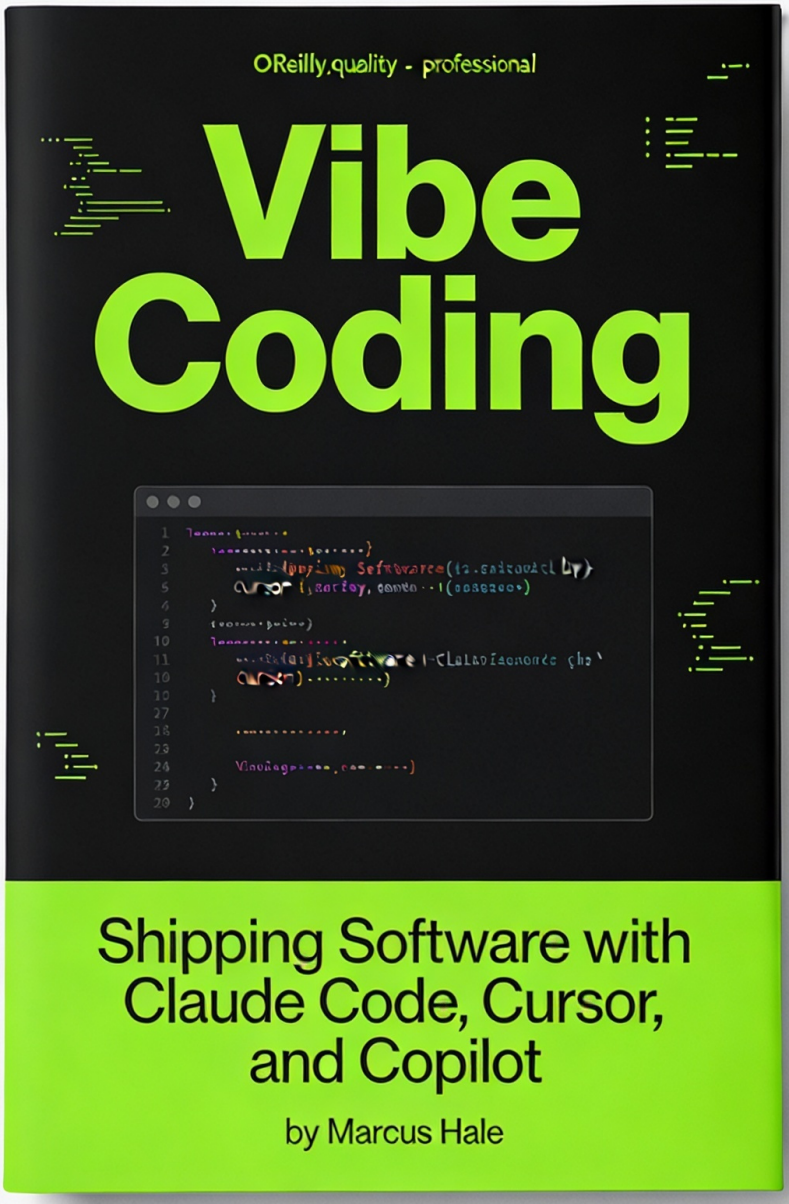

OReilly.quality - professional
Vibe Coding
Shipping Software with Claude Code, Cursor, and Copilot
by Marcus Hale

Vibe Coding

Shipping Software with Claude Code, Cursor, and Copilot

by Marcus Hale

CONTENTS

I Vibe Coding Is a Stupid Name for a Real Thing

II The Only Three Tools That Matter Right Now

III Setting Up a Workflow That Doesn't Fight You

IV Prompts for Code Review

V Prompts for Refactoring

VI Prompts for New Features

VII Prompts for Tests

VIII Prompts for Debugging

IX When to Put the AI Away

X Verifying Generated Code Without Re-Doing It by Hand

XI Working on a Team Where Not Everyone Uses AI

XII Security, IP, and the Paste Rule

XIII The Honest Case For and Against Agent Loops

XIV Where This Is Going by 2027

CHAPTER I

Vibe Coding Is a Stupid Name for a Real Thing

Andrej Karpathy posted it on February 2, 2025. Twelve words that immediately annoyed half the software industry and got the other half nodding along in secret.

"There's a new kind of coding I call 'vibe coding,'" he wrote on X, "where you fully give in to the vibes, embrace LLMs, and forget that the code even exists."

Within forty-eight hours, the term was everywhere. Senior engineers were dunking on it. Startup founders were putting it in their pitch decks. Product managers were forwarding it to their engineering teams with zero irony. And somewhere in the middle of all that noise, something actually interesting got buried.

This chapter is about digging it back up.

Where the Name Came From and Why It Landed Wrong

Karpathy is not a crank. He co-founded OpenAI, ran AI at Tesla for years, and has a track record of identifying real patterns in how the field moves before most people notice them. When he named something, people paid attention — including people who immediately decided they hated the name.

The backlash was fast and, to be fair, partially earned. "Vibe coding" sounds like what a 22-year-old who learned Python last month tells people he does at parties. It sounds deliberately anti-serious. It has the same energy as calling surgery "vibe operating." Engineers who spent years learning systems programming, type theory, and the dark arts of debugging memory corruption felt, not unreasonably, that someone had just suggested their craft could be replaced by vibes.

Some of the criticism got specific. The term, critics argued, glorified a practice that produces code nobody understands. It encouraged developers to ship things without comprehension. It was one step from "I don't know why it works, I just kept asking Claude until it stopped crashing." That's not a programming methodology. That's a liability.

All of that is defensible. None of it is the whole story.

What Karpathy Actually Described

Here's what the critics mostly glossed over: Karpathy wasn't prescribing a workflow. He was describing a phenomenon. He was naming the sensation of a mode of working that millions of developers had already started doing quietly, without a label for it, often a little embarrassed about how much they were enjoying it.

Read his original post slowly. He said he "fully gives in to the vibes." He said he's "not really coding," he's "seeing stuff, saying stuff, running stuff." He said "it mostly works." He was not claiming this was the correct way to build production software for a Fortune 500 bank. He was describing what it feels like to prototype fast with an AI in the loop — the lowered friction, the reduced ceremony, the strange flow state where you're thinking about what you want rather than how to express it syntactically.

That's a real thing. It happens. If you've spent more than two hours with Claude Code or Cursor, you've experienced some version of it.

The problem was never the observation. The problem was that "vibe coding" became a banner for two very different populations: people using AI tools thoughtfully while acknowledging the phenomenology is genuinely different, and people using AI tools recklessly while using the term as cover. The second group gave the first group a branding problem.

The Sensation, Accurately Described

Let me try to describe what actually happens, stripped of the hype.

You're building something. New feature, greenfield module, small API wrapper — doesn't matter. In the old workflow, there's a gap between thinking about the problem and expressing it in code. That gap is filled with syntax lookup, boilerplate typing, mental juggling of variable names, and the minor friction of remembering exactly how Python handles nested list comprehensions versus how JavaScript does. The thinking and the typing are entangled. Sometimes the typing interrupts the thinking.

With a capable AI assistant in the loop, that gap compresses. You describe the shape of what you want, and a plausible implementation appears. You react to it — this part is right, this part is wrong, this edge case is missing — and the thing refines. You're still doing cognitive work. You're evaluating, directing, correcting. But the ratio of thinking to typing has shifted dramatically. The thinking, for stretches of time, is all you're doing.

That's what the vibe is. It's not sloppiness. It's a different allocation of your attention.

GitHub's 2022 study — 95 developers, controlled task conditions, the results published in October of that year — found that developers using Copilot completed the assigned task 55% faster than the control group. Fifty-five percent. That's not a rounding error. The developers in the Copilot group also reported higher satisfaction with their code. They weren't just faster; they enjoyed the work more.

Now, that study had real limitations. The tasks were contained and well-defined. The measure was task completion, not production readiness, not long-term maintainability, not code review time downstream. You can't extrapolate from "write a function to parse HTTP headers" to "maintain a distributed system across a five-person team for three years." But the productivity signal is real. It shows up repeatedly across different studies, different tools, different task types.

The vibe that Karpathy named has a measurable substrate. It's not purely aesthetic.

Why Experienced Engineers Are Suspicious Anyway

Here's the thing about being a senior engineer: you've been burned before. Not by AI — by every previous wave of "this will change how we code" tooling.

Visual programming environments in the nineties. Low-code platforms in the 2000s. No-code platforms in the 2010s. Every one of them arrived with demos that looked amazing and production realities that were humbling. Every one of them required you to fight the tool the moment you left the happy path. Every one of them generated enthusiasts who insisted you just weren't using it right.

So when a new generation of developers says "I just describe what I want and the AI writes it," the experienced engineer hears an echo. They've heard something like this before. And the last several times they heard it, the technology eventually revealed its ceiling, usually at the worst possible moment — in production, under load, with a bug that nobody could debug because nobody understood the generated code.

That suspicion is not technophobia. It's pattern recognition. It's earned.

Simon Willison, who runs simonwillison.net and has been writing about LLMs more rigorously and honestly than almost anyone else in public, put his finger on something important. He's argued consistently that the question isn't whether AI tools can write code — they obviously can — but whether the developer using them actually understands what they're shipping. The tool's capability and the user's comprehension are two separate variables. Conflating them is where things go wrong.

That's the correct skepticism. Not "this technology is fake" — it isn't. But "are you actually in control of the code you're shipping?" That's the question that matters, and it's the one that the vibe-coding discourse mostly skipped over.

The Two Failure Modes

There are two ways to get this badly wrong, and they're almost mirror images of each other.

Failure mode one: full conversion. You read Karpathy's post, nodded enthusiastically, started prompting your way through every task, and stopped reading the output carefully somewhere around week three. Things worked, mostly, until they didn't. The bug that surfaced six weeks later turned out to be in a function you'd accepted without really reviewing, generated by a model that had confidently misunderstood your intent. You spent more time debugging that function than it would have taken to write it yourself. You've now made the exact mistake the skeptics predicted.

Failure mode two: principled refusal. You read the Karpathy post, rolled your eyes, closed the tab, and continued working exactly as you had before. You have thoughts about craft and comprehension that are not wrong, but you've also quietly become slower than your peers on certain classes of tasks. You're spending forty-five minutes writing boilerplate that someone else's AI generated in ninety seconds. Your throughput on

new features is lower than it needs to be. You're spending your limited cognitive budget on things that don't require it, leaving less for the things that do.

Both of these developers are making errors. The first has outsourced their judgment. The second has refused a tool that would genuinely help them. The goal of this book is to put you somewhere between those two points — using the tools where they help, keeping your hands on the wheel, and knowing the difference between the two situations before you're already in one.

What Changed in 2025

It's worth being specific about why 2025 is the year we're having this conversation rather than 2022 or 2023.

The earlier generations of coding AI tools — early Copilot, GPT-3 completions, the first round of code-gen experiments — had real capability but real limits. They were good at completing lines. They were decent at suggesting short functions. They struggled with anything requiring sustained context: a function that needed to coordinate with five other functions, a refactor that touched multiple files, a debugging task where the error was ten layers away from the symptom.

The 2024-2025 generation of tools is meaningfully different. Context windows grew. Models improved at multi-step reasoning. Claude 3.5 Sonnet, released in mid-2024, raised the bar for code tasks noticeably. Claude Code — Anthropic's terminal-native agent released in early 2025 — introduced something qualitatively different from autocomplete: an AI that could read your repo, propose a plan, execute it across multiple files, run tests, and iterate on failures. Cursor's integration of frontier models into an editor-native workflow gave developers an experience where the AI had genuine awareness of the project, not just the current file.

These aren't incremental improvements to the same thing. They're a different thing.

Which is partly why the vibe-coding discourse arrived when it did. The phenomenon Karpathy named became visible at scale because the tools finally had enough capability to produce it consistently. Earlier tools could occasionally give you that feeling. The 2025 generation of tools gives it to you most of the time, across a much wider range of tasks. That's a phase transition, not a point on a curve.

The Hype Gradient

Every real technology arrives bundled with hype that doesn't match the technology. The hype and the reality have different shapes, different timelines, different costs. Getting them confused is how you end up disappointed, or reckless, or both.

The honest case for AI-assisted development in 2026 looks like this:

You will write certain classes of code faster. Boilerplate, repetitive transformations, API integrations with well-documented endpoints, first-draft unit tests for functions with obvious inputs and outputs — these tasks accelerate meaningfully. Not 10% faster. Sometimes 5x faster. The GitHub numbers are directionally correct

even if you can't apply them mechanically to every situation.

You will make certain classes of mistakes more easily. Accepting code you don't understand is now a much lower-friction mistake than it used to be, because the code looks authoritative and is syntactically correct. The model can be confidently wrong. It hallucinates API signatures, gets library versions wrong, makes assumptions about your data that don't match your actual data. These errors look like regular code until they don't.

You will feel occasional spurts of genuine flow state. The thing Karpathy described is real and it's pleasant. It can also be a trap, because flow state feels productive regardless of whether you're actually being productive.

You will not be replaced by a language model. Not this year. Probably not by 2027 either, though the role is shifting in ways worth thinking about. The chapter at the end of this book goes into this more carefully. For now: the developers who will have the worst time in the next few years are the ones who either refuse to adapt or adapt in ways that hollow out their own understanding.

What "Vibe" Actually Names

Here's my actual working theory about why the word "vibe" stuck, despite how much it annoys people.

It names a relationship to code that's genuinely new. Before LLMs, the developer's relationship to their codebase was primarily expressive — you were the author, the code was your text, and every character in it was something you had consciously chosen. You might use libraries and frameworks written by others, but you understood (or could understand) what you were invoking. The mental model of authorship fit.

That model is strained now. When Claude Code writes a 200-line module and you read it, test it, correct two things, and ship it — are you the author? In some sense yes. In another sense, this is more like the relationship an architect has to a building they designed but didn't physically construct. The intellectual ownership is real. The line-by-line authorship is not.

"Vibe" captures something about this imprecision. You're navigating by intention and evaluation rather than by word-by-word composition. You know what you want. You judge whether you got it. The middle part — the actual code generation — happens in a black box you don't fully control. That's a new kind of relationship with your software, and it requires a new kind of discipline that the old authorship model didn't prepare you for.

The discipline is verification. The habit is skepticism. The skill is knowing how much to trust a given output on a given class of task. This book is mostly about developing those three things.

The Premise, Stated Plainly

I'm not here to sell you on AI tools. You probably already have opinions and some of them are probably correct. I'm not here to validate the hype cycle or tell you that the skeptics are all Luddites. They're not.

What I think is true, based on using these tools daily for the past year across real projects:

The productivity gains are real and unevenly distributed. You capture them through deliberate practice, not passive adoption. The risks are also real and mostly concentrated in a few specific failure modes that are straightforward to avoid once you know what they look like. The tools are not magic. They're also not snake oil. They're powerful, weird, and require a kind of meta-cognitive awareness that most developer education didn't prepare us for.

The vibe is real. The hype is optional. You can use these tools — seriously, professionally, without embarrassment — without pretending they're something they're not.

That's the entire premise of this book. Everything that follows is operational detail.

On Drinking the Kool-Aid

One more thing before we move on.

There's a version of AI-tools enthusiasm that functions like a religion. True believers who can't hear criticism. Converts who are visibly impatient with anyone who hasn't yet seen the light. People who explain, gently but repeatedly, that if you're skeptical it's only because you haven't tried it correctly. That version is annoying and it's also bad epistemics.

Willison has been a useful counter-voice here. He uses LLMs extensively — he's documented this publicly across hundreds of posts — and he maintains a healthy suspicion of his own enthusiasm. He writes about what they're bad at with the same specificity he uses to write about what they're good at. He doesn't treat any particular tool as a personality. He updates his views when the tools change.

That's the posture. Not cynicism. Not evangelism. Calibrated, revisable, grounded in actual use.

The developers I know who are getting the most value from AI tools in 2026 share a few traits. They've developed a fast inner signal for when the AI output is trustworthy on a specific type of task versus when it needs careful review. They've built workflows that make verification easy rather than bolting it on as an afterthought. They don't use AI for tasks where their own understanding of the system is irreplaceable. And they don't particularly care whether other people consider their workflow legitimate.

They're not vibe coders in the reckless sense. They're just developers who added a capable new tool to a competent existing practice. The vibe is present. The judgment is still theirs.

Where This Leaves Us

Karpathy named something real with a name that will probably embarrass its subject slightly. The name will either stick or be replaced by something more respectable-sounding. Either way, the phenomenon it describes — the changed texture of working with AI as a close collaborator on code — isn't going away.

The question worth asking isn't "should I vibe code?" That question has bad framing on both sides. The question is: what do I actually want from AI tools, what am I willing to trade for it, and how do I keep my engineering judgment intact while capturing the genuine productivity available here?

Those questions have specific answers, not vibes. The rest of this book is the answers.

Claude Code, Cursor, and Copilot are the three tools you'll hear most about in the chapters that follow. They're not interchangeable, and which one earns your primary trust depends on how you work and what you're building. But they share the same basic dynamic: they're fast, they're capable, they're confident even when they're wrong, and they will reward you in proportion to how clearly you think before you prompt.

The vibe is a starting point. Understanding is the thing you can't outsource.

That's the setup. Let's get to work.

CHAPTER II

The Only Three Tools That Matter Right Now

The AI coding tool market currently contains somewhere north of forty products that will each tell you, with full sincerity, that they're the only one you need. Substack newsletters have reviewed most of them. Indie hackers have launched half of them. A fair number will exist for another eighteen months before quietly shutting down or pivoting to enterprise sales.

Three actually matter for working developers in 2026. Not because the others are uniformly bad, but because these three have crossed a threshold — in capability, in adoption, in toolchain integration — where they're genuinely worth building habits around. The interesting question isn't which one is best. It's which one is best for your specific situation and what it costs you when it's not.

This is not a feature matrix. Feature matrices are how product marketers compare software. You and I think about software differently: does it work when I need it, does it get out of the way when I don't, and what does it cost — in money, in time, in the debugging sessions that follow when I trust it too much?

Claude Code, Cursor, and GitHub Copilot give very different answers to those questions. Here's the honest accounting.

Claude Code: The Contractor You Brief in the Morning

Anthropic shipped Claude Code as a generally available product in early 2026. The core pitch is simple: an agentic coding assistant that runs in your terminal, reads your filesystem, executes shell commands, reads your git history, and works across a whole codebase rather than a single open file. You describe a task. It decides what to do about it.

The UX philosophy here is delegation. Not augmentation, not suggestion — delegation. Cursor and Copilot both sit beside you while you work. Claude Code is closer to a contractor you brief on a task and then leave alone for forty-five minutes. That distinction matters more than any capability benchmark.

When Claude Code wins

The sweet spot is complex, multi-file work that requires holding a lot of context simultaneously. Take a realistic example: you need to refactor a service layer to use dependency injection across thirty files that all touch the same interface. Doing that with inline AI suggestions is miserable — you're prompting file by file, losing track of what changed where, manually stitching together a coherent result across twenty separate sessions. Claude Code, given a clear task description and permission to explore the repo, will map the dependencies, plan the edit sequence, execute the changes in order, and report what it did. It doesn't hand you a suggestion. It hands you a diff.

Understanding work is another genuine strength. Before you touch a legacy codebase you've never seen, asking Claude Code to orient you — trace how a request flows from entry point to database, identify where the business logic is concentrated, flag anything that looks inconsistent with the rest of the codebase — saves you two to three hours of cold reading. When that reading would otherwise be expensive senior developer time, the economics are obvious.

Simon Willison — who has been writing about LLM behavior longer and more rigorously than most practitioners, at simonwillison.net — has observed that Claude's models tend to excel specifically on tasks requiring multi-step reasoning and careful, explicit chain-of-thought work. That matches the practical experience. When a task requires holding multiple competing constraints in mind, tracking state across a long reasoning chain, and catching its own contradictions, Claude Code outperforms the alternatives in this list.

Long context is a real advantage rather than a spec sheet number. Claude 3.7 Sonnet, Claude Code's current default model, handles 200,000 tokens of context. In practical terms: you can pass it a large module, several related test files, a database migration, a style guide excerpt, and a plain-language description of your goal without hitting a wall. That wall-hitting is a real UX cost in other tools. When you lose context midway through a session, you lose the thread.

When Claude Code embarrasses you

It is slow for interactive work. Meaningfully slower than inline autocomplete. If you want a function signature completed while you're in flow, Claude Code is the wrong tool. It is not where you go for fluency. It's where you go for thoroughness on tasks that justify the overhead.

The terminal interface is alienating for developers who've spent their entire career inside a GUI IDE. There's no inline diff renderer by default, no click-to-accept button beside a suggestion. Everything happens in a separate window, which is a workflow adjustment that takes a week to internalize. Some developers never fully make peace with it.

Cost at scale is real and easy to underestimate. Claude Code runs on usage-based billing unless you're on a Max subscription, which runs $100/month as of mid-2026. Heavy agentic loops on large codebases — long back-and-forth sessions on big context windows — accumulate charges in ways that will surprise you if you're used to flat-rate tools. The first month of heavy use has caused at least one developer I know to have an uncomfortable conversation with their accounting software.

The honest failure mode: Claude Code will make confidently wrong architectural decisions when given too much latitude and insufficient constraint context. It doesn't know your team has a rule against adding new dependencies without a review cycle. It doesn't know the function it just replaced was doing something subtle that the name didn't convey. Agentic tools fail at the edges of their context window. The broader the task and the thinner your briefing, the more you need to be watching the output before you run git commit.

Cost reality in practice: Developers using Claude Code for focused 30-to-60-minute sessions on genuinely hard tasks — not as a constant background assistant — typically land around $30 to $60/month at current usage-based pricing. The Max subscription makes sense for heavy users. The discipline is treating it like a

professional engagement rather than an open faucet: session with a clear objective, explicit constraints, human review at the end.

Cursor: The Editor That Assumes You're Always Working With It

Cursor is a fork of VS Code with AI integrated at the foundation rather than added as an extension. That architectural distinction sounds minor. It's not.

When you install Copilot into VS Code, you're adding AI to an editor. When you use Cursor, you're using an editor that was designed from the start on the assumption that you and a model are working together at all times. Every interface decision reflects that assumption: the tab completion model, the chat panel that already knows your cursor position, the composer that diffs across multiple files while you stay in place.

The UX philosophy is residency. The AI doesn't live in a sidebar you open when you have a question. It lives in the editor with you, continuously, holding context about what you're writing, what you wrote yesterday, and what the rest of your project looks like.

When Cursor wins

Tab completion at its best is the capability that genuinely surprises experienced developers the first time they encounter it. Not single-line autocomplete — full multiline completions that finish the function you're writing, infer the next method you need, and sometimes complete three lines simultaneously in exactly the pattern you would have chosen. Developers who use this daily describe it as the sense that the editor has learned how they think. That's not mysticism. It's a model with deep context on your codebase and your recent edits making probabilistic bets on what comes next. When those bets are right, the velocity gain is real and it compounds. You start typing less and shipping more.

The contextual chat interface has one practical advantage over every external tool: the model already has your file, your cursor position, your recent edits, and your project structure loaded before you type your first character. When you ask "why is this function returning nil here?" you don't paste anything. It already knows where "here" is. That zero-friction context handoff, used twenty times a day across a week, adds up to meaningful saved time.

Composer — Cursor's multi-file edit mode — handles medium-complexity refactoring well. Not as autonomously as Claude Code on a large agentic task, but for "change this function's signature and update all callers," it's faster and more controllable. You see the diff. You accept or reject file by file. You never leave the editor. That continuous feedback loop is a real advantage over tools that hand you a finished result and ask you to review it cold.

Model flexibility is underrated. Cursor lets you route requests to different underlying models — Claude 3.7 Sonnet, Opus, GPT-4o, and others — per task. For complex reasoning, you route to the heavy model. For quick completions, you use a faster, cheaper one. You're not locked into a single cost-capability tradeoff for

all work.

When Cursor embarrasses you

The "it already knows your context" advantage reverses when the important context is in a file you haven't touched recently. Cursor's project model is built from what's open, what you've edited, and what its indexer has processed. If the critical constraint for your current task lives in a configuration file three directories away that you haven't touched in two months, Cursor may answer your question confidently and incorrectly, because that file isn't in its active context. It will not signal uncertainty. It will give you an answer.

The forked-editor model imposes a real migration cost. VS Code extensions don't all work correctly in Cursor. Some do, some crash, some work but behave differently, and the list changes with each Cursor release. If you have a carefully tuned VS Code setup — language servers, linters, custom keybindings, a collection of extensions you've accumulated over years — plan for a migration day that is unglamorous and occasionally annoying.

Tab completion also has a specific failure mode when you're learning. Working in an unfamiliar library with Cursor aggressively completing your code can produce fluent, plausible output that is subtly wrong for the version you're using or the pattern the library actually expects. When you need to read and understand — when the goal is learning, not speed — constant suggestions become interference. This is a tool with a specific personality: it rewards expertise and punishes ignorance in ways that are easy to miss until something fails at runtime.

Cost reality: Cursor Pro is $20/month. That includes the tab completion model, 500 fast requests per month to frontier models, and unlimited slower requests. For most developers using it as a primary editor, the value-per-dollar is high. The ceiling to watch: heavy usage of Opus or the most powerful frontier models through Cursor will burn through your fast request quota quickly, slowing you down or surfacing additional cost.

GitHub Copilot: The Tool That Already Won the Adoption War

Let's be direct: Copilot is not the most capable tool on this list. On complex reasoning tasks, Claude Code will outperform it. On tab completion in a well-configured project, Cursor is at minimum competitive. Copilot does not win capability benchmarks against focused competitors.

What it won is distribution. GitHub reported over 1.8 million paid subscribers by mid-2024, and Microsoft has since integrated Copilot into VS Code, GitHub PRs, GitHub Actions, and the broader Microsoft 365 ecosystem in ways that make it simply present in the tools a large percentage of working developers use every day. Enterprise adoption has followed Microsoft's sales apparatus, and those deals compound. For a huge number of teams, Copilot is not chosen — it is inherited, because the organization already has a GitHub or Microsoft enterprise agreement.

The UX philosophy is ambient assistance. Copilot does not ask you to change how you work. It watches what you're typing and offers completions inline, in the editor you already have, without requiring a workflow change. That's a real virtue. The lowest adoption friction on this list.

When Copilot wins

Boilerplate and repetitive patterns. If the code you're writing is correct, predictable, and structurally common, Copilot is genuinely excellent at it. Generating a REST handler that matches your project's established pattern, writing five variations on a utility function, adding docstrings to a module you're about to publish — Copilot handles these faster than any other tool here, because it needs no briefing. It watches the pattern and continues it.

GitHub Actions integration is better than it gets credit for in most comparisons. Copilot's inline suggestions for CI/CD YAML, informed by the context of your repository's existing workflows and the broader GitHub Actions ecosystem, are specifically good. Being an integrated GitHub product matters here in a way it doesn't on most other task types.

The enterprise IP risk management features are real and affect professional decisions. Copilot Business and Enterprise include code referencing filters — a system that checks generated suggestions against a database of public code and flags or suppresses suggestions that match licensed open-source code above a configurable similarity threshold. That's not a vanity feature. That's the reason legal teams at non-trivial companies approved the Copilot purchase. Microsoft's IP indemnification policy for Business and Enterprise customers — which commits to defending customers in lawsuits arising from Copilot-generated code — is similarly real, similarly untested at scale in court as of 2026, and similarly the kind of thing enterprise procurement notices.

When Copilot embarrasses you

API staleness is a persistent problem. Models trained on data with a cutoff date will suggest APIs that have since changed, libraries that have deprecated the method, and patterns that were correct in 2022 and wrong now. You'll see syntactically valid code in your editor, and your IDE won't complain until something fails. The newer Copilot models are better at this than earlier versions, but the problem hasn't gone away. Any tool that generates code from statistical patterns over historical training data has this failure mode, and Copilot has it prominently.

The completion model's statistical bias toward common patterns is a feature for common tasks and a liability for unusual ones. Copilot will consistently nudge you toward the generic approach even when your situation requires something specific to your architecture. When you resist the suggestion and write what you actually need, the experience degrades — you're fighting the autocomplete rather than using it.

GitHub's 2022 controlled study found that developers completed benchmark tasks 55% faster with Copilot. That number is real but narrow. The tasks in that study were isolated, well-defined, and representative of the kind of work where autocomplete acceleration applies most directly. They were not representative of the cross-cutting, context-dependent, ambiguous work that consumes most of a senior developer's day. The

headline applies, just not uniformly.

Copilot's agentic features — Copilot Chat, Copilot Edits, the newer agent-mode additions — feel like a company catching up to a paradigm its original product wasn't designed for. They work. They're not the reason you'd choose Copilot as your primary in 2026.

Cost reality: $10/month individual. $19/user/month for Business. $39/user/month for Enterprise. Individual cost is the lowest barrier on this list by a significant margin. Business tier is competitive for teams that need the compliance and org management features. Enterprise value depends almost entirely on whether your team genuinely uses the GitHub-native capabilities — PR summaries, internal knowledge bases, deep Actions integration — or pays for them and ignores them.

The Decision Heuristic

You don't need to pick one and delete the others. But you probably should pick a primary — the tool you reach for without consciously deciding — and use the others when they're genuinely better suited.

If most of your work is large, complex, multi-file — architectural changes, unfamiliar legacy codebases, significant structural refactoring — Claude Code is your primary. Budget accordingly. Learn to scope tasks tightly. Treat each session as a briefing, not a chat.

If most of your work is iterative feature development in a codebase you know — new components, incremental improvements, the daily texture of building within a familiar structure — Cursor is your primary. The tab completion and always-present context will compound over time in ways that produce real velocity.

If your primary constraint is team adoption, existing GitHub enterprise contracts, or legal IP risk management — Copilot is your primary. Accept its capability ceilings, use it for what it does well, and stop expecting it to do the things it doesn't.

A practical overlay that holds for most working developers: run Copilot or Cursor as your ambient tool, and reach for Claude Code when a task is genuinely hard enough to justify the briefing overhead and the usage cost. They're not competing for the same tasks. They operate at different altitudes.

On Not Picking a Religion

The worst thing you can do with these tools is choose a tribe.

The developer who insists Copilot is a toy because Claude Code handled their last refactor better is making the same mistake as the developer who wrote off static typing because they worked on a dynamic codebase where it didn't seem to matter. The situation changed. The opinion didn't.

These are instruments. Instruments get selected based on the work in front of you, not on identity.

Three weeks of real work — actual tasks, not weekend toy projects — is enough to form an honest opinion about any of these tools for your specific workflow. Track, loosely: where did it help, where did it waste your time, where did you catch an error before it shipped. Then make a decision with real data from your own situation, not from a product launch video or a conference talk or this chapter.

These tools are moving fast enough that any static ranking has a shelf life of six months, maybe less. The mental model that ages well is not "Claude Code is best" or "Copilot is for enterprise." The model that ages well is: understand what the tool optimizes for, recognize when your task matches that optimization, and use it accordingly. When it doesn't match, put it down.

That's not indecision. That's the habit of thought that keeps you effective when the landscape shifts again — and it will, probably before you finish this book.

CHAPTER III

Setting Up a Workflow That Doesn't Fight You

Most AI-assisted dev setups look great on day one. You've installed the extension, you've watched the autocomplete fire, you've felt that particular buzz when Claude rewrites a 40-line function in four seconds. Everything is smooth. You are, you think, finally living in the future.

By day eight, the context is a mess. The model is suggesting code that contradicts your actual architecture. You're pasting the same project conventions into every new session like some kind of manual human memory system. The autocomplete is completing things in a style that belongs to a different codebase entirely, maybe a different decade. You spend twenty minutes cleaning up a suggestion that would have taken you ten to write from scratch. You briefly wonder if you should just go back to Vim.

The setup didn't fail because AI coding tools are bad. It failed because good tools still require a workflow, and "just install it and see what happens" is not a workflow.

This chapter is about building a setup that lasts past week one — specifically, the small structural decisions around context management, model selection, editor config, and daily habit that separate a durable AI-assisted workflow from an expensive experiment in frustration.

Context Is the Whole Game

Here's the thing nobody puts in the marketing copy: the quality of output from any of these tools is almost entirely determined by the quality of the context you give them. Not the model. Not the temperature setting. Not whether you paid for the Pro tier. Context.

Simon Willison, who has been writing about LLM-assisted development on his blog with more rigor and less hype than almost anyone else working in public, made this point bluntly in a 2024 post: the difference between a useful LLM output and a useless one is usually what you put in, not which model you called. The models have gotten good enough that this gap is the bottleneck now.

Context has three layers, and you need to manage all three.

Project-level context is the stuff that's always true: your language, your framework, your naming conventions, your architectural constraints, your dependencies, your test strategy. This is the context you need every session without exception.

Session-level context is the stuff that's true for this working block: which file you're in, what problem you're solving, what you've already tried, what the relevant data structures look like right now.

Per-prompt context is what you paste in for a specific question — a stack trace, a function, a PR diff, a failing test output.

Most developers only think about the third one. The first two determine whether the third one produces anything coherent.

The Instruction Files

The single highest-leverage thing you can do is write a persistent instruction file for each tool. All three major tools support this; they just call it different things and put it in different places.

For Claude Code, the file is CLAUDE.md, and it lives at the root of your project. Claude Code reads it automatically at the start of every session. This is not a configuration file in the INI-file sense — it's prose, written to Claude the way you'd write a brief to a new contractor who is technically excellent but knows nothing about your specific situation. Anthropic's documentation describes it as "project memory," which is accurate but undersells what it actually does. It's your chance to answer every question that would otherwise be answered wrong by default.

A minimal CLAUDE.md for a production Node.js service might look like this:

```markdown

## Project: payments-api

Node 20, TypeScript strict mode, Express 5, Postgres 16 via pg-promise. All handlers return a typed ServiceResult<T> — never throw raw errors. Tests live in /test, use Vitest, run with npm test. No Lodash. No Moment. No new dependencies without asking. The /src/legacy/ directory exists for a reason. Do not refactor it. ```

That last line matters. AI tools will helpfully refactor code that you are not allowed to touch, that is in the middle of a migration, that has a legal review pending. You have to tell them to leave it alone. They will not guess.

For Cursor, the equivalent is .cursor/rules (previously .cursorrules before Cursor updated their format in 2025). Same idea. Cursor's documentation recommends structuring these as project rules — markdown files inside .cursor/rules/ that can be tagged with alwaysApply or scoped to specific file globs. This is more granular than a single CLAUDE.md and worth the extra two minutes to set up if you have a monorepo with different conventions per package.

For GitHub Copilot, the file is .github/copilot-instructions.md. Microsoft added this in late 2024 and it's now picked up by both Copilot Chat and the inline completions engine. Copilot's instruction parsing is still somewhat less reliable than Claude Code's — in my experience, complex conditional instructions get fuzzy — but the basics work.

Write these files. Commit them. Treat them as living documents. Update them every time the model does something wrong that it shouldn't have done if it had known your codebase.
```

Model Selection Defaults

Not every task deserves your most expensive model, and the defaults that ship with these tools are often optimized for impressiveness rather than efficiency.

In Claude Code, you're primarily choosing between Claude Sonnet and Claude Opus (with Haiku available for lighter tasks). Sonnet 4 is fast, cheap, and correct about 85–90% of the time on ordinary development work. Opus is slower, meaningfully more expensive, and noticeably better on tasks that require sustained multi-step reasoning — architectural decisions, complex refactoring with non-obvious constraints, debugging a failure mode that involves five interacting components. Use Sonnet for the bulk of your work. Use Opus when you're genuinely stuck or when the cost of a wrong answer is high.

In Cursor, you can configure a default model per task type. This is worth doing. Set the inline completion model to something fast — Claude Sonnet or GPT-4o-mini — because you'll trigger it hundreds of times a day and latency matters more than raw capability for tab-completion. Set the Chat panel to a stronger model for deliberate, pull-based conversations.

In Copilot, the model selector is in the Chat panel. As of mid-2026, Copilot Chat supports Claude Sonnet, GPT-4o, and o3, and the right choice depends on what you're doing. For exploratory conversations about design or architecture, o3's reasoning mode is genuinely useful. For "generate a migration for this schema change," 4o is fast enough.

The heuristic is: fast and cheap for ambient assistance, slow and expensive for stuck-and-thinking. The default everywhere is probably too expensive for what you're doing in the 80% case.

Editor Config That Stays Out of Your Way

The actual VS Code or Cursor settings that matter for an AI-assisted workflow are less exotic than you'd think. A few that make a real difference:

Keep your workspace tight. The more files in the active workspace, the fuzzier the context Cursor can pull in. In large monorepos, open a workspace from the specific package you're working on rather than the root. This isn't a settings change — it's a habit — but it meaningfully improves suggestion quality.

Turn off completions in comments and strings. Both Cursor and Copilot will autocomplete inside string literals and code comments. This is almost never what you want. In VS Code/Cursor settings, "editor.quickSuggestions" lets you disable suggestions in strings and comments independently of regular code context. Do this.

Set a short autocomplete delay. The default delay before inline suggestions appear is often 300ms or more. In a fast-moving flow state, this feels like friction. "editor.inlineSuggest.delay" can be reduced to 75ms. This is personal preference, but developers who switch to a shorter delay rarely switch back.

Configure your terminal model explicitly. Claude Code runs in the terminal, not the editor, which means its context window is separate from whatever Cursor has loaded. Don't assume they share state. They don't. If you're switching between Cursor chat and a Claude Code session in the same codebase, you need to be explicit about context in both places.

One setting that trips people up: Cursor's "auto-context" feature, which automatically pulls in files it thinks are relevant. This is useful in small projects and unreliable in large ones. You can see what files Cursor added to your context by looking at the chat UI — there are pills at the top of the response showing attached files. Get in the habit of glancing at those. If Cursor pulled in /src/legacy/v1/auth.ts when you're working on /src/api/v2/, that's why the suggestion looks weird.

File Structure Conventions for AI-Assisted Projects

Your file structure signals intent to both humans and AI tools. A few conventions that work well:

Keep related context co-located. AI tools build their understanding of what you're doing from the files physically near what you're editing. If your types live in /types/, your utils in /utils/, and your tests in /test/, the model has to chase files across the project to understand a single function. Colocation — keeping the component, its types, its utils, and its tests in the same directory — results in noticeably better suggestions.

Use explicit naming over clever naming. getUserByEmailWithRetry is better than fetchUserE for AI-assisted development even if the shorter name is fine for humans who know the codebase. The model has to infer intent from names constantly. Give it clear signals.

Put your CLAUDE.md (and equivalent files) in source control and treat them with the same seriousness as README.md. I've seen teams maintain beautiful READMEs and garbage CLAUDE.md files that haven't been updated since the initial project scaffold. The instruction files are actually load-bearing now. They deserve the same attention.

Create a /prompts/ directory for reusable prompts. Not everyone does this, but it's worth considering for any project where you're running the same types of AI-assisted tasks repeatedly. A file called prompts/review-pr.md with your current preferred review prompt is version-controlled, team-shareable, and beats "I think I had a good prompt for this last month somewhere."

Mark AI-generated files explicitly when it matters. For generated boilerplate, tests scaffolded from specs, or docs generated from code, a comment at the top of the file — # generated by Claude Code from spec.md, 2026-04-11, reviewed by [name] — costs you fifteen seconds and provides meaningful context for future reviewers, including yourself six months later. This is especially relevant given the ongoing conversation about AI attribution; Chapter 11 covers the team dynamics in more detail.

The Small Habits That Actually Matter

The workflow collapses when the habits are wrong. These are the ones that matter most:

Commit before you start an AI session. Always. If you have uncommitted changes and the AI suggests a broad refactoring that you accept, and it's wrong, your recovery path is a mess. A clean git status before you start a vibe session means you can git checkout . your way out of anything in under five seconds. This one habit prevents more wasted time than any model upgrade.

Scope requests to one change at a time. The error rate on "refactor this entire module to use async/await and also add error handling and also update the tests" is dramatically higher than doing three separate passes. It's not that the model can't attempt compound requests — it's that each additional requirement multiplies the probability of something going wrong, and when something does go wrong in a multi-step output, the diff is large and the errors are buried.

The 2022 GitHub and Microsoft research on Copilot's impact on developer productivity — the one that showed developers using Copilot completed certain tasks 55.8% faster — was measuring focused, bounded tasks. A new HTTP handler, a utility function, a unit test for a specific behavior. Not "refactor this whole service." Scope accordingly.

Read the diff before you apply it. This sounds obvious. It's routinely skipped. Cursor's compose panel shows you a full diff before you accept changes. Claude Code shows the changes inline. In the flow state, it's tempting to see green checkmarks and hit accept. Don't. Five seconds of reading the diff has caught more subtle bugs for me than an hour of post-hoc debugging. The model is very good at appearing confident while being wrong in a small but load-bearing way.

Say what you don't want, not just what you do. Negative constraints are underused. "Refactor this to remove the callback pattern — don't change the function signature, don't change the error handling behavior, don't add any new dependencies" produces a more targeted result than "make this more modern." The model interprets omission as permission.

Keep a session log for complex problems. For debugging sessions or architectural explorations that span multiple exchanges, paste the key decision points into a scratch file. The context window has a limit; long sessions degrade. Knowing that you tried approach A and it failed for reason X is context you'll want when you come back to the problem after lunch, or when the model starts forgetting it because the context was flushed.

The Author's Setup

This is where you're entitled to be skeptical, so: here's what I actually use, stated plainly, with the caveat that your situation is different and you should treat this as one data point.

Primary tool is Claude Code for any task that requires extended back-and-forth reasoning — debugging sessions, architectural decisions, and writing code that needs to fit precisely into a specific context. The terminal interface is an advantage, not a drawback. It keeps the tool separate from the editor, which keeps the

cognitive modes separate. I am either writing code or I am talking to the model. When both happen in the same panel, I find my judgment gets blurry.

Cursor for ambient assistance in TypeScript and Python projects. The inline completions are fast enough now that they've largely replaced my habit of typing out boilerplate by hand. I have Cursor configured with two rules files: one that applies to all files in the project, one that applies specifically to test files (different style conventions, different constraints on what to import).

GitHub Copilot is open in a secondary VS Code window for reading unfamiliar codebases I don't own. Not for writing — for asking questions about what something does. Copilot's "explain this code" capability is genuinely good, and I'm not polluting my primary editor with a second AI instance competing for the same keystrokes.

My CLAUDE.md template, which I fork for new projects, has five sections: language and runtime, framework and libraries, code style (short, and not a replacement for an actual linter config), what to avoid, and what to ask before changing. That last section is where I put things like "this function has a specific side effect on startup that is not obvious from its name — ask me before changing it." It's the section I update most often.

I run a linter and formatter in pre-commit hooks. Always. AI-generated code skips style enforcement constantly, even with explicit instructions. The hook catches it without requiring me to remember.

I do not use agent mode for anything that touches production. Not because it doesn't work — it often does — but because the blast radius of a wrong autonomous action on a production-adjacent system is large and I am not yet confident enough in the guardrails to accept it. Chapter 13 covers agents properly. The short version is: in 2026, supervised loops for bounded tasks, not autonomous end-to-end execution on real infrastructure.

The Collapse Patterns

Setups fail in predictable ways. If you know the patterns, you can catch them early.

Context drift. The CLAUDE.md gets out of date, the model starts making suggestions based on old conventions, you stop trusting it, you start ignoring it. Solution: add a line to your weekly cleanup ritual — five minutes, update the instruction files to reflect what changed.

Scope creep in prompts. You start asking compound questions because the answers were good, the requests get bigger, the responses get longer and less reliable, you accept a large diff without fully reading it, something breaks in production. Solution: reset to single-task prompts when you notice the output quality dropping.

Over-reliance on session continuity. Long sessions where you assume the model remembers everything from the beginning. It doesn't — not reliably. At around 80k tokens of context in a Claude session, the effective memory of early exchanges degrades. Solution: summarize and re-anchor at natural breakpoints ("To summarize where we are: we're trying to do X, we've tried A and B, the constraint is C").

Tool competition. Cursor's AI and Claude Code both have opinions about the same file. You accept Cursor's suggestion, then ask Claude Code a question, and it answers based on the file as it was before Cursor's change. The tools don't share state. Be explicit about which version of the file the model is looking at. Paste the relevant section if there's any doubt.

The goal of this setup is not to maximize the number of AI interactions. It's to make each interaction reliable enough that you trust the output — which means your decision-making time drops, your error rate stays low, and the net effect on your velocity is actually positive rather than theoretically positive. That requires structure. The structure is not complicated. But you do have to build it.

Start with the instruction files, the commit habit, and the diff-reading habit. Those three alone will move the needle. Add the rest as the friction points become clear. This is a starting point, not a final answer — the tools are changing fast enough that any setup is provisional anyway. What matters is building the habit of maintaining the setup as the tools evolve, rather than letting it calcify into something that fights you.

CHAPTER IV

Prompts for Code Review

The first time most developers use AI for code review, they do something like this: paste the diff, type "review this," and wait. The model responds with a numbered list. Some observations are obvious. Some are wrong. None of them would have stopped the bug that got into production last month. You close the tab and go back to your usual process.

That's the rubber-stamp version. Not useless, exactly — but not a reviewer. It's a mirror that's slightly too polite to show you what's actually wrong.

The gap between that experience and genuine AI-assisted code review is almost entirely about how you prompt. Not the model. Not the tool. The prompt.

The Rubber-Stamp Problem

Here's the thing about asking an AI to "review this code": the model has no idea what it's optimizing for. Review for what? Correctness? Security? Performance? Style? Whether it matches your team's existing patterns? Whether it'll survive a Monday morning when someone else is modifying it without the original context?

When you leave that ambiguous, the model fills the gap with the most agreeable answer it can generate. It'll note that your variable names are "clear and descriptive." It'll say the logic "appears correct." It'll flag that you might want to add error handling, which you already know. This is not a code review. This is a compliment with footnotes.

Simon Willison — whose blog is probably the most consistently grounded writing available on practical AI usage — has made this point in various forms: models are trained to be helpful, and "helpful" in the absence of specific instructions usually means "reassuring." If you want the model to find problems, you have to explicitly ask it to find problems. You have to make finding problems the actual assignment.

That reframe — from "review this" to "find what's wrong with this" — is where the chapter actually starts.

What You're Asking For vs. What You Need

Before we get into specific prompts, it's worth being precise about what code review is actually trying to accomplish. Most PR reviews are checking for at least four distinct things simultaneously:

- Correctness — does the code do what the ticket says it should do?
- Logic errors — does the code do what the author thinks it does?

- Security — does the code introduce vulnerabilities, even unintentionally?
- Maintainability — will this code be comprehensible and modifiable in six months?

Human reviewers handle all four in a single pass, which is why code review takes time and why it's frequently shallow when everyone is busy. The AI advantage isn't that it's better at any of these individually. It's that it's tireless and fast, and it doesn't get bored on the third function in a 600-line diff.

But you have to ask for each dimension separately, or the model blurs them together into a generic assessment that addresses none of them adequately. The most effective review workflow is exactly that: separate passes, separate prompts, each optimized for one kind of signal. This sounds slower than it is. Each pass takes two to five minutes. You're not spending thirty minutes iterating; you're spending ten minutes getting three passes of focused analysis that a single human review might partially cover in twice the time.

Prompt Pattern One: Logic Errors

This is the one where AI earns its keep. Not style nits, not documentation suggestions — actual logic errors, the kind a tired human reviewer misses because the code looks right even when it isn't.

The failure mode is asking the model to "check for bugs." Too broad. The model will either produce a superficial scan or hallucinate issues that don't exist.

What works is forcing the model into an adversarial posture. You're not asking it to evaluate the code; you're asking it to try to break it.

The prompt:

``` Here is a function. Your job is not to improve it or suggest rewrites — your job is to find conditions under which it produces wrong output, crashes, or silently does something the caller didn't expect.

For each issue you find, describe:

- The exact input or state that triggers it
- What actually happens
- Why the author probably missed it

If you don't find any issues, say so plainly. Do not suggest stylistic improvements.

[paste code] ```

The "do not suggest stylistic improvements" clause matters. Without it, the model pads its response with style observations when the logic is actually clean. That pollutes the signal you're looking for. You want a short response with sharp findings, not a long response that buries findings under noise.

Here's a real example of what this surfaces. A Python function was batching API calls — chunking a list of IDs into groups of 50 and making one request per chunk. The test suite passed. The logic looked correct. What the adversarial prompt found: when the input list was empty, the function made one API call with an empty payload. The remote API interpreted that as "give me everything" — behavior documented two pages
```

from the method signature in the vendor's fine print. No human on the team caught it in review. The model caught it in four seconds because I asked it specifically to find inputs that produce wrong output. The fix was a two-line guard clause at the top of the function.

That's the goal. Not to replace your review — to catch what you, specifically, are likely to miss.

Scope-Limiting for Logic Review

The logic-error prompt works better when you constrain the surface area. Paste a 400-line file and the model spreads its attention thin, producing shallow observations across the whole thing. Paste a single function or a 30-line block and you get deeper analysis on that specific surface.

A practical heuristic: review logic in units no larger than a single responsibility. If you have a function that does three things, run three prompts. This sounds slower. It's actually faster, because you spend less time parsing verbose output and more time acting on specific findings.

One more useful addition to the prompt: "Here is the existing test for this function." When the model knows what's already tested, it focuses its adversarial analysis on the untested surface. Without that context, it may spend its response re-deriving edge cases your tests already cover.

Prompt Pattern Two: Security Smells

Security review is where overconfidence in AI output will hurt you most. The model can surface a class of common vulnerabilities quickly — SQL and NoSQL injection patterns, hardcoded credentials, unvalidated inputs, insufficient output escaping, missing rate limits. It is genuinely useful for those. What it cannot reliably do is reason about your system's specific threat model, your authorization architecture, or whether a particular code path is reachable by an adversary in your production environment.

The framing that works: ask the model to list security-relevant observations, not to certify the code as secure. Those are different asks with meaningfully different outputs.

The prompt:

``` Review the following code for security concerns. Focus specifically on:

- Input validation and sanitization
- Authentication and authorization assumptions
- Data exposure (logging, error messages, API responses)
- Common injection patterns relevant to this language and framework
- Any hardcoded secrets, tokens, or credentials

For each concern, classify it as:

- DEFINITE: this is a vulnerability as written
```

- POSSIBLE: this is worth investigating depending on context I can't see
- PATTERN: this matches a known risky pattern but may be fine here

Do not suggest performance improvements or style changes.

[paste code] ```

The three-tier classification is doing real work. Without it, the model mixes genuine vulnerabilities with theoretical concerns with stylistic observations, and you waste time triaging a flat list. With it, you get a prioritized queue and a built-in signal about where to focus first.

What this looks like in practice: on a recent Express.js route handler, this prompt surfaced one DEFINITE — a req.params value being interpolated directly into a MongoDB query string, the kind of NoSQL injection that's embarrassingly easy to introduce when you're moving fast — and two POSSIBLEs. The first POSSIBLE: an error handler that echoed user input back in the response body, which could expose stack traces depending on middleware config elsewhere in the application. The second: a missing rate limit check that mattered depending on whether the endpoint was public-facing. All three were real findings. Two required me to check context the model didn't have access to. One was a fix-it-right-now situation that would have been awkward to discover in a security audit.

The Ceiling on AI Security Review

Be explicit with yourself about where this stops working. AI review will not catch authorization logic errors that span multiple files. It will not detect race conditions in distributed systems. It will not reason correctly about the downstream consequences of your specific data model. It will not flag business logic vulnerabilities — the ones where the code is technically valid but does the wrong thing in context.

Those require a human with system knowledge. AI review catches the mechanical patterns fast and reliably. Don't let that speed create the impression it caught everything, because it demonstrably did not.

Prompt Pattern Three: Style and Consistency

This is the one that feels least important and is actually quite valuable on larger teams. Style review is the review nobody enjoys doing. It's tedious, contentious, and responsible for a meaningful fraction of PR comment threads that would have been better spent on something else.

AI handles this well because it doesn't have opinions. It has instructions. You give it your style guide, you give it the code, you ask for deviations.

The prompt:

``` We follow these conventions in this codebase: [paste relevant sections of your style guide or CONTRIBUTING.md]

Review the following code for deviations from these conventions. List each deviation with the line reference and the convention it violates. Do not suggest changes that are not covered by the conventions listed above.
```

Do not comment on logic or functionality.

[paste code] ```

The constraint "do not suggest changes not covered by the conventions listed above" is critical. Without it, the model applies general best-practice style recommendations from its training data, which may conflict with your house style. You want it enforcing your rules, not its default preferences. These are not the same thing.

One practical note: keep the pasted conventions short. If you paste 3,000 words of style documentation, the model's attention diffuses across all of it and misses obvious violations of the rules you actually care about. Pick the ten conventions that cause the most friction in your reviews. Write them in plain language. Use those.

A Worked PR Review: All Three Passes

Here's a realistic multi-pass workflow using Claude Code — roughly as I actually run it. The PR is a new API endpoint in a Node.js service: about 120 lines total, one new route handler, one service layer function, one update to existing middleware.

Pass one — logic errors (5 minutes):

Paste the route handler separately, then the service function separately. Use the adversarial logic prompt. The route handler comes back clean. The service function has a finding: it assumed userId would always be a string and called .toLowerCase() without a type guard. If the upstream caller ever passed a numeric ID — which it occasionally did in a legacy integration path, something only visible in the calling code — it would throw at runtime rather than failing gracefully or coercing correctly.

Pass two — security (3 minutes):

Paste both functions together with the three-tier classification prompt. One POSSIBLE: the response object returned the full user record from the database, including passwordHash, mfaSecret, and several internal flag fields that the frontend had no business receiving. Not a hard exploit, but it violated the principle of minimal data exposure and would have been a one-line fix to address. Nobody would have caught it in a normal review because nobody checks what a user object contains until something leaks.

Pass three — style (2 minutes):

Paste the full diff against the team's condensed style guide — eight rules, plainly written. Two deviations: inconsistent destructuring spacing in one line, one async function missing explicit try/catch wrapping that the conventions required.

Total time: ten minutes. The PR went back with three substantive comments and two style comments. All five addressed within the same afternoon. No back-and-forth, no clarification threads. The senior reviewer who would have done this manually would have caught maybe two of the five, depending on how the afternoon was going.

That's not a magic trick. That's a workflow.

The Critical Difference: Rubber-Stamp vs. Surfacing Real Problems

There's a version of AI code review that developers hate, and reasonably so. It's the version where the AI produces a wall of text, the developer skims it, decides none of it matters, merges the PR anyway. The AI contributed nothing except a false sense that someone looked at the code.

That version exists because the prompt was designed to get a response, not a useful response. "Review this code" invites a response. "Find conditions under which this crashes" invites a useful response. The difference is whether you've given the model a goal it can actually fail to meet.

This matters because of how human psychology interacts with confident AI output. Research out of Microsoft and Carnegie Mellon — specifically work led by Harmanpreet Kaur on what she termed "overreliance" in AI-assisted programming tasks — found that developers accepted AI-generated suggestions at substantially higher rates when those suggestions were presented with high confidence, regardless of whether the suggestions were actually correct. Confidence is free in language models. It costs a model nothing to sound certain. The rubber-stamp problem is partly a prompting problem, but it's also a calibration problem: you have to actively resist the gravity of a system that sounds sure of itself.

Structuring your prompts to produce skeptical output — explicitly asking for problems, explicitly asking for confidence classification, explicitly asking the model to flag when it lacks enough context to make a judgment — is how you counteract that gravity. The goal isn't to distrust AI output. The goal is to receive it with the same skepticism you'd apply to a first review from a junior developer who's technically sharp but doesn't know your system yet.

Tool Differences That Actually Matter

The three-pass workflow above is mostly tool-agnostic. Cursor's chat panel handles it fine — paste the code, run the prompts, read the output, follow up as needed. Claude Code handles it natively. GitHub Copilot's review feature, which graduated from beta in late 2024 and has been iterated through 2025, takes a different approach: it hooks into the PR review UI directly and generates inline comments automatically.

The inline approach is seductive. It also has a structural problem. When the model generates suggestions automatically without you specifying what to look for, you're back in rubber-stamp territory — it's surfacing whatever it decided was worth noting, not what you decided you needed. Copilot's built-in review is most useful for style and obvious mechanical issues, where automated pattern matching is genuinely fine. For logic and security passes, you still need to drive the prompt deliberately.

Cursor has an edge in iterative review because the chat interface makes it trivial to follow up. "Focus specifically on the error handling in the retry loop" takes five seconds. That back-and-forth is where review quality actually increases, because you're progressively narrowing toward the things you're uncertain about.

Treat Cursor review like a conversation with a technically competent colleague who needs very explicit direction, and you'll get significantly more out of it than treating it like a static analysis scan.

Claude Code's advantage is in longer-context reviews — if your PR spans multiple files with interdependencies, Claude Code's extended context window (up to 200k tokens as of 2025 models) means you can give it more of the relevant code without hitting the wall. Whether that context actually improves review quality depends entirely on whether you've given the model clear instructions about what to look for across that surface. More context with a vague prompt is worse than focused context with a sharp prompt. Every time.

On Not Outsourcing Your Judgment

There's a version of this workflow that can go wrong slowly: if you start treating AI review findings as authoritative without reading and understanding each one, you're no longer reviewing code — you're delegating code review to a system that makes confident mistakes.

Worth being direct about this. The purpose of this workflow is to catch things you might have missed, not to replace the part where you understand what the code does. If the logic-error prompt surfaces a finding and you don't understand it well enough to say "yes, that's a real issue" or "no, that's fine for this reason," the right response is to go learn enough to make that judgment. Not to file the comment and move on.

The AI is good at pattern recognition. You're good at understanding the system. The combination is where you actually catch things. Either one alone is worse than the combination, and the AI alone — without your verification — is an active liability in high-stakes code paths.

Willison has written about this specifically in security contexts: the danger isn't that AI will miss vulnerabilities, because it will, predictably, and you can account for that. The danger is that it finds something that looks like a vulnerability, you file it as a finding without verifying it, and you ship a fix that either doesn't address the actual issue or introduces a new one. Confident output requires confident verification.

A Short Pre-Review Checklist

Before running any of these prompts, a few preparation steps that consistently improve output quality:

Strip what the model doesn't need. Reviewing a function? Paste the function. Don't paste the entire file and hope the model finds the relevant section. Irrelevant context dilutes focus and inflates output you'll have to read through.

Include type information. Missing type context is the single most common cause of false positives in logic reviews. If the model doesn't know what type a parameter is, it flags things your type system already prevents. "TypeScript strict mode, userId is always a string" costs you one line and eliminates a category of noise.

Name your framework and version. "Express 4.18, Node 20" produces more accurate security observations than leaving the model to guess. Models have training cutoffs. Your framework has changed since then.

Tell it what's already tested. "There's an existing test for the happy path — I'm looking for untested edge cases." This focuses the analysis on uncovered surface rather than re-deriving what you already handle.

Set the scope of the review explicitly. "I am not looking for refactoring suggestions. I am not looking for performance improvements. I am specifically looking for [X]." This keeps the response tight and actionable.

None of these are complicated. They're the difference between a response you act on in five minutes and a response you spend five minutes deciding to ignore.

Code review is the professional obligation everyone nominally performs and nobody has fully solved. Human reviewers are valuable because they know the system; they're limited because they're tired, busy, and sometimes trying to get through the queue before standup. AI reviewers are available, consistent, and tireless; they're limited because they only see what you show them and have no idea how your service behaves at 2 a.m. on the third Tuesday of the month when the job queue backs up.

The workflow that works uses both. You provide system knowledge and final judgment. The AI provides pattern recognition and tireless scrutiny over mechanics. You drive the prompts. You evaluate the output. You decide what gets filed and what gets closed as won't-fix.

The rubber-stamp is what happens when you forget the second half of that sentence.

Don't forget it.

CHAPTER V

Prompts for Refactoring

Refactoring is the task where AI assistance is most likely to betray you. Not because the models are bad at it. Because they're too good at something adjacent to it — and that something is rewriting your code from scratch with different tradeoffs than you wanted, in a style you didn't ask for, while quietly removing behavior you were relying on.

You ask for one thing. You get three things, two of which you didn't want.

This chapter is about preventing that. About getting AI to do the unglamorous, genuinely useful work of improving existing code without torching the building in the process.

The Core Problem With AI and Refactoring

Tell any of the major models — Claude, GPT-4o, whatever's running under Cursor's hood today — to "refactor this function," and you'll probably get back something that compiles, looks cleaner, and has subtly different behavior. Sometimes the change is fine. Sometimes it's a bug that won't show up until 2 AM on a Tuesday.

The problem is structural. These models were trained on enormous amounts of code, including countless examples of code being replaced rather than incrementally improved. Rewrites outnumber refactors in training data. So when you say "clean this up," the model's prior is: produce a better version from scratch. It doesn't have a strong instinct for preservation. You have to install that instinct manually, in every prompt.

GitClear's 2024 analysis of AI-assisted code changes found a meaningful increase in what they called "churn" — code that gets added and then modified or reverted within two weeks. Their hypothesis was that developers were accepting AI output that looked correct but wasn't quite right, requiring follow-up fixes. Refactoring, where the expected behavior is already defined by existing code, is exactly the context where that kind of invisible divergence happens most.

Simon Willison, who has been writing about LLM behavior probably longer than most of us have been using these tools seriously, put it plainly on his blog: the model doesn't know what you care about. It only knows what you said. The delta between those two things is where the bugs live.

So before you open a prompt box, you need to be precise about what you want. Not "clean this up." Not "make this better." Specific. Constrained. Verifiable.

Scope-Limiting Before You Type a Single Word

The most important prompt technique for refactoring isn't a phrase or a template. It's a mental habit you develop before you write anything: define the boundary of the change.

Here's what that means in practice.

Suppose you have a 200-line Python module with a data-parsing function that does four distinct things — it validates input, transforms it, handles errors, and formats the output. All in one function. Classic legacy mess. You want to break it apart.

The naive prompt: "Refactor this module to separate concerns."

That prompt will get you a rewritten module. Maybe with a class. Maybe with async somewhere it wasn't before. Maybe with a logging pattern you don't use. The model will make decisions on your behalf that you didn't authorize.

The scoped prompt does this instead:

> *"I want to extract the validation logic from parserecord() into a separate function called validaterecord(). The new function should accept the same input parameter as parserecord() and return a boolean. Do not change any other behavior. Do not rename variables. Do not change parserecord()'s public signature. Show me only the new function and the modified first 10 lines of parse_record() where the call would be inserted."*

That prompt is longer. It's also doing five things: naming the target function, specifying the extraction, defining the new function's signature, explicitly forbidding scope creep, and limiting the output to the relevant section. Each of those constraints saves you a debugging session.

The three scope-limiting strategies that work most reliably are:

Function-level isolation. Never hand the model a file and ask it to improve the file. Hand it one function and ask it to improve that function. If you're using Claude Code, you can literally select the function in your editor and use the selection as context. Cursor's Cmd+K (or Ctrl+K on Linux/Windows) works the same way. Your job before the prompt is to draw that box.

Explicit preservation lists. Tell the model what it may not touch. "Do not modify the error handling. Do not change the return type. Do not add imports." Negative constraints are as important as positive instructions. Maybe more important.

Output format restrictions. Ask for a diff, or ask for only the modified function, not the whole file. If you get back 300 lines when you expected 40, that's a signal the model overstepped before you even read it.

The "Intent Preservation" Prompt Pattern

There's a specific failure mode that shows up in refactoring more than anywhere else: the model changes what the code means while keeping what it looks like intact.

Consider a function that increments a counter and returns the new value. A refactor might turn that into a function that returns the incremented value without persisting the change — same return value in most tests, different behavior in any stateful context. The model wasn't wrong about what the code does line-by-line. It was wrong about what the code is for.

The fix is to make intent explicit in the prompt.

Before you ask for the refactor, write one or two sentences describing what the function is supposed to do from a behavioral standpoint — not what the code says, but what the system relies on. Then include that in the prompt as a constraint.

Like this:

> *"This function is called in a request handler and its side effect — updating self.request_count — must be preserved after the refactor. The caller never checks the return value. Clean up the control flow but do not eliminate the side effect."*

Now the model has your intent, not just your code. That's a meaningful difference.

In Anthropic's documentation for Claude Code, there's guidance around using system prompts and context to establish constraints before generating code. The same principle applies here: the more you front-load what you care about, the less you have to fix afterward. It's not about writing longer prompts. It's about writing prompts that contain the right information.

Claude in particular tends to respond well to what you might call "behavioral contracts" — plain-English descriptions of what a function's invariants are. Something like: "This function must always return a list, even if empty. It must never raise. It must not mutate its input." Give it that and ask it to refactor; it'll use those constraints the way a careful human would.

Prompt Patterns for the Four Refactoring Scenarios You'll Actually Face

1. Extract and Rename

The most common refactoring task: something is doing too much, or something is named confusingly. You want to pull a piece out or rename it consistently across the codebase.

For extraction, the pattern is:

> *"Extract [specific behavior] from [function name] into a new function named [new name]. The new function should take [parameters] and return [type]. Call it from [original function] in place of the inline logic. Do not change any other function signatures or behavior."*

For renaming, especially with something like Cursor that has codebase-wide context:

> *"Rename [OldName] to [NewName] throughout this file. Show me every location where the name appears so I can verify the list before you make changes."*

That last sentence — "show me every location before you make changes" — is valuable. You want the model to surface scope before it acts, not after. Cursor's Composer mode handles multi-file changes, but you should still ask for the change list explicitly before accepting a diff that touches 14 files.

2. Simplify Control Flow

Nested conditionals. Guard clauses shoved into the middle of logic. A function that reads like a flowchart from 2009. This is where AI can genuinely save time, but it's also where intent drift is highest.

The pattern:

> *"Simplify the control flow in this function using early returns. Do not change the behavior for any input case. If you're uncertain whether a transformation is behavior-preserving, leave that section alone and comment it with 'REVIEW'."*

That last instruction — flag uncertainty rather than guess — is one of the most useful things you can add to any refactoring prompt. Models will sometimes smooth over an edge case rather than admit they're not sure. Telling them to flag it instead of resolve it gives you a review list rather than a hidden bug.

3. Break Apart a God Object or Module

This is the hardest refactoring to do with AI, because it requires understanding the system, not just the file. A 2,000-line class with 40 methods isn't a single-prompt job. It's a planning job first.

Do not start with "refactor this class." Start with:

> *"List the distinct responsibilities this class currently handles. Group the methods by responsibility. Do not generate any code yet."*

That's a recon prompt. You're using the model to build a map before you start moving things. Once you have the groupings, you can do one extraction at a time, each with its own scoped prompt, each verified before you move to the next.

Simon Willison has described this as "using the LLM as a thinking partner before using it as a code generator" — getting the analysis first and the synthesis second. It's good advice. The models are often better at describing what code does than at changing it without breaking things, so lean on that in the early stages.

4. Performance Refactoring

Rewriting something for speed is a context where AI is genuinely useful but can introduce correctness regressions disguised as improvements. An $O(n^2)$ algorithm replaced with an $O(n \log n)$ one is only better if it produces the same output.

The pattern here combines scope-limiting with an explicit benchmark request:

> *"Refactor this function to reduce its time complexity. Preserve all current outputs for the same inputs. After the refactored version, write three assertions that verify the output matches the original for the following cases: [specific cases]. Do not modify anything outside this function."*

Asking for assertions in the same prompt as the refactor is a technique worth building into your habits. You'll generate the verification at the same moment you generate the change, while the model has full context about what changed. Doing it separately, later, is slower and the second prompt loses context about what specifically changed.

Verifying That the Refactor Did What You Think

Here's the uncomfortable truth about AI-assisted refactoring: the model can't run your code. It can reason about it, and that reasoning is often good, but "often good" is not the same as "always correct." Verification is your responsibility.

The practical toolkit has three layers.

Layer one: the diff review. Before you accept any AI-generated change, read the diff the way a careful reviewer would — not to understand the intent (you already know the intent, you wrote the prompt), but to check that the only changes are the ones you asked for. You're looking for surprise. Anything that changed that you didn't explicitly ask to change is a question. Maybe it's fine. Maybe it's not. But you need to notice it.

Cursor shows diffs inline. Claude Code shows them in the terminal. Both are readable. The discipline is actually reading them rather than accepting the summary.

Layer two: behavioral tests. If the function you're refactoring has existing tests, run them immediately. If it doesn't — and in a legacy codebase, it often doesn't — write at least three before you accept the refactor. Characterization tests. Input !’ output pairs that document what the function whether "what it currently does" is what it should do. The refactor shouldn't change those outputs.

Here's a prompt for generating those quickly:

> *"Write five pytest characterization tests for this function that cover its current behavior, including edge cases and error cases. Don't test correctness — test that the behavior is what it currently is. Use the function as-is for the expected values."*

That last instruction matters. "Use the function as-is for expected values" means the model should call the original function to produce the expected output, or reason about what it would return, not invent expected values from scratch. You want tests that will catch divergence from current behavior, not tests that express what you wish the function did.

Layer three: the adversarial second pass. Take the refactored code and ask the model to find problems with it — not from the perspective of style, but from the perspective of behavioral correctness.

> *"Here is the original function and here is the refactored version. List any cases where the refactored version might behave differently from the original. Be specific. If you're not sure, say so."*

This is one of the more reliable uses of a second AI pass. Not to generate more code, but to critique existing code from a specific angle. Claude tends to do this better when you frame it as adversarial — "find problems" rather than "review this." The latter gets you a polished summary. The former gets you edge cases.

The Incremental Commit Pattern

One habit that prevents a lot of pain: treat each AI-assisted refactoring step as its own commit.

Not a commit at the end of the refactoring session. A commit per logical cha
commit. Rename the variable!' commit. Simplify the conditional!' commit.

This sounds tedious. It isn't. It takes thirty seconds and it buys you the ability to bisect a regression down to a single operation rather than hunting through a 400-line diff that mixes three different kinds of changes. You can also describe each commit message from the prompt you used, which is a form of documentation that's actually useful later.

When you're using Claude Code in particular, the agentic mode will sometimes attempt multi-step refactors in a single pass. Let it. But before you commit the output, break it into logical stages manually and verify each one. The model's convenience doesn't override your review hygiene.

Cursor's Notepads feature (introduced in late 2024) lets you maintain a persistent note with your project's coding conventions and refactoring rules that gets injected into prompts. If you're doing a long refactoring session on a specific module, write your constraints into a Notepad once and reference it in each subsequent prompt rather than re-typing the same restrictions. It's a small thing. It adds up over a day of work.

When the AI Pushes Back — And When It's Right

Occasionally the model will tell you that the refactor you're asking for is going to be difficult, or that it sees something in the code that complicates the change. Pay attention to this.

It's tempting to treat AI pushback as friction to overcome — just re-prompt more firmly and it'll comply. Often that works. Sometimes working means "the model did what you asked and introduced a subtle bug because you steamrolled a legitimate concern."

If the model says something like "this function has a side effect in the error path that might be affected by the proposed change," treat that as a flag. Investigate before you insist. The model isn't always right. But it's sometimes pointing at something real that you glossed over when you wrote the prompt.

The opposite is also true. Models will occasionally refuse a refactoring step on grounds that seem overly cautious — worried about breaking something in a way that your actual test suite would immediately catch. In those cases, the answer is to add the test coverage to the prompt:

> *"Here are the tests that currently cover this function. Given that these tests exist and will be run after the change, is the refactoring still risky?"*

That reframes the conversation. You're not asking the model to be brave. You're giving it evidence that the risk is managed.

What "Done" Looks Like

Refactoring is one of those tasks that can expand indefinitely if you're not careful. The code can always be a little cleaner, a little more idiomatic, a little more testable. AI makes it easier to keep going past the point where you should stop, because generating the next improvement costs almost nothing.

Set a definition of done before you start the session. Write it down. Something like: "This session ends when parse_record() has been split into three functions, each under 30 lines, and all existing tests pass." When you hit that, stop. Commit. Close the session.

The value of incrementalism isn't just safety — it's also forcing yourself to make a decision about what the goal actually is before the tools make it too easy to just keep going. Refactoring sessions that don't have an exit condition don't end; they get abandoned when you run out of time or patience, leaving the codebase in a half-transformed state that's worse than either the original or the target.

AI is very good at the motion of refactoring. The forward momentum, the constant generation of "one more improvement." You have to supply the stopping condition. The tools won't.

The through-line of all of this is the same as everything else in AI-assisted development: the model is a capable collaborator and a terrible decision-maker. When you're refactoring, the decisions are: what to change, how much to change at once, what must be preserved, and when to stop. None of those belong to the model. All of them belong to you.

Give the model specific work within a defined boundary. Verify the output against concrete behavioral expectations. Commit incrementally. And when the refactored code passes its tests and does exactly what the original did — except clearly and maintainably — that's the thing you were after.

Not cleaner-looking code that you hope still works. Code you know still works, and can prove it.

That's the difference between AI-assisted refactoring and just letting a model touch your codebase and seeing what happens. One of those is a workflow. The other is a prayer.

CHAPTER VI

Prompts for New Features

The ticket lands in your queue at 9 AM. "Add CSV export to the reporting dashboard." Simple enough on the surface — a button, a file, done. By 10 AM you've got 400 lines of AI-generated TypeScript that exports something, sort of, except it doesn't handle null dates, ignores the user's active filters, and the column order is alphabetical for no reason. You spend the rest of the day fixing it.

That is the canonical way to misuse AI for new feature work. Not because the tool failed you — it gave you exactly what you asked for. You just asked for the wrong thing, at the wrong moment, with the wrong context. This chapter is about fixing that.

The Decomposition Step You're Skipping

Here is the single most common mistake developers make when using AI to build a new feature: they paste the ticket into the prompt and hit Enter.

Tickets are not specs. A Jira ticket that says "Add CSV export to reporting dashboard" is a label on a box, not a blueprint for what's inside. Before you type a single word to Claude Code or Cursor, you need to do the thinking that turns that label into a description of actual work. This isn't extra ceremony. It's the same thinking you'd do before writing the code yourself — except now you're also doing it before writing the prompt.

The decomposition step has a few distinct parts.

What does this feature actually touch? Not in the abstract, but in your specific codebase. Which models, which services, which existing API endpoints. If your reporting dashboard has a /api/reports/{id} endpoint that returns JSON, the CSV export probably needs to either call that and transform the output, or go directly to the database with the same query. Those are two different implementations. The AI will pick one, probably the wrong one, unless you tell it which.

What are the edge cases you already know about? Before you prompt, list them. Null values. Empty result sets. Reports with 50,000 rows. Special characters in string fields. Date formatting across timezones. You know your data. The AI does not. If you don't tell it that your user_notes field can contain commas and line breaks — fields that will obliterate a naïve CSV format — it will give you a naïve CSV format.

What constraints exist that aren't in the ticket? Performance requirements. Authentication. The fact that this is a multi-tenant system and the export must be scoped to the requesting user's organization. Accessibility requirements if there's a UI component. The existing code style. Whether you're already using a CSV library or need to add a dependency.

Write these down. Seriously, just write them down — even if it's a quick bullet list in a scratch file. Simon Willison has made this point repeatedly in his writing about LLM-assisted development: the value of the model is roughly proportional to the quality of the specification you give it. He's been building with LLMs long enough to have learned this the hard way, and so have I.

This decomposition usually takes fifteen to twenty minutes for a moderately complex feature. That time pays back immediately. The first output from the AI is dramatically more useful when it's responding to a real spec instead of a wishful ticket summary.

Context: The Goldilocks Problem

Once you know what you're building, you face the context question. How much of your codebase should you show the AI?

The instinct is to show everything. More context, better answers, right? In practice, no. Flooding the context window with tangentially related files creates two problems. First, the model's attention gets diluted — this is a real effect, not a metaphor. Anthropic's documentation on Claude's context handling is explicit about the tradeoffs at very long contexts, and Cursor's codebase indexing exists precisely because unguided retrieval over a large repo produces worse results than targeted retrieval. Second, you make it harder to iterate, because every clarification and follow-up now has to wade through the same ocean of files.

The right approach is surgical. Include:

- The files the feature will directly modify
- The files that define types or interfaces the feature will use
- One working example of a similar feature, if one exists

That last point is underrated. If your app already has an Excel export somewhere, or a PDF download, the pattern for how your team handles file-generation endpoints is right there. Show it to the model. Not as a constraint — the AI shouldn't just copy it — but as context for how your codebase handles this class of problem.

Leave out:

- General utility files the feature won't touch
- Tests for unrelated components
- The entire node_modules directory (this is a joke, but I have seen it happen)
- README files and documentation unless the feature has unusual setup requirements
- Configuration files that don't affect this feature's behavior

For a mid-size feature in a TypeScript/React project, you're typically looking at three to eight files of context. That's it. If you find yourself thinking "I should include the entire src/ directory so it understands the project," that's a sign your decomposition step wasn't specific enough. Go back and sharpen what you're actually building.

Claude Code handles context differently from Cursor. In Claude Code, you're explicitly passing files or content. In Cursor, the @ mention system lets you pull in specific files, symbols, or even docs. Copilot in the editor works from what's open and what's adjacent. All three reward specificity. None of them are helped by vagueness.

Writing the Prompt

With your spec and context ready, the prompt itself becomes almost mechanical. You are describing a concrete transformation: take this codebase state, add this behavior, produce these outputs.

A feature prompt that works has four components:

1. What exists. "There is an existing /api/reports/{id} endpoint in src/api/reports.ts that returns report data as JSON. The Report type is defined in src/types/report.ts."

2. What you're building. "Add a /api/reports/{id}/export/csv endpoint that returns the same data as a downloadable CSV file."

3. The constraints. "Must use the existing auth middleware in src/middleware/auth.ts. Must respect the user's active filters (passed as query params). Must handle null values by outputting empty strings. Max expected row count is 100,000 — don't buffer everything in memory."

4. What you want back. "Write the new endpoint handler and any helper functions needed. Use the papaparse library which is already in the project. Don't modify the existing JSON endpoint."

That's the whole prompt. No philosophical preamble. No "You are a senior TypeScript developer with ten years of experience" — that framing does essentially nothing for code generation tasks, and you look a little silly typing it. Just tell the model what exists, what you want, and what the rules are.

One thing to add if you're using Claude Code: ask it to explain its approach before it writes the code. Not a long explanation — a paragraph is enough. "Before writing the code, describe your approach in two to three sentences." This serves two purposes. It exposes wrong assumptions before they get embedded in a hundred lines of generated code, and it forces the model to make its architectural choices explicit so you can redirect if needed. It's essentially asking the AI to show its work.

The First Output Is a Draft

Here's the expectation calibration that matters most: the first thing the AI writes is a draft. Not a prototype. Not a first cut. A draft, like the first draft of a document — structurally plausible, probably containing real errors, almost certainly needing revision.

This isn't a limitation. This is how good software gets written, with or without AI. The difference is that AI drafts are fast enough that you can afford to treat them as genuinely disposable starting points, iterate twice, and still be faster than you would have been writing from scratch.

What you're looking for in the first output:

- Does the overall structure make sense? Is the approach reasonable for the problem?
- Are there obvious logical errors you can spot without running it?
- What assumptions did the AI make that you didn't specify? There will always be some.

What you're not doing with the first output:

- Shipping it
- Convincing yourself it's fine because it looks plausible
- Skipping your normal review process because it came from a "smart" tool

The second prompt is almost always more valuable than the first. Once you can see the shape of the generated code, you can give much more targeted feedback. "The memory issue: you're loading all rows into an array before piping to the response. Use a streaming approach with Transform instead." That's a precise, testable correction. You could not have written that prompt cold, before seeing the first draft.

A Realistic Walkthrough

Let's make this concrete. The feature: add CSV export to a reporting dashboard, exactly the scenario from the opening. Here's how this actually goes from ticket to merged PR.

Step 1: Decomposition (20 minutes)

The ticket says "Add CSV export to reporting dashboard." You pull up the codebase and figure out:

- Reports are served from src/api/reports.ts. The GET /api/reports/:id endpoint calls ReportService.getById(id), applies user-scoped access control, and returns a ReportData object.
- ReportData has seventeen fields. Three of them (completedAt, startedAt, reviewedAt) are nullable Date objects. Two (notes, description) are free-text strings that may contain commas, quotes, and newlines.
- The dashboard currently uses recharts for visualization. There's no existing file-export code anywhere in the project.
- papaparse is already a dependency (it's used in a CSV importer elsewhere in the app).
- Authentication is handled by a middleware that attaches req.user to the request. The ReportService takes userId as a parameter and enforces org-level scoping internally.
- The PM's Figma mockup shows a "Download CSV" button in the report header. It should trigger an immediate download, not a link to a URL.

You now have a real spec. The ticket was six words. Your spec is a paragraph.

Step 2: Context selection

You open Claude Code and decide to include:

- src/api/reports.ts (the existing endpoint)
- src/services/reportService.ts (the service layer)
- src/types/report.ts (the ReportData type definition)
- src/middleware/auth.ts (twelve lines, shows the pattern)

You leave out the frontend components — you'll handle the button separately — and the test files, which you'll write after seeing what gets generated.

Step 3: The first prompt

``` I'm adding a CSV export endpoint to an existing reporting API.

Existing context (files attached):

- src/api/reports.ts — existing GET /api/reports/:id endpoint
- src/services/reportService.ts — ReportService with getById(userId, reportId)
- src/types/report.ts — ReportData type (17 fields, 3 nullable dates, 2 free-text)
- src/middleware/auth.ts — auth middleware pattern

Task: Add a new endpoint GET /api/reports/:id/export that:

- Uses the same auth middleware as the existing endpoint
- Calls ReportService.getById with the same scoping
- Returns a CSV file download (Content-Disposition: attachment)
- Uses papaparse (already installed) for CSV generation
- Handles null dates as empty strings
- Properly escapes free-text fields (notes, description)
- Streams the response rather than buffering all rows in memory

Before writing the code, describe your approach in 2-3 sentences.

Don't modify the existing GET /api/reports/:id endpoint. ```

## Step 4: Review the approach explanation

The model comes back with something like: "I'll add the export endpoint to reports.ts following the same middleware pattern. For the ReportData object (single report), streaming isn't necessary since it's one record — I'll use papaparse.unparse() to generate the CSV string and send it. I'll map nullable dates using an ?.toISOString() ?? '' pattern and rely on papaparse's default quoting for the free-text fields."

Stop. Read this. Notice that the model is treating this as a single-record export. But your reporting dashboard's "report" is actually a collection of events — your ReportData has a rows: EventRow[] field with potentially thousands of items. The model read the type wrong, or you didn't make the shape explicit enough.
```

This is why you ask for the approach first. Correcting a two-sentence misunderstanding costs thirty seconds. Correcting it after 150 lines of generated code costs twenty minutes.

You follow up: "The ReportData object contains a rows: EventRow[] field with potentially 100,000 items. The export is these rows, not the top-level report fields. Please stream the response using a Transform stream or equivalent to avoid loading all rows into memory at once."

Now the model revises its approach and generates the code.

Step 5: Review the generated code

What comes back is mostly right. There's a streaming Transform setup, correct Content-Disposition header, papaparse handling. But you notice two things:

First, the generated code doesn't include the Content-Type: text/csv header. Minor, but some browsers will behave badly without it.

Second, the date handling has a subtle bug: row.completedAt?.toISOString() will throw if completedAt is a string in some of your older data records instead of a Date object. You know this because you've been in the codebase for eighteen months. The AI doesn't know this.

You make the first fix yourself — it's one line. For the second, you prompt: "The date fields may be strings or Date objects in existing data. Update the date formatting to handle both: if it's already a string, use it directly; if it's a Date, call toISOString()."

The model gives you the updated helper function. You eyeball it, it's right, you copy it in.

Step 6: The frontend button

Now you turn to the "Download CSV" button. Separate prompt, separate context. You include the relevant React component file and the existing API client utilities. Short prompt: "Add a Download CSV button to the report header that hits GET /api/reports/:id/export and triggers a browser download. Use the existing apiClient from src/utils/apiClient.ts."

First output: fine, but it uses window.open() which won't work correctly for authenticated endpoints. You know this. The model doesn't know your auth setup without being told.

Second prompt: "The API requires an auth token in headers. window.open() won't work. Instead, fetch the endpoint with the auth token, convert the response to a Blob, and use URL.createObjectURL() to trigger the download."

Second output: correct. Standard pattern. You've seen it a hundred times; the AI generated it accurately when pointed in the right direction.

Step 7: Tests

You're not going to prompt for tests blindly. You'll write the tests yourself using the generated code as your reference, because tests for AI-generated code are where you actually verify it does what you think it does. (Chapter 7 goes deeper on this — the short version is that AI-generated tests for AI-generated code often test the AI's assumptions back at themselves, which is circular and useless.)

For this feature, you write three tests manually: one for a normal export, one for a report with null dates, one for a report where the notes field contains commas and double-quotes. The second test fails. Turns out the null date handling works, but the streaming code has an off-by-one error in the header row. Fifteen minutes to find and fix. This is a bug the AI introduced and a test you caught it with. The process worked.

Step 8: PR

You open the PR. The diff is 187 lines of new code, zero modifications to existing files, and three test cases. The description explains what the endpoint does, what papaparse handles, and why you're using streaming. You wrote that description yourself in four minutes because you understand every line of the code — you reviewed it, corrected it, and tested it.

A teammate reviews it. They ask one question about the streaming buffer size. You have an answer because you read the code. The PR merges same day.

That's the workflow. Not magic, not embarrassing. Just faster than writing all of it from scratch.

What the Iteration Looks Like in Practice

The CSV example involved four prompts total: initial approach, correction on data shape, date type fix, and the frontend auth correction. Four prompts over maybe forty-five minutes of actual AI interaction time, spread across a few hours of review and testing.

That ratio — several targeted prompts, each sharpening the previous output — is typical for a feature of this size. You are not going to get a five-prompt sequence that produces merged code for anything nontrivial. And you shouldn't want to. A feature built in five prompts with no iteration probably wasn't complex enough to need AI in the first place, or it's shipping risk you haven't examined.

The prompts that matter most are the corrective ones. The initial prompt structures the problem. The follow-ups expose the errors. Most of the value in AI-assisted feature work is in that second and third turn, where you're redirecting based on what you can see rather than speculating about what you'll get.

This is also where Cursor's inline editing has a specific advantage for iterative work — you can see the diff immediately and accept or reject individual chunks, which makes the back-and-forth faster than copy-pasting between a chat interface and an editor. For longer sequences of iteration on a complex feature, that UX difference is real and it adds up.

A Few Patterns That Consistently Help

Separate UI from API. Don't ask for "the full feature including backend and frontend" in one prompt. You'll get a coherent-looking mess that couples things incorrectly. Backend first, review it, then frontend as a second task with the finalized API contract as context.

Tell it what not to do. "Don't modify the existing endpoint" and "don't add new dependencies" and "don't change the type definitions" are all legitimate constraints that will save you from surprises. Models have a mild tendency to helpfully refactor adjacent code. You don't want that.

Ask for the minimal change first. "What's the smallest change that achieves this feature?" is a genuinely useful prompt for scoping. You can always ask for more. Unwinding an over-engineered first draft is tedious.

Verify library usage. If the generated code uses a library function you're not familiar with, look it up. AI assistants will confidently call methods that don't exist, especially for libraries that have changed their API since the model's training data. papaparse.unparse() is real. Some of the other things models confidently reach for are not.

The goal of all of this is not to automate feature development. It's to spend your cognitive budget on the parts that actually require your judgment — the architecture decisions, the edge case identification, the review, the testing — and let the model handle the mechanical translation of a clear spec into code structure. When that division of labor is right, you ship faster and you understand what you're shipping. When it's wrong, you spend a Friday afternoon debugging code you don't really understand because you trusted the first draft.

The first draft is never the thing you ship. The discipline is in the iteration.

CHAPTER VII

Prompts for Tests

Tests are where AI-assisted development either earns its keep or betrays you quietly. The betrayal is particularly insidious because it looks like productivity. You ask for test coverage, you get test coverage. The CI pipeline goes green. You feel like you did the right engineering thing. Then, three months later, a bug slips through that the tests were supposedly covering, and you spend an afternoon realizing that your test suite was never actually testing your code — it was testing the AI's model of your code, which is a subtly different thing.

This chapter is about not letting that happen.

What You're Actually Asking For

Before you write a single prompt, it helps to be precise about what kind of test you want. "Write tests for this function" is how you get a fast green suite that proves nothing. The model will write tests that exercise the happy path, match the assumptions embedded in the implementation, and technically achieve 100% line coverage of the function as written. Congratulations, you have metrics.

The problem is that tests aren't supposed to validate an implementation. They're supposed to validate intent. And the AI wrote the implementation. So when you ask it to test that implementation, there's a structural circularity problem baked in from the start. The model's internal representation of what the function does is the same representation it used to write the function. It will test against that representation. It will not test against what the function was supposed to do according to the ticket, the spec, the user story, or the conversation you had with your product manager last Tuesday.

You have to supply that gap. That's your job. The AI is very fast at the mechanical parts of test writing — scaffolding, boilerplate, parameterization, assertion syntax. Let it be fast at those things. Your job is to make sure you're telling it what to actually test, not just asking it to test whatever it thinks is being done.

Unit Tests: Give It the Spec, Not Just the Code

Here's the prompt pattern that consistently produces useful unit tests instead of decorative ones. Instead of:

> *"Write unit tests for this function."*

Try:

> *"Here is a function. Here is what it is supposed to do according to this spec: [paste spec]. Write unit tests that verify the function meets the spec. Do not assume the implementation is correct — test the intended behavior."*

The second version gives the model an anchor that isn't its own output. It has to compare implementation against specification, which is what a test should do. When you don't give it a spec, it will infer one from the code, and the inferred spec will match the code perfectly. Of course it will.

This matters more with non-trivial functions. If you have a calculateShippingCost(weight, zone, discount) function and you ask for tests without context, you'll probably get tests that confirm the existing conditionals. What you want are tests that confirm the business rules: that a 10kg package to Zone 3 with a 20% discount costs exactly what finance said it should cost in Q3. That test requires a number. A real number from a real source. The AI won't invent that number correctly unless you give it to it.

So the concrete habit is: when you start a test-writing session, open the relevant spec, ticket, or requirements document alongside the code. Feed both to the model. Force the comparison. Simon Willison made this point in a 2024 post on his blog — that the value of AI in testing isn't in automating the obvious cases, it's in forcing you to articulate what "correct" actually means before you write the assertion. The discipline of writing the prompt well is the discipline of knowing what you're testing.

Edge Cases: This Is Where AI Actually Shines

Unit tests for the happy path are the boring part. Any developer can write assert calculateShippingCost(5, 1, 0) == 12.50 given ten minutes. The expensive part of test writing has always been enumerating the edges: zero values, negative inputs, null, empty strings, max integers, concurrent access, malformed JSON, the customer who somehow has a negative loyalty points balance.

This is where AI assistance is genuinely valuable. Not because it knows your domain better than you do — it doesn't — but because it's faster at systematic enumeration. You can prompt it specifically for this:

> *"Given this function signature and spec, generate a list of edge cases I should test. For each one, explain why it might break the function and what the expected behavior should be."*

Note that you're asking for a list of cases and the reasoning, not the test code. Not yet. You want to review the edge case list first, because the AI will include some cases that don't apply to your context and will miss some that do. Take two minutes to read through the list. Remove the irrelevant ones. Add the ones it missed — and there will be some, because there are always cases only you know about from domain experience. Then ask it to generate tests for the approved list.

This two-step approach catches something important. When you get a generated edge case list, you'll occasionally see the model propose a case that reveals a misunderstanding of the spec. It might suggest testing what happens when discount exceeds 100%, and your immediate reaction is "wait, we validate that upstream, there's no way that reaches this function." Correct — and now you've just confirmed that your upstream validation is handling it. Good catch. Or your reaction is "oh, we actually don't validate that upstream, do we." Also a good catch. Either way, the edge case enumeration step produces value beyond just test boilerplate.

Anthropic's Claude Code is particularly effective here because you can keep the spec document in context alongside the implementation and run the edge case generation iteratively. If you ask for ten edge cases, review them, and then say "you missed the timezone handling edge case — add tests for date calculations that cross DST boundaries," it will update the list intelligently without requiring you to re-paste everything. Cursor's context management works similarly well for this workflow, since it can see the file tree and pull in related modules automatically.

Integration Stubs: Setting Boundaries Carefully

Integration tests are a different beast. The failure mode here isn't circularity — it's scope explosion. Ask an AI to "write integration tests for the user authentication flow" and it will cheerfully try to write tests that touch your database, your email service, your session store, and possibly your Stripe integration, depending on what it can see in the codebase. The resulting test suite will either require an enormous amount of infrastructure to run locally, or it will have so many mocks that it's essentially a unit test wearing integration test clothing.

The prompting discipline for integration tests is explicit boundary declaration:

> *"Write integration tests for the authentication flow. The boundary of this test is the HTTP layer — mock the database with [specific mock pattern], mock the email provider with [specific interface], and treat the session store as an in-memory implementation. Test that the HTTP contract between client and server is correct."*

You're doing the architectural thinking first, then handing the model a constrained problem. This is almost always the right pattern: you think about what the test should and shouldn't touch, you declare that explicitly in the prompt, and the model handles the mechanical expression of that decision.

Where developers get into trouble is treating the AI as an architect. It isn't. It will make reasonable-seeming decisions about test boundaries — often based on what it has seen in similar open-source codebases — but those decisions may not match your team's actual infrastructure, your deployment environment, or the way your CI pipeline is configured. A test that works perfectly in the model's assumed environment might fail immediately in yours because you use a non-standard Redis client or your database migrations run differently in test mode.

The practical rule: you own the architecture of the test, the AI writes the code of the test.

The Three Failure Modes You'll Actually Encounter

Circular Tests

This is the failure mode the chapter summary names, and it deserves specificity. A circular test is one that passes precisely because the implementation is what it is, not because the implementation is correct. Here's a simplified example.

Suppose the implementation has a bug:

``python def applydiscount(price, discountpct): return price - (price * discountpct)  # bug: discountpct should be /100 ``

If you ask the AI to write tests for this function without giving it the spec, it will likely write:

``python def testapplydiscount(): assert apply_discount(100, 0.2) == 80.0  # passes ``

The test passes. The bug is still there. The function expects a fraction, not a percentage, and the test was written to match that expectation — because the AI inferred the expectation from the implementation. If your spec said "apply a 20% discount to a price of $100," the correct test input would be apply_discount(100, 20) and the expected output would be 80.0. That test would fail immediately and surface the bug.

Spot circular tests by checking: does this assertion value come from me, from a spec, or from running the function and copying the output? If it came from running the function, it's circular. The assertion was generated by the same code it's asserting against.

Mocks That Test the Mock

Integration and unit tests that rely heavily on mocking have a second failure mode: the mock becomes so detailed that the test is actually testing that the mock behaves as expected, not that the real implementation does. You see this when AI-generated tests create elaborate mock setups that precisely mirror the production dependency's API. When the production dependency changes behavior, the mock keeps returning the old values, the tests keep passing, and you find out at runtime.

Prompt against this explicitly:

> *"For any mocks in these tests, use the simplest possible mock that satisfies the interface. Do not recreate the full behavior of the dependency — just enough to let the unit under test execute."*

Tests That Don't Actually Assert Anything

This sounds too obvious to mention. It isn't. Language models will sometimes generate tests that call a function, perform some operations, and then... assert that no exception was raised. That's it. No assertion about output. Just:

``python def testprocessorder(): result = processorder(sampleorder) assert result is not None ``

That test tells you the function doesn't crash on a sample input. It tells you nothing about whether it processed the order correctly. This pattern shows up more often in generated tests for functions that return complex objects, where the model isn't sure what to assert about the return value. The fix is to tell it:

> *"Every test must include at least one assertion about the specific content or state change produced by the function under test, not just that it ran without raising an exception."*

How to Prompt for a Complete Test Suite

The mistake most developers make is trying to generate an entire test suite in one shot. You get a massive file, you skim it, it looks reasonable, you commit it, and then weeks later you start finding the hollow tests. One shot is fine for boilerplate scaffolding, but it produces shallow coverage.

A better structure is three passes.

Pass one: spec alignment. Feed the model the spec or acceptance criteria and ask it to generate the test cases as text descriptions — no code. Just sentences like "Test that a user with an expired subscription cannot access premium endpoints" and "Test that the rate limiter resets after 60 seconds." Review this list. This is your test plan. Add cases the model missed. Remove cases that are duplicates or out of scope. This list should match your understanding of what the feature is supposed to do.

Pass two: implementation. Now feed the model the approved test case list, the implementation, and any relevant fixtures or helpers, and ask it to write the actual tests. You've already decided what to test; now it's filling in the code.

Pass three: edge case sweep. After the main tests are written, run one more pass specifically asking for edge cases and negative tests — inputs that should fail, boundary conditions, malformed inputs. The model is good at enumerating these systematically. Review the list as before, then have it generate code for the approved cases.

This three-pass approach takes longer than "write tests for this file" but produces a meaningfully better outcome. The first pass, which usually takes five or ten minutes, is doing the work that makes the difference between a test suite and a test collection.

Reading Generated Tests Before You Run Them

There's a tempting habit that AI tooling quietly encourages: run first, read second. The feedback loop is fast — you generate tests, you run them, they pass or fail, you adjust. Reading the test code feels slower and less informative than just running it.

Resist this. Generated tests should be read before they're run, at least until you've developed a strong sense of where your particular model and prompt patterns tend to fail. Reading a test takes thirty seconds. You're looking for three things:

One: does the assertion value make sense? If a function that calculates a percentage suddenly has assert result == 2.33333333, ask yourself where that number came from. The model probably ran the implementation mentally and hardcoded the output. Check whether it's the right output.

Two: is the setup realistic? If a test creates a user with admin=True and verified=False and subscription_active=True, does that combination actually represent a valid system state? Or is it a combination the model invented that would never occur in production?

Three: are the mocks minimal? A mock that is more complex than the thing it's mocking is usually a sign the test has scope problems.

This review is faster if you ask the model to annotate its own tests during generation:

> *"For each test, add a one-line comment explaining what invariant or behavior it is verifying."*

Those comments make the review pass dramatically quicker. They also occasionally reveal tests where the comment says "verifies that the function returns correctly" — which means even the model couldn't articulate what it was verifying. Delete those tests. They're noise.

A Worked Example

Say you're adding a feature: a function mergeuserprofiles(primaryid, secondaryid) that combines two user accounts. The secondary account's activity history attaches to the primary account, the secondary account gets deactivated, and if both accounts have email addresses, the primary email wins.

Before you touch a prompt, you write down the invariants:

- Primary account's email is unchanged after merge
- Secondary account's status becomes deactivated after merge
- All activity records previously belonging to secondary now have userid == primaryid
- If primary has no activity and secondary has 5 records, primary has 5 records after merge
- Merging an account with itself raises an error
- Merging a non-existent account raises an error
- Merge is not reversible — no undo mechanism

Now you open Claude Code or Cursor and paste:

> *"Here is a function: [paste code]. Here are the invariants it must satisfy: [paste your list]. Write unit tests that verify each invariant. Treat the database as an in-memory dict keyed by user_id. Use the simplest possible fixture setup. Every test should have a comment identifying which invariant it covers."*

What comes back is vastly more useful than what you'd get from "write tests for this function." You'll get tests that actually cover the business logic. You'll probably need to tweak a few assertions and adjust the fixture setup to match your actual test infrastructure. But the skeleton is solid because it was anchored to something real.

Also notice the "merging an account with itself" case. That's a reasonable edge case that the model might have caught on its own. It might not have. You know your codebase; you know this is the kind of thing that slips through. Writing it down explicitly costs you twenty seconds.

On Coverage Numbers

One thing worth saying plainly: 100% test coverage from AI-generated tests is meaningless. The number is easy to hit and means almost nothing about test quality. A test suite that hits 100% line coverage by running each line once and asserting nothing substantial is worse than a test suite with 70% coverage that actually verifies behavior, because the 100% suite creates false confidence.

When you're reviewing AI-generated test output, don't look at the coverage report first. Look at the assertions. Count the assertions that check specific values against specific expected outcomes, versus assertions that just check types, not-null, or successful execution. The ratio tells you more than the coverage number.

This is not a new insight — experienced testers have said it for decades. AI tooling just makes it possible to hit 100% coverage without thinking, so it's worth naming explicitly. The metric was always gameable; now it's effortlessly gameable.

The Useful Mental Model

Think of the AI as a test scaffolding contractor who is fast, thorough on the obvious stuff, and knows exactly zero about your business requirements. You would not hire a general contractor to make architectural decisions about your building. You would hire them to execute decisions you've already made. Same principle.

Your role in AI-assisted test writing is:

- Define what correct behavior looks like (the spec)
- Approve or reject the test cases before they're written as code
- Read the generated tests for circular assertions and hollow mocks
- Add the domain-specific cases the model couldn't know to include

The AI's role is:

- Generate the test scaffolding and boilerplate fast
- Enumerate edge cases you can review and filter
- Write parameterized test suites without the tedium
- Keep assertion syntax correct for whatever testing framework you're using

When the division of labor is clear, AI-assisted test writing is genuinely faster than writing tests by hand — without producing a suite that deceives you. The speed comes from the AI handling the mechanical expression. The value comes from you handling the judgment.

That split won't feel natural immediately. The friction is in the review pass, which feels slower than just running the tests and seeing green. Sit with the friction. A green suite built on circular assertions is a liability, not an asset. You want tests that can catch bugs, and bugs come from places where implementation diverged from intent. Only you know where those places are likely to be.

The AI doesn't know what the function was supposed to do. You do. Tell it.

CHAPTER VIII

Prompts for Debugging

The gap between something is broken and I understand why is where most developer hours actually go. Not writing new code. Not planning features. Sitting in a broken environment, reading logs backward, adding console.log statements like someone trying to echolocate in a dark room.

A 2018 Stripe report, conducted with Harris Poll across 850 developers, found that developers spend roughly 17.3 hours per week — nearly half a working week — on maintenance, debugging, and dealing with technical debt. You probably didn't need a survey to confirm that. It felt true the moment you read it.

AI changes this equation, but not in the way the demos suggest. The demo version shows you paste a broken function, the AI spots the off-by-one error in seconds, and everyone applauds. That happens sometimes. The more common reality is murkier: your bug isn't in one function, the error message is cryptic, and the codebase has three layers of abstraction between where the exception lands and where the problem actually lives.

This chapter is about what actually helps — and what wastes your time — when you bring AI into a debugging session.

The Rubber Duck Got an Upgrade

The classic rubber duck technique is simple: explain your problem out loud to an inanimate object, and in the act of articulating it, you often find the answer yourself. It works because narrating forces you to slow down and structure your thinking. The duck contributes nothing. That's not a bug; it's the feature.

AI-assisted debugging is rubber-duck debugging with a duck that occasionally says something useful. And occasionally says something confidently wrong. The key insight is that the articulation process is still where a lot of the value lives — not just in whatever the model says back.

Simon Willison, whose writing on practical LLM usage at simonwillison.net is some of the most grounded you'll find, has described his experience debugging with AI in terms of "thinking out loud to something that sometimes finishes the thought for you." That framing is honest. You're not outsourcing the debugging. You're using the process of explaining the problem to sharpen your own thinking, with the occasional real hit when the model recognizes a pattern from its training data.

This matters because it sets the right expectations. You're not handing the AI a bug and waiting. You're in a dialogue. And like any dialogue, what you put in shapes what you get back.

The best debugging prompts aren't requests. They're narrations.

Instead of: "Why is this throwing a null pointer exception?"

Try: "I have a Node.js service that processes webhook events. It calls getUserById with an ID extracted from the payload. The function returns null in staging but never in local dev. Here's the stack trace. Here's the relevant function. My hypothesis is the database connection pool is exhausted, but I'm not sure why that would only happen in staging. What am I missing?"

That second prompt is longer. It's also worth twenty times more. You've already done a third of the debugging by writing it.

What to Actually Give the AI

This is where most people go wrong.

There's a temptation — especially early in a debugging session when you're frustrated and just want to hand the problem to something that might fix it — to dump everything. The entire stack trace. The error logs. The function. The file. Sometimes half the project. You paste it all in and ask: what's wrong?

This almost never works well.

Stack traces are almost always the right starting point. They're dense with signal: the exception type, the call chain, the line numbers, the context. A clean stack trace is to a debugger what a patient's vitals are to a doctor — a structured summary of what the system reported about its own failure. Paste the full trace, not a cropped version. Do not summarize it in prose. The model reads raw traces well.

But stack traces have a failure mode: they tell you where the exception surfaced, not necessarily where the bug lives. A null reference exception twenty frames into the call stack often has its root cause in the function that should have returned a valid object three frames earlier. The AI knows this. So does every experienced developer. Help both of you by including the relevant surrounding code — not just the line that threw, but the caller that invoked it with data that turned out to be wrong.

Minimal reproductions are more powerful than stack traces when the bug is behavioral rather than exceptional. If your function returns the wrong value but doesn't throw, if a UI component renders incorrectly under a specific state combination, if a race condition surfaces only under load — these aren't stack trace bugs. They're logic bugs. For logic bugs, what you want to give the AI is the smallest possible code that demonstrates the wrong behavior, plus your observation of what "wrong" actually looks like.

"Here's a 15-line function. When I pass { count: 0 }, I expect false, I get true. I've read it three times." That's a debuggable prompt. The AI can reason through it cleanly.

A rough decision rule:

- Exception with a trace !’ paste the trace + the immediately relevant code
the caller that invoked it with bad input)
- Wrong output, no exception !’ paste the smallest reproduction you can co
expected versus actual output

- Intermittent or timing bug !’ describe the conditions under which it appe
  and lead with your hypotheses

What you almost never need to include: unrelated files, configuration files the bug couldn't touch, your full database schema when the issue is a JavaScript runtime error, comments explaining code that isn't involved in the bug.

Every token of noise you add to the context is a token of signal you're diluting. Claude's context window on the Sonnet-class models is 200K tokens as of early 2026. A large window doesn't mean a large window improves results. Context quality beats context quantity, nearly every time.

The Context Trap

Let's sit with this for a moment, because it's the single most common mistake I see developers make when AI-debugging.

It goes like this: something is broken, you don't know where the bug is, so you don't know what's relevant. Since you don't know what's relevant, you include everything. The entire service. Four related files. The types. The config. The error. You paste it all in and ask: "what's wrong?"

This almost never works well. Here's why.

First, long contexts shift model behavior in ways that aren't always visible. Research on long-context language models has documented that models tend to weight the beginning and end of long inputs more heavily than the middle — the so-called "lost in the middle" effect. If your actual bug is in a function buried on page three of a wall of code, there's a real chance the model spends its analysis budget on the wrong thing.

Second, the model will try to answer something. It won't say "I can't find the bug in this wall of code." It will point to something. Sometimes it'll be right. Often it'll identify a minor style issue or a theoretical edge case that has nothing to do with your immediate problem. You'll read the response, feel like progress is happening, and then realize ten minutes later you're no closer to the actual bug.

Third — and this is the subtle one — when you paste everything, you outsource the diagnosis step. And the diagnosis step is where the debugging actually happens. Forcing yourself to isolate the problem, to cut the reproduction down to 20 lines from 200, to articulate exactly what you expect versus what you observe — that process often reveals the bug before the AI says a word.

Willison describes this as the "explain it until it fixes itself" phenomenon. You're 200 words into writing out your debugging prompt and you suddenly stop, because you just realized you never validated that the upstream function was returning what you thought it was.

Use the AI as the forcing function for that articulation. Give it enough context to be useful, not enough to be confused. A practical ceiling: if you're pasting more than 150 lines for a single debugging prompt, stop and ask yourself what you're actually trying to get the model to reason about. If you can't answer that clearly, the prompt isn't ready yet.

Prompt Patterns That Actually Work

Let's get specific. These structures consistently produce useful debugging output across Claude Code, Cursor, and Copilot Chat.

The hypothesis prompt. You have a theory about what's wrong. You want the AI to stress-test it, not confirm it.

> *"My hypothesis is that the session token is expiring before the refresh logic kicks in. Here's the auth middleware and the token refresh function. Tell me whether my hypothesis holds up, and if not, what else could explain this."*

The explicit ask — tell me if I'm wrong — matters. Without it, the model tends to elaborate on your hypothesis rather than challenge it. You want a skeptic, not a yes-man.

The differential diagnosis prompt. You've ruled things out. You want to narrow the remaining candidates.

> *"I've confirmed the input data is valid before it hits this function. I've confirmed the database connection is healthy. The function still returns null for user IDs above 10,000. Here are three possible causes I've thought of. What am I missing, and which would you prioritize testing first?"*

Listing what you've already ruled out is important. It stops the AI from walking you through steps you've already taken, which is one of the more frustrating failure modes in AI debugging assistance.

The log analysis prompt. Hand it a log segment with context about what those logs represent.

> *"This is 40 lines from an application log around the time a payment processing job failed. The service is Python, running Celery workers against a Postgres database. I'm looking for anything that suggests a worker timeout, a retry storm, or a dropped connection. Walk me through what you see."*

Note that you've told it what you're looking for. "What's wrong with these logs?" produces generic output. "I'm looking for evidence of X, Y, or Z" produces analysis that's actually useful. You're directing attention rather than hoping it focuses on the right signal.

The rubber duck prompt. You don't have a theory yet. You just want to articulate the problem clearly.

> *"I'm going to describe a bug and I want you to ask me clarifying questions until you have enough information to suggest root causes. Don't guess early. The bug: our image upload endpoint works fine in production, fails silently in staging, and I cannot reproduce it locally."*

Inviting the model to ask questions before answering shifts the dynamic from "AI guesses, you evaluate" to something closer to a structured diagnostic conversation. It slows things down deliberately. For complex intermittent bugs where you don't have enough signal yet, the clarifying questions themselves are often diagnostic.

Reading Log Output With AI

Log analysis deserves its own treatment because it's a place where AI genuinely outperforms most developers in raw throughput — but fails in specific, predictable ways you need to account for.

The throughput advantage is real. You can paste 300 lines of nginx access logs, application logs, and database slow query logs into Claude Code and ask it to correlate the timing across all three to find where a request went wrong. Doing that manually is tedious and error-prone. The AI will often spot a pattern — the database slow query fired 40ms before the request timed out, every single time — that would have taken you twenty minutes to trace by hand.

But log analysis has a specific failure mode: the model finds patterns, and it doesn't always know whether those patterns are significant. If your logs contain a benign warning that fires every thirty seconds regardless of application state, the AI will often flag it as suspicious because it appears near error events — even though it appears near everything. It correlates. It doesn't reliably distinguish correlation from causation.

How to handle this. First, tell the model what normal looks like. "This WARN: connection pool at 80% message fires regularly during peak traffic and is not the issue. Focus on anything unusual between 14:03:00 and 14:03:45." Second, ask it to separate what it sees from what it's inferring: "For each finding, tell me whether it's directly stated in the logs or whether you're inferring it from timing." This produces far more useful output than a simple "here's what's wrong."

Third — verify the timestamps yourself. Before you chase a lead, confirm with your own eyes that the events the model pointed to actually happened in the sequence it claims. Models can misread log timestamps, particularly across format variations or when logs from different services use different timestamp schemas in the same paste.

Hypothesis Generation When You Have No Leads

Sometimes you're not testing a hypothesis. You have nothing. The bug is new, the stack trace doesn't point anywhere obvious, and you've been staring at it for forty minutes without a lead.

This is where AI earns its keep in a different way: generating the list of hypotheses you haven't thought of yet.

The prompt structure here is deliberate:

> *"I'm going to describe a bug and I want you to generate 6 to 8 possible root causes, ordered from most likely to least likely, with one sentence of reasoning for each. Do not elaborate on any of them — just give me the ordered list. The bug: [description]."*

The request for an ordered list without elaboration is intentional. If you ask "what could cause this?" you get an essay. If you ask for an ordered list, you get a structured thing you can scan in thirty seconds. You read the top three, check whether any of them are obviously wrong given what you know, and start testing from there.

The other prompt that works here — and I use this more than almost anything else — is asking the AI to identify the assumptions your code is making that you forgot to validate.

> *"Here is a function I wrote six months ago that is now failing. List every assumption the code makes — about input types, about state, about external service behavior, about timing — that I might not have explicitly validated."*

That prompt is uncomfortable to read the output of. It's supposed to be. You wrote the function with a dozen implicit assumptions baked in, and seeing them listed makes the potential failure points obvious. One of them is almost always the bug.

When the AI Gives You a Confident Wrong Answer

This happens. It happens more often in debugging than in code generation, because debugging requires genuine causal reasoning about a specific runtime system the AI has never actually executed.

The signs:

The explanation is smooth and authoritative but doesn't account for something you know to be true. ("The issue is that your Redis client is reconnecting on every request" — but you can see in your metrics dashboard that connection count is stable.)

The suggested fix is a real pattern but isn't connected to the actual error. ("You should add a null check here" — the null check is reasonable advice, but the error is coming from somewhere else entirely.)

The AI references behavior of a library or framework version you're not running. This is common with fast-moving ecosystems. If you're on Next.js 15, a recent Rust async runtime, or anything that shipped a breaking change in the last twelve months, the model may describe behavior from a prior version with complete confidence.

Your defense against all three: don't accept the diagnosis before you can explain why it would cause the specific behavior you observed. If the AI's explanation doesn't connect mechanistically to the error you're seeing, treat it as a hypothesis to test, not an answer to accept. The model produced a plausible story. Plausible stories can be wrong.

This is where a second-pass prompt earns its keep:

> *"Play devil's advocate on the diagnosis you just gave. What would have to be true for that explanation to be wrong?"*

That single follow-up surfaces weaknesses in the model's own reasoning in a way that "are you sure?" never does. Ask it to challenge itself.

A Realistic Debugging Session

Here's what a real AI-assisted debugging session looks like when it goes well — not the demo version where everything resolves in thirty seconds, but the actual working version.

A background job that processes order fulfillment emails starts silently dropping messages in production. No exceptions in the logs. No errors in the queue consumer. Messages are being acknowledged but orders aren't being fulfilled. You have a few hours to fix it before the business notices.

Round one. You paste the 22-line stack trace from the dead-letter queue processor, along with the 35-line function that handles message acknowledgment. You ask Claude Code to describe what should happen versus what might cause silent drops. It identifies three candidates: the ack firing before processing completes, an exception being swallowed somewhere in the handler, or a filter condition silently excluding messages. You hadn't thought of the filter condition.

Round two. You check the filter. There's a condition that excludes orders flagged as "test." Someone three days ago ran a load test in production and left 400 test-flagged orders sitting in the queue. You verify against the database in two minutes. That's the bug. Not the code you were looking at at all.

Round three doesn't happen. You fix the filter condition, drain the dead-letter queue, write a test that validates the filter excludes only what it should, and close the incident.

The AI didn't solve the bug. It generated the hypothesis you hadn't thought of. The total AI interaction was six minutes. The debugging session, from alert to fix, was forty minutes. Compare that to a two-hour session of manually reading queue processing code without a lead.

That's the honest version. Not magic. Not a replacement for knowing your system. A thinking partner that occasionally hands you the right hypothesis at the right moment.

The Setup That Makes All of This Work

None of these patterns work well if your setup fights you. In Claude Code, the --context flag and the #file syntax in Cursor let you include specific files without dumping your whole project into the window. Use them deliberately — point at the file containing the broken function, not the directory containing your entire service.

Keep a debugging scratch file. A DEBUG.md or a scratch buffer where you narrate the problem, paste your prompts, and record what you've ruled out. This isn't busywork. It's the written version of the rubber duck, and it doubles as a record you can hand to a teammate — or a second model — if the first pass doesn't crack it.

The habit that matters most: before you paste anything into an AI debugging prompt, write one sentence that completes this template — "I believe the bug is in [X] because [Y], and what I can't explain is [Z]." If you can't complete that sentence, you're not ready to prompt yet. Do five more minutes of solo investigation first.

The AI is fastest when you're already 60% of the way to the answer. Feed it precisely. Direct its attention. Verify before you trust.

The rubber duck became a debate partner. That's still not a bad deal.

CHAPTER IX

When to Put the AI Away

There's a specific feeling that should make you stop. You've been working with the AI for an hour, things have been moving fast, and you're about to accept a suggestion — a block of code that looks right, reads cleanly, and solves the problem you described. But somewhere in the back of your head there's a small, quiet alarm. You don't fully understand what the code does. You're not sure where the edge cases land. You're mostly trusting the output because it sounds confident and you want to keep moving.

That feeling is the signal. The moment before you hit accept is exactly when you need to ask yourself whether the AI should still be in the loop.

This chapter is about learning to recognize that moment earlier, and about the specific categories of work where AI assistance doesn't help you — it degrades your output while making you feel productive. Not as a general anti-AI argument. As a practical inventory of the situations where the tools reliably fail working engineers, so you can route around them instead of discovering them in production.

The Speed Illusion

The first thing to understand is that AI assistance can feel fast while actually being slow. This is one of its sneakier failure modes, because you're measuring the wrong thing.

You get a suggestion in two seconds. You read it for fifteen seconds, decide it looks roughly right, accept it. Four minutes later you realize it doesn't handle the case where the input list is empty. You fix that. Six minutes later you realize the fix introduced a subtle off-by-one on the index. You find that during code review. By the time you've made three passes through code you only half-understand, you've spent more time than you would have spent just writing it.

The illusion is that generation is fast, so the whole workflow must be fast. It isn't. Generation is fast. Comprehension takes as long as it takes. And if the generated code is doing something non-trivial, comprehension doesn't get a discount.

Simon Willison, who has written more clear-eyed analysis of practical AI coding than almost anyone working publicly, put it plainly on his blog: the problem isn't that the AI is wrong, it's that when it's wrong, it's wrong in ways that are difficult to spot because the code looks like code a competent developer would write. The syntax is correct. The naming is reasonable. The logic looks plausible. The bug is in the semantics, or the assumptions, or a subtle misunderstanding of what you actually asked for.

That's different from a bug you introduce yourself. When you write buggy code, you usually have a model of what you intended, which makes the bug easier to find. When AI writes buggy code, you sometimes lack that

model entirely.

Where the Training Data Runs Out

AI coding tools are very good at well-traveled ground. They've seen thousands of implementations of JWT validation, hundreds of examples of parsing CSV with edge cases, dozens of ways to set up a React context. The model has strong priors on these things. Output quality is high.

Move off the beaten path and the quality drops fast, often without any visible warning.

Proprietary internal systems. Domain logic that only exists in your company's codebase. Regulatory calculations under rules specific to a particular jurisdiction. Anything built on a library that's either very new or has a small user base. Code interacting with a third-party API that changed its behavior six months ago and hasn't been widely written about. In all of these areas, the model is pattern-matching to superficially similar things it has seen, and the match may be close or it may be badly off.

This matters in a specific way. If you're working in a domain where you're also not the expert — you're a backend engineer writing financial calculation logic you've only recently learned, or a web developer implementing compliance rules you got in a document from legal — you're in a particularly dangerous position. You're not expert enough to notice when the model's output misses something important. The model isn't expert enough to get it right reliably. And the result can look perfectly fine to both of you.

The working rule: the less you know about a domain, the more dangerous it is to rely heavily on AI for that domain. Your expertise is part of the quality-control loop. When you can't play that role, the loop is broken.

Security Logic Is a Special Case. Treat It as One.

Authentication flows. Authorization checks. Input sanitization. Cryptographic operations. Session management. SQL query construction.

These are the categories where plausible-looking wrong code has catastrophically different consequences than plausible-looking wrong code anywhere else. A generated function that sorts a list incorrectly is a minor bug. A generated function that validates a JWT incorrectly is a security hole.

The specific risk isn't that AI produces insecure code most of the time. It doesn't. The specific risk is that AI produces insecure code occasionally, and in a form that's hard to distinguish from correct code on quick inspection, and in categories of code where the consequences of missing it are severe.

Anthropic's own guidance on Claude Code usage notes that models can produce code that works for the happy path and fails on adversarial inputs. This is almost a structural property of how models are trained — they're optimized on normal cases. Your threat model is exactly the abnormal cases.

Some concrete patterns to watch for:

A generated access control check that works correctly when a user has the expected role, but whose logic inverts unexpectedly when the role field is null or missing. A password comparison that uses == instead of a constant-time comparison function, opening a timing side channel that's invisible in a code review unless you know to look for it. An input validation function that sanitizes HTML entities but misses Unicode normalization, letting a carefully crafted string bypass the check. SQL query construction that parameterizes the values correctly but interpolates the column name directly, because you phrased your prompt as "make this dynamic."

None of these look obviously wrong. None of them will throw an error. All of them are things models have been observed producing.

The working rule: for any code that touches authentication, authorization, cryptography, or user-supplied input used in queries or system calls, write it yourself or perform a thorough adversarial review that takes significantly longer than writing it would have. Use the AI for research — "what are the common vulnerabilities in JWT implementations" — not for implementation.

The Confident-Wrong Problem

Models don't have uncertainty the way you do. When you're unsure about something, you slow down. You add a comment saying "I'm not sure this handles concurrent writes." You ask a colleague. You go check the docs. You emit some kind of signal that there's a question here.

Models don't do that by default. They produce well-formatted, confident-sounding output whether they're on solid ground or not. This is one of the most important things to internalize about working with these tools, and it's not a bug in any particular model — it's a structural feature of how they generate text.

OpenAI's own technical documentation on model limitations acknowledges this directly: models can state incorrect information with the same fluency and apparent authority as correct information. The confidence of the output is not a signal about the accuracy of the output.

This creates a specific cognitive trap. You develop a feel for what "good" AI output looks like — it's detailed, it anticipates edge cases, it includes comments, it handles errors. And you start to associate that style with correctness. But the same style can appear on output that's deeply wrong. The model has learned to produce output that looks competent, and the appearance of competence is not competence.

The failure mode that follows from this: you read generated code with a filter set to "does this look like the right kind of thing" rather than "is this actually correct." Those are different questions. The first is a pattern match. The second requires actual reasoning about what the code does.

Willison has documented this repeatedly in public posts — models producing Python that references libraries that don't exist, citing functions with plausible names that aren't in the actual API, giving explanations for why code works that are internally coherent but factually backwards. All of it delivered with the same smooth, professional tone as correct output.

Cognitive Capture

There's a concept that gets discussed in automation research sometimes called automation bias — the tendency for humans working alongside automated systems to over-trust the system's outputs, especially when the system is usually right. Air crews have been studied on this. Medical diagnostic software, too. The pattern repeats: when a system is correct ninety-something percent of the time, humans working with it develop a mental habit of accepting its outputs, and that habit persists past the edge of the system's competence.

AI coding tools produce this same dynamic, and they produce it fast. Within a few days of working with a capable coding assistant, you start trusting it. That trust is not entirely misplaced — the tool is often right. But the trust doesn't come with a reliable calibration for when it's about to be wrong.

The specific version of this in software development: you accept code you don't fully understand because you're tired, or you're behind, or the code looks clean, or you've accepted the last four suggestions and they were all fine. The model has earned credit it's spending in a place where it shouldn't be spending it.

This is cognitive capture. The AI has captured your cognitive load, and you've let it. You're no longer reasoning about the code — you're supervising a process you believe is working. When the process stops working, you may not notice until the damage is done.

The moment you notice you're reading generated code for style rather than semantics, stop. You're not reviewing code. You're proofreading it, which is not the same thing.

The Specific Situations to Route Around

Some categories where you should reach for the AI less or not at all:

Novel architectures with no established pattern. If you're the first person on your team to implement a particular distributed systems pattern, or you're using a new infrastructure product that shipped three months ago with sparse documentation, the model's training data on this is thin. It will generate something that looks like it should work, assembled from patterns that superficially resemble your use case. Often that works. Just as often, it misses a critical constraint specific to the thing you're actually using. Read the source code and the official docs yourself.

Numerical code in regulated domains. Financial calculations, actuarial computations, clinical dosage logic. These domains have specific precision requirements, specific rounding rules, specific edge cases defined by regulation or standard. A model that doesn't have precise knowledge of GAAP revenue recognition rules, or the IEEE 754 corner cases that matter for your specific operation, will write code that looks correct and fails audits or produces wrong numbers in production. The cost of getting these wrong is not a bug report.

State-machine logic at scale. Models are good at simple state transitions. They're much less reliable on state machines with more than a handful of states, especially when transitions have side effects or when invalid

state combinations are meaningful. The model tends to implement the happy path cleanly and handle error transitions poorly or incorrectly. In complex workflow engines or protocol implementations, this is exactly where bugs live.

Debugging non-deterministic failures. Race conditions. Heisenbugs. Timing-dependent failures that only manifest under load. Memory corruption. These are problems where understanding comes from reasoning carefully about execution order, memory layout, and system behavior over time — not from pattern-matching to known bug types. A model can help you brainstorm hypotheses. It cannot reliably reason its way to the correct one. When you've got a bug that disappears under the debugger and only appears in production at 3am, the AI is a sounding board at best.

Code under active litigation or patent dispute. If you're working on code that's the subject of IP claims, or that touches areas with known active litigation, the last thing you need is AI-generated output with uncertain provenance. This is a legal question, not a technical one. Ask your legal team.

Code that touches compliance boundaries. HIPAA, GDPR, PCI DSS, SOX. These frameworks have specific technical requirements that can be easy to implement incorrectly. A model can tell you that "you should encrypt PHI at rest" and even give you encryption code. It's much less reliable on whether the key management approach you've implemented actually satisfies the HIPAA technical safeguard requirements in your specific architecture. Compliance isn't just a code-correctness problem. It's a regulatory interpretation problem.

The "Just One More Pass" Trap

Here's a failure mode that doesn't get talked about enough: the incremental accept.

You ask the model to write a function. It's not quite right. You ask it to revise. The revision is closer but introduces a new issue. You ask it to fix that. Now you've got a function that's been through three AI-generated iterations, and you've been reading it continuously, and each pass looked like an improvement, so you've been mentally accepting each delta rather than evaluating the whole.

Step back and read the final function cold, as if you'd never seen it, and sometimes you'll find it's worse than where you started. Not wrong in an obvious way. Wrong in a slow-accumulation way, where each revision patched one thing while subtly breaking something else.

The antidote is simple and uncomfortable: periodically throw away the generated version and reread the specification. Ask yourself what the function needs to do. Then look at what you have and ask whether it does that, not whether it's an improvement on the previous version. Improvement-from-last-version and correctness-of-final-version are not the same question.

The Gut-Check Checklist

This is the practical part. Before you accept a significant piece of generated code — let's say anything more complex than a ten-line utility function — run through these questions. Not as a ceremony. As a genuine check.

Can you explain what this code does to someone who hasn't seen it? Not at a hand-wavy level. At a specific, line-by-line level for any section that matters. If there are sections you'd have to say "I think it's doing something like..." about, that's a flag.

Do you understand the failure modes? What happens when the input is null? When the list is empty? When the network is down? When the user has a role your access control logic didn't anticipate? If you can't answer these questions, you don't understand the code well enough to own it.

Would you catch a bug in this code in code review? This is a useful forcing function. Imagine a colleague submitted this as a PR and you were reviewing it cold. Would you be able to spot if the logic was wrong? If the answer is no, you probably shouldn't be shipping it either.

Does this code touch a security boundary? Auth, authz, crypto, input validation, query construction, external API calls that return privileged data. If yes, apply a higher standard. Take the time. Read it adversarially.

Is this in a domain where you're also not the expert? Financial calculations, compliance logic, medical data handling, regulatory edge cases. If you're new to the domain and the AI is your main source of domain knowledge, you have no expert in the loop. Bring one in.

Has the model been confidently wrong in this session before? If the AI suggested something that turned out to be wrong, or referenced an API incorrectly, or hallucinated a function that doesn't exist, treat subsequent outputs in this session with more skepticism. Models don't recalibrate after being wrong. You should.

Are you accepting this because it's correct, or because you're tired? This one requires honesty. If it's the latter, stop for ten minutes. Come back. Review the code when you're actually thinking.

Can you delete the tests the model generated and write them from the spec? If the model wrote both the implementation and the tests, you need to verify that the tests actually test the spec, not just the implementation. The best way to check this is to write at least a few critical tests yourself, from scratch, against the behavior you want.

Is this going into a part of the codebase where a bug would be difficult to detect? Background jobs. Queue consumers. Rarely-triggered code paths. Code that runs once during initialization. These areas are where subtle bugs live the longest before surfacing. They deserve more human attention, not less.

If this code had a subtle bug, how would you find out? Monitoring? Tests? Customer reports? If the answer is "customer reports," or "we'd find out eventually," that's a sign you need more validation before this ships, regardless of whether AI wrote it or you did.

The Productive Use of "No"

None of this is an argument for using these tools less. It's an argument for knowing where to draw a line and drawing it without guilt.

The engineers who get the most out of AI coding assistance are not the ones who use it for everything. They're the ones who've developed a clear internal model of what the tools are reliably good at and what they're not, and who route their work accordingly. They use AI fast and confidently in the categories where it's genuinely strong. They route around it — or apply much more scrutiny — in the categories where it isn't.

That internal model takes time to build. You build it by noticing when the output was wrong and what kind of wrong it was. By noticing when you accepted something you didn't fully understand and had to go back later. By noticing which categories of problem leave you less confident after the AI helps than you were before.

The gut-check checklist above isn't about making AI assistance slower. It's about making it reliable. Speed without reliability isn't useful velocity — it's just fast debt accumulation.

Most working developers overuse these tools in exactly the failure modes described here, not because they're naive, but because the tools work well enough, often enough, that the habit of trusting them generalizes past where it should. The correction isn't skepticism as a posture. It's calibration as a skill.

Know what you're accepting. Know what you'd catch if it were wrong. Own what you ship.

When you can answer all three of those comfortably, use the AI as aggressively as you want. When you can't, put it away and do the work yourself. Not forever. Just for this function, this module, this security boundary, this thing that has to be right.

That's the whole checklist, really. Two questions dressed up as ten. Do I understand this? Would I catch a bug in it? If yes to both, accept it. If no to either, not yet.

CHAPTER X

Verifying Generated Code Without Re-Doing It by Hand

The verification problem has a shape that most developers get wrong on their first serious month with AI tools. They swing between two bad strategies: accept everything because the output looks reasonable, or reconstruct everything because they don't trust anything they didn't write. The first gets you paged at 2 a.m. The second evaporates the productivity gain you were trying to capture, and leaves you wondering why you're paying for a subscription that just makes your afternoon slower.

There's a third way. And it's not a compromise between the two — it's a different frame entirely.

What you're building, when you do this right, is earned trust in generated code. Not blind acceptance. Not paranoid line-by-line archaeology. A structured, fast process that tells you, with enough confidence to actually ship, whether the thing Claude or Copilot or Cursor just produced does what it claims to do and doesn't do something it shouldn't.

The difference between understanding the shape of a change and understanding every line of it is real and important. Sometimes you need the latter. Usually you don't. Knowing when you do is the whole game.

The Diff Is Your Primary Instrument

The single biggest leverage point in reviewing AI-generated code is something most developers already have in their workflow: the diff.

When you accept a generation — whether it's a new function, a refactored module, or a test suite — your first job is not to read the file. It's to read the change. What was there before. What replaced it. What got added. What disappeared.

This sounds obvious. It mostly isn't how people actually work. The natural instinct is to open the file and read the new version top to bottom, because that's how code review happens when a colleague submits a PR. But a colleague's PR arrives with context you already have: you know the codebase they were working in, you've often discussed the approach in advance, and you've reviewed their style before. With AI output, you have none of that implicit trust scaffolding. Reading the file without the diff means you're reading code in a vacuum, and the human brain is terrible at spotting problems in code that looks coherent on its face.

The diff forces specificity. It makes you answer the question: what actually changed, and is that change correct?

In Cursor, this is the panel that opens inline before you accept a suggestion. In Claude Code, it's the diff preview before you confirm a file write. In Copilot, it's your git diff after accepting a block. Make it a habit to read the diff before you read anything else. If the diff is small and legible, you can often validate the change in thirty seconds. If the diff is 400 lines across six files, that's the signal to slow down — not because AI is untrustworthy in principle, but because you cannot adequately verify 400 changed lines in a single review pass regardless of who wrote them.

Keep generations small enough that the diff is reviewable. This is a prompting discipline, not just a verification discipline. Ask for one function at a time. Ask for the refactor of one module, not the whole package. The size of the diff you're willing to accept in one shot should be roughly the same size you'd be comfortable reviewing in a colleague's PR before merging.

What to Read Closely and What to Skim

Not all generated code deserves equal scrutiny. Here is a rough triage.

Read closely: Any code that touches security boundaries — authentication, authorization checks, input validation, anything that handles user-supplied data before it hits persistence. Read this line by line. No exception. Simon Willison, who has written candidly about his AI-assisted workflow on simonwillison.net since 2023, has made the point repeatedly that the cases where AI coding tools produce the most confident-looking output are often the cases where the correctness conditions are most subtle. A permissions check that looks syntactically perfect can still be logically inverted. An input validation function can sanitize the right things in the wrong order. Read this stuff as if you'll be explaining it in a postmortem, because someday you might be.

Read closely: Any code that implements business logic the AI had to infer rather than being told explicitly. If you gave the model a function signature and a docstring but didn't specify the edge case behavior, it made decisions. Those decisions may be reasonable. They may also be subtly wrong for your domain in ways that won't surface in a happy-path test. Find the decisions and check them.

Read closely: Error handling. AI-generated error handling is frequently aspirational. It catches the right exception types, logs something plausible, and returns a sane-looking default. Then in production it silently swallows an IOException that was actually a sign the disk was full, and you spend four hours figuring out why writes are disappearing. Look at every catch block. Look at every return None or return [] and ask what it's masking.

Skim: The structural boilerplate that you can verify at the type level. If the function signature is correct, the types check, and the imports resolve, the skeleton is probably fine. You're looking for logical errors, not syntactic ones — your compiler or linter handles the latter.

Skim: Code that's a near-verbatim rearrangement of something that already exists in the codebase and that you know works. If the model generated a new endpoint handler that's structurally identical to three handlers you already have, and the diff shows it followed the same pattern, you're mainly checking that it didn't

introduce a deviation in the one place where your pattern has a subtle required invariant.

Don't bother reading in detail: Generated docstrings, comments, and README updates, unless documentation accuracy is the specific thing you're shipping. Read them for gross errors and move on.

Probe Tests: Fast Verification Without Full Coverage

Here is a technique that doesn't get talked about enough: the probe test.

A probe test is a small, throwaway test — or sometimes a set of assertions in a scratch file — written specifically to validate the decisions you couldn't verify by reading. Not a proper unit test with mocking and fixtures. A fast, blunt instrument.

Say the AI generated a function that parses a config file format you use internally. The logic looks right. You're not sure about the edge cases — what happens with empty sections, what happens with duplicate keys, what happens if the file has a BOM. You don't want to write a full test suite for this right now. What you want is three assertions that hit those specific cases, run in four seconds, and tell you whether the model's assumptions match your requirements.

Write those. Run them. Delete them when you're done, or keep them if they belong in the actual test suite. The point isn't test coverage. The point is targeted verification of the specific decisions you couldn't confirm through reading.

This is different from just running the existing test suite, though you should do that too. The existing tests verify behavior against requirements that existed before the change. Probe tests verify the model's inferred decisions against requirements that may have only been implicit in your prompt.

GitClear's 2024 analysis of code written with AI assistance — drawn from over 153 million lines of code across a sample of repositories — found that code churn (code added and then removed or rewritten within two weeks) increased by 39% year-over-year in heavily AI-assisted repositories. That's not purely a quality problem; some of that churn is iterative. But some of it is changes that looked fine on review and turned out to be wrong in production. Probe tests are cheap insurance against the second category.

The specific things worth probing, in order of payoff:

- Boundary conditions the model had to guess about. Off-by-one errors. Empty collection handling. Zero, negative, and maximum values.
- Concurrency behavior, if the function touches shared state. Even a trivial race condition test can surface an assumption the model made that doesn't hold.
- The contract between this code and its callers. If the function returns a different shape than callers expect, that's often invisible in a pure unit view.
- Error paths. What actually happens when you pass in something malformed? Does it raise? Does it return a sentinel? Does it silently succeed and do the wrong thing?

The Second AI Pass as Skeptic

Here's a technique that takes a bit of getting used to but pays off consistently: use a second AI pass to review the first one. Not to generate more code. To find problems.

The key is the prompt framing. There is a significant difference between asking "does this code look good?" and asking "what are the three most likely ways this code could fail in production?" The first prompt invites agreement. The second invites adversarial analysis. Language models are genuinely better at finding problems when you explicitly instruct them to look for problems rather than confirm correctness.

A prompt that works well in Claude Code or in Cursor's chat panel:

> *Here is a function that was just generated. I'm going to ship it. Before I do, play skeptic: what are the three most likely failure modes? Don't explain what it does. Just tell me where it could go wrong, with specific reference to the code.*

You'll get some false positives. The model will sometimes flag things that are actually fine, or describe failure modes that your architecture already prevents. That's okay. Your job is to triage the response, not to act on every item. What you're fishing for is the one real problem that slipped past your read, surfaced by a system that's good at pattern-matching on "code that looks like code that has failed before."

This works best when you use a different model for the review pass than the one that generated the code. If Claude Code generated the function, paste it into Cursor's chat running a different underlying model, or into ChatGPT. Same-model review can sometimes reproduce the same blind spots that produced the original error; a different model's training will have different weak spots.

A note on what this technique is not: it's not a substitute for your own judgment. The second AI pass is a net, not a guarantee. It catches things. It misses things. Your brain is still the last line of verification for anything where the stakes are high.

The Confidence-Complexity Inversion

This is the failure mode that bites experienced developers harder than beginners, because experienced developers know enough to recognize when output sounds right.

AI coding tools produce code with essentially uniform surface confidence. The function that reverses a string and the function that implements a distributed lock look equally authoritative in the output. There are no hedges in the syntax, no "I'm less sure about this part" in the generated code itself. This is structurally dangerous, because human reviewers unconsciously use confidence cues to calibrate scrutiny. You slow down for the uncertain-looking code. You speed through the code that reads like it was written by someone who knows what they're doing.

The problem is that complexity and confidence are inversely correlated in practice. The simple string-reversal function is almost certainly correct. The distributed lock implementation — which has to handle network

partitions, clock skew, and leader election edge cases — is exactly where you'd want to slow down. But both look equally polished.

Neil Perry and colleagues at Stanford published research in 2022 showing that developers who used AI coding assistance were measurably more likely to introduce security vulnerabilities than those who didn't — particularly when the task involved non-trivial security logic. The mechanism wasn't that developers trusted the AI blindly; it was that the AI output reduced their sense of uncertainty, which reduced the scrutiny they applied. Confident-looking code gets less review. Less review means more bugs ship.

The practical countermeasure: build your scrutiny around domain complexity, not output appearance. Keep a mental list of the categories where AI tools consistently make subtle errors — security logic, concurrency, numerical precision, anything involving your specific business domain that the model couldn't have learned from public code. Treat code in those categories as always requiring close review, regardless of how clean the output looks.

Reading for Patterns, Not Lines

When you do need to read generated code carefully, there's a technique borrowed from security code review that dramatically increases your hit rate: read for patterns, not for lines.

Security auditors don't read code the way you'd read a novel, sequentially. They read for shapes — structures that have historically been associated with vulnerabilities. A call to eval() anywhere. String interpolation into a SQL clause. os.system() with any variable component. These patterns don't always mean there's a bug. They mean there's a place worth looking.

Apply the same thing to AI-generated code. Have a mental list of the patterns that AI tools frequently get wrong in your specific stack. In Python async code: is the model doing blocking I/O inside an async function? In JavaScript: is it calling .then() on something that's already being awaited? In Rust: is it doing an unwrap() in a place that will panic in production? In SQL: is it constructing queries with string concatenation in a context where parameterization is possible?

You're not reading every line. You're scanning for the signatures of the specific failure modes that matter in your codebase. This is faster than sequential reading and — for the failure modes you're actually worried about — more reliable.

The failure patterns worth hard-coding into your mental checklist (adapt for your stack):

- Silent failure. Returned error values that aren't checked. Exceptions caught and swallowed.
- Resource leaks. File handles, database connections, goroutines, threads. Does the generated code clean up after itself in the error path as well as the happy path?
- Incorrect scoping. Variables captured in closures at the wrong time. State shared between requests in a context where it shouldn't be.

- Off-by-one in collection operations. Especially in pagination logic, chunking, and any loop that terminates on a count.
- Hardcoded assumptions about environment. File paths, port numbers, environment variable names. The model may have hardcoded something that belongs in configuration.

The Verification Checklist

Before you commit generated code, run through this quickly. It takes under two minutes when the code is small and well-scoped.

1. Did the diff match what I asked for? Not "does the code look good." Did the change scope match your prompt scope? If you asked for a function and got a function plus edits to three other files, understand why before you accept.

2. Do I understand every decision the code makes? Not every line — every decision. If control flow branches, do you know why? If a specific library was chosen, was that the right choice for your environment?

3. Is there anything in here I can't verify by reading? If yes, write a probe test for it before you commit.

4. Does it touch any of my high-scrutiny categories? Security, concurrency, business-critical logic, anything under regulatory requirements. If yes, slow down and read it properly.

5. Have I run the existing tests? Yes, obviously. But also: did any tests change in the generation? If existing tests were deleted or modified as part of the output, understand why.

6. Is there anything I'm accepting because I'm tired of reading, rather than because I'm confident? This is the honest question. It's easy to rationalize acceptance after you've been staring at a diff for ten minutes. If the answer is yes, take a break, then look again.

What Earned Trust Actually Means

The goal of all of this isn't to make AI-assisted development feel more like traditional development. It's to build a verification practice that's calibrated to actual risk rather than performed thoroughness.

Some generated code you can review in thirty seconds and ship with confidence. Some code deserves an hour of careful reading and a conversation with a colleague. The practice described in this chapter doesn't treat all code the same — it routes your attention to where it matters, and gives you fast-path options for the code that doesn't need deep scrutiny.

Anthropic's own documentation for Claude Code, updated through 2025, recommends treating generated code with the same review standards you'd apply to code from an unknown contributor — not because the model is untrustworthy, but because that framing correctly calibrates the scrutiny level. An unknown contributor's code gets read. It gets tested. It gets questioned where it's surprising. And then, if it passes those checks, it gets merged.

That's the model. Not paranoia. Not credulity. Just the same engineering rigor you'd apply to any code you didn't write yourself, applied efficiently to a class of output that happens to be very fast and very voluminous.

The developers who get the most out of AI coding tools long-term are the ones who built this muscle early. They're faster than they were before the tools existed. They're also not running from incidents caused by code they accepted without understanding. Both things are true at once, and neither one requires any particular heroism — just a workflow built on earned trust rather than assumed trust.

The vibe is real. The verification is how you keep it.

CHAPTER XI

Working on a Team Where Not Everyone Uses AI

Somewhere around the third sprint after you started using Claude Code seriously, you noticed something uncomfortable. Your PRs were bigger. They arrived faster. And the comments on them started carrying a faint undertone you couldn't quite name — something between curiosity and suspicion.

Your teammate Marcus, who has been doing this for eleven years and writes careful, meticulous code, looked at your 400-line feature branch and said, "How long did this take you?" Not good work. Not let me look at this more carefully. Just: how long.

That's the moment this chapter is about. Not the AI tools. The team.

The Speed Gap Is Visible, and That's the Problem

When GitHub published their Copilot productivity study in 2022 — the Peng et al. controlled experiment where developers using Copilot completed an HTTP server task 55.8% faster than those without it — the headline went wide. Faster. Good. Done.

What the headline didn't say was: faster is socially complicated.

On a team where output velocity has been roughly calibrated over months or years, a sudden jump is conspicuous. If you routinely closed three tickets a sprint and now you're closing seven, people will notice before they understand why. And in the absence of an explanation, they will generate their own. The most common ones are: you've stopped being careful, you got easy tickets, or something about the work isn't real.

None of this is irrational. On any healthy team, unusual speed is a signal worth examining. The problem is that if you haven't established the context — that your process changed, not your rigor — the conclusion your teammates reach will fill the vacuum.

This isn't about managing perception. It's about preventing a friction that compounds quietly until someone raises it in a retro, or in your performance review, or in an offhand comment that sticks. The gap between "using AI" and "everyone understands that you're using AI and what that means" is where most of the social pain lives.

Review Asymmetry

Here's the specific problem that doesn't get talked about enough: AI-assisted developers and non-AI-assisted developers have a structurally different experience during code review, and it cuts in both directions.

When you're reviewing code that your teammate hand-wrote, you can lean on the shared assumption that every line was considered. The author's fingerprints are on the logic. If something is weird, there's a reason — you just need to find it. That assumption guides your review. You're looking for what was meant as much as what was written.

When you're reviewing AI-generated code — whether it's yours or someone else's — that assumption doesn't hold. The code may be syntactically correct, locally coherent, and functionally plausible while still being wrong in a way that's hard to spot on a surface read. The AI had no stakes. It wasn't solving your problem; it was pattern-matching to code that looked like it solved your problem.

Now flip it. You're using AI. Your non-AI teammate is reviewing your PR. They're applying the first model — every line was considered, everything weird has a reason — to code that was partly generated and accepted. If you didn't catch the weird thing, they may not either, but for different reasons. You didn't catch it because you were moving fast and the code looked right. They didn't catch it because they assumed you'd already looked at it carefully.

This is review asymmetry: two people in the same review, working from different mental models of what happened to produce the diff. Neither of them is making a mistake, exactly. The process is just silently miscalibrated.

The fix isn't complicated, but it requires you to do something that feels slightly uncomfortable: communicate more about how code was produced, not just what it does.

"Who Wrote This?" and Why That Question Gets Thorny

Attribution on AI-assisted work is genuinely murky, and the murkiness matters at the team level in a few specific ways.

The clearest one is ownership. When something breaks in production at 2am, someone needs to understand the code well enough to fix it. If that code was largely generated by Claude and you accepted it after a quick look, your depth of understanding is different than if you'd written it line by line. That's not a moral failing. It's just true, and your on-call rotation depends on the team being honest about it.

Simon Willison, whose writing on AI-assisted development has been some of the most technically grounded of anyone in the field, has made the point that the question isn't whether AI helped you write the code — it's whether you understand the code well enough to own it. He's described a rule he applies to his own work: if he can't explain what a generated piece of code does without re-reading it slowly, he rewrites it or at minimum sits with it until he can. Not for ethical reasons. For maintenance reasons.

That's a useful framing for attribution on teams. The question isn't credit. The question is who can answer questions about this code at 2am. If the answer is "me, but I'll need fifteen minutes to reconstruct what I was thinking," say so. Not as a confession. As useful information.

The second attribution issue is more political. Some organizations — and some individual engineers — have opinions about AI-assisted code ranging from mild skepticism to active hostility. If those opinions exist on your team and you haven't disclosed that you're using AI, you're not just withholding information. You're creating a situation where the disclosure, when it eventually happens, will carry a layer of "why didn't you say so earlier" that makes everything harder.

This doesn't mean you need to put an AI badge on every commit. It means you shouldn't actively conceal something that's relevant to how your teammates will work with your output.

Marcus, Who Is Vocally Suspicious

Let's return to Marcus.

He's on your team. He's good. He has strong opinions about correctness and he's usually right. He doesn't trust AI tools — not on philosophical grounds, but because he's been burned by plausible-sounding wrong answers in his own experiments, and he's extrapolated appropriately. He hasn't seen evidence that AI output can be consistently trusted at the level the marketing implies, and he's not wrong to be skeptical.

Marcus is also, in some meetings, your loudest friction source. When you ship fast, he reviews slow. When he reviews slow, he finds things — sometimes real things, sometimes overfitting to style — and he's not shy about raising them. The velocity difference has become a kind of implicit argument, with neither of you stating it directly.

Here's what doesn't work with Marcus: evangelizing. Do not explain to Marcus why AI tools are good, or offer to show him a demo, or tell him the productivity numbers from the GitHub study. He knows the numbers. He's read more about this than you have. He's unconvinced for reasons that are more specific than general skepticism, and treating him like he just hasn't encountered the right information is condescending.

What works with Marcus is slower PRs with better context. If you're shipping AI-assisted code that covers 400 lines, write a PR description that makes the review tractable. Not "added feature X" — a description that tells Marcus where to look, what the interesting decisions were, and where you're least confident. Make the review easier and you remove the justification for the slow, suspicious read. You also give him something concrete to engage with, which means if he has a real objection, it surfaces as a specific comment rather than a generalized mistrust.

The other thing that works with Marcus is being right. Consistently. The skepticism toward AI-assisted code is, in substantial part, a fear that the work is shallower than it looks. If your AI-assisted code holds up — if it passes review, handles edge cases, survives production — that's evidence. It accumulates. You don't get to cash it in as a philosophical point, but it changes the baseline.

Disclosure Norms That Don't Feel Like Confession

What should you actually say about using AI, and when?

A few things that are clearly reasonable:

In PR descriptions: If a significant portion of a PR was generated or substantially shaped by AI, a line like "drafted with Claude, reviewed and tested by me" costs nothing and provides useful context. It tells reviewers that the code came from a different generative process than hand-writing, which changes how they should read it. It doesn't imply lower quality. It implies different provenance.

Some teams have made this a standard part of their PR template. A single optional field: "AI involvement: none / assisted / substantial." Low overhead, high signal.

In code comments: For complex generated logic — the kind you accepted because it worked but would take you an hour to reconstruct from scratch — a comment that says "generated implementation, here's what it does and why" is useful technical documentation. Not "I got this from Claude." Just: here's the intent, here's the behavior, here's what to watch for if you change it.

In standups and planning: You don't need to report your tooling in every standup. But if someone asks how you estimated a task and you estimated it assuming AI assistance, say so. Estimates made with AI assistance are different from estimates made without it, and if a non-AI teammate picks up the ticket, your estimate may be wrong for them.

When you're uncertain: If you shipped AI-generated code that you're not fully confident in, say so. "I'm not 100% sure about the edge behavior here, could use a close look" is a normal thing to write in a PR comment. The fact that your uncertainty comes from accepting generated code rather than writing uncertain code yourself doesn't change its relevance.

What you shouldn't do is perform transparency as a ritual that substitutes for actual understanding. Writing "generated by AI" on code you don't understand doesn't discharge your responsibility for that code — it just labels your uncertainty. The disclosure is for your team's benefit. The understanding is for yours.

PR Hygiene for AI-Assisted Work

AI-assisted development produces a specific kind of PR problem that doesn't exist at the same scale for hand-written code: the large, heterogeneous diff.

When you're writing code manually, there's a natural friction to scope. You write what you need, you stop when you're tired, the PR is roughly proportional to your effort. When you're working with Claude Code or Cursor, the proportionality breaks. A well-scoped prompt can produce two hundred useful lines in eight minutes. A slightly too-broad prompt can produce four hundred lines that are 80% right and 20% quietly wrong in ways that take twice as long to review as the good parts.

This is the primary PR hygiene problem with AI-assisted work: the diff is larger, more varied, and less uniformly owned than a diff of comparable size from a careful human author.

The discipline that helps here is scope control before prompting, not cleaning up after. If you know you're going to generate a significant chunk of code, decide the boundaries of that chunk before you start. What files does it touch? What does it explicitly not do? What are the success criteria you'll evaluate the output against? Writing this down — even just for yourself, in the PR description you'll fill in later — forces the scoping that AI tools don't impose on you automatically.

Then, after you have output: before you open the PR, read the diff the way a reviewer will. Not as someone who knows what was intended. As someone seeing this for the first time, with no context about the prompts, the iterations, or which parts you're confident in. The places where you'd want context as a reviewer are the places where you should add it.

A few specific PR hygiene practices that reduce friction on mixed teams:

- Break AI-assisted PRs by concern, not by generation session. If one Claude conversation produced code that touches authentication, routing, and database schema, those should be separate PRs if possible. The fact that they were generated together doesn't mean they should be reviewed together.
- Delete generated code you don't understand. If you accepted a function because it seemed to work but you can't explain its logic, either understand it before the PR or replace it with something you can. This is the Willison rule applied to PRs: don't ship what you can't own.
- Call out the uncertain parts. A PR comment saying "this error handling is what Claude suggested, I'm not sure it's complete, specifically around the case where X" is useful. It directs the reviewer. It's also honest.
- Test coverage as proof of understanding. Writing the tests for AI-generated code yourself — not generating them, actually writing them — is one of the strongest signals that you understand what the code is supposed to do. It's also practically valuable, because generated tests frequently test the AI's assumptions rather than your requirements. More on this in chapter seven.

The Norms Conversation You Should Have Once and Not Repeatedly

If AI tool adoption is creating genuine friction on your team — slowed reviews, implicit skepticism, uneven contribution rhythms — that's worth one explicit conversation, not an accumulation of indirect signals.

The conversation doesn't need to be dramatic. "I've been using Claude Code heavily for the last month, here's what that looks like for my PRs, here's what it means for reviews, does anyone want to establish norms around this" is a complete agenda item. Fifteen minutes. Maybe twenty if Marcus has feelings.

What you're trying to establish is shared mental models about three things:

- What AI assistance means for code ownership. Who is accountable for AI-assisted code? What depth of understanding is expected? Is "it passed review" sufficient, or do we want something more explicit?
- What AI assistance means for review. Should reviewers treat AI-assisted code differently? Should they be told when a PR contains substantial generated content? Is there a different standard for test coverage?

- What AI assistance means for estimates and velocity. If some developers are using AI and some aren't, your sprint velocity is measuring different things for different people. That doesn't automatically cause problems, but it's worth knowing.

You probably won't get perfect answers to all three in one meeting. What you'll get is a team that has said these things out loud, which means the next time friction arises, it's a calibration problem rather than a trust problem. That's a much better place to work from.

What Not to Do

Since this chapter is at risk of sounding relentlessly reasonable, here's a short inventory of approaches that reliably make things worse.

Hiding your tools. If you're using AI heavily and you haven't said so, the disclosure will happen eventually — through a question you can't answer about your own code, through a stylistic inconsistency that looks off, through a teammate who notices you typing a prompt when they walk by. Discovery is worse than disclosure. Say something now, at low stakes, rather than explaining yourself later at high stakes.

Evangelizing during a code review. If your teammate raises a concern about a PR, "this was generated by AI, which is why it looks different" is not a response to the concern. Address the concern. Separately, maybe, have the conversation about tooling. Mixing them makes it look like you're invoking the AI as an authority, which is exactly the wrong message to send to someone already skeptical.

Generating your way out of a misunderstanding. If Marcus doesn't like how you structured something, don't go back to Claude, regenerate an alternative, and present it as your own considered revision. This is the move that, if discovered, destroys trust faster than almost anything else. It's also cognitively lazy in a way that will eventually bite you. You need to understand the tradeoffs well enough to make a choice, not just to produce a new option.

Using AI adoption as a status signal. There is a version of this — usually deployed by people who are genuinely excited about the tools and have poor audience awareness — where AI use becomes a way of performing modernity at slower colleagues. If you find yourself saying "I just don't understand why you wouldn't use these tools" in a tone that's more about your identity than their workflow, stop. It doesn't help adoption and it poisons the working relationship.

Assuming non-AI teammates will stay non-AI. The colleague who's skeptical today may be an enthusiastic user in six months, once they've worked through their own concerns at their own pace. Leave the door open. Don't create a dynamic where coming around means admitting they were wrong. Make it easy for them to try things quietly without it being an event.

The Reviewer on the Other Side of the Asymmetry

One more angle worth taking seriously: if you're on a team where some colleagues use AI and you don't, the review asymmetry runs in your direction too.

When you're reviewing AI-assisted code, the reasonable response isn't heightened suspicion across the board — it's calibrated attention. The failure modes of AI-generated code are somewhat consistent: edge case handling at system boundaries, logic that works for the happy path but doesn't consider state transitions correctly, error handling that's syntactically complete but semantically wrong. These are worth looking for specifically, not because AI is untrustworthy, but because they're where the pattern-matching approach tends to break down.

The other reasonable response is to ask questions that verify understanding. "Walk me through how this handles the case where the upstream returns a 429" is a reasonable review comment whether the code was generated or hand-written. If your colleague can answer it clearly and immediately, the code was understood before it was shipped. If they have to go look, that tells you something useful about the ownership model.

This isn't punitive. It's the same standard of understanding you'd want applied to your own work, regardless of tools.

The Longer Game

Teams that handle this well tend to arrive at something simple: a shared norm that AI tools are part of the environment, like linters or automated tests, and that the responsibility for what ships remains with the person who proposed it. The tool doesn't change accountability. It changes throughput. The social contract stays the same.

That's not a complex idea. Getting there takes a few deliberate conversations, a bit of friction, and probably one uncomfortable PR review where someone asks a question you should have been able to answer and couldn't. That experience is useful. It's how you calibrate the depth of review your AI-assisted work actually needs.

The teams that handle it badly tend to land in one of two failure modes. Either AI use is invisible — people are using it, no one is talking about it, and mismatches between code quality and apparent author expertise are quietly eroding trust. Or AI use is a battlefield — every PR from an AI-assisted developer is reviewed with extra scrutiny by people who feel like the tools represent a shortcut to something that shouldn't be shortened.

Neither of those is stable. Both are avoidable with a modest amount of directness early on.

You don't have to be the person who runs the meeting, writes the RFC, and changes the team culture. You do have to be someone who doesn't hide what they're doing, writes PRs that are honest about their provenance, and meets skeptical colleagues with patience rather than evangelism. That's a low bar. It's also, apparently, not universal.

Start there.

CHAPTER XII

Security, IP, and the Paste Rule

The premise is embarrassingly simple. If you wouldn't paste something into a Slack channel visible to every employee at your AI vendor's company, don't paste it into a prompt.

That's the Paste Rule. It doesn't require legal training. It doesn't demand a formal threat model. It requires the same judgment you already apply to email subject lines and screen-sharing during a call with contractors. The problem is that the AI chat interface feels private in a way that Slack obviously doesn't. The box is just sitting there, waiting. It feels like thinking out loud. It feels like a conversation. Not a data transfer.

That feeling is wrong. And it has already cost people their jobs.

The Samsung Problem (And Why It Keeps Happening)

In early 2023, Samsung permitted engineers to use ChatGPT internally. Within twenty days, at least three separate incidents were reported where employees had pasted sensitive internal source code directly into the chat — including semiconductor equipment measurement data and proprietary meeting notes. Samsung responded by banning generative AI tools on company devices and internal networks. The engineers involved weren't reckless people. They were solving real problems under deadline pressure, and the fastest path to a solution ran straight through the chat box.

This pattern repeats constantly. It just usually doesn't make the news.

The reason it keeps happening is a mismatch between how the interface feels and how the system actually works. When you type into Claude Code or Cursor or the Copilot chat panel, that text travels to a server. It gets processed. Depending on your subscription tier, your organization's contract, and settings you may never have read, it may or may not be stored. It may or may not be used to improve the model. The people who built the tool almost certainly have reasonable privacy practices. But "reasonable" and "completely private" are not the same thing, and the terms of service know that even if your subconscious doesn't.

What the Tools Actually Do With Your Prompts

Before getting into categories of things you shouldn't paste, it helps to understand the data flows behind the tools you're actually using.

Claude Code / Anthropic API. When you use Claude through the API — which is what Claude Code runs on — Anthropic's default API terms state that they don't train on API inputs and outputs. Enterprise agreements can additionally include data processing agreements that clarify retention windows and regional processing. Anthropic's privacy documentation distinguishes between consumer Claude.ai (where data may be used for

training by default unless you opt out) and the API tier (where it isn't). That distinction matters. If your company has a Claude Code enterprise arrangement, read the contract. If you're running personal Claude Code and pasting in work code, you are almost certainly operating outside your employer's data handling agreements, regardless of what Anthropic's terms say.

Cursor. Cursor has a Privacy Mode — toggleable in settings — that prevents your code from being stored on their servers or used for training. When Privacy Mode is off, Cursor indexes your codebase locally and sends relevant chunks to the model as context for completions. When it's on, that indexing still happens locally, but Cursor states that code is not persisted server-side. Their SOC 2 Type II certification covers the enterprise tier. The relevant detail for sensitive work: Cursor's autocomplete feature is always sending code snippets to the model in real time. Privacy Mode controls retention and training use, not the act of transmission.

GitHub Copilot. GitHub has historically offered business and enterprise tiers with explicit no-training guarantees and data handling commitments. The free and individual tiers have had murkier histories. As of 2025, GitHub Copilot Business promises that prompts and suggestions are not used to train the model and are not retained beyond the session. Copilot Enterprise adds additional audit logging and network-level controls. The individual tier's defaults have changed multiple times; check current terms if you're using it on personal projects that happen to contain sensitive material.

Simon Willison, who has written more carefully about LLM security and data handling than almost anyone working publicly in this space, puts it plainly: the question isn't just what the vendor promises, but what surface area you're creating. Every piece of data you send to a third-party API is data that has left your control at the moment of transmission, regardless of what happens to it afterward. Network interception, vendor breaches, subpoena risk — none of those depend on the retention policy you agreed to.

The No-Paste List

Here are the categories that warrant actual policy — not paranoia, but a standing decision you don't relitigate every time you're under pressure.

Credentials and Secrets

API keys. Database connection strings. OAuth tokens. Private keys. Anything in a .env file. Anything in a secrets manager. Anything you'd rotate if you thought it might have been compromised.

Do not paste these. Not even as an example. Not even with the actual value replaced by a placeholder that you're planning to swap out. "Here's my code but I replaced the real API key with REPLACE_THIS" is fine — but check that you actually did replace it before you hit enter, because the number of times developers have pasted a prompt they were "about to sanitize" is higher than you want to believe.

The specific risk here is concrete. If a vendor does experience a breach, or if your conversation is visible to a support engineer reviewing a bug report, a credential in a prompt has now leaked outside the rotation cycle you control. Even credentials with very short TTLs create a window. And keys that live in old conversation

history create an indefinitely long window for anything that was captured in logs.

Rotate any credential that has already appeared in a prompt, regardless of the tool and regardless of what you think the retention policy says. This is not overcaution. It's the only response proportionate to not knowing exactly what happened to that text.

PII and Client Data

Personally identifiable information — real names, email addresses, phone numbers, government IDs, IP addresses tied to real people, health information, financial records — does not go into prompts. Full stop.

This isn't just a privacy norm. It's a legal exposure question in most jurisdictions you care about. GDPR's definition of "processing" is broad enough that transmitting personal data to a third-party AI service almost certainly qualifies, which means you need a legal basis for doing it and the third party needs to be a data processor under an appropriate agreement. Your AI vendor's standard terms are probably not that agreement. If you're in healthcare in the United States, HIPAA is similarly unambiguous: protected health information cannot leave your covered environment without appropriate safeguards that a commercial chat API does not provide by default.

The workaround is simple and almost never actually painful. Anonymize before you paste. Replace real names with User A and User B. Replace real email addresses with user@example.com. Replace real dollar amounts with generic values if the specific figures are confidential. Replace real IP addresses with 192.0.2.x (the RFC 5737 documentation range). The code problem you're trying to solve with AI doesn't require the real data. It requires the structure of the data. Preserve the shape; strip the substance.

The slightly harder version of this problem is logs. Stack traces and application logs often contain more PII than you realize — user IDs, query parameters that include names or emails, session tokens that can be reversed to identify individuals. Before you paste a log snippet for debugging help, scan it. Ten seconds of scanning is worth more than the explanation you'll have to give your DPO later.

Proprietary Algorithms and Trade Secrets

This one is trickier because the line is fuzzier. "My database connection logic" is probably not a trade secret. The pricing algorithm your company has spent three years building to outcompete the market might be. The model your risk team runs to flag fraud — the one that took 18 months to calibrate — probably is.

The practical test: would you describe this code in a patent application or keep it confidential precisely because disclosure would benefit a competitor? If the latter, it doesn't go in the prompt.

The frustrating part is that the most valuable code is often the most complex, and complex code is exactly what you want AI help with. The answer isn't to avoid AI for hard problems. It's to decompose. Paste the structural problem — "I have a function that takes a weighted graph and returns optimal paths under dynamic cost constraints, and I need to refactor it to be testable" — without pasting the actual weight calculation logic that is your company's secret sauce. The AI can help you with the architecture. You keep the specifics.

Anything Under NDA

If you signed a nondisclosure agreement about it, it doesn't go in the prompt without legal sign-off. This is less about whether the AI vendor will actually misuse the information and more about whether you are in breach of a contract the moment the transmission happens. Some NDAs are broad enough that disclosure to any third party, regardless of intent or subsequent handling, constitutes a violation. If you don't know whether yours is that broad, assume it is and ask your legal team. This takes one email and less than 24 hours in most organizations. It is dramatically less painful than the alternative.

Your Colleagues' Code Without Their Knowledge

This one doesn't come up in legal discussions often, but it should come up in team ethics discussions. When you paste someone else's code — a function written by the senior engineer who wrote most of your payment service, say — into an external AI service, you are making a decision about that code that they didn't consent to. At organizations where there are reasonable expectations about code confidentiality, this is a social and potentially a policy violation even if it doesn't trip any of the legal wires above.

This is a team norm question more than a rule. But it's worth establishing explicitly. We'll return to team dynamics in the next chapter.

The Licensing Problem Is Real, And Also Not a Reason to Panic

Separately from what you send in, there's the question of what you get out.

GitHub Copilot was the subject of a class action lawsuit — Doe v. GitHub, filed in November 2022 — arguing that code suggestions derived from training data were reproducing copyrighted code without proper attribution or license compliance. The lawsuit has moved slowly through the courts in the way these things do. But it raised questions that are genuinely unresolved: when an AI model trained on GPL-licensed code produces output that resembles GPL-licensed code, is the output subject to the GPL?

The honest answer as of 2026 is: nobody knows for certain, and courts haven't definitively resolved it. What we do know is a few things worth keeping in mind.

First, AI models can reproduce training data verbatim, especially for common patterns. GitHub's duplication detection filter — available in Copilot's settings — is designed to flag suggestions that closely match code found in training data. Turn it on. It doesn't eliminate the risk, but it surfaces the obvious cases.

Second, the risk is heavily context-dependent. Generic utility functions — sorting algorithms, string manipulation, simple HTTP handlers — are implemented the same way in thousands of codebases. The probability that any given implementation of binary search is meaningfully "derived" from a specific copyrighted version is low in ways that are hard to articulate legally but are probably correct. Unique, novel algorithms with distinctive implementation choices are a different matter.

Third, if you are generating code for a context where licensing matters critically — open source libraries you're releasing, code for heavily regulated industries, anything where your legal team cares about provenance — you want explicit provenance checking as part of your review process. This means: turn on whatever duplication detection the tool offers, treat any unfamiliar long code block as something to search for verbatim before shipping, and document that you did this in cases where you need the paper trail.

Simon Willison has argued that the licensing question is somewhat overblown for most practical use cases, and probably correctly. The realistic exposure for most developers using AI assistance to write application code that will never be open-sourced is low. But "low" is not "zero," and knowing the actual risk helps you make a proportionate response — which is: use the duplication filters, do a quick search on any substantial block of generated code that looks like it might have a source, and don't pretend the question doesn't exist.

Enterprise vs. Personal Tools for Sensitive Work

If you work for an organization that handles genuinely sensitive data, the single highest-leverage thing you can do is make sure you're using enterprise or API tier tools under a proper data handling agreement, not personal or free tier tools.

The difference matters. Enterprise-tier agreements with Anthropic, GitHub, and Cursor include data processing addendums that you can actually show to your legal team. Personal tiers generally don't. This isn't a gotcha — it's a structural reality of how B2C and B2B products work. Consumer products have privacy policies. Enterprise products have contracts.

If your organization has standardized on a specific tool with an enterprise agreement, use that tool for sensitive work. If you're using a personal subscription because you find it more convenient, you are making a data handling decision that your employer probably hasn't blessed and may not know about. That's a conversation to have proactively, not after an incident.

The Pre-Prompt Checklist

Before you paste code, data, or context into a prompt on any sensitive task, run through this. It takes thirty seconds. Do it until it's automatic.

1. Does this contain credentials or secrets? Keys, tokens, passwords, connection strings. If yes, remove them before pasting. All of them. Check twice.

2. Does this contain real personal data? Names, emails, phone numbers, IDs, IP addresses tied to users, health or financial records. If yes, anonymize before pasting. Replace with generic placeholders. The AI doesn't need the real values.

3. Is this covered by an NDA or contains client-confidential material? If you're not sure, assume yes and decompose the problem to remove the sensitive specifics before you paste.

4. Is this a proprietary algorithm, pricing model, or anything a competitor would pay to see? If yes, abstract the problem. Describe the pattern without pasting the implementation. Ask for architectural guidance on a sanitized version.

5. Am I on an enterprise-tier tool with a data handling agreement? If you're using a personal subscription, is this the right tool for this task? Could you wait until you're on a company-licensed tool?

6. Did someone else write part of this code? If it's a colleague's work, are you operating within your team's norms about what goes to external AI services?

That's it. Six questions. If you clear all six, paste away. If you stumble on any of them, spend sixty seconds cleaning up the input before you continue. The bottleneck in your AI-assisted workflow is almost never the thirty seconds you spent on the checklist.

Why This Chapter Exists Next to the Others

There's a version of this conversation that turns into security theater — an 80-slide deck from the InfoSec team, a list of forbidden tools, a monthly sign-off form. That's not useful, and it doesn't actually improve behavior under pressure. Engineers in a hurry make different decisions than engineers sitting through a training session, and most of the data leaks happen when people are in a hurry.

The alternative is to have the right instincts load automatically. The Paste Rule is short enough to remember. The checklist is fast enough to run. The underlying principle — your prompt is a data transfer, not a private thought — is simple enough to stay true under deadline pressure once it's actually internalized.

The tools you're using are genuinely useful. The security and IP considerations around them are real but not unmanageable. The engineers who get into trouble aren't stupid. They're just running on autopilot in moments that required a beat of conscious thought. This chapter is that beat, written down, so you can read it once and carry it as a reflex.

Paste thoughtfully. Verify the output. Keep the sensitive stuff out of the box.

Everything else is just good engineering hygiene — which, if you've been doing this long enough, you already know isn't glamorous. It's just the thing you do consistently so that nothing bad happens, which means when nothing bad happens, nobody notices. That's the goal. That's the whole goal.

One more thing worth saying plainly: the tools themselves are not the enemy here. Anthropic, GitHub, and the Cursor team have all made meaningful commitments to responsible data handling, and their enterprise documentation is worth reading rather than ignoring. The risk isn't that these companies are malicious. The risk is that the interface hides complexity, that default settings aren't always the right settings, and that the implicit model most people carry around — "it's just a chat, it disappears" — is not an accurate model of what's happening.

Update the model. The checklist follows from the model. The behavior follows from the checklist.

That's the whole argument. Three sentences. The rest of this chapter was just making sure you believe them.

CHAPTER XIII

The Honest Case For and Against Agent Loops

The demos are always spectacular.

An AI agent gets handed a GitHub issue number, locates the relevant repo, reads the code, identifies the bug, writes a fix, runs the full test suite, watches four tests fail, figures out why, adjusts the fix, runs tests again, everything green, opens a pull request with a coherent description and a linked issue number. Four minutes. No human at the keyboard. Someone plays this recording at a developer conference, or drops the link on X, and the reaction is sincere. You can feel the audience quietly recalculating things.

Then you try it on your codebase.

You give the agent a ticket that would take you two hours. It confidently reads the wrong files first, writes a fix that makes the described symptom disappear by special-casing it rather than addressing the underlying cause, writes tests that pass only because it mocked the broken behavior it was supposed to test, and opens a PR with the commit message "fix: resolve issue." You spend an hour understanding what it actually did. You undo most of it. You write the real fix yourself, which takes forty minutes. Net result: sixty minutes lost, plus the low-grade unease of wondering whether you caught everything the agent quietly changed.

Both scenarios are real. Not cherry-picked. Not the credulous version versus the cynical version—those are two honest descriptions of the same tools operating under different conditions. The gap between them is not hype versus reality. It's conditions versus conditions. The whole question, the only useful question, is which conditions produce which result.

What an Agent Loop Actually Is

Before discussing when agents work and when they don't, it's worth being precise about the mechanism. "AI agent" has become one of those terms that means everything and nothing, depending on who's using it and what they're trying to sell. Get specific.

An agent loop is a model with access to tools—bash, file read/write, web search, API calls—running in a cycle where it decides which tool to call, observes the result, and continues until it determines the task is complete. That's the whole architecture. The sophistication isn't in the loop structure; it's in the quality of the model's decisions at each step and the design of the scaffolding around it.

Claude Code implements this directly. In agentic mode, it can read files, run commands in the terminal, edit code, execute your test suite, and observe the output—looping until it reaches a stopping point or you interrupt it. The tool calls are sequential: the model says it wants to run a bash command, the framework executes it, the output comes back, the model decides what to do next. Each decision is made by the language

model. Each tool call is real. The model can read a file and then delete it, and both of those things actually happen on your filesystem.

Cursor's Agent mode in Composer works similarly, with multi-file awareness and optional terminal access. Their Background Agents feature, which reached general availability in 2025, extends this further: you dispatch tasks to agents running in isolated cloud containers, where they have full access to a copy of your codebase but are sandboxed away from your local environment and any production systems. The isolation is deliberate. It's not a performance optimization. It's damage control.

OpenAI's Agents SDK, released in early 2025, adds a routing abstraction on top: you define specialized agents with specific tools and responsibilities, and an orchestration layer manages handoffs between them. One agent reads documentation, another writes code, a third runs validation. Elegant on paper. In practice, handoff points introduce new failure modes where context can be partially lost or misinterpreted between agents, and the coordination overhead is real.

Simon Willison, who has been building production tools on language models since before most companies had an AI strategy, and documenting the results with unusual frankness at simonwillison.net, put the core tension plainly: the thing that makes agents powerful is that they take real actions in the world. Reading a file has no consequences. Running a migration script does. Calling a write API does. The loop has no built-in instinct to calibrate its caution to the stakes of the action it's about to take. You have to engineer that caution in from outside.

There's a reason Anthropic's documentation on Claude Code's agentic capabilities dedicates considerable space to interruption, oversight, and recognizing when to pause and ask. The model itself will tell you, if you ask it directly, that unsupervised long-horizon tasks are where it's most likely to make consequential mistakes. That's not a liability disclaimer. It's accurate self-assessment.

What Agents Can Actually Do in 2026

There is a real category of work where agents are faster and more reliable than a human doing the same thing manually. Worth describing carefully, because it gets buried under the backlash.

Mechanical, rule-based transformations across large codebases

This is the strongest category by a wide margin. Renaming a function and updating every call site. Converting a codebase from CommonJS require to ES module imports. Adding TypeScript annotations to an untyped JavaScript project. Migrating from one version of an internal API to another where the mapping is fully documented. Replacing deprecated patterns with their modern equivalents across hundreds of files.

These tasks share a structure: a clear rule, uniform application, machine-verifiable output, and failure modes that are local rather than systemic. If the agent makes an error in file forty-seven of ninety, it shows up as a failing test or a type error—not a subtle architectural problem that festers for weeks before surfacing. The feedback loop is fast and external.

For this category, agents are faster than a human by an order of magnitude. Not slightly faster. An order of magnitude. Across a hundred-thousand-line codebase, a human doing some of these tasks manually needs days. An agent does them in minutes, and the review takes less time than the migration would have.

Scaffolding and boilerplate generation

Generate a new service: directory structure, config files, basic request handlers, a Dockerfile, a test skeleton, the CI workflow definition. Stub out an SDK client for a third-party API from its documentation. Generate CRUD endpoints for a new data model. The value here is producing something complete to edit rather than building from nothing. Having a full scaffold that needs 30% adjustment is meaningfully faster than starting with a blank file, even accounting for the review time.

Targeted search and synthesis through unfamiliar code

Give an agent a large codebase and a specific question: "Where is authentication middleware applied and are there routes that skip it?" It traces through imports and function calls faster than you navigate manually, and gives you an accurate answer in under a minute. Anthropic's documentation on Claude Code describes this use case as exploration—agentic file access used to understand rather than change. It works well, and it's underused.

Self-correcting loops with fast external validators

The single most important predictor of agent reliability is whether "done" can be expressed as a command that returns a success code. Tests pass. The build compiles. The linter has no errors. When the agent has a clear, mechanical, external signal that it's finished, it can self-correct through iteration. It runs the tests, sees failures, revises, runs again.

SWE-bench Verified—the benchmark specifically testing this pattern, using verified test suites as exit conditions against real GitHub issues—was showing top agent scores around 50–55% by late 2025 for the best scaffolded configurations of frontier models. That means roughly one in two real-world software engineering tasks, given a clear issue description and a working test suite, resolved correctly without human intervention. Under 20% was the figure a year prior. The trajectory is genuine.

But 50% also means failure half the time. Hold that number.

Where They Fail Embarrassingly

The failure modes are not exotic edge cases. They're consistent, they're predictable once you've seen them, and they cluster around a handful of patterns.

Compounding errors

The most dangerous property of an autonomous loop and the hardest to counter. A human developer making a wrong assumption usually notices within a few steps—cognitive friction, something that doesn't compile, logic that visibly doesn't cohere. The developer backtracks. Agents don't experience friction. They register no

cognitive state at all.

An agent that makes a subtly wrong assumption in step three of a fifteen-step task builds coherently on that assumption through steps four, five, six, and beyond. By step fifteen, the output is internally consistent, confidently presented, and wrong in a way that can take you an hour to untangle—precisely because the wrongness has the shape of correctness. It hangs together. It's just hanging together around a mistaken premise. Willison's writing on this is direct: the failure mode he's most wary of is not the obvious crash but the plausible-looking result that embeds a quiet error you won't catch until it matters in production.

Prompt injection

If an agent accesses external content—URLs it browses, files it reads from external sources, third-party API responses it processes—it can be redirected by adversarial instructions embedded in that content. A README that contains instructions formatted to look like system directives. A comment in a public library's source. A response from an external API that includes text phrased as commands. Willison has documented this attack class at length and argued it remains an unsolved problem for any agent that operates on untrusted input. The model can be instructed to ignore directives from within tool call results, but that instruction is not reliably respected under adversarial conditions.

For agents running against an internal codebase where all input sources are controlled, the risk is low. For agents that browse documentation, process user-submitted content, or act on external API responses, the attack surface is live and the mitigations are incomplete.

Hallucinated tool behavior

Agents sometimes call functions that don't exist in the codebase, import libraries that aren't installed, or reference environment variables that haven't been set. The model is pattern-matching against codebases in training data, not querying your actual dependency tree. If your codebase uses a pinned version of a library with different API surface than the current release, or a niche framework with limited public presence, the agent will confidently invoke methods that don't exist in your environment. These calls look plausible in a diff. You catch them when the code runs.

Silent scope creep

Ask an agent to fix a specific bug and it will sometimes "improve" adjacent code it wasn't asked to touch—refactoring a function it passed through, renaming a variable it found unclear, adjusting formatting in a file it opened to read. None of these changes are in the task description. All of them appear in the diff. Each one requires you to decide whether it was correct, whether it was wanted, whether it introduces any interaction effects. The agent was trying to be helpful. The result is noise in your review and the possibility of unintended changes you don't fully scrutinize.

Non-deterministic judgment calls

Should this function be split into two? Should this error be recoverable or fatal? Is this a bug or an intentional undocumented behavior? Agents make these calls. They make them without flagging that a decision was

made. You discover a choice embedded in the code that you would have made differently—maybe correctly, maybe not—and you have to decide whether to leave it or reverse it. The judgment calls that are outright wrong cost you time to fix. The judgment calls that are merely different from yours cost you time to evaluate. Both costs are real.

Circular test generation

An agent asked to write tests for a function it also implemented will often write tests that verify the behavior it implemented—including any bugs in that implementation. If the agent misunderstood the requirements, the tests will verify the misunderstood behavior. Everything passes. Coverage metrics improve. The function is wrong. The test suite confirms it. Chapter 7 covers this failure mode in depth; the relevant point here is that agent-generated tests require a specifically different skepticism than human-written tests, focused on whether the tests are verifying the right behavior at all.

Cost at scale

Long agent runs against frontier models are not cheap. A task that consumes forty minutes of active inference at Claude Opus or GPT-4o class pricing is a real line item, especially if you're running several per day. The economics favor short, precise, high-supervision runs over extended exploratory ones. Which is also better engineering practice for the failure mode reasons above. Two things aligning in the same direction is usually a reliable signal.

The Conditions That Separate Reasonable Bet From Debugging Time Bomb

Model quality matters less than conditions. This is the counterintuitive lesson from consistent production use, and it holds up against the benchmarks. A weaker model in well-controlled conditions outperforms a frontier model given an ambiguous task and unrestricted scope.

The conditions that predict a good outcome:

The exit condition is mechanical and external. If "done" can be expressed as a command with an exit code—tests pass, build succeeds, linter clean—the agent can target it and self-correct toward it. If "done" means "looks right to me," you are the exit condition, and you need to be present to apply it.

The blast radius is bounded. Map out the worst sequence of actions the agent could plausibly take. If that sequence touches production data, deployed infrastructure, shared state that can't be rolled back, or credentials with broad write access, reduce the scope before you start. Feature branch in an isolated container: appropriate. Agent with database write access and deployment credentials: not appropriate. This seems obvious until you've watched someone construct exactly this situation and then spend three hours with git reflog.

The task has an unambiguous stopping point. "Migrate all usages of the old authentication helper to the new one and verify the existing tests pass" is a stopping point you can evaluate against. "Improve the auth

module" is not a stopping point. Vague objectives produce agents that either over-run—changing things you didn't ask for—or under-run, declaring success arbitrarily when the model's confidence about completion exceeds its actual confidence about correctness.

The domain is common in the model's training data. Agents are substantially more reliable on TypeScript, Python, and mainstream frameworks than on internal DSLs, niche libraries with limited public presence, or proprietary systems that don't appear in public code. If your codebase makes heavy use of a custom ORM, a homegrown configuration system, or an unusual framework, the agent is inferring by analogy. It will sound exactly as confident as it does on common stacks. It will be wrong more often.

You can verify the output faster than you could have produced it. If the agent gives you 600 changed lines across eighteen files and you cannot determine in fifteen minutes whether those changes are correct, the economic argument for using it doesn't hold. Good agent task selection produces outputs you can check in a fraction of the time they would have taken to write. If verification approaches the cost of original production, you have gained the possibility of a wrong answer and not much else.

What Production Use Actually Looks Like

Not the demos. Not the benchmark leaderboards. The day-to-day.

The teams using agent loops well in 2026 have converged, mostly independently, on a pattern that looks nothing like the four-minute end-to-end recording. The demos show full autonomy. The production pattern looks more like supervised scaffolding with deliberate human checkpoints.

Short, watched runs. Not "here's a ticket, come back when it's done." More like: hand the agent a narrowly-defined subtask, watch the first three or four tool calls to confirm it's heading the right direction, and interrupt early if it's not. An agent that's been running unsupervised for four minutes has made dozens of decisions. Catching one wrong decision at minute two costs almost nothing. Catching it at minute forty is a different situation entirely.

Agents on the mechanical work, developers on the judgment calls. The agent does the migration, the stub generation, the test skeleton, the repetitive wiring. The developer handles the architectural decisions, the edge-case logic, the integration with constraints that aren't visible in the code itself. This isn't "AI does 80%, developer does 20%." It's closer to: AI does 80% of the keystrokes, developer does 100% of the thinking that matters.

Negative constraints in the prompt, consistently. "Don't modify files outside the src/auth directory" is more valuable than a detailed positive description of what to do. "If you encounter a decision that isn't clearly specified, stop and surface it rather than making a guess" will save more debugging time than almost any positive instruction you can write. The best production agent prompts read partly like task descriptions and partly like security policies.

A second pass with deliberate skepticism. Treat agent output as a pull request from an unknown contributor. Read it looking specifically for: functions or methods that don't exist in the codebase, test mocking patterns that make tests circular, happy-path logic with absent error handling, and behavior changes that weren't in the task description. The techniques in chapter 10 apply directly. The goal is targeted review for the specific failure modes agents produce—not paranoid line-by-line reconstruction, and not a quick scroll and merge.

Where That Leaves You

The dismissive position—agents are toys, hallucinations make them useless, real engineers write real code—is coherent but expensive. It declines a genuine efficiency gain because of a real but manageable set of risks, and it stays expensive as the capability trajectory continues. The maximalist position—agents handle the toil, I just direct—is currently ahead of what the tools reliably deliver and produces debugging overhead that eats the efficiency gain it promised.

The calibrated position is situational. For this specific task, with these specific conditions, today, in this codebase: do the conditions apply? Does the exit condition exist? Is the blast radius bounded? Is the domain common? Can you verify the output efficiently? If yes, run the agent. If not, write the code. The decision is not a philosophical stance on AI. It's an engineering judgment about conditions.

The engineers getting the most out of agent loops right now are not the ones who've handed off their judgment. They're the ones who've handed off the work that didn't require judgment in the first place—and kept their judgment exactly where it was before, applied to the things that need it.

That shift is smaller than the demos suggest and larger than the skeptics allow. The tedious migration that would have taken three days now takes an afternoon. The test coverage on the legacy module you couldn't justify time for now exists. The scaffold for the new service is there when you need it. The category of work you were putting off because it was pure labor is no longer the bottleneck.

Not nothing. Worth understanding clearly and using accordingly.

CHAPTER XIV

Where This Is Going by 2027

The honest answer to "where is this going?" is that nobody knows. Not Dario Amodei, not Sam Altman, not the researchers at Google DeepMind who have spent their entire careers thinking about little else. The people building these models are genuinely surprised by what emerges from them at scale. That should make you appropriately skeptical of anyone — including me — who opens a chapter like this with numbered predictions and a tidy timeline.

But "nobody knows" is a cop-out when you're trying to make practical decisions about your career and your workflow. There's a meaningful difference between knowing nothing and being unable to extrapolate from a direction that is clearly in motion. Meteorologists don't know exactly where a hurricane will make landfall. They can tell you with confidence it won't hit Denver. That kind of bounded thinking is what this chapter is going for.

So here's the frame: what trajectories are already in motion? What do two years of capability jumps look like as a trend line instead of a point? And given those trajectories, what does a working developer's skill set need to look like by 2027 to be on the right side of the shift — not winning a prediction contest, but positioned to stay effective when the tools are materially different from what they are today?

The Trend Lines That Are Actually Legible

Context windows are still growing, and the downstream effects haven't fully landed.

In 2022, GPT-4 shipped with an 8K token context window. By early 2025, Claude's 200K context was almost mundane — Anthropic had made it the standard rather than the premium offering — and Gemini 1.5 Pro was handling 1M tokens in preview. That's a 125x increase in roughly three years. The raw capability gain is obvious. What's less obvious, and more interesting, is that the effects on how developers work are still catching up to the capability.

When your context window was 8K, you had to summarize. You had to be selective. You had to understand your codebase well enough to know which pieces were relevant and worth feeding to the model. That constraint enforced a kind of rigor. You couldn't be lazy about it. Now you can drop 150,000 tokens of context into a prompt and the model will read all of it. Which means the bottleneck has shifted from "can the model see enough of the codebase?" to "can you describe what you actually want precisely enough that the model can prioritize correctly inside that enormous context?"

That's a different skill. A harder one, arguably. Expect context windows to keep expanding toward whatever ceiling exists — or possibly no ceiling in practical terms. And expect the skill of writing precise intent to matter more, not less, as the context capacity grows. More context, paradoxically, demands better

specification.

Benchmark scores on coding tasks are climbing fast. The benchmarks are also narrow.

SWE-bench Verified is currently the closest thing to a credible independent measure of whether an AI can do the kind of work software engineers actually do. It takes real GitHub issues from open-source repositories and asks: can a model generate a patch that resolves the issue and passes the existing tests? It's an imperfect proxy — all benchmarks are — but it's a real-world proxy, not a constructed toy problem.

In late 2024, Claude 3.5 Sonnet was scoring around 49% on SWE-bench Verified. OpenAI's o3 hit 71.7% in December 2024. Those numbers will be higher by the time you read this.

But notice what SWE-bench actually tests. The issues are typically self-contained bug fixes in well-maintained open-source Python libraries: Astropy, Flask, Sympy, Django. They're tractable. They have clear success criteria built in — the test suite. They don't require understanding a three-year-old architectural decision made under deadline pressure by someone who left the company eighteen months ago and didn't document their reasoning. They don't require negotiating with a product manager who keeps changing the requirements. They don't require weighing the performance cost of the correct fix against the timeline cost of the complete fix.

Still. 71.7% on real code, real issues, real tests. That's not nothing, and it wasn't 71.7% twelve months prior.

By 2027, those scores will be higher. Whether the ceiling on SWE-bench-style tasks is 85% or something close to 95% is hard to say. What matters more is that the class of problems models can handle autonomously will expand — gradually, unevenly, with surprises in specific narrow domains and disappointments in others that currently seem approachable.

Agentic loops are getting more reliable. Slowly, and with important asterisks.

The demo of an AI agent writing a complete feature end-to-end, committing it, running the tests, fixing the failures, and opening a PR has existed since at least 2023. The demo is always impressive. Production reliability for anything nontrivial is still not there.

Anthropic released Claude's computer use capability in late 2024, enabling models to interact with a full desktop environment — click buttons, fill forms, read the screen, respond to what it sees. OpenAI has been investing heavily in agentic infrastructure. Cursor's background agents, running autonomously while you're doing other work, are migrating from experimental feature to mainstream product offering. The trajectory is clear.

The timeline is not, and the failure modes are not random. Simon Willison has been documenting them methodically on his blog since 2023, and the pattern is consistent: prompt injection vulnerabilities that hijack agent behavior mid-task, tools that silently return wrong values without error, context that gets subtly corrupted across a long agent loop, and the compounding problem that an agent with the power to execute actions also has the power to execute the wrong actions at scale. One bad hallucination in a long-running agent loop is categorically worse than a bad hallucination in a single completion, because the agent has acted

on it — potentially many times — before you notice anything is wrong.

By 2027, agents will handle a broader range of well-scoped tasks reliably. The qualifying phrase is "well-scoped." The ability to scope tasks for agents — recognizing when a problem is structured enough to hand off, and being precise about what success looks like — is one of the most underrated skills in the current AI toolkit. It will become more valuable as the agents improve, not less, because it becomes the rate-limiting step.

What Gets Easier

Specifically. Not "everything will be easier." Specifically.

Boilerplate dies a real death. The mechanical work of scaffolding a new service — REST or gRPC endpoints, auth middleware, logging configuration, database migrations, test harness setup — is already most of the way to automated for standard stacks. By 2027, for any widely-used combination of language, framework, and infrastructure, the boilerplate will be a prompt and a review pass. Not judgment-free. The review still matters. But the part where you're typing things you've typed fifty times before: largely gone.

GitHub's 2022 research showed developers completing certain tasks 55% faster with Copilot. Stack Overflow's 2024 survey data put 62% of developers reporting meaningful time savings specifically on routine implementation. The 2027 version of this is not faster boilerplate. It's boilerplate you don't personally produce at all, reviewed for correctness rather than authored from scratch.

Migration assistance improves meaningfully. Upgrading from Django 4.2 to 5.1. Moving a codebase from CommonJS to ESM. Migrating off a deprecated AWS SDK version across forty service files. These are tasks where AI tools in 2025 are useful enough to accelerate the work but not reliable enough to hand off entirely — they lose the thread across large codebases, miss a dependency chain, fail to catch an edge case introduced by the version change. Both of those failure modes are being actively addressed by longer context and improved long-horizon consistency. By 2027, a model-assisted migration on a medium-sized codebase that currently takes a week of careful human work will plausibly take a day of directed AI work plus review. Not zero. A day.

Test coverage for standard patterns stops being a negotiation. The argument "we don't have time to write tests" gets substantially harder to make when writing tests is a prompt and a thirty-minute review pass rather than an afternoon. For pure functions, well-defined transformations, and clearly-scoped units of behavior, AI-generated test suites by 2027 will be structurally sound and edge-case-aware for common patterns. The harder cases — behavior that depends on external service state, timing-sensitive logic, distributed consistency conditions — will still require humans who understand what the tests are supposed to prove. But the mass of the testing pyramid, the unit and integration layer for standard code, becomes a much faster operation.

Documentation generation that doesn't embarrass you. Current AI-generated documentation is often technically accurate and stylistically inert — repetitive, over-formal, structured as though the model has read thousands of API docs but hasn't thought about what a developer actually needs at 11pm when they're stuck.

This is improving faster than most other capabilities, probably because writing quality is easier to evaluate at scale than code correctness. By 2027, inline documentation and README generation for clean, well-structured code will be close to ship-ready without heavy editing. For messy code — code with tribal knowledge and implicit assumptions baked into its structure — human correction will remain necessary, because a model can only document what it can infer, and messy code is messy specifically because some of its meaning isn't on the surface.

What Stays Hard

This section is longer than the previous one. The optimism already saturates every other source you'll read about AI in 2026. The constraints deserve more air.

Architecture. The decision about whether to introduce a message queue at a given layer, whether to normalize or denormalize a particular schema, whether microservices or a modular monolith is the right shape for this specific team — none of this gets easier to delegate. Not because AI can't reason about tradeoffs. It can. Ask any current model about the tradeoffs between event-driven and request-response architecture and you'll get a competent taxonomy of considerations. The problem is that those tradeoffs are only partially about technical properties. They're about your team's operational maturity, your on-call load, your deployment pipeline, your specific performance envelope, your relationship with the three other services you can't change. An AI can surface what the tradeoffs are in the abstract. It cannot weigh them for your specific context unless you can articulate that context — at which point you're most of the way to the decision anyway.

Architecture stays human not because it's harder cognitively than writing code, but because it's contextually denser. The inputs are partially tacit. The success criteria span eighteen months. You won't know whether the architectural call was right until you're living with the consequences of it.

Security, genuinely. AI tools catch common vulnerability patterns. They know the OWASP Top Ten by heart. They can flag a SQL injection hole in sample code and explain why environment variables aren't a safe place for secrets. That's the approachable layer of security. The hard layer is the logic that makes your application specific: your permission model, your trust boundaries, the conditions under which your authentication layer can be confused by a sequence of requests that each individually look fine.

A model generating code for a new access control feature doesn't know what your existing access control assumptions are unless you tell it — and even then, it doesn't know what an attacker will try. Willison has made this point precisely and repeatedly: LLMs understand the vocabulary of security. They don't understand the adversarial mindset. Those are different things. The failure mode is not that the AI will generate obviously vulnerable code. The failure mode is that it will generate internally consistent code that passes your tests and has a subtle logical flaw that creates a privilege escalation under a condition your tests don't cover and your reviewer didn't think to check. That gap — between vocabulary and adversarial reasoning — does not close in 18 months.

Novel problem domains. If you're working on a problem with abundant analogies in public training data — web services, standard data pipelines, CRUD applications on common stacks — AI assistance will keep improving. If you're working on something genuinely uncommon — embedded systems for a niche industrial protocol, a distributed database with an unusual consistency model, numerical code where floating-point behavior actually matters — the model's confidence does not reliably track its competence. It delivers plausible-sounding output at the same tone it uses when it actually knows what it's talking about. The tell is not hesitation. The tell is subtle wrongness that requires domain knowledge to detect.

This doesn't make AI useless in uncommon domains. It means the verification burden stays high. You still need deep enough expertise to catch what's wrong. The value proposition shifts from "AI does the reasoning" to "AI saves time on lookup and mechanical expression while I own the correctness."

Ambiguity resolution. "The checkout flow is slow." Slow under what load? Slow for which users? Slow compared to what baseline, and who defined that baseline? Slow because of a database query, a downstream API call, a rendering bottleneck, or a perceived slowness caused by something entirely outside the codebase? Getting from a vague complaint to a testable hypothesis requires clarifying questions, informed prioritization, and knowledge of which parts of the system have historically been problematic. AI can participate in this — it can generate hypotheses. It cannot decide which to test first without context that exists only in your head, and the conversation with the stakeholder who noticed the slowness still needs a human who can read the room.

Skills on the Right Side of the Line

If you're trying to calibrate where to invest over the next 18 months, here's the clearest framing I can offer.

Skills that appreciate in value:

Systems thinking — the ability to hold a whole system in your head, including the data flows, the failure modes, the dependencies, the performance characteristics — becomes more valuable as AI handles more implementation. You become the person who knows what should be built, which is worth more than the person who builds what they're told.

Precise specification. The ability to write a requirement that is genuinely unambiguous, testable, and complete is already a scarce skill on most engineering teams. As AI assistants take over implementation, the quality of what they produce is increasingly limited by the quality of the specification they receive. Writing a good spec for an AI is not trivially different from writing a good spec for a junior engineer. Both require thinking through edge cases, defining success criteria explicitly, identifying constraints you'd otherwise leave implicit. This skill was valuable before AI. It's more valuable now. It will be more valuable still.

Fast, accurate code reading. If you're generating more code than you write, you need to read faster and more carefully. Code review — real code review, the kind that catches logic errors and not just style violations — is a skill that can be practiced and sharpened deliberately. It is worth sharpening now.

Domain expertise. Whatever you know about medical billing, financial clearing, logistics optimization, game networking, or any other specific domain — it compounds. The AI doesn't have it. You do. The deeper your domain knowledge, the better you can specify what the AI should build and the more reliably you can evaluate whether what it built is actually correct.

Skills under real pressure:

Rote implementation. If a task is well-defined enough to describe clearly, it's well-defined enough to hand to an AI. The developer who differentiates primarily on speed of mechanical implementation — fast at writing CRUD endpoints, fast at plumbing a new config option through a layered system — is competing directly with a tool that improves every six months on that exact capability. This is not a reason to panic. It's a reason to look clearly at the shape of your current work and think about what in it is mechanical versus what requires genuine judgment.

Lookup expertise. Knowing the exact argument order for a particular Linux syscall, or the correct syntax for a specific pandas operation, was a skill with value when the reference material wasn't already in your editor. It's less valuable today and will be less valuable still. This is genuinely fine. Treat this knowledge as reference material, not a differentiator.

Your Practical Posture for Moving Targets

None of this matters unless it translates into habits.

Stay close to the output. The mistake that will bite developers in 2026 and 2027 is progressive trust escalation — each time the AI gets something right, you review the next output a little less carefully, until you're approving without reading. Set a deliberate floor on your own review and hold it even when that feels slower than necessary. Every PR still gets a real look. Not because you distrust the AI as a matter of principle, but because the cost of a mistake is yours, not the model's.

Use agents for defined work, not undefined work. Before handing a task to an agent loop, ask: could I write a clear spec for this? If yes, the agent can probably handle it. If no, you're going to spend several hours unwinding what the agent decided to do with the ambiguity. The clearer your specification, the more useful the automation — and the more confident you can be that what comes back is what you meant to build.

Keep one environment you fully understand. As AI generates more of your infrastructure configuration, migration scripts, and scaffolding, it becomes easy to end up running code in production that you couldn't explain from memory if asked. Deliberately maintain at least one context — a service, a subsystem, a test harness — that you built yourself and understand end-to-end without AI filling the gaps. Not as nostalgia. As a calibration tool. It keeps you honest about the difference between what you know and what you're trusting.

Read the changelog. This sounds trivial. It's not. Claude Code, Cursor, and Copilot are shipping meaningful capability changes on timescales of weeks. The workflow you optimized three months ago is already partially outdated. Willison publishes regular roundups of what's changed in the major models; his blog is one of the

few sources that is both technically precise and genuinely current. Anthropic's model release notes are dense and worth reading. The tools you're using today are not the tools you'll be using in twelve months. Treating them as static will cost you.

Don't optimize for today's tools specifically. The prompt patterns in this book, the context management tricks, the model-selection heuristics — these have a shelf life measured in months, not years. What has a longer shelf life is the meta-skill: how do I figure out what this tool can and can't do? How do I verify its output efficiently? How do I decompose a problem into pieces this system can handle reliably? That transfers across tool generations. The specific techniques don't.

Here's what nobody in the AI discourse wants to say plainly: the people with the most confident predictions about AI capabilities have been consistently wrong in both directions. The pessimists in 2022 who called Copilot an expensive autocomplete missed how it would change the cost of routine implementation across entire engineering organizations. The optimists in 2023 who said autonomous agents would be handling entire sprint backlogs by end of year missed how brittle those systems were in practice, and still are.

The right posture is not confident prediction. It's a short update cycle. Form a view, act on it, test it against reality in six months, revise. The developers who handle 2027 well are not the ones who called the future most accurately today. They're the ones who stay close enough to the actual tools — actual benchmarks, actual production behavior, actual failure modes under real conditions — to update their view when something meaningful changes.

Which is, when you step back, just good engineering. You don't architect a system once and stop paying attention to how it behaves. You instrument it, observe it, and adjust.

Same thing here. The forecast is that it keeps moving. Your job is to keep reading the instruments and adjusting heading — not once, at the start, but continuously, for as long as the tools are still changing.

Which will be a while yet.