobile installers are still distributed by download from a website or app store.

Making software run inside the browser, not just be downloaded from it, took two more steps. JavaScript, created by Brendan Eich at Netscape, was the next necessary piece. Static HTML pages couldn't respond to user input without sending a request back to the server and reloading the entire page. JavaScript ran directly in the browser, so a form could validate input before submission, a button could trigger an action, a page could change without disappearing and reappearing. Web pages started feeling less like documents and more like applications. The shift completed with AJAX. Jesse James Garrett named the pattern [20] that programmers had already begun using. Applications sent requests in the background and updated only the changed parts of the page instead of reloading. Gmail (2004) proved this worked at scale. An entire productivity application ran in the browser, feeling as responsive as desktop software. The Web had evolved from a tool for sharing scientific documents into the primary platform for delivering software to users.

1991. Python becomes the default for scripting, automation, and data science

Problem. The dominant assumption at the time was that runtime performance mattered most. C and FORTRAN optimized for execution speed. The opposite insight was that programmer time is more expensive than machine time. Most code runs once or rarely (scripts, glue code, prototypes). The cost of writing, debugging, and maintaining it dwarfs execution time. A language that made the common case fast to write, even if slow to run, would win.

Solution. Guido van Rossum released Python in 1991 [21]. Python prioritized readability and ease of use over raw performance. It required no compile step, used clear syntax, and would become "batteries included" as its standard library grew. The crucial design choice was extensibility. When a hot path needed speed, programmers could drop into C. NumPy (2006) demonstrated the pattern. Python for glue code and control flow, C (via extensions) for the numerical inner loops. Programmers got productivity for the 95% of code that wasn't performance-critical, and C-level speed where it mattered. pandas, Django, TensorFlow, and PyTorch followed the same model. Python became the default for data science, ML, and glue code because it optimized for the right variable (programmer time).

1994–1998. Standard algorithm libraries make common algorithms and data structures reusable

Problem. Programmers implementing a sorted collection had to build their own balanced tree or settle for a slower linked list. Those needing O(log n) lookup implemented a red-black tree. Those needing to sort implemented quicksort or merge sort. These implementations were subtle. Off-by-one errors, edge cases with empty collections, and incorrect handling of

equal elements were common. Every team duplicated the same work, and bugs in algorithm implementations were hard to detect because the logic was buried in application code.

Algorithm theory (Big O notation, complexity analysis) had given programmers a vocabulary for reasoning about performance, but it did not eliminate the need to implement. The gap between theory and practice remained.

Solution. Alexander Stepanov and Meng Lee developed the Standard Template Library (STL) for C++ at Hewlett-Packard in 1994 [22]. The design separated containers, iterators, algorithms, and functors. The key insight was generic programming. Algorithms were written once in terms of iterators and worked with any container. `std::sort` worked on a vector, a queue, or a custom container, as long as it provided random-access iterators. HP released the STL freely. It was incorporated into the C++ standard and shipped with every C++ compiler.

In 1998, Java followed with the Collections Framework in JDK 1.2 [23]. List, Set, Map, and their implementations (ArrayList, HashMap, TreeMap) became part of the standard library, with interfaces for sorting, searching, and bulk operations. Like the STL, it provided complexity guarantees. Donald Knuth had established the theoretical basis in *The Art of Computer Programming*, from 1968 onwards [24]. The STL and Java Collections made those guarantees practical. Programmers chose by need (sorted, O(1) lookup, ordered iteration) and the library supplied a correct implementation. The pattern spread to Python's `list`, `dict`, and `set`, C#'s `System.Collections`, and Rust's `std::collections`. Algorithms and data structures became part of the standard toolkit. Programmers used them without implementing them.

1995. Garbage collection makes entire categories of memory errors impossible

Problem. Through the early 1990s, most commercial software was written in C and C++ requiring manual memory management. Programmers explicitly allocated memory with `malloc()` or `new` and deallocated with `free()` or `delete`. Getting the pairing wrong led to memory leaks, use-after-free, double-free, and corruption. Memory leaks consumed all available memory in long-running programs. Use-after-free errors occurred when code freed memory but later accessed it through a dangling pointer, often causing data corruption. Double-free errors corrupted the allocator's internal structures. These bugs were insidious because they might not manifest during testing but caused failures only after days of production operation.

The consequences were real. BlueKeep (2019), a use-after-free in Windows RDP, let attackers execute arbitrary code with kernel privileges over the network without authentication. The NSA issued a rare advisory. Microsoft patched even end-of-life systems. Microsoft estimated 70% of their 2006–2018 security vulnerabilities were memory-safety issues [25].

Solution. Java, released in 1995 [26], popularized garbage collection for mainstream commercial development. Java's innovation was demonstrating that automatic memory management was practical despite performance overhead.

Java eliminated manual deallocation. Programmers allocated objects with `new` but never freed them. The garbage collector, a background process that ran as part of the Java runtime during program execution, periodically scanned the heap to identify objects that were no longer reachable from any live variable or reference, and reclaimed their memory. Because this happened automatically at runtime, programmers could not introduce use-after-free or double-free bugs. The entire class of dangling pointer errors disappeared by construction.

The idea of automatic memory reclamation was not new. John McCarthy's Lisp in 1960 [27] included the first garbage collector. But early collectors were too slow for commercial systems. The breakthrough was generational collection, developed in the 1980s [28]. Most objects die young, so collecting younger generations frequently and older ones rarely reduced overhead enough that GC became viable for production. That made Java's approach credible when it launched.

The performance overhead was non-zero but acceptable. Early collectors had pause times of seconds. Improvements reduced pauses to milliseconds for typical applications. An entire category of serious bugs disappeared. Garbage collection also simplified concurrent programming. Threads could share object references without complex deallocation coordination.

Following Java's success, garbage collection became standard in new languages. C#, Go, JavaScript, Python, and Ruby all adopted it. Rust (2015) took a different approach. Its ownership and borrowing model, a zero-cost abstraction, enforces memory safety at compile time with no runtime overhead or GC pauses [29].

1995–2010. Package managers make dependency management automatic

Problem. Before package managers, reusing code meant finding it, downloading it, manually placing it in your project, and ensuring it worked with other dependencies. Version conflicts were discovered only when builds failed. There was no central registry, no automated resolution. Dependency management was manual and error-prone.

Solution. CPAN, launched in 1995 for Perl, established the pattern. It provided a central archive, a standard layout for how modules were packaged, and a tool that installed modules and dependencies with a single command. Maven, released in 2004, brought the same approach to Java with declarative `pom.xml` files. npm, launched in 2010, did the same for Node and became the largest package ecosystem in history. The abstraction was declarative. Programmers specified what they needed, not how to get it. The same pattern spread to Python (pip), Ruby (RubyGems), and virtually every language. Dependency management became part of the standard toolkit.

1998. Open source makes collaborative, publicly developed software the default

Problem. Through the 1990s, most software was proprietary. Companies kept source code secret to protect their competitive advantage. Reusing code meant licensing it or rewriting it. Richard Stallman had founded the free software movement (GNU, GPL) and framed it as "free as in free speech, not free beer." That established legal and ethical foundations, but "free" carried political baggage that made businesses hesitant. Linux and Apache had proven that open collaboration could produce production-grade software, yet there was no neutral term that invited broad adoption. Programmers who wanted to share code faced a fragmented landscape of licenses and ideologies.

Solution. In 1998, Christine Peterson coined the term "open source," and Bruce Perens and Eric S. Raymond founded the Open Source Initiative (OSI) [30]. The shift from "free software" to "open source" was deliberate. It emphasized practical benefits (peer review, faster iteration, no vendor lock-in) over the freedom-first stance that Stallman had championed. The OSI

defined criteria for open source licenses and certified them. Apache, MIT, and GPL became mainstream choices rather than ideological statements.

The model proved itself. Linux, Apache, MySQL, PHP (LAMP) powered the early web. Firefox challenged Internet Explorer. Android was built on Linux. Companies from Google to Microsoft adopted open source as strategy. GitHub (2008) reduced contribution friction (fork, change, pull request). By the 2010s, open source was the default for infrastructure, frameworks, and tools. Programmers no longer needed to build or buy everything. They could adopt, adapt, and contribute back. The abstraction was organizational. The efficiency gain was communal. The best software in the world was built collaboratively, in public, by anyone who cared to participate.

2000. REST APIs standardize how web services communicate

Problem. Business-to-business commerce and supply-chain integration required one organization's applications to talk to another's over the Internet. Existing distributed computing (CORBA, DCOM, Java RMI) was vendor-specific, complex, and did not work across organizational boundaries. HTTP could carry requests and XML could encode data. What was missing was an agreed way to structure a call. How should one program ask another for a record or submit an update? Without a standard, every service invented its own. SOAP (Microsoft, IBM, W3C 2000) proposed one approach. It was heavyweight (XML envelopes, WSDL, WS-*). No lightweight alternative existed.

Solution. Roy Fielding [31], a co-author of HTTP/1.1, had helped design the Web's protocol. In his 2000 doctoral dissertation he named and formalized the architectural style already present in the Web. Resources were identified by URLs. The interface was the HTTP methods (`GET`, `POST`, `PUT`, `DELETE`). Requests were stateless. He called this style REST. He was not inventing a new protocol. He was documenting what had made the Web scale. That gave programmers a clear model for designing application programming interfaces (APIs). To get a customer, use `GET` on a URL. To create one, use `POST`. No XML envelope, no WSDL. When JSON replaced XML as the preferred format, REST with JSON was easy to use and to test in a browser. Public APIs that let external developers access a site's data (Amazon, Google, and others) adopted REST. By the 2010s, REST had become the default way machines talked to each other on the Web.

2001. IDEs automate the mechanical scaffolding of programming, an early step toward code generation

Problem. As Java and enterprise applications grew into systems of hundreds of thousands of lines, the mechanical overhead of programming became a significant drag on productivity. Adding a new method to a class meant manually hunting through dozens of files to find every call site that needed updating. Renaming a class required running `grep` across the entire codebase and editing each hit by hand. Finding where a method was actually defined meant navigating through directories of source files. These tasks required no deep thought. They were purely mechanical, but they consumed hours each day.

Solution. IntelliJ IDEA (2001) [32] and Eclipse (2001, first release 2004) [33] represented a generational leap in development tools. They parsed entire codebases and built an internal model of every class, method, and reference. This let them provide intelligent code completion. As a programmer typed, the IDE suggested valid method names and parameter types.

Automated refactoring made operations like renaming a class or extracting a method into a single action that propagated correctly across the entire project. Integrated debugging let programmers step through code without leaving the editor. Visual Studio provided similar capabilities for C# and .NET development. Design patterns (Gamma et al., 1994) [14] had codified common OOP solutions. Refactoring (1999) [34], JUnit (1997) [35], and test-driven development (TDD) made restructuring and automated testing mainstream practices.

The productivity gains were substantial. Operations that previously required manual searching and editing across a codebase became instantaneous. By the mid-2010s, IDEs had become so essential that programmers who worked without one felt as disadvantaged as those without version control.

2002. Dependency injection frees enterprise programmers from framework boilerplate

Problem. Java 2 Enterprise Edition (J2EE) and Enterprise JavaBeans (EJBs) were the standard platform for enterprise Java in the early 2000s. J2EE was the platform. EJBs were the component model for server-side business logic, objects that ran in a container and handled transactions and persistence. In practice it required extensive XML, deployment descriptors, and boilerplate just to wire objects together. A simple database service might need dozens of config files and hundreds of lines of scaffolding. Objects created their own dependencies. Testing and swapping implementations required rewriting wiring code throughout. Programmer time went to infrastructure, not features.

Solution. Spring (2003) [36] replaced J2EE's heavy wiring with dependency injection. Rod Johnson's 2002 book [37] had argued that Plain Old Java Objects (POJOs) and a lightweight container could replace EJBs, and Spring implemented that idea. Instead of objects creating their own dependencies, a container created and injected them, so a class that needed a database connection could simply declare the dependency and Spring would provide it. Testing became straightforward because tests could inject mocks. Wiring became explicit and centralized rather than scattered throughout the codebase.

J2EE had established the enterprise Java market but created the complexity Spring addressed. Practitioner-built frameworks could outcompete committee-designed standards. Spring Boot (2014) took the next step by providing sensible defaults and auto-configuration, so programmers could start a Spring application with minimal or no config files. By the mid-2010s, dependency injection had become standard across languages and frameworks.

Cloud and infrastructure. This era saw cloud computing, mobile, big data, and the commoditization of previously specialized infrastructure.

2004–2009. MapReduce and Hadoop make processing massive datasets accessible

Problem. By the early 2000s, companies like Google were crawling and indexing billions of web pages. The sheer volume of data dwarfed what any single machine could store or process. Google solved this internally by building the Google File System (GFS) in 2003 [38], a distributed file system that spread data across hundreds or thousands of commodity servers, and MapReduce in 2004 [39]. MapReduce was a programming model that let programmers express massively parallel computation in a simple way. A Map function processed individual records and a Reduce function aggregated results. The framework handled distributing work,

shuffling data, and recovering from failures. Google published papers but did not release the code.

Solution. Doug Cutting and Mike Cafarella had started Nutch, an open-source web crawler, in 2002. When Google's GFS and MapReduce papers appeared, they implemented the techniques in Nutch but needed institutional backing. Yahoo hired Cutting in 2006 to build distributed data processing for its search engine. He extracted the distributed file system and MapReduce implementation from Nutch into a new project, Hadoop. Hadoop comprised HDFS (the file system) and MapReduce (the processing framework). The same name, MapReduce, was intentional. It implemented the same model from the Google papers. Yahoo dedicated a large team to developing it. By 2007, Yahoo was running Hadoop on a 1,000-node cluster.

Yahoo open-sourced its Hadoop work in 2009, ran Hadoop at scale, and adoption followed quickly. Facebook ran Hadoop and built Presto for interactive SQL, Twitter built Scalding (a Scala API on Cascading and Hadoop MapReduce), and LinkedIn built Kafka for event streaming [40]. eBay and others adopted the ecosystem, and eventually more than half of the Fortune 500 ran big data pipelines on open-source tools. Spark emerged in 2009 [41] as an alternative to MapReduce that kept intermediate data in memory rather than writing it to disk, improving performance for iterative workloads while requiring more memory. Kafka became the de facto standard for event streaming, Flink (2014) [42] offered true stream processing at lower latency than Spark's micro-batch model, and Tez optimized batch DAGs for Hadoop. Big data processing went from out of reach to something any company could run.

2005. Git enables distributed collaboration at global scale

Problem. For the first decade of Linux kernel development (1991–2002), there was no formal version control at all. Contributors emailed patches to mailing lists, and Linus Torvalds manually applied them to his own source tree before cutting releases. This worked when the project was small, but Linux had grown into the most important open-source project in the world, with thousands of contributors. The manual process became a serious bottleneck.

In 2002, Torvalds adopted BitKeeper, a proprietary distributed system that was far ahead of CVS or Subversion. In early 2005, BitMover revoked the free license and the kernel community lost its version control overnight.

Solution. Torvalds had spent months considering what kernel development required. CVS and Subversion were centralized, which made cheap branching and offline work impossible, and no open-source alternative was mature. He began writing Git on April 3, 2005, and had a working system within roughly 10 days. The design was fully distributed. Every clone contained the complete repository history, which allowed programmers to commit, branch, and merge locally without network access. Branching became a lightweight operation, a pointer to a commit that made it essentially free. The Linux kernel 2.6.12 release in June 2005 was the first managed entirely by Git.

Git-based workflows later enabled continuous integration and deployment. Jenkins (2011) and Travis CI (2011) automated testing and deployment pipelines. Programmers pushed code to Git repositories, triggering automated builds, tests, and deployments. GitHub launched in 2008 [43], adding pull requests and code review workflows that made open-source collaboration frictionless. The model enabled global collaboration at unprecedented scale. Projects like Linux, with thousands of contributors across continents, could coordinate effectively. This

DevOps movement reduced the time between writing code and running it in production from weeks to minutes.

2006. Cloud platforms transform infrastructure into elastic, pay-per-use resources

Problem. Before 2006, running applications meant purchasing servers, networking equipment, and storage, renting rack space and power in a data center, and hiring system administrators to maintain all of it. For a startup launching a web service, the upfront capital was substantial. Ordering, installing, and configuring new hardware took weeks or months. Capacity planning made this worse. Organizations had to forecast future demand and either overprovision and pay for idle capacity or underprovision and risk outages. Spiky workloads, such as retail at holidays or tax software in filing season, made the tradeoff brutal.

Solution. Amazon Web Services launched Elastic Compute Cloud (EC2) in August 2006 [44], providing virtual servers provisionable through an API in minutes with pay-per-hour billing. This transformed infrastructure from capital expenditure (CapEx) to operational expense (OpEx) and from static to elastic. EC2 is the foundational example of what the industry came to call Infrastructure as a Service (IaaS). The cloud provider manages physical hardware, networking, and virtualization, while the customer retains responsibility for operating systems, applications, and data. The customer rents compute, storage, and network capacity rather than purchasing it.

This transformed capacity planning. Organizations could scale elastically to match current demand. The cloud model expanded in layers. PaaS (Heroku, Google App Engine, Elastic Beanstalk) shifted OS and runtime management to the provider, so programmers could deploy applications without configuring servers. SaaS (Salesforce, Gmail, Dropbox) delivered entirely managed applications. Infrastructure as code (Chef, Puppet, Terraform) automated the provisioning of IaaS resources through version-controlled scripts, replacing weeks of manual setup. AWS operated data centers worldwide. Startups could deploy globally with the same API calls. Following AWS's success, Azure and Google Cloud emerged. Cloud computing became the dominant deployment model.

2007. Mobile platforms turn the phone into a general-purpose computer with app ecosystems

Problem. At its peak, Nokia controlled over 40% of the global mobile phone market. Within six years, that share had collapsed to under 5%. Hardware was not the issue. Nokia's model treated phones as closed appliances. SDKs were fragmented (J2ME on some devices, proprietary on others), there was no unified channel for programmers to distribute software to users, and Symbian was not built for a phone as a general-purpose computer running third-party software.

Solution. Apple released the iPhone in June 2007. The multitouch screen and full web browser were significant, but the deeper change was conceptual. The iPhone was positioned as a general-purpose computing device that also made calls, with a browser that rendered full web pages rather than a stripped-down mobile experience. The iPhone SDK launched in March 2008 [45] and the App Store opened in July 2008. For software distribution, Apple provided a single channel. Programmers could submit apps and reach millions of devices without going through carriers or OEMs. Google followed with Android in September 2008

[46] and the Android Market, with a more permissive review process. Both platforms provided high-level APIs and enabled instant global distribution. By the mid-2010s, mobile had created new categories such as ridesharing, mobile payments, and social photography, and changed how software was discovered, distributed, and monetized.

2008–2012. Microservices replace monoliths as the architecture for large-scale applications

Problem. Large web companies in the late 2000s built their platforms as monolithic applications, that is, large codebases deployed as one unit. Early Netflix illustrates the pattern. Its core system was a Java application backed by an Oracle database. In August 2008, a hardware failure took the entire service down for three days. The cause was initially suspected to be database corruption. Every part of the system depended on the same database, so a failure in one place propagated everywhere. Such failures are inherent to monolithic architecture.

Beyond availability, monoliths created organizational bottlenecks. Because the application was a single deployment unit, teams working on different features, such as recommendations, billing, and streaming playback, had to deploy together. A bug in one component could take down the whole process and break unrelated features. Because the codebase had no service boundaries, adding a feature required understanding and testing the entire application. Because all components ran in the same process, scaling meant adding more copies of the entire application. Provisioning more compute for one component required scaling everything, wasting capacity on components that needed none. As the codebase grew, development slowed and onboarding became increasingly difficult.

Solution. Amazon arrived at the architecture first [47]. Its monolithic e-commerce platform became unmanageable as it expanded. Architects required every internal capability to be exposed as an independent service. That restructuring produced the infrastructure that became AWS. Netflix began migrating to that model on AWS in 2009, a seven-year process. The core idea was to break the monolith into small, independently deployable services, each owning its own database. A failure in one service no longer took down the whole platform. Netflix eventually decomposed into over 700 microservices. At that scale, services must find each other, handle failures gracefully, and distribute load across instances. Netflix open-sourced its operational tooling as Eureka (service discovery), Hystrix (circuit breaker), and Ribbon (load balancing) [48]. Fowler and Lewis gave the pattern its name and a widely cited reference in 2014 [49]. By the 2020s microservices had become the dominant architecture for large-scale web applications, adopted by most large enterprises.

2009. NoSQL databases trade consistency for scale and flexibility

Problem. By the late 2000s, web-scale data and traffic exceeded what relational databases could handle. Horizontal scaling required ACID across partitions and two-phase commit, which did not scale. The CAP theorem [50] formalized the tradeoff. Relational databases chose consistency and became unavailable during partitions. Fixed schemas forced sparse tables or many joins for heterogeneous data. Schema changes required migrations that locked tables. Row-oriented storage made analytical scans expensive. Relational full-text search did not scale.

Solution. NoSQL databases relax relational constraints in exchange for scale. Different designs addressed different constraints.

Relational databases had required two-phase commit across partitions and went offline when partitions occurred. Two designs from 2006–2007 showed that abandoning ACID across partitions enabled horizontal scaling. They chose opposite sides of the CAP tradeoff.

BigTable [51] (Google, 2006) chose consistency. It used a sparse, multi-dimensional sorted map. Data was organized in column families, groups of columns stored together within each row, so each row could have different columns. Heterogeneous data such as web crawl records with varying fields per URL no longer required sparse tables or many joins. It offered strong consistency within a row and suited read-heavy workloads such as Google's search index and Google Maps. That consistency required a central coordinator, at the cost of availability. BigTable's design was adopted in the open-source Apache HBase.

Dynamo [52] (Amazon, 2007) chose availability. It stayed writable during partitions when relational systems went offline. Its key-value model had no central coordinator. It suited shopping carts and session data, where availability mattered more than immediate consistency. Dynamo's design was adopted in Amazon's DynamoDB (commercial) and Riak (open-source).

Document stores addressed sparse data and schema evolution. Fixed schemas had forced migrations that locked tables. MongoDB (2009) [53] and CouchDB stored JSON-like documents. Applications could add fields without migrations. The model suited user profiles with varying attributes, product catalogs with nested specifications, and content with arbitrary metadata. The tradeoff was eventual consistency and no ACID across documents. Cassandra (2008) [54] combined BigTable's column-family model with Dynamo's decentralized distribution. It handled sparse data, scaled horizontally, and offered tunable consistency. Use cases included activity feeds and time-series data. The cost was eventual consistency by default.

Columnar stores and search engines addressed analytical scans and full-text search. Row-oriented storage had made broad aggregations slow. Vertica and ClickHouse stored each column separately, so scans could read only the columns needed for aggregations. They suited analytical dashboards, sales reports, and click analytics (OLAP) but were poor for transactional point updates (OLTP). Elasticsearch (2010) [55] and Solr, built on Lucene, provided full-text search over HTTP for product search, log analysis, and site search. NoSQL made web-scale storage reachable without purpose-built hardware or specialist teams.

2009. Node.js makes JavaScript full-stack and enables the npm ecosystem

Problem. By the late 2000s, web applications had a split identity. The browser ran JavaScript. The server ran Java, PHP, Python, or Ruby. Programmers wrote frontend and backend in different languages, with different runtimes and toolchains. Building a real-time feature meant WebSockets on the server and JavaScript in the browser. Full-stack development meant context-switching between languages, deployment targets, and debugging environments. Programmer time was spent on integration friction, not features. There was no way to share code reliably between client and server.

Solution. Ryan Dahl released Node.js in 2009 [56]. Node.js ran JavaScript on the server using Google's V8 engine, the same one powering Chrome. The key innovation was non-blocking I/O. Instead of threads, it used an event loop. A single process could handle thousands of concurrent connections. This suited I/O-bound workloads such as APIs, proxies, and real-time applications that had dominated server-side scaling challenges.

Node.js made JavaScript full-stack. Programmers could write client and server in the same

language. npm, launched in 2010, became the package registry for Node and the browser. The same `require` or `import` worked on both sides. Rails (2004) and Django (2005) had popularized convention-over-configuration, relying on sensible defaults (e.g. a `Post` model maps to a `posts` table) instead of explicit configuration for every detail. Node.js and frameworks like Express (2010) brought the same model to JavaScript. The ecosystem expanded rapidly. By the mid-2010s, Node.js powered Netflix, LinkedIn, Uber, and PayPal. JavaScript went from a browser scripting language to the most widely used language for web development. One language and one ecosystem spanned the stack from end to end.

2010–2015. Type safety, component architecture, and safer concurrency reach mainstream development

Problem. Dynamic languages such as JavaScript, Python, and Ruby had become central to web and backend development, but type checking and clear separation of concerns lagged. Dynamic typing deferred errors to runtime that static typing would have caught at compile time. In JavaScript, jQuery-based applications entangled DOM manipulation, business logic, and data fetching with no clear separation.

Concurrent programming in Java, C++, and similar languages faced a separate set of challenges. Mutable shared state and race conditions produced bugs that were difficult to reproduce and debug. Locks offered a remedy but introduced deadlocks and contention, and correctness depended on precise lock ordering that most programmers found impractical to maintain.

Solution. TypeScript (2012) [57] and React (2013) [58] added type checking and component architecture to JavaScript, transforming jQuery spaghetti into structured development.

Functional concepts entered mainstream languages. Scala bridged object-oriented and functional programming on the JVM, making functional ideas accessible to Java programmers. Twitter's adoption of Scala for high-concurrency systems [59] demonstrated that type-safe functional programming could handle production scale. Java 8 (2014) [60] followed with lambdas and streams, bringing functional patterns to the mainstream. Immutable data and pure functions addressed concurrency without locks.

2013–2014. Containers and orchestration make deployment portable and scalable

Problem. Two constraints slowed deployment. The first was environment inconsistency. Applications ran in development but crashed in production due to different library versions, missing dependencies, or configuration drift. Virtual machines provided isolation but were heavyweight. Each VM required a full OS and consumed gigabytes. Manual configuration and documentation were brittle.

The second constraint was orchestration at scale. Distributing workloads across clusters required placement decisions, failure handling, traffic routing, and elastic scaling. Manual coordination did not scale. Cluster managers such as Apache Mesos offered resource scheduling but did not provide declarative desired-state configuration, automated rollouts and rollbacks, integrated service discovery, or self-healing of failed workloads.

Solution. Docker (2013) [61] addressed environment inconsistency. Linux containers packaged applications into images that ran identically anywhere. A Dockerfile replaced manual

setup. Containers were lightweight compared to VMs and required no full OS per instance.

Kubernetes (2014) [62], based on Google's Borg, addressed orchestration at scale. Programmers declared desired state in YAML, for example `replicas: 10` and `containerPort: 80`, and Kubernetes reconciled the cluster to match. The system restarted crashed instances, scaled in response to load, and exposed infrastructure as declarative YAML stored in version control. Cloud providers offered managed Kubernetes. Within a decade, containers and Kubernetes had become the standard for cloud-native deployment that scales and recovers across many machines.

2014. Serverless computing shifts the unit of deployment from servers to functions

Problem. Containers and orchestration made deployment portable and scalable, but containers were always-on. A workload handling one request per hour still required a running container and continuous compute cost. For bursty or infrequent workloads, organizations paid for idle capacity, the same inefficiency that cloud computing had aimed to eliminate.

Solution. AWS Lambda, launched in November 2014 [63], introduced Function-as-a-Service, or serverless. The platform invoked functions only when triggered by events such as HTTP requests, file uploads, queue messages, or scheduled jobs. Each execution was ephemeral. The runtime was torn down afterward, so no compute was allocated between runs and pricing was per-invocation and per-duration. Cost aligned with actual usage rather than provisioned capacity, eliminating idle cost for bursty or infrequent workloads. Lambda scaled automatically from zero to thousands of concurrent executions.

The stateless model imposed constraints. Cold starts introduced latency after inactivity, and execution time was capped, so serverless suited event-driven workloads such as APIs, background processing, data pipelines, and scheduled tasks but was not a universal replacement for containers. Google Cloud Functions and Azure Functions followed. Serverless became a standard deployment option alongside IaaS and containers, selected by workload characteristics.

2015–2016. ML frameworks democratize machine learning without research-level expertise

Problem. Before 2015, applying machine learning meant implementing algorithms from academic papers and using tools like R and Python that required statistical expertise. Deep learning had emerged as a research direction, but implementing backpropagation, designing architectures, and training at scale demanded deep knowledge of linear algebra, optimization, and distributed systems. Google, Facebook, Twitter, and a few labs built their ow internal frameworks. They either had to hire PhDs or stay out. The gap between "research breakthrough" and "programmer can use it" was enormous. The bottleneck was expertise, not compute.

Solution. TensorFlow (2015) [64] and PyTorch (2016) [65] made deep learning tractable for programmers without research-level expertise. Both provided backpropagation, GPU acceleration, and a high-level API. Programmers defined computation as a graph or in imperative code. The frameworks handled the math, learning-rate tuning, and distributed training across GPUs. Both integrated with NumPy, pandas, and Jupyter. Transfer learning allowed fine-tuning of pretrained models with minimal data. Scikit-learn had already made classical ML

such as regression, classification, and clustering accessible. TensorFlow and PyTorch did the same for deep learning, and the language models behind AI coding assistants such as Copilot and Codex were trained with these frameworks [66].

AI coding. AI coding assistants depend on language models that map input sequences to output sequences. The following milestones trace the architectural and scaling developments that made those models possible.

2017. Transformers replace recurrence with self-attention

Problem. By 2017, neural networks had replaced rule-based systems, count-based n-grams, phrase tables, and hand-engineered features for tasks like machine translation, language modeling, and question answering. The dominant architecture was the encoder-decoder, implemented with recurrent neural networks (typically LSTMs). It mapped an input sequence to an output sequence, for example one sentence to another or a natural language description to code. The encoder processed the input token by token and produced a fixed-length vector (the final hidden state). The decoder consumed that vector and generated the output token by token, autoregressively. The limitation was recurrence. At each step t, the hidden state h_t depended on the previous hidden state h_{t-1} and the current input x_t, so computation was strictly sequential.

$$h_t = f(h_{t-1}, x_t) \tag{1}$$

Because h_t depends on h_{t-1}, the forward pass required n sequential steps and could not be parallelized. Information from position t to $t+k$ propagated through k steps. During backpropagation, the gradient was multiplied by $\partial h_t / \partial h_{t-1}$ at each step. The product of k Jacobians often had spectral norm below one, so the gradient decayed exponentially and long-range dependencies received negligible signal. An architecture that allowed parallel computation and direct flow between arbitrary positions was needed.

Solution. Vaswani et al. [67] introduced the Transformer, an encoder-decoder that dispenses with recurrence. Instead of the recurrent update above, each layer uses self-attention. Let i and j denote sequence indices (positions). At each layer, the representation at position i is computed as a weighted sum over all positions j,

$$h_i = \sum_j \alpha_{ij} V_j \qquad \alpha_{ij} = \text{softmax}\left(q_i \cdot k_j / \sqrt{d}\right) \tag{2}$$

The query q_i, key k_j, and value V_j are learned linear projections of the input representations. Unlike recurrence, h_i depends on all h_j in one step, with no sequential dependency. All positions are updated in parallel, and information between any two positions flows in one layer regardless of their distance in the sequence. This design yields three effects. Training is fully parallelizable over the sequence. Long-range dependencies avoid vanishing gradients because the gradient between any two positions traverses one layer, not many. The architecture scales to very large models and datasets. Transformers underpin BERT (2018), GPT-2 (2019), GPT-3 (2020), Codex (2021), and all subsequent code-capable language models.

2020. Large language models demonstrate in-context learning

Problem. By 2019, Transformer-based language models such as BERT and GPT-2 had been pretrained on large text corpora. The standard way to apply these models to a specific task was supervised fine-tuning. A practitioner took a pretrained model, collected labeled examples for the target task, and trained the model on those examples. Translation required labeled translation pairs. Sentiment analysis required labeled sentences. Code generation required labeled specification-code pairs. Each task demanded its own dataset and its own training run. Deploying a new capability meant fine-tuning, validating, and shipping a new model variant. The cost of data collection and the expertise required for training and deployment limited adoption to organizations with dedicated ML infrastructure.

Solution. Brown et al. [68] showed that fine-tuning could be dropped and demonstrated it at scale. A 175-billion-parameter model, trained only on next-token prediction over text, performed well across many tasks when given a few in-context examples and no gradient update. Smaller models had not shown the same capability, so scale mattered. The pretraining corpus contained many input–output style subsequences, such as translations, Q&A, and code with comments, so the model had already learned to continue them without task labels. At inference the input was a prompt of a few pairs plus the new query. A translation prompt could look like:

"Hello, world." $\rightarrow$ "Bonjour, le monde." "Good morning." $\rightarrow$ "Bonjour." "See you tomorrow." $\rightarrow$?

The model produced the continuation by computing $P(\text{next token} \mid \text{prefix})$ with the prompt as prefix, with no second phase or parameter update. They called this in-context learning because the task was specified only in the prompt at inference.

2021. Copilot and Codex bring AI code generation to mainstream development

Problem. Software engineering continued to face productivity bottlenecks. Significant time was spent on mechanical tasks, including implementing CRUD endpoints and validation logic, consulting documentation for library and API usage, searching codebases for analogous implementations, translating schemas to types and API specs to stubs, and writing unit tests with conventional arrange-act-assert structure.

Solution. Research had already shown that pretraining on code improved over general-purpose LMs. CodeBERT [69] and related work demonstrated that joint representations of code and natural language supported search, summarization, and completion. Codex [66] was a GPT model fine-tuned on publicly available code from GitHub. It used the same next-token, in-context paradigm as Brown et al., but with a code-heavy training distribution, and outperformed general-purpose models on code. GitHub Copilot [70] (June 2021) was the first mainstream assistant, with 55% faster task completion [71]. The model completed code as programmers typed. The abstraction was autocomplete at the level of functions and blocks. Verification remained necessary. Output was statistically plausible, not formally correct.

2022. RLHF aligns code models to programmer intent

Problem. Models optimized for next-token prediction did not reliably follow instructions or match user preference. A programmer asking to "add error handling" might receive techni-

cally valid code that didn't match their error-handling conventions. Early Copilot and Codex produced code that was statistically plausible but often misaligned with intent.

Solution. The fix was to add a second training phase that optimized the policy for human preference, not only for next-token likelihood. Here the policy is the code model that, given a prompt x, defines a distribution over completions y, written $\pi_\theta(y \mid x)$ with parameters θ. The procedure is reinforcement learning from human feedback (RLHF). It has two components. (1) A reward model and (2) an RL phase.

(1) The reward model is a network that assigns each prompt-completion pair a scalar reward, e.g. $r(x, y) \in \mathbb{R}$ where x is the prompt and y is the completion. It is trained on human pairwise preferences so that preferred completions receive higher reward. That required new human-labeled data, but only preference labels that indicate which of two completions is better, not full target completions. The scale of this preference data is on the order of tens of thousands of comparisons, far less than pretraining data.

(2) The RL phase has one goal. The policy is adjusted so that its completions get high reward from the reward model (i.e. what humans prefer), without drifting so far from the pretrained reference that outputs degenerate into gibberish or reward-hacking (e.g. repeating phrases the reward model likes). The policy is fine-tuned with PPO (proximal policy optimization), an RL algorithm that updates the policy in constrained steps. In plain language, the training objective is to maximize the average reward on completions the policy produces, then subtract a penalty for how far the policy has drifted from the pretrained reference. So the policy is pushed toward high-reward outputs but kept close to the reference so that outputs stay readable, valid code.

Formally, the objective is

$$\mathbb{E}_{y\sim\pi_\theta}[r(y)] - \beta\,\mathrm{KL}(\pi_\theta \| \pi_{\mathrm{ref}}) \tag{3}$$

where $\mathbb{E}_{y\sim\pi_\theta}$ is the expectation (average over completions drawn from the policy), y is a completion, and $r(y)$ is its reward. Thus the objective maximizes the first term (expected reward under the policy) and minimizes the second (deviation from the reference policy). The coefficient β controls the tradeoff between reward and staying close. KL denotes the Kullback–Leibler divergence,

$$\mathrm{KL}(\pi_\theta \| \pi_{\mathrm{ref}}) = \mathbb{E}_{y\sim\pi_\theta}\left[\log \pi_\theta(y) - \log \pi_{\mathrm{ref}}(y)\right], \tag{4}$$

Without the KL term, the policy can collapse toward high-reward, low-fluency or reward-hacking outputs. The KL term keeps outputs close to the reference distribution so that they stay readable, valid code. The policy is thus optimized for preference, not only for likelihood on a fixed corpus.

InstructGPT (March 2022) [72] and ChatGPT (November 2022) established the pipeline. Code assistants adopted it with programmer labelers and code completions.

2023. RAG grounds code generation in the codebase

Problem. A language model's context window is the maximum number of tokens (roughly, words or subwords) it can take as input in one call. Code-capable models of the Codex and Copilot era (2021–2022) had context windows of 2k–8k tokens. That was enough for a short prompt and a few in-context examples, but not for real codebases. Typical limits remained

4k–8k tokens through 2022. A programmer fixing a bug needed relevant files in context, but those limits could not hold them. Even a modest service spanning dozens of files and tens of thousands of lines exceeded the window, so the model never saw most of the code.

Solution. RAG (retrieval-augmented generation) [73] was introduced for knowledge-intensive NLP in 2020. Code assistants adopted it for the codebase context problem in 2023. The delay reflected two factors. Code-specific retrieval infrastructure (repository indexing, code-aware embeddings) had to be developed. In addition, context limits became a pressing constraint only once coding assistants were widely adopted. RAG sidesteps the context limit by not sending the whole codebase. A retrieval step (e.g. semantic search over embeddings or a code index) selects a subset of files or snippets relevant to the programmer's request. Only that subset is concatenated into the prompt, so the model's fixed context window holds the query plus the retrieved material instead of the entire repo. The model's output is therefore grounded in actual codebase structure rather than generic patterns. Cursor, GitHub Copilot Chat, and others adopted RAG for codebase search. Programmers could point the assistant at a repo and get answers grounded in its structure and contents.

2023–2024. Long-context and agentic interfaces expand scope

Problem. RAG addressed context limits by supplying a retrieved subset of the codebase to the model, but the assistant remained a single-turn completer. It produced output only in response to the current prompt and had no ability to execute tools, query the repository, run tests, or incorporate execution results into the next step. Any task that required multiple steps (for example, fixing failing tests by running the test suite, reading failures, editing code, and re-running until green) therefore had to be orchestrated entirely by the programmer, who ran each step, read the outcome, and re-prompted by hand. The cognitive and manual burden of multi-step tasks stayed with the programmer rather than shifting to the assistant.

Solution. Two developments unfolded over 2023 and 2024.

First, context windows grew. Models with 100k-token context (e.g. Claude 2, GPT-4 Turbo) reached production in 2023, and 200k-token windows became available by 2024. Entire moderate-sized repositories could fit in context, so the model could reason about architectural patterns, cross-file dependencies, and project-wide conventions without retrieval.

Second, agentic interfaces enabled multi-step behaviour. The enabling mechanism is tool use (function calling). The model emits structured tool invocations (e.g. run command, read file, edit file, run tests). The host executes them and appends the results to the model context, so that the model chooses the next action in a repeating plan, act, observe loop. Cursor embedded this pattern in the IDE. Devin (Cognition, March 2024) applied it to autonomous multi-file coding. Claude's "computer use" capability [74] (Anthropic, October 2024) extended it to direct desktop control (cursor, keyboard, screen) in addition to structured tool APIs.

A single request such as "fix the failing tests" could thus trigger a multi-step workflow (run tests, read failures, locate code, generate fixes, rerun tests, iterate) without the programmer re-prompting at each step. Multi-agent systems (e.g. MetaGPT, Devin) went beyond a single model driving tools by deploying several agents that divide the work. Each agent has a distinct role (e.g. planning, coding, reviewing, testing), and they pass outputs to one another so that planning, implementation, and verification are separated and sequenced rather than performed by one monolithic assistant.

2024. Extended reasoning and enterprise fine-tuning complete the AI coding assistant stack

Problem. By 2024, coding assistants combined RLHF, RAG, long-context windows, and agentic tool use. Two gaps remained.

The first gap was the absence of an explicit reasoning phase before the model produced code. Many tasks benefited from weighing options before committing, such as diagnosing a failing test whose cause might lie in several files or choosing among plausible implementations. The autoregressive model used in these assistants did not. It generated one token at a time, at each step computing the distribution over the next token given the prefix and then emitting it.

$$P(x_t \mid x_1, \ldots, x_{t-1}) \tag{5}$$

That autoregressive model did not weigh alternatives before committing. When asked to fix a bug, it could output the first line of a patch immediately, without having considered other possible causes. A human might consider several alternatives before writing any code; such a model did not, and on those tasks its outputs were often wrong or suboptimal.

The second gap was a distribution mismatch between model output and each organization's codebase. RAG and long-context windows both supplied the organization's code as input in the prompt, so that the model had access to it at inference. The weights, however, had been learned only on public corpora and did not change at inference. The model could reuse names or patterns from the prompt, but when the prompt did not fully determine style, structure, or naming, it fell back on what it had learned in training. Output often looked more like public repos than the organization's code, so programmers edited heavily or rejected it.

Solution. Each gap had a direct technical fix.

The first gap was the absence of explicit reasoning before committing to code. Extended-reasoning models solved it. Models such as o1 [75] add an internal chain-of-thought phase before the final output. Instead of generating code tokens directly from the user prompt, the model first generates a sequence of reasoning tokens $r_1, \ldots, r_k$ and then the answer tokens $y_1, \ldots, y_n$ (the code). The user sees only the answer. The next-token distribution at each step conditions on the full prefix, including the model's own reasoning, so the model can explore steps or alternatives before committing to code. Formally, the output distribution is

$$P(y_{1:n} \mid x) = \sum_{r_{1:k}} P(r_{1:k} \mid x)\, P(y_{1:n} \mid x, r_{1:k}). \tag{6}$$

In practice the model is trained to produce $(r_{1:k}, y_{1:n})$ and is given more inference-time compute for the reasoning segment. For complex tasks (e.g. multi-file debugging or choosing among implementations) this yielded substantially better results than direct generation.

The second gap was a distribution mismatch between model output and each organization's codebase. Enterprise fine-tuning solved it. The model's parameters are updated on the organization's code. Let θ denote the base parameters (trained on public corpora). Let $\mathcal{D}_{\text{org}}$ denote the organization's dataset (e.g. proprietary code or prompt–completion pairs). Fine-tuning minimizes the negative log-likelihood on $\mathcal{D}_{\text{org}}$,

$$\mathcal{L}(\theta) = -\sum_{(x,y)\in\mathcal{D}_{\text{org}}} \log P_\theta(y \mid x), \tag{7}$$

yielding parameters θ_{org} that assign higher probability to continuations consistent with the organization's style, naming, and structure. The model's default behaviour at inference then reflects the fine-tuning corpus rather than public code. Copilot Enterprise (2024) [76] offered such customization on proprietary repositories.

By 2024 the ecosystem had diversified. Developers could choose among multiple leading models (Claude, GPT-4, Gemini, open code models such as DeepSeek Coder) and AI-native IDEs (Cursor, Windsurf) alongside incumbent tools. The jump from 2021 Copilot to 2025-era assistants came not mainly from larger base models but from adding RAG, long context, tool use, extended reasoning, and enterprise fine-tuning. Those additions changed what the assistant can do and how well it matches an organization's codebase.

2024. Code evals established comparable benchmarks and revealed the gap to real-world tasks.

Problem. HumanEval was introduced alongside Codex in the 2021 Codex paper [66] and gave the field its first standardized benchmark for code generation. However, it only measured function-level generation from docstrings. Modifying a large, unfamiliar codebase from an ambiguous bug report was a different kind of work and still had no shared evaluation. The field could not separate algorithmic performance from real-world codebase capability, so capability claims that mixed the two were not distinguishable.

Solution. SWE-bench [77] in 2024 supplied the missing benchmark for codebase-editing evaluation. SWE-bench Verified is the curated subset with validated, solvable tasks used for the results reported here. Each instance is an actual bug from open-source repos such as Django, Flask, Matplotlib, and Scikit-learn. The model gets the GitHub issue and must produce a patch that passes the project's test suite. Success depends on locating the relevant code, respecting architecture and invariants, and avoiding regressions.

In June 2024 Claude 3.5 Sonnet reached 93% on HumanEval and 33.5% on SWE-bench Verified [78]. The same model thus showed a wide spread between the two benchmarks. On SWE-bench Verified, GPT-4 reached 1.74% in early 2024, OpenAI o1 reached 48.9% in December 2024 [75], and Claude 4 reached 72.5% in May 2025 [79]. The spread between function-level generation under clear specs and codebase editing under ambiguous, multi-file constraints is therefore substantial. Leaderboards at swebench.com track current results, and any capability claim must state the benchmark and task class.

Discussion

The historical framework above equips us with a lens to understand where AI coding stands today and what its impact may be. The following sections use that lens to further assess the impact of AI coding on software engineering.

The internet, cloud, and mobile eras put AI in context

The internet (TCP/IP, 1983) became a universal substrate for connecting machines and distributing software. Cloud computing (AWS EC2, 2006) turned infrastructure from capital expenditure into operational expense and enabled elastic scaling. Mobile (iPhone and Android, 2007–2008) made the phone a general-purpose computer and established app stores as a dominant distribution channel. All three changed how software reached users. AI coding operates at a different layer. It alters how code is produced, not the substrate or platform.

Nevertheless, our framework does not settle the magnitude of AI's economic impact relative to the internet, cloud, or mobile.

Verification and maintenance costs determine whether AI displaces SaaS

SaaS prevails where the cost of building, operating, and maintaining software in-house has historically exceeded the cost of subscription. Vendors amortize development, maintenance, security, and compliance across many customers. AI may lower the cost of initial construction and can reduce ongoing maintenance, integration, and compliance. In each use case, subscription is displaced only when AI-assisted in-house development costs less in total than subscribing. Bacchelli and Bird [80] find that the expertise to verify code matches the expertise to write it, so verification cannot be offloaded yet and remains a large share of in-house cost. Where that holds, total in-house cost may stay above subscription even when AI lowers the cost of producing code.

SaaS has other moats that in-house builds do not easily reproduce. Vendors spread the cost of compliance certifications (e.g. SOC 2, HIPAA), availability and SLAs, ongoing R&D, and data that grows with the customer base. A single organization replicating that must bear the full cost of audits, redundancy, feature development, and acquiring equivalent data.

Successive abstractions removed entire domains of knowledge from the programmer's burden. FORTRAN let programmers write in a high-level language instead of coding machine instructions. Unix removed vendor-specific system calls and device interfaces. Relational databases removed the need to understand the physical storage layout. TCP/IP obviated the knowledge of each network's internals so any machine could talk to any other on a single global network. In each case the abstraction was sound. Programmers could rely on it without verifying the layer below.

The AI case differs. Coding assistants significantly reduce the effort of producing code, but not the need to verify it. Past abstractions eliminated the need to acquire certain knowledge. AI may accelerate production without removing the expertise required to evaluate and maintain the result. Whether that distinction holds as tools evolve remains an open question.

English is not a programming language

One obstacle persists no matter how capable AI becomes at testing and verification. The artifact that is stored, run, reviewed, and maintained is code, not the natural-language prompts that may have produced it. Meyer puts it directly. Programmers save the source code, not the prompts, because prompts cannot serve as reproducible specification [81]. English is not a programming language because the code is not in English.

Programming languages serve two important purposes. First, they eliminate ambiguity for machines. Natural language is inherently ambiguous. Berry and Kamsties show that ambiguity in requirements is inescapable; different readers take different meanings from the same text [82]. "Export recent orders" leaves format, date range, and fields unspecified. "Retry on failure" leaves how many attempts, which exceptions, and whether to back off unspecified. "Delete inactive users" leaves the inactivity threshold and soft-delete versus purge unspecified. The same prompt yields different code from an LLM on different runs.

Second, they force precision in human thinking. Writing code commits to each choice. In Python you might call `export_orders(since_days=7, format='csv', fields=['id', 'total', 'created_at'])`, `retry(times=3, on=TimeoutError)`, or `delete_inactive_users(inactive_since_days=90, soft=False)`. Each argument answers a question the English left open. The language demands answers. The discipline of

expressing logic in code makes the logic itself clearer.

Intentional Software and generations of research aimed to let humans specify intent without writing code. The idea was that domain experts would edit in their own notation (e.g. tax rules or business logic in domain vocabulary) and the system would maintain a single representation and generate code, like WYSIWYG for documents but for software [83]. The vision was influential. Intentional Software was acquired by Microsoft in 2017, but the approach never became mainstream. Nevertheless, the dream persists and may be more accessible than ever.

Open source creation and maintenance both benefit from AI

Empirical work on open source suggests that both creation and maintenance benefit from AI. Hoffmann et al. find that maintainers with access to GitHub Copilot increase coding activity and reduce project management load, and that these effects persist for at least two years [84]. Yeverechyahu et al. find that maintenance-related contributions rise more than original contributions [85], so the larger gain appears to be in maintenance.

Languages, frameworks, and tools are consolidating, and AI may accelerate the trend

Consolidation around a small number of languages, frameworks, and ecosystems has long been the norm. Language adoption follows a power law, with a few languages accounting for most use [86]. Gu et al. run thousands of algorithmic coding tasks and hundreds of framework selection tasks and find that mainstream languages and frameworks achieve significantly higher success rates in AI-generated code than niche ones [87].

Twist et al. give eight LLMs coding tasks with expert-written reference solutions that specify which libraries to use. They find that LLMs import dominant libraries like NumPy even when those libraries do not appear in the reference solution, in up to 48% of cases. For language choice, the same models are given project initialization tasks in domains where Python is suboptimal for performance. Python is still chosen in 58% of cases and Rust zero times [88].

On the frontend, observers note that models default to React because it dominates training data, even when simpler approaches would serve the task [89]. The winner-take-all pattern predates AI; however, current models are accelerating it.

Can AI improve existing abstraction layers?

Research shows AI improving existing abstraction layers in several domains. Learned query optimizers outperform classical optimizers on some workloads, and GenJoin consistently outperforms PostgreSQL on standard benchmarks [90]. In compilers, models trained on LLVM IR and assembly reach a substantial fraction of autotuning search potential [91]. In cloud infrastructure, reinforcement learning for dynamic resource allocation has been shown to reduce CPU allocation and improve utilization over rule-based autoscaling [92]. Nevertheless, these results show that AI is already delivering real gains within existing abstraction layers.

The cost framework in this article implies that new abstractions emerge when the cost of the incumbent exceeds the cost of the alternative. Further, past transitions such as relational algebra, garbage collection, and TCP/IP required conceptual shifts. AI may lower the cost of exploring new designs. Whether that yields qualitatively new abstractions, such as new models of concurrency, persistence, or distribution, or meaningfully better cloud, databases, or languages remains to be seen.

Conclusion

Seven decades of software engineering followed one dynamic. When the cost of manual work exceeded the cost of automation, the abstraction won. What counted as cost varied. Programmer time, portability, errors, capital, and scale each drove different shifts and produced layer upon layer of abstractions that reduced cost and expanded what was possible.

AI coding fits the same cost logic. Assistants reduce programmer time on mechanical work and accelerate production. Open source evidence suggests both creation and maintenance benefit. Code remains the durable artifact that teams can review, refine, and own. Consolidation around dominant languages and frameworks may deepen, a pattern that has long applied.

Research already shows AI improving existing layers. Query optimizers, compilers, and cloud resource allocation all see gains. Whether AI will yield qualitatively new abstractions remains open. Better tooling and formal methods may narrow the verification gap further. Software engineering has absorbed every prior shift, and there is good reason to expect it to absorb this one.

References

[1] Metropolis, N. et al. 1959. "Early Computing at Los Alamos." *Annals of the History of Computing* 1(1):23-34. Available at `https://ieeexplore.ieee.org/document/4640758`

[2] Backus, J. 1957. "The FORTRAN Automatic Coding System." *Western Joint Computer Conference*. ACM. Available at `https://dl.acm.org/doi/10.1145/1455567.1455599`

[3] Dijkstra, E. W. 1968. "Go To Statement Considered Harmful." *Communications of the ACM* 11(3):147-148. Available at `https://dl.acm.org/doi/10.1145/362929.362947`

[4] Naur, P. & Randell, B. 1969. *Software Engineering: Report on NATO Conference*. NATO Science Committee. Available at `https://eprints.ncl.ac.uk/158767`

[5] Böhm, C. & Jacopini, G. 1966. "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules." *Communications of the ACM* 9(5):366-371. Available at `https://dl.acm.org/doi/10.1145/355592.365646`

[6] Floyd, R. W. 1967. "Assigning Meanings to Programs." *Proceedings of Symposium in Applied Mathematics* 19:19-32. Available at `https://people.eecs.berkeley.edu/~necula/Papers/FloydMeaning.pdf`

[7] Hoare, C. A. R. 1969. "An Axiomatic Basis for Computer Programming." *Communications of the ACM* 12(10):576-580. Available at `https://dl.acm.org/doi/10.1145/363235.363259`

[8] Wirth, N. 1971. "The Programming Language Pascal." *Acta Informatica* 1(1):35-63. Available at `https://link.springer.com/article/10.1007/BF00264291`

[9] Codd, E. F. 1970. "A Relational Model of Data for Large Shared Data Banks." *Communications of the ACM* 13(6):377-387. Available at `https://dl.acm.org/doi/10.1145/362384.362685`

[10] Härder, T. & Reuter, A. 1983. "Principles of Transaction-Oriented Database Recovery." *ACM Computing Surveys* 15(4):287-315. Available at `https://dl.acm.org/doi/10.1145/289.291`

[11] Ritchie, D. M. & Thompson, K. 1974. "The UNIX Time-Sharing System." *Communications of the ACM* 17(7):365-375. Available at https://dl.acm.org/doi/10.1145/361011.361061

[12] Ritchie, D. M. 1993. "The Development of the C Language." *ACM SIGPLAN Conference on History of Programming Languages*, 201-208. Available at https://dl.acm.org/doi/10.1145/155360.155580

[13] Stroustrup, B. 1985. *The C++ Programming Language*. Addison-Wesley.

[14] Gamma, E., Helm, R., Johnson, R., & Vlissides, J. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

[15] Cerf, V. G. & Kahn, R. E. 1974. "A Protocol for Packet Network Intercommunication." *IEEE Transactions on Communications* 22(5):637-648. Available at https://ieeexplore.ieee.org/document/1092259

[16] Postel, J. 1981. "NCP/TCP Transition Plan." RFC 801. Available at https://www.rfc-editor.org/rfc/rfc801

[17] Berners-Lee, T. 1989. "Information Management: A Proposal." CERN. Available at https://www.w3.org/History/1989/proposal.html

[18] CERN. 1993. "CERN Puts Web into Public Domain." Available at https://home.cern/science/computing/birth-web/licensing-web

[19] Andreessen, M. & Bina, E. 1993. "NCSA Mosaic: A Global Hypermedia System." *Internet Research* 3(1). Available at https://www.emerald.com/insight/content/doi/10.1108/10662249410798803/full/html

[20] Garrett, J. J. 2005. "Ajax: A New Approach to Web Applications." Available at https://web.archive.org/web/20060207160552/https://adaptivepath.com/ideas/ajax-a-new-approach-to-web-applications/ (original: adaptivepath.com). See also https://jessejamesgarrett.com/2025/02/18/ajax-at-20/.

[21] Python Software Foundation. "History of Python." Available at https://www.python.org/doc/essays/blurb/

[22] Stepanov, A. & Lee, M. 1994. "The Standard Template Library." Hewlett-Packard. Available at https://www.stepanovpapers.com/Stepanov-The_Standard_Template_Library-1994.pdf

[23] Oracle. 1998. "Java Collections Framework." JDK 1.2. Design by J. Bloch. See Java Collections Design FAQ.

[24] Knuth, D. E. 1968. *The Art of Computer Programming, Volume 1. Fundamental Algorithms*. Addison-Wesley. Subsequent volumes from 1969 onwards.

[25] Microsoft Security Response Center. 2019. "A Proactive Approach to More Secure Code." Available at https://msrc-blog.microsoft.com/2019/07/16/a-proactive-approach-to-more-secure-code/

[26] Gosling, J., Joy, B., & Steele, G. 1996. *The Java Language Specification*. Addison-Wesley. Available at https://docs.oracle.com/javase/specs/

[27] McCarthy, J. 1960. "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I." *Communications of the ACM* 3(4):184-195. Available at https://dl.acm.org/doi/10.1145/367177.367199

[28] Lieberman, H. & Hewitt, C. 1983. "A Real-Time Garbage Collector Based on the Lifetimes of Objects." *Communications of the ACM* 26(6):419-429. Available at https://dl.acm.org/doi/10.1145/358141.358147

[29] Jung, R., et al. 2018. "RustBelt: Securing the Foundations of the Rust Programming Language." *Proceedings of POPL*, 66:1-66:34. Available at https://dl.acm.org/doi/10.1145/3158154

[30] Open Source Initiative. "History of the OSI." Available at https://opensource.org/history

[31] Fielding, R. T. 2000. "Architectural Styles and the Design of Network-based Software Architectures." Doctoral dissertation, University of California, Irvine. Available at https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

[32] JetBrains. 2001. "IntelliJ IDEA." Available at https://www.jetbrains.com/idea/

[33] Eclipse. 2001. "Eclipse IDE." Available at https://www.eclipse.org/

[34] Fowler, M., Beck, K., Brant, J., Opdyke, W., & Roberts, D. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley. Available at https://martinfowler.com/books/refactoring.html

[35] Beck, K. & Gamma, E. 1997. JUnit. Available at https://junit.org

[36] Spring. 2003. "Spring Framework." Available at https://spring.io

[37] Johnson, R. 2002. *Expert One-on-One J2EE Design and Development*. Wrox. Available at https://www.wiley.com/en-us/Expert+One+on+One+J2EE+Design+and+Development-p-9780764543852

[38] Ghemawat, S., Gobioff, H., & Leung, S.-T. 2003. "The Google File System." *Proceedings of SOSP*, 29-43. Available at https://research.google/pubs/the-google-file-system/

[39] Dean, J. & Ghemawat, S. 2004. "MapReduce: Simplified Data Processing on Large Clusters." *Proceedings of OSDI*, 137-150. Available at https://www.usenix.org/legacy/publications/library/proceedings/osdi04/tech/full_papers/dean/dean_html/

[40] Kreps, J., Narkhede, N., & Rao, J. 2011. "Kafka: A Distributed Messaging System for Log Processing." *Proceedings of NetDB*. Available at https://engineering.linkedin.com/27/project-kafka-distributed-publish-subscribe-messaging-system-reaches-v06

[41] Zaharia, M., et al. 2010. "Spark: Cluster Computing with Working Sets." *Proceedings of HotCloud*. Available at https://www.usenix.org/legacy/events/hotcloud10/tech/full_papers/Zaharia.pdf

[42] Carbone, P., et al. 2015. "Apache Flink: Stream and Batch Processing in a Single Engine." *IEEE Data Engineering Bulletin* 36(4):28-38. Available at https://arxiv.org/abs/1506.08603

[43] Dabbish, L., et al. 2012. "Social Coding in GitHub." *Proceedings of CSCW*, 1277-1286. Available at https://dl.acm.org/doi/10.1145/2145204.2145396

[44] Amazon Web Services. 2006. "Announcing Amazon Elastic Compute Cloud (Amazon EC2)." Available at https://aws.amazon.com/about-aws/whats-new/2006/08/24/announcing-amazon-elastic-compute-cloud-amazon-ec2---beta/

[45] Apple. 2008. "iPhone SDK Announcement." Available at https://www.apple.com/newsroom/2008/03/06/Apple-Announces-iPhone-2-0-Software-Beta/

[46] Android Open Source Project. 2008. "Android Platform Overview." Available at https://source.android.com

[47] Vogels, W. 2022. "The Distributed Computing Manifesto." *All Things Distributed*. Available at https://www.allthingsdistributed.com/2022/11/amazon-1998-distributed-computing-manifesto.html

[48] Netflix. 2012. "Netflix Shares Cloud Load Balancing And Failover Tool: Eureka!" Available at https://netflixtechblog.com/netflix-shares-cloud-load-balancing-and-failover-tool-eureka-c10647ef95e5

[49] Fowler, M. & Lewis, J. 2014. "Microservices." Available at https://martinfowler.com/microservices/

[50] Brewer, E. 2000. "Towards Robust Distributed Systems." *Proceedings of ACM PODC*. Available at https://www.researchgate.net/publication/221343719_Towards_robust_distributed_systems

[51] Chang, F., Dean, J., Ghemawat, S., et al. 2006. "Bigtable: A Distributed Storage System for Structured Data." *Proceedings of OSDI*, 205-218. Available at https://research.google/pubs/bigtable-a-distributed-storage-system-for-structured-data/

[52] DeCandia, G., Hastorun, D., Jampani, M., et al. 2007. "Dynamo: Amazon's Highly Available Key-value Store." *Proceedings of ACM SOSP*, 205-220. Available at https://dl.acm.org/doi/10.1145/1294261.1294281

[53] Chodorow, K. & Dirolf, M. 2010. *MongoDB: The Definitive Guide*. O'Reilly Media. Available at https://www.oreilly.com/library/view/mongodb-the-definitive/9781449381578/

[54] Lakshman, A. & Malik, P. 2010. "Cassandra: A Decentralized Structured Storage System." *ACM SIGOPS Operating Systems Review* 44(2):35-40. Available at https://dl.acm.org/doi/10.1145/1773912.1773922

[55] Banon, S. 2010. "You Know, for Search." Elasticsearch. Available at https://www.elastic.co/guide/en/elasticsearch/guide/current/intro.html

[56] Node.js. "About Node.js." Available at https://nodejs.org/en/about

[57] Microsoft. 2012. "Introducing TypeScript." Available at https://devblogs.microsoft.com/typescript/announcing-typescript-1-0/

[58] Facebook. 2013. "React: A JavaScript Library for Building User Interfaces." Available at https://react.dev

[59] Eriksen, M. et al. 2012. "Effective Scala." Twitter Engineering. Available at https://twitter.github.io/effectivescala/

[60] Oracle. 2014. "What's New in Java SE 8." Available at https://docs.oracle.com/javase/8/docs/technotes/guides/whats-new/java-se-8.html

[61] Merkel, D. 2014. "Docker: Lightweight Linux Containers for Consistent Development and Deployment." *Linux Journal* 2014(239):2. Available at https://www.linuxjournal.com/content/docker-lightweight-linux-containers-consistent-development-and-deployment

[62] Burns, B., Grant, B., Oppenheimer, D., Brewer, E., & Wilkes, J. 2016. "Borg, Omega, and Kubernetes." *ACM Queue* 14(1):70-93. Available at https://dl.acm.org/doi/10.1145/2898442.2898444

[63] Amazon Web Services. 2014. "Announcing AWS Lambda." Available at https://aws.amazon.com/blogs/aws/run-code-cloud/

[64] Abadi, M., et al. 2016. "TensorFlow: A System for Large-Scale Machine Learning." *Proceedings of OSDI*, 265-283. (Released 2015.) Available at https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi

[65] Paszke, A., et al. 2019. "PyTorch: An Imperative Style, High-Performance Deep Learning Library." *Advances in NeurIPS* 32. (Released 2016.) Available at https://arxiv.org/abs/1912.01703

[66] Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. D. O., Kaplan, J., et al. 2021. "Evaluating Large Language Models Trained on Code." *arXiv:2107.03374*. Available at https://arxiv.org/abs/2107.03374

[67] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. 2017. "Attention Is All You Need." *Advances in Neural Information Processing Systems* 30. Available at https://arxiv.org/abs/1706.03762

[68] Brown, T. B., Mann, B., Ryder, N., et al. 2020. "Language Models are Few-Shot Learners." *Advances in Neural Information Processing Systems* 33:1877-1901. Available at https://arxiv.org/abs/2005.14165

[69] Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., et al. 2020. "CodeBERT: A Pre-Trained Model for Programming and Natural Languages." *Findings of EMNLP*. Available at https://arxiv.org/abs/2002.08155

[70] GitHub. 2021. "Introducing GitHub Copilot: Your AI pair programmer." Available at https://github.blog/2021-06-29-introducing-github-copilot-ai-pair-programmer/

[71] GitHub. 2022. "Research: Quantifying GitHub Copilot's impact on developer productivity and happiness." Available at https://github.blog/2022-09-07-research-quantifying-github-copilots-impact-on-developer-productivity-and-happiness/

[72] Ouyang, L., et al. 2022. "Training language models to follow instructions with human feedback." *Advances in NeurIPS* 35. Available at https://arxiv.org/abs/2203.02155

[73] Lewis, P., et al. 2020. "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks." *Advances in NeurIPS* 33. Available at https://arxiv.org/abs/2005.11401

[74] Anthropic. 2024. "Introducing computer use, a new Claude 3.5 Sonnet, and Claude 3.5 Haiku." Available at https://www.anthropic.com/news/3-5-models-and-computer-use

[75] OpenAI. 2024. "Introducing OpenAI o1." Available at https://openai.com/o1/

[76] GitHub. 2024. "Fine-tuned models are now in limited public beta for GitHub Copilot Enterprise." Available at https://github.blog/news-insights/product-news/fine-tuned-models-are-now-in-limited-public-beta-for-github-copilot-enterprise

[77] Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., & Narasimhan, K. 2024. "SWE-bench: Can Language Models Resolve Real-World GitHub Issues?" *ICLR 2024*. Available at https://arxiv.org/abs/2310.06770

[78] Anthropic. 2024. "Claude 3.5 Sonnet." Available at https://www.anthropic.com/claude/sonnet

[79] Anthropic. 2025. "Introducing Claude 4." Available at https://www.anthropic.com/news/claude-4

[80] Bacchelli, A. & Bird, C. 2013. "Expectations, Outcomes, and Challenges of Modern Code Review." *Proceedings of ICSE*, 712-721. Available at https://dl.acm.org/doi/10.1109/ICSE.2013.6606617

[81] Meyer, C. 2025. "English Isn't a Programming Language." *Substack*. Available at https://csmeyer.substack.com/p/english-isnt-a-programming-language

[82] Berry, D. M. & Kamsties, E. 2004. "Ambiguity in Requirements Specification." In *Perspectives on Software Requirements*, Springer, 7-44. Available at https://link.springer.com/chapter/10.1007/978-1-4615-0465-8_2

[83] Simonyi, C. 1995. "The Death of Computer Languages, The Birth of Intentional Programming." Microsoft Research Technical Report MSR-TR-95-52. Available at https://www.microsoft.com/en-us/research/publication/the-death-of-computer-languages-the-birth-of-intentional-programming/

[84] Hoffmann, M., Boysel, S., et al. 2024. "Generative AI and the Nature of Work." *SSRN*. Available at https://papers.ssrn.com/sol3/papers.cfm?abstract_id=5007084

[85] Yeverechyahu, D., Mayya, R., & Oestreicher-Singer, G. 2024. "The Impact of Large Language Models on Open-source Innovation." *SSRN*. Available at https://papers.ssrn.com/sol3/papers.cfm?abstract_id=4684662

[86] Meyerovich, L. A. & Rabkin, A. S. 2013. "Empirical analysis of programming language adoption." *Proceedings of OOPSLA*. ACM. Available at https://dl.acm.org/doi/10.1145/2509136.2509515

[87] Gu, F., Liang, Z., Ma, J., & Li, H. 2025. "The Matthew Effect of AI Programming Assistants: A Hidden Bias in Software Evolution." *arXiv:2509.23261*. Available at https://arxiv.org/abs/2509.23261

[88] Twist, L., Zhang, J. M., Harman, M., Syme, D., Noppen, J., Yannakoudakis, H., & Nauck, D. 2025. "A Study of LLMs' Preferences for Libraries and Programming Languages." *arXiv:2503.17181*. Available at https://arxiv.org/abs/2503.17181

[89] Cass, S. 2025. "Web Development in 2025. AI's React Bias vs. Native Web." *The New Stack*. Available at https://thenewstack.io/web-development-in-2025-ais-react-bias-vs-native-web/

[90] Sulimov, P., Lehmann, C., & Stockinger, K. 2024. "GenJoin: Conditional Generative Plan-to-Plan Query Optimizer." *arXiv:2411.04525*. Available at https://arxiv.org/abs/2411.04525

[91] Meta. 2024. "LLM Compiler: Foundation Models of Compiler Optimization." *arXiv:2407.02524*. Available at https://arxiv.org/abs/2407.02524

[92] Fettes, Q., Karanth, A., Bunescu, R., Beckwith, B., & Subramoney, S. 2023. "Reclaimer: A Reinforcement Learning Approach to Dynamic Resource Allocation for Cloud Microservices." *arXiv:2304.07941*. Available at https://arxiv.org/abs/2304.07941