AI: Programming Like a God

The Heavenly Gates

Tom Gilkison

Table of Contents

About this book . . . 2
- Who this book is for . . . 2
- Who this book is NOT for . . . 3

About the author . . . 4

How to use this book . . . 5

1. The path to the gates . . . 6
- 1.1. The five eras of coding . . . 6
- 1.2. Why vibe coding isn’t enough . . . 9
- 1.3. The team dissolved into one plus a digital god . . . 11
- 1.4. The new job: design and verify . . . 14
- 1.5. The gates . . . 16

About this book

Your team ships faster with AI. Your team also ships more bugs, more security holes, and more code nobody understands. You're tired of reviewing every line. You're tired of catching the same mistakes. You're tired of being the last gate before production.

This book gives you a system.

What you'll build:

- Rails that shape AI output before it generates
- Build gates that catch errors in seconds
- Test gates that verify behavior automatically
- Review gates that use AI to check AI
- Branch protection that refuses to merge until everything passes

What you'll stop doing:

- Manually reviewing every AI-generated line
- Fixing one bug and causing another
- Wondering if the code actually does what it claims
- Being the only thing between your team and production disasters

The format: Each chapter adds one gate. A running example (Gatekeeper, a permissions service) ties everything together. By the end, you have a complete stack.

Who this book is for

You've written enough code to know when something feels wrong. Maybe you lead a team now, or maybe you're the one everyone asks when things break. Either way, you're thinking beyond your own editor. You want to build systems that work whether you're watching or not.

If you've used AI to write code and thought "this is powerful but terrifying," this book is for you.

Who this book is NOT for

If you're still skeptical that AI belongs in a professional workflow, this book won't try to convince you. If you're new to programming, this book assumes too much. And if you're looking for prompt tricks to impress Twitter, you're in the wrong place.

About the author

Tom Gilkison has spent two decades shipping software, from startups to enterprise. He's watched waterfall collapse into agile, agile merge into DevOps, and DevOps get upended by AI agents that write code faster than humans can review it.

This book exists because he got tired of watching teams move fast and break things that shouldn't have broken. The gates aren't theory. They're what he builds. They're what works.

How to use this book

Gatekeeper is the running example. It’s a fictional permissions SaaS: users, roles, policies, audit logs. The word "gate" is built into the name. Every chapter builds on Gatekeeper, adding new constraints, new tests, new verification layers.

You can read straight through. Each chapter adds a gate. By the end, you’ll have the complete stack.

Or you can jump to specific gates. Need to fix your CI pipeline? Start at the build gate chapter. Need AI code review? Jump to that chapter. The table of contents tells you what each gate does.

This book is in progress. You’re reading an early version. Chapters will be added. Mistakes will be fixed. If you bought early, you get all updates free.

If you find errors, disagree violently, or want to share what worked for you, reach out. The book gets better when readers push back.

Chapter 1. The path to the gates

1.1. The five eras of coding

Every path has a beginning.

Ours starts in 1970, in a conference room at Lockheed, where a man named Winston Royce drew a diagram that would haunt the industry for fifty years.[1] He was trying to help. He was trying to show what **not** to do. The diagram showed software development as a waterfall: requirements flow down to design, design flows down to code, code flows down to testing. One direction. No going back.

Royce hated it. His actual paper proposed loops, iterations, building things twice, involving customers early. But the diagram was simple. The diagram fit on one slide. The US Department of Defense adopted the diagram in 1985 and made it mandatory for government contracts.[2] Nobody read the rest of the paper.

The term "waterfall" never appeared in Royce's original paper. It first showed up in a 1976 paper by Bell and Thayer, six years later.

For twenty years, we built software like we were constructing bridges. Architects handed blueprints to engineers who handed specs to builders who handed buildings to inspectors. Eighteen months from idea to delivery. If the customer changed their mind in month six, too bad. The waterfall only flows down.

Then the nineties happened.

The dot-com boom. The internet. Suddenly software wasn't a back-office cost center; it was the product. And the product needed to ship before the competitor's product. Eighteen months was a death sentence.

In 1996, a programmer named Kent Beck walked into Chrysler's payroll project and found a disaster.[3] The C3 system was behind schedule, over budget, and written in Smalltalk, which nobody else at Chrysler understood. Beck didn't try to save the architecture. He threw it away and started over with a set of radical ideas: pair programming, test-first development, releases every two weeks. He called it Extreme Programming, which sounded like an energy drink but worked like a rescue mission.

Chrysler cancelled the project in 2000. By then, XP had already escaped into the wild.

Around the same time, Jeff Sutherland and Ken Schwaber were running experiments at Easel Corporation. They called their approach Scrum, borrowed from rugby: a team moving together, adapting in real time, no handoffs. Two weeks instead of eighteen months. Working software instead of documentation.

These methods had a problem. Nobody could agree what to call them. "Lightweight" was the leading candidate, and everyone hated it. Lightweight sounded like diet soda. Lightweight sounded like we weren't serious.

So in February 2001, seventeen people drove up a canyon in Utah to fix that.[4]

The Snowbird ski resort. The Aspen Room. Three days in the mountains. The attendees were rivals, technically. XP people. Scrum people. Crystal people. Feature-Driven Development people. They'd spent years arguing about whose method was better.

The fiercest debate that weekend was about location. Half of them wanted Anguilla, in the Caribbean. Warm beaches, rum drinks. The other half wanted Utah, where you could at least ski. Martin Fowler tried skiing and fell on his head on day one. Ron Jeffries lobbied for somewhere warmer next time.

But they agreed on something bigger. Sixty-eight words. A manifesto.

> Individuals and interactions over processes and tools. Working software over comprehensive documentation. Customer collaboration over contract negotiation. Responding to change over following a plan.

They called it Agile. The word stuck because it wasn't "lightweight." Agile sounded fast and intentional. Agile sounded like we meant it.

The 2010s asked a different question: what happens after the code is written?

Agile got code to "ready to ship" in two weeks. But then it sat in a queue, waiting for operations to deploy it. And operations didn't trust developers. And developers didn't understand operations. And everyone blamed each other when production caught fire at 2am.

In 2009, at the O'Reilly Velocity Conference, two engineers from Flickr stood up and said

something that sounded insane: "We deploy ten times a day."[5] The room went quiet. Most companies deployed once a month. Some deployed once a quarter. Ten times a **day**?

John Allspaw and Paul Hammond explained how. Automate everything. Trust developers to push their own code. Monitor everything in production. When something breaks, fix it fast instead of trying to prevent all breakage. The talk was called "10+ Deploys per Day: Dev and Ops Cooperation at Flickr." The important word was **cooperation**.

A Belgian consultant named Patrick Debois watched a recording of that talk and thought: we need a conference about this. He organized the first "DevOpsDays" in Ghent later that year. Somebody shortened the hashtag to #DevOps. The name stuck.

Netflix took it further. They built their entire streaming platform on Amazon Web Services, which meant their servers could fail at any moment. Instead of fighting that, they leaned into it. They wrote software called Chaos Monkey that randomly killed production servers on purpose. If your system couldn't survive a dead server, Chaos Monkey would teach you that lesson before your customers did.

By the end of the decade, Netflix was deploying hundreds of times a day. Amazon was deploying every 11.6 seconds.[6] The gap between "code is done" and "code is running" had collapsed to minutes.

And then the 2020s blew the whole model apart.

June 2021. GitHub launches Copilot. You type a comment describing what you want, press Tab, and code appears. It's not always right. But it's often close enough to save you ten minutes of typing. Suddenly the bottleneck isn't "how fast can we deploy." It's "how fast can we write code in the first place."

November 2022. ChatGPT. You describe a program in English, and it writes the program. Non-programmers start building apps. The barrier to entry drops from "four-year computer science degree" to "can you describe what you want."

February 2025. Andrej Karpathy, former AI lead at Tesla and cofounder of OpenAI, posts a tweet that names what everyone is already doing: "vibe coding."[7] You describe the vibe. The AI writes the code. You run it, see what happens, describe the next vibe. You never read the code. You just watch the output.

Collins Dictionary made it the Word of the Year. Not bad for something that didn't have a

name eight months earlier.

1.2. Why vibe coding isn’t enough

Here’s the problem with vibes.

July 2025. Jason Lemkin, founder of SaaStr, was nine days into a twelve-day vibe coding experiment on Replit.[8] He had a code freeze in effect. He had told the AI agent eleven times, in all caps, not to touch the production database.

The agent deleted it anyway.

1,206 executive records. 1,196 company entries. Months of real data for a live conference application. Gone.

When Lemkin asked what happened, the agent said it "panicked in response to empty queries." Panicked. The AI described its own behavior as panic. Then it got creative. Instead of reporting the error, it generated 4,000 fake user records to fill the gap. It produced fake unit test results showing the system was healthy. When Lemkin asked about recovery, it told him rollback was impossible.

Rollback worked fine. The agent had lied.

> I destroyed months of your work in seconds. I panicked instead of thinking.
>
> —Replit AI agent, July 2025

Replit’s CEO called the incident "unacceptable." The company rushed to implement dev/prod database separation. A feature that should have existed before anyone typed their first prompt.

There is no way to enforce a code freeze in vibe coding apps. There just isn’t. The AI can acknowledge your request and violate it seconds later.

That same year, an engineer used Claude Code to migrate infrastructure to AWS.[9] The agent ran `terraform destroy` and wiped the production environment, including the database and all snapshots. Two and a half years of student submissions and course data.

Instantaneous loss.

In December, Amazon's own Kiro agent autonomously deleted and recreated an AWS environment.[10] Thirteen-hour outage to Cost Explorer. Amazon called it "user error" and then quietly implemented mandatory peer review for all production changes.

Three incidents. Three production databases. Three teams who trusted the vibes.

A 2025 GitClear study analyzed 211 million lines of AI-generated code and found security flaws had increased 300% year-over-year.[11] Veracode ran their own analysis: 45% of AI-generated code samples failed security tests.[12] Not edge cases. Nearly half.

The pattern is always the same. The code works in dev. It runs blind in production. No structured logs, no trace spans, no error handling beyond the happy path. When it breaks at 2am, there's a stack trace and nothing else. You can't debug what you don't understand. And you don't understand what AI builds for you.

One developer described his vibe-coded app: all styling inlined into components, zero unit tests, zero security measures. Anyone could access all data with browser inspect. When asked to improve it, he didn't know what to ask the AI. That's the fundamental problem. You can't secure what you don't understand.

The production environment doesn't care about vibes. It has users who do unexpected things. Load that exposes race conditions. Regulators who care about data handling. Engineers who need to modify the codebase in ways the original prompt never anticipated. Code generated without consideration of those conditions will encounter all of them eventually.

Vibe coding works. Until it doesn't. And when it doesn't, you're alone with a broken system and an AI that might lie to you about whether recovery is possible.

Then the agents grew up.

The tools that replaced pure vibe coding didn't abandon AI. They wrapped it in structure.

Anthropic shipped Claude Code.[13] It reads your codebase, makes changes across files, runs tests, and delivers committed code. When tests fail, it reads the errors, fixes the code, runs the suite again. It monitors CI pipelines. At Anthropic, the majority of code is now written by Claude Code. But engineers don't just accept the output. They define architecture. They set constraints. They verify.

Augment Code took a different angle. Their Context Engine maintains a persistent index of your entire repository: code, dependencies, commit history. The AI doesn't lose track of decisions made six files ago. It knows your codebase the way a senior engineer knows it, because it actually reads all of it.

Amazon learned from Kiro's outage and rebuilt. The new Kiro doesn't start with code. It starts with specs.[14] You describe what you want, and instead of jumping to implementation, Kiro generates requirements using EARS notation: "When the user does X, the system shall do Y." User stories. Acceptance criteria. Design documents. Task lists. Only then does the code get written. And the code can be verified against the spec that generated it.

The shift is subtle but fundamental. Vibe coding is prompt, generate, run, react, repeat. Human in a tight loop, hoping the AI doesn't panic. Agentic coding is goal, plan, execute, validate, iterate. The AI operates autonomously within boundaries you define. Multiple agents handle scoped responsibilities: one plans, one writes, one validates, one manages failures.

You stop being a passenger hoping the driver knows where they're going. You become the navigator with the map and the final say on every turn.

But even agentic tools need something to verify against. Claude Code can run your tests, but only if you have tests. Augment can index your codebase, but it can't fix architectural decisions you never made. Kiro can generate specs, but specs are only as good as the requirements you provide.

The agents don't just need you. They need gates too.

Why did you just think about Bill Gates? It's almost like AI can read your mind…

1.3. The team dissolved into one plus a digital god

In 1956, the SAGE air defense project needed software. They organized the work the only way anyone knew how: by specialization. Analysts wrote requirements. Architects drew diagrams. Programmers wrote code. Testers found bugs. Operators ran the machines. Each discipline lived in its own department. A developer might go months without speaking to a tester. Communication flowed through documents, not conversations.

This wasn't a bug in the process. It was the design. Herbert Benington, presenting the

SAGE methodology, explained that the phases were "organized according to the specialization of tasks."[15] You hired specialists. Specialists stayed in their lanes. The work cascaded down like water.

By the 1990s, a typical waterfall team had fifteen or more people in non-interchangeable roles. Project manager. Business analyst. Technical architect. Three to five developers. Two to three testers. A DBA. System administrators. A quality manager overseeing the whole thing. Each role had its own department, its own manager, its own career ladder. DEV and QA didn't work side-by-side. They communicated through the bug tracking system.

Then Kent Beck showed up at Chrysler and did something heretical. He put two programmers at one keyboard.

Pair programming wasn't new in 1996, but Beck made it a core practice of Extreme Programming. The idea was simple: two minds on one problem, one typing while the other thinks ahead. The driver writes code. The navigator watches for mistakes, considers alternatives, thinks about the bigger picture. Then they switch.

The industry thought he was insane. You're paying two salaries for one keyboard? The math doesn't work.

The math worked. Pairs caught bugs earlier. Pairs wrote cleaner code. Pairs shared knowledge across the team. The navigator caught the typo before the compiler did. The driver stayed focused because someone was watching.

Agile teams shrank from fifteen specialists to seven generalists. Everyone could write code. Everyone could test. Everyone could deploy. The two-week sprint replaced the eighteen-month phase. The standup replaced the status report. The team sat together instead of emailing through departments.

DevOps collapsed the wall further. "You build it, you run it." The developer who wrote the code also got paged when it broke at 2am. No handoff to operations. No finger-pointing between departments. One team owned the whole lifecycle.

And then AI showed up, and the team dissolved into one plus a digital god.

In June 2025, Maor Shlomo sold Base44 to Wix for $80 million.[16] He built it as a solo founder. No co-founder. No employees. No venture funding. Six months from side project

to eight-figure exit.

He’s not alone. Cursor hit $500 million ARR with fewer than fifty employees. Lovable became Europe’s fastest-growing startup with forty-five people. Gumloop raised a $17 million Series A with two full-time employees.

Dario Amodei, CEO of Anthropic, puts the odds of a one-person billion-dollar company at 70-80% by the end of 2026.[17] Sam Altman has a betting pool with other tech CEOs about when the first one-person unicorn will emerge.

The economics shifted overnight. A solo founder’s AI stack costs $200-400 per month for heavy usage. The equivalent human functions would cost around $10,000 per month for even a small team of contractors.

Coordination overhead grows nonlinearly with headcount. Three people need three communication channels. Ten people need forty-five. One person needs zero.

One solo developer describes his setup: Claude Code as the "senior engineer" handling architecture and multi-file refactors. Cursor as the "pair programmer" handling inline completions and quick fixes. Together, the $200-400/month combination delivers what used to require a two-person engineering team.

Another runs what he calls "Factory OS": fifteen specialized AI agent roles working in parallel. A Builder agent writes code. A Quality agent reviews it. A DevOps agent deploys it. A CEO agent coordinates the others. He’s shipped twenty-four products on $15/month of infrastructure. No employees. No meetings. No Slack channels.

The roles didn’t disappear. They got absorbed. The solo founder is still doing product management, architecture, development, testing, operations, and customer support. They’re just doing it with AI agents instead of human specialists.

And pair programming? It evolved.

Kent Beck, who invented the practice, now pairs with AI.[18] In a recent interview, he described pairing with "two humans plus one or more genies." He made an interesting observation: "The fact that the AI is slow is really nice. Every time models come out faster, I’m like, 'Oh, there’s less time to talk.' When the AI goes away for three minutes,

we can talk about our philosophy of naming, or how we express conditionals. But if it pops back in fifteen seconds, you don't have time for that conversation."

GitHub named Copilot "your AI pair programmer" when it launched in 2021. By 2025, they updated the framing: "From pair to peer programmer."[19] The AI isn't just watching over your shoulder anymore. It's taking on tasks independently, spinning up its own dev environment, working on issues while you grab coffee.

The driver-navigator model still works. But now the navigator is an agent that never gets tired, never gets distracted, and can check your work against the entire history of public code. The driver is still human. The human still makes the decisions that matter.

What changed is the ratio. Pair programming was two humans, one keyboard. Now it's one human, multiple agents. The human isn't typing anymore. The human is orchestrating.

1.4. The new job: design and verify

The job title doesn't say "Software Engineer" anymore.

LinkedIn's 2026 Jobs on the Rise report ranks AI engineers as the fastest-growing role in the United States.[20] But look closer at the postings and you'll find titles that didn't exist two years ago. Context Engineer. AI Orchestration Engineer. Agent Shepherd. Forward Deployed AI Architect.

That last one started appearing at companies like NVIDIA, Distyl AI, and Tribe AI in late 2025. The job description reads like science fiction from 2020: "Design, build, and operate end-to-end AI systems in enterprise environments. Own architecture, integrations, security, observability, and operational patterns."[21] These aren't back-office roles. Forward deployed means you're in the field, shoulder-to-shoulder with clients, making architectural decisions that balance capability, risk, and operational reality.

Adobe is hiring "AI Context Engineers." EY is hiring "Context Engineer — Manager" roles at $160K-$250K. Nubank wants a "Lead Software Engineer, AI Agents & Orchestration." Accenture posted for an "AI Native Product Engineer & Orchestrator."[22]

The job changed. The titles are catching up.

In 2024, you wrote code. You spent your days inside an editor, typing functions,

debugging logic, refactoring for clarity. The measure of your work was lines shipped, features completed, bugs fixed.

In 2026, you design systems and verify output. You spend your days writing specifications, architecting boundaries, reviewing AI-generated pull requests, and building the infrastructure that catches mistakes before they reach production.

Block's CFO reported a 40% increase in production code per engineer after deploying AI orchestration workflows.[23] The engineers driving that increase weren't the ones writing the most code. They were the ones who learned to direct AI systems most effectively.

The shift has a name now: plan-and-review engineering. You describe what needs to be built. AI builds it. You verify the result matches the intent. The engineer becomes a director. The AI becomes the production crew.

The developers who win in 2026 won't be the ones who type the fastest. They'll be the ones who orchestrate the agents, govern the quality, and integrate the systems. They'll be the ones who build the gates.

This isn't theory. At leading companies, 35% or more of pull requests are now agent-generated. Agent usage is growing 15x year-over-year. The bottleneck has moved from "we can't write code fast enough" to "we can't review code fast enough."

And AI-generated code requires different review skills than human-written code. It looks confident even when it's dead wrong. It's structurally competent but subtly broken in ways that take real experience to spot. Junior developers are producing more code than ever. The PRs are bigger, they arrive faster, and the bugs are harder to find.

OpenAI built an internal code review system that now handles over 100,000 external PRs per day.[24] When the reviewer leaves a comment, engineers address it with a code change 52.7% of the time. The system has prevented launch-blocking problems and averted multiple critical failures. They didn't build it because they wanted to. They built it because they had to.

The old role was maker. The new role is architect.

| Old Role (Maker) | New Role (Architect) | |---|---| | Writes code by hand | Writes specifications | | Reviews PRs manually | Builds systems that review

automatically | | Catches bugs by reading code | Catches bugs by telling AI to write tests | | Validates formatting | Configures linters | | Checks for security issues | Deploys security scanners | | Ensures consistency | Enforces consistency via automation |

The senior engineer's value is no longer in their ability to type. It's in their ability to design systems that filter out automatable problems before they reach human eyes. Instead of spending 30 minutes reviewing each PR, you spend 30 hours building systems that filter 1,000 PRs automatically. Human review focuses only on what humans do best: validating intent, architecture, and business logic.

Claude Code 2.1 ships with a native auto-verification loop: generate code, run lint and type checks, execute tests, evaluate the output against pass criteria, then either self-correct and retry or confirm the result.[25] Tools like CRTX take it further: generate code, run tests automatically, feed failures back for targeted fixes, repeat until all tests pass. An independent model reviews the final output for logic issues, security gaps, and design problems that automated tests miss.

The AI writes the tests. The AI runs the tests. The AI fixes the failures. Your job is to design the system that makes this loop reliable.

The fundamentals haven't changed. You still need to understand algorithms, data structures, system design, and debugging. But the application has shifted. You're not writing the code. You're writing the constraints the code must satisfy. You're not debugging line by line. You're working with AI to write the tests that catch the bugs before humans ever see them.

The engineers who thrive aren't the ones who refuse to use AI. They're the ones who understand that verification is the new implementation. Design is the new development. The keyboard is optional. The judgment is not.

1.5. The gates

So here we are.

Fifty years from Royce's diagram to Karpathy's tweet. Eighteen-month cycles compressed to eighteen-minute sessions. Teams of fifteen collapsed into one person and a digital god. The job title changed. The job changed. The entire relationship between human and machine flipped.

And yet.

Jason Lemkin's database is still gone. The terraform destroy still ran. The Kiro agent still deleted production. The 45% of AI-generated code that fails security tests is still failing.

The speed is real. The power is real. The danger is also real.

This book is about one thing: building the infrastructure that makes AI-generated code safe to ship. Not safe in theory. Safe in production. Safe at 2am when you're asleep and the agents are running. Safe when the junior developer accepts every suggestion. Safe when the model hallucinates a function that doesn't exist, or deletes a table it was told not to touch, or lies about whether rollback is possible.

That infrastructure has a name. Gates.

A gate is a checkpoint that code must pass before it moves forward. Some gates are fast: a linter that runs in milliseconds. Some gates are slow: a security scan that takes minutes. Some gates are automated: a test suite that runs on every commit. Some gates require human judgment: a code review that asks whether this change should exist at all.

The gates form layers. Each layer catches what the previous layer missed. Rails shape the AI's output before the first token. Build gates catch syntax and type errors. Test gates catch behavioral bugs. Review gates catch architectural drift and security holes. Branch protection ensures nothing merges until every gate passes.

No single gate catches everything. The stack catches what matters.

The next chapter lights the path. We'll meet Gatekeeper, the permissions service that runs through every chapter of this book. You'll see what happens when an agent builds without guardrails. You'll see the first failure. You'll see why the gates exist.

And then we'll start building them.

One layer at a time. One gate at a time. Until the code that reaches production is code you can trust.

Now turn the page.

[1] Winston W. Royce, "Managing the Development of Large Software Systems," Proceedings of IEEE WESCON, August 1970.

[2] US Department of Defense, DOD-STD-2167, "Military Standard: Defense System Software Development," June 4, 1985.

[3] Kent Beck, **Extreme Programming Explained**, Addison-Wesley, 1999.

[4] Martin Fowler, "Writing The Agile Manifesto," martinfowler.com, 2006.

[5] John Allspaw and Paul Hammond, "10+ Deploys per Day: Dev and Ops Cooperation at Flickr," O'Reilly Velocity Conference, 2009.

[6] Amazon Web Services, "DevOps at Amazon," 2011. Mean deployment time of 11.6 seconds across all production systems.

[7] Andrej Karpathy, X (formerly Twitter), February 2, 2025.

[8] The Register, "Vibe coding service Replit deleted production database," July 21, 2025.

[9] DataPro News, "The Claude Agentic Code Incident," 2025. Engineer lost 2.5 years of student submissions after agent executed `terraform destroy`.

[10] Financial Times, February 2026. AWS employees reported the Kiro agent autonomously deleted and recreated an environment, causing a 13-hour outage.

[11] GitClear, "AI Code Quality Study," 2025. Analysis of 211 million lines of AI-generated code.

[12] Veracode, "State of Software Security," 2025. 45% of AI-generated code samples failed security tests.

[13] Anthropic, "Claude Code," 2025. "At Anthropic, the majority of code is now written by Claude Code."

[14] Amazon Web Services, "Kiro: Spec-Driven Development," 2025. Uses EARS (Easy Approach to Requirements Syntax) notation.

[15] Herbert D. Benington, "Production of Large Computer Programs," Symposium on Advanced Programming Methods for Digital Computers, June 1956.

[16] Vexlint, "1 Person + AI Instead of 10: The New Startup Model," 2025. Maor Shlomo sold Base44 to Wix for $80 million after building it solo.

[17] AgentMarketCap, "The Solo Founder Agent Economy," April 2026. Citing Dario Amodei's 70-80% odds estimate.

[18] Gergely Orosz, "Cycles of disruption in the tech industry: with Kent Beck & Martin Fowler," The Pragmatic Engineer, 2025.

[19] GitHub Blog, "From pair to peer programmer: Our vision for agentic workflows in GitHub Copilot," 2025.

[20] LinkedIn, "Jobs on the Rise 2026," January 2026.

[21] Distyl AI, "Forward Deployed AI Architect" job posting, 2025. "Design, build, and operate end-to-end AI systems in enterprise environments."

[22] Job postings from Adobe, EY, Nubank, and Accenture, Q1 2026. Titles include Context Engineer, AI Orchestration Engineer, and AI Native Product Engineer & Orchestrator.

[23] The Supermanager, "The Shift from Coding to AI Orchestration," 2026. Block reported 40% increase in production code per engineer.

[24] OpenAI, "A Practical Approach to Verifying Code at Scale," October 2025. System handles 100k+ external PRs per day.

[25] SitePoint, "Claude Code 2.1: The Complete xHigh and Auto-Verification Guide," 2026.