A FIELD GUIDE

# The Agentic Crew

*Engineering in the age of AI agents*

Rasmus Bornhøft Schlünsen

March 2026

rev 2

# Foreword

It was a Tuesday evening, sometime last year. My kids were asleep, and I was staring at a migration script I'd been dreading all week — the kind of tedious, table-by-table reshuffling that eats an entire day if you're careful, and destroys production if you're not. On a whim, I described the problem to an agent. Schema here, constraints there, watch out for this foreign key. Then I hit enter and went to make tea.

When I came back, the script was done. Not a rough draft. *Done.* Correct edge cases, rollback logic, comments I would have written myself. I sat there for a long time, tea going cold, feeling two things at once: genuine awe — and a quiet, creeping dread.

Because that migration? That was *my* thing. I was the person on the team who could hold the whole schema in my head, who knew which joins were cursed, who could write the careful SQL by hand. Fifteen years of muscle memory, and an LLM had just matched it in four minutes while I boiled water.

I want to be honest with you: I didn't sleep well that night. I lay in bed running the same loop every engineer I know has run. *What am I for now? What happens to the craft I spent half my life building? Am I training my replacement?*

It took me months — and a lot of building, failing, and rebuilding with these tools — to find the answer. And the answer surprised me. The craft isn't dying. It's *molting.* The outer shell — the keystrokes, the syntax, the boilerplate — that part is falling away. But the animal underneath? The part that knows *what* to build and *why*, that smells a bad abstraction from three files away, that can hold a whole system in mind and feel where it's fragile? That part is more alive than ever.

We're not being replaced. We're being promoted. From typists to thinkers. From writing code to directing it — orchestrating, reviewing, shaping. The skills that made you a good engineer — systems thinking, taste,

judgement, the instinct for simplicity — those are the *whole game* now, not just the background hum.

But nobody gave us a manual for this transition. It's messy and uncomfortable and sometimes humbling. I wrote this book because I'm living through it, and I have a feeling you are too. These pages are everything I've learned about working *with* the agents instead of against them — or worse, pretending they don't exist.

If you've ever watched an AI write code that looked like yours and felt your stomach drop, this book is for you. Keep reading. It gets better — and stranger — than you think.

*Rasmus Bornhøft Schlünsen*
*March 2026*

# Contents

# 1 Introduction

Last year I watched a junior developer on my team — two years of experience, still nervous in code reviews — ship a complete API endpoint in forty-five minutes. Data model, validation, error handling, tests, documentation. The code was clean. The tests were thorough. The PR passed review on the first try.

It would have taken me an hour to do the same work. And I've been doing this for twenty years.

She didn't type most of it. She described what she needed, pointed an agent at the codebase, and steered it to the finish line. Her skill wasn't in writing the code — it was in knowing what to ask for, recognising when the output was good, and catching the one edge case the agent missed. She was *engineering*. Just not the way I learned to engineer.

I went home that evening and sat with an uncomfortable question: if the gap between twenty years of experience and two years of experience just got a lot narrower, what exactly am I bringing to the table?

The answer, I eventually realized, is everything that isn't typing. But getting to that answer took months, a lot of mistakes, and this book.

## 1.1 The Ground Is Shifting

For twenty years, being a software engineer meant one thing: you open an editor, you write code, you ship it. The tools changed — from Vim to VS Code, from SVN to Git, from bare metal to Kubernetes — but the fundamental loop stayed the same. You, a keyboard, and a problem.

That loop is breaking. And it's breaking fast.

AI agents don't just autocomplete your code. They read your entire codebase, reason about architecture, make changes across dozens of files, run your tests, and iterate on failures — all without you touching the

keyboard. They're not replacing the editor. They're replacing the *workflow*. The engineer who used to spend 80% of their time typing is now spending 80% of their time thinking, reviewing, and steering.

Some engineers are thriving in this shift. They're shipping more, with higher quality, and they'll tell you they're enjoying their work more than they have in years. Others are frustrated, skeptical, or quietly terrified that the craft they spent a decade mastering is evaporating underneath them.

Both reactions are rational. The truth is somewhere in the uncomfortable middle.

## 1.2 Why This Book

Because nobody handed us a playbook.

The tools showed up fast — Copilot, then Claude, then agents that can autonomously run tasks end-to-end — and we're all figuring it out in real time. I looked for the book that would tell me how to actually work with these things. Not the marketing pitch. Not the academic paper. Not the Twitter thread from someone who tried it for twenty minutes. I wanted the honest engineering guide — written by someone who ships production code and has watched agents do brilliant things and catastrophically stupid things in equal measure.

That book didn't exist. So I wrote it.

I wrote it because I was lost too. I was the senior engineer who couldn't figure out why the agent kept rewriting my entire component library when I asked it to fix a button color. I was the guy who burned 500,000 tokens on a task that should have taken ten minutes, because I didn't know how to set boundaries. I made every mistake in this book before I learned to avoid them.

This is the guide I wish someone had given me.

## 1.3 Who This Is For

You're a software engineer. You've shipped real things. You know what a production incident feels like at 2am. You're not afraid of the terminal.

But lately, something feels different. Maybe you've tried AI coding tools and found them impressive but chaotic — like pairing with someone who's incredibly fast but has no concept of scope. Maybe you've watched a developer with two years of experience ship a full feature in an afternoon using agent assistance, and it made you feel something you didn't expect. Maybe you're excited but don't know where to start. Maybe you're skeptical and want someone to convince you with substance, not hype.

This book is for you. It assumes you can code. It assumes you've been around. It meets you where you are.

## 1.4 How to Read This Book

This isn't a reference manual. It's a journey, and it's structured like one.

We start by understanding what's actually changing and why — the shift in how software gets built, not just the tools but the *thinking*. Then we get into what agents actually are, stripped of the marketing language, so you have a mental model that holds up when the tools change next quarter.

From there, we get our hands dirty. You'll learn how agents work in the terminal, how to set up guardrails so they don't wreck your codebase, how Git workflows change when agents are committing code, and why sandboxing isn't optional. We'll dig into testing as the feedback loop that makes agents *reliable* instead of just fast, and conventions as the secret weapon that most people overlook.

Then we go deeper — local versus commercial models, prompt engineering as a real discipline, and multi-agent orchestration. We'll look at war stories: real failures, real lessons, real scar tissue. We'll talk about when *not* to use agents, because knowing when to put down the tool is as important as knowing how to use it. And we'll close with how teams adopt this stuff without imploding.

By the end, you won't just know how to use AI agents. You'll know how to *think* about them — which is the skill that survives when the current generation of tools is obsolete.

## 1.5 What This Book Is Not

Let me save you some time.

This is not a prompt cookbook. You won't find "50 ChatGPT prompts for developers" here. Prompting matters, and we cover it, but copy-pasting prompts without understanding the system underneath is a recipe for expensive frustration.

This is not an AI hype manifesto. I'm not here to tell you that agents will replace all programmers by next Tuesday. They won't. The gap between demo and production is as wide as it's ever been, and someone still has to mind that gap.

This is not a doom narrative either. The "AI is coming for your job" framing is lazy and mostly wrong. What's coming is a fundamental change in *how* the job works, and that's a different conversation entirely.

This is an engineering book. For engineers. Written by someone who spends his days writing code with agents and has the git history to prove it. We're going to be practical, honest, and specific. If that sounds like your kind of thing, turn the page.

# 2 Context

Last month, a bug came in from a customer: payments were silently failing for users with non-ASCII characters in their billing address. A tricky one — the kind that lives in the seam between your frontend validation and your payment gateway's character encoding.

An engineer on my team grabbed the ticket first. He opened his agent and typed: "There's a bug with payments for international users. Can you look into it?" The agent gamely read through the payments module, made some plausible guesses about Unicode handling, and produced a patch that normalised all input to ASCII. It would have stripped every accent, every umlaut, every character outside the English alphabet from every user's billing address. A fix that was technically worse than the bug.

An hour later, a second engineer picked up the same ticket after the first attempt was rejected in review. She pasted in the customer's error log, the failing Sentry trace, the specific payment gateway response code, the relevant section of the gateway's character encoding documentation, and the three files involved in the billing pipeline. Her agent diagnosed the issue in under two minutes: a UTF-8 string was being passed through a function that assumed Latin-1 encoding before hitting the gateway's API. The fix was four lines. It shipped that afternoon.

The model was the same. The agent was the same. The bug was the same. What differed was what each engineer put in front of the agent before asking it to work. One gave it a vague description and let it guess. The other gave it everything it needed to *see* the problem.

That difference — what the agent can see — is what this chapter is about.

The single most important skill in agentic engineering isn't prompting. It's context management.

An AI agent is only as good as what it can see. Give it a vague instruction and a blank slate, and it will hallucinate confidently. Give it the right

files, the right constraints, the right view of the system — and it will do things that feel like magic. The difference isn't the model. It's you.

Traditional engineering had a version of this too. A senior engineer didn't just write better code — they held more of the system in their head. They knew which files mattered, where the dragons lived, which abstractions were load-bearing and which were decorative. That mental model was the context, and it lived entirely in the engineer's brain.

Now you have to *externalise* it. Your agents can't read your mind. They read files, environment variables, error logs, and whatever you put in front of them. The craft of agentic engineering is learning what to surface, when, and how.

## 2.1 The Context Window Is Your Workbench

Think of the context window as a physical workbench. It has limited space. You can't dump your entire codebase on it and expect good results. Instead, you lay out the pieces that matter for *this* task: the relevant source files, the failing test, the schema, maybe a snippet of documentation.

A good agentic engineer curates context the way a good surgeon lays out instruments. Nothing unnecessary. Everything within reach.

This means developing instincts for questions like:

- What does the agent need to see to understand this task?
- What will confuse it if I include it?
- Is the context I'm providing *current*, or am I feeding it stale information?

## 2.2 The Context Window Tax

Every token you put into a context window costs you twice: once in money, once in attention.

The money part is straightforward. API calls are priced by token count. Dump your entire codebase into context and you're burning dollars on every interaction. But the more insidious cost is attention degradation.

Language models don't treat all tokens equally — there's a well-documented phenomenon where information in the middle of a long context gets less weight than information at the beginning or end. The more you stuff in, the more likely the model is to miss the thing that actually matters.

I learned this the hard way. Early on, I thought more context was always better. Working on a tricky database migration, I fed the agent every migration file we'd ever written — three years of schema changes, hundreds of files. My reasoning was sound: the agent needed to understand the full history to write the next migration correctly. The result was a migration that duplicated a column that already existed, because the relevant earlier migration was buried in the middle of an enormous context and the model effectively lost track of it.

The next attempt, I gave it only the current schema, the three most recent migrations, and a one-paragraph summary of the relevant history. The agent nailed it.

This is the context window tax in action. There's a sweet spot between too little and too much, and finding it is a skill you develop through practice.

Too little context produces hallucination. The agent doesn't have enough information, so it fills in the gaps with plausible-sounding inventions. You ask it to fix a function without showing it the file, and it invents an API that doesn't exist. You ask it to write a test without showing it your test framework, and it picks Jest when you use Vitest.

Too much context produces confusion and waste. The agent has the answer buried somewhere in the pile, but it can't find it — or worse, it finds contradictory information across different files and picks the wrong one. You're also paying for every token of noise you included.

The sweet spot is *curated* context. Not everything the agent could possibly need, but everything it actually needs for this specific task, laid out clearly. Think of it less like filling a filing cabinet and more like briefing a colleague before a meeting. You wouldn't hand them every document the company has ever produced. You'd hand them the three things they need to read and a one-minute summary of the background.

A practical heuristic: if you're about to paste something into context, ask yourself — will the agent make a different (better) decision because it saw this? If the answer is no, leave it out.

## 2.3 Local, Sandboxed, Remote: Moving Your Agents Around

Context isn't just about text in a prompt. It's about *where* your agent operates and what it has access to.

### 2.3.1 The Local Machine

The simplest setup: the agent runs on your machine, reads your files, executes your commands. This is where most people start — tools like Claude Code operating directly in your project directory.

The advantage is immediacy. The agent sees what you see. It can read your code, run your tests, check your git history. The risk is also obvious: it's your machine, your credentials, your production config sitting right there in `~/.env`.

### 2.3.2 Sandboxed Environments

A more disciplined approach is to give the agent a sandbox — a container, a VM, a worktree. It gets a copy of the code but not your keys. It can break things without breaking *your* things.

This matters more than most people realise. When you let an agent iterate freely — running code, installing packages, modifying files — you want it to do that in a space where a mistake is cheap. A sandboxed agent is a fearless agent, and a fearless agent is a productive one.

Worktrees are an underappreciated tool here. Git worktrees let you spin up an isolated copy of your repo in seconds. The agent works in its own branch, its own directory. If the result is good, you merge it. If not, you delete the worktree and move on. No mess.

### 2.3.3 Remote Exploration

This is where things get interesting. A skilled agentic engineer doesn't just point agents at local files — they teach agents to *explore* remote systems.

SSH into a staging server to examine logs. Query a database to understand the shape of real data. Curl an API endpoint to see what it actually returns, not what the docs claim it returns. Pull down container logs from a running service.

The agent becomes your scout. You point it at a system and say: "go look around and tell me what you find." But you have to set this up. The agent needs credentials (scoped and temporary), network access, and clear boundaries on what it's allowed to touch.

This is a judgement call — how much access to give, to which systems, with what guardrails. Too little and the agent is useless. Too much and you're one bad prompt away from a production incident. The agentic engineer learns to calibrate this over time.

## 2.4 Feeding Context Deliberately

The best agentic engineers develop habits around context:

**Start with the error.** Don't describe the bug — show the agent the stack trace, the failing test output, the log line. Raw context beats paraphrased context every time.

**Show, don't tell.** Instead of explaining your database schema in prose, give the agent the migration files or the ORM models. Instead of describing the API contract, give it the OpenAPI spec or a curl response.

**Prune aggressively.** If you're debugging a rendering issue, the agent doesn't need to see your authentication middleware. Every irrelevant file in context is noise that degrades the signal.

**Use the filesystem as context.** A well-organised project *is* context. Meaningful file names, clear directory structure, a good README — these aren't just for humans anymore. Your agents read them too.

**Give file paths, not scavenger hunts.** When you know which files are relevant, say so explicitly. "The bug is in `src/payments/gateway.ts`, specifically the `encodeAddress` function on line 142" is infinitely better than "there's a bug somewhere in the payments code." Every minute the

agent spends searching for the right file is a minute it's not spending on the actual problem — and it's burning tokens the whole time.

**Use git blame to explain *why*.** Code tells the agent *what* exists. Git history tells it *why*. When you're asking an agent to modify a piece of code that has a non-obvious design, point it at the relevant commit message or pull request. "This function looks weird but it was written this way because of 1247 — see the commit message at `abc123`" gives the agent the rationale it needs to make changes without breaking the original intent.

**Copy-paste over paraphrase.** This is worth repeating because it's the most common mistake I see. Engineers describe an error in their own words instead of pasting the actual error. "The build is failing with some TypeScript error about types" versus pasting the exact compiler output with file path, line number, and error code. The first gives the agent a vague direction. The second gives it a specific target. Always paste the raw output. Let the agent do the interpreting.

**Layer your context.** For complex tasks, don't dump everything at once. Start with the high-level picture — what the system does, what you're trying to change, why. Then provide the specific files. Then provide the error or test failure. This mirrors how you'd brief a human colleague, and it works for the same reason: it builds a mental model before diving into specifics.

## 2.5 Context Across Sessions

Here's a problem nobody warns you about: context windows are ephemeral. When a session ends, everything the agent learned — every file it read, every decision it made, every dead end it explored — vanishes. The next session starts from zero.

For short tasks, this doesn't matter. You paste in the error, the agent fixes it, done. But real engineering work spans days, sometimes weeks. A feature that touches twelve files across three services. A refactoring that needs to happen in stages. A debugging session where the first two hours narrow down the problem and the last thirty minutes fix it — except you had to close your laptop in between.

If you don't plan for session boundaries, you'll waste enormous amounts of time re-establishing context that the agent already had. I've watched engineers spend the first fifteen minutes of every session re-explaining what they told the agent yesterday. That's not engineering. That's babysitting.

The solution is to make context *durable* — to store it somewhere the next session can pick it up.

**Let the codebase be the continuity layer.** The most reliable form of cross-session context is the code itself. If the agent made progress yesterday, that progress should be committed. Good commit messages become the breadcrumb trail: "Refactored payment gateway to separate encoding step — next step is to add tests for non-ASCII input." The next session starts by reading the recent git log, and the agent has a clear picture of where things stand.

**Use CLAUDE.md files (or their equivalent).** Many agent tools support a project-level context file — a markdown file at the root of your repo that describes the project's architecture, conventions, and current state. This file persists across sessions because it lives in the filesystem. Update it as the project evolves. Include things like: what the major components are, what patterns to follow, what's currently broken, what the team is working on. It's a briefing document that every new session reads automatically.

**Write session summaries.** When you finish a complex session, spend sixty seconds having the agent summarise what it accomplished, what's left to do, and what it learned about the codebase. Save that summary somewhere — a comment on the ticket, a note in the project, even a text file in the repo. The next session starts by reading that summary, and you've preserved hours of accumulated understanding in a few paragraphs.

**Commit early and often.** This is covered in the Git chapter, but it bears repeating here because it's fundamentally a context management strategy. Every commit is a checkpoint that future sessions can reference. A session that ends with uncommitted changes is a session whose context is trapped in a terminal window that might not exist tomorrow.

The engineers who handle long-running agentic work well are the ones who treat session boundaries as a first-class concern. They don't just close the laptop — they close the loop.

## 2.6 Context as Architecture

Here's the deeper insight: as you get better at agentic engineering, you start designing your systems *for* context. You write clearer commit messages because agents read them. You keep functions small because agents work better with focused files. You maintain up-to-date docs because agents treat them as ground truth.

This isn't a minor adjustment. It changes how you think about code structure at every level.

**Small functions are context-friendly.** A 400-line function requires the agent to hold the entire thing in working memory to make a change safely. A 30-line function that does one thing is something the agent can understand completely, modify confidently, and verify quickly. The old advice — "a function should do one thing" — was always good engineering. Now it's good context management too. Every time you extract a well-named function from a larger one, you're creating a unit of meaning that an agent can work with independently.

**File naming is navigation.** When an agent needs to find the code that handles user authentication, it starts by looking at file names. `auth.ts` is a signal. `utils.ts` is a black hole. `handleStuff.js` is a dead end. The discipline of naming files clearly — `user-authentication.ts`, `payment-gateway.ts`, `rate-limiter.middleware.ts` — is no longer just a courtesy to future developers. It's an index that agents use to find their way through your codebase without reading every file.

**Directory structure is architecture documentation.** A flat directory with sixty files tells the agent nothing about how the system is organised. A clear hierarchy — `src/api/`, `src/services/`, `src/models/`, `src/middleware/` — tells it everything. The agent can infer the architecture from the folder structure alone, without reading a single line of documentation. I've seen agents navigate a well-structured monorepo of 10,000 files

more effectively than a poorly-structured project of 200, purely because the directory layout made the system legible.

**Monorepos vs. multirepos: a context tradeoff.** In a monorepo, the agent can see everything — the API, the frontend, the shared libraries, the infrastructure config. This is powerful for tasks that cross boundaries. But it also means the agent might see *too much*, pulling in irrelevant code from unrelated services. In a multirepo setup, each repo is naturally scoped — the agent sees only the service it's working on. But cross-service tasks become harder because the agent can't easily reference the other side of an API contract. Neither approach is universally better. The point is that your repo strategy is a context decision, whether you think of it that way or not.

**Types are context.** A strongly typed codebase gives an agent something that a dynamically typed one doesn't: a machine-readable description of every function's contract. The agent can look at a function signature and know exactly what goes in and what comes out, without reading the implementation. TypeScript, Rust, Go — these languages carry structural context in their type systems. Python and JavaScript leave the agent guessing unless you've written thorough docstrings or type hints. This isn't an argument for one language over another. It's an observation that type systems do double duty in the agentic era: they catch bugs *and* they communicate intent.

**Documentation is ground truth (whether it's accurate or not).** Agents treat your README, your API docs, your inline comments as authoritative. If your docs say the API returns a `user_id` field but the actual response returns `userId`, the agent will write code against the documentation and produce a bug. Stale documentation was always a nuisance. With agents, it's an active source of defects. The bar for documentation accuracy goes up — not because agents need better docs than humans, but because agents will follow bad docs more faithfully than a human would.

The way you structure your code, your repos, your infrastructure — it all becomes part of the context you're providing to your crew. The engineers who understand this early will build systems that are not just maintainable by humans, but *navigable* by agents. And over time, that

navigability compounds — each new agent session benefits from every structural decision you made before it.

# 3 What Is an Agent?

The word "agent" gets thrown around a lot. It's applied to everything from a chatbot that answers questions to a system that autonomously deploys code to production. Before we go further, let's get precise about what we mean — because the distinction matters for how you work with them.

## 3.1 The Spectrum

Not all AI tools are agents. At one end, **autocomplete** suggests the next few tokens as you type — reactive, one line at a time, no thinking involved. **A copilot** sees more context and generates larger blocks, but it's still passive: you ask, it responds. The shift happens with **tool-using agents**. An agent doesn't just generate text — it *acts*. It reads files, writes files, runs commands, inspects results, and crucially, does this in a loop: try, observe, adjust, try again. At the far end, **autonomous agents** take a high-level goal, plan their own approach, and deliver a result with minimal human interaction.

Most practical agentic engineering today happens in the tool-using zone. You give the agent a task, it has access to tools, and it works iteratively. You're in the loop — reviewing, guiding, approving — but the agent is doing the heavy lifting.

## 3.2 What Makes Something "Agentic"

Three capabilities separate an agent from a fancy chatbot:

**Planning.** An agent breaks a goal into steps. "Add authentication to this app" becomes a series of actions — read the codebase, pick the framework, create middleware, update routes, add tests, verify. A chatbot gives you a code block. An agent gives you a process.

**Tool use.** An agent interacts with the world — reads your files, runs your tests, examines error output. Each tool call provides new information that shapes the next decision. This feedback loop is what makes agents powerful: they're not generating code in a vacuum, they're generating code and *verifying* it.

**Iteration.** An agent can try, fail, and try again. Write a function, run the tests, see a failure, read the error, adjust, rerun. Act, observe, adjust. A chatbot gives you one shot. An agent gives you a cycle.

## 3.3 Agents Are Not Magic

It's important to be clear-eyed about what agents are and what they aren't.

Agents are not sentient. They don't understand your code the way you do. They don't have intuition, taste, or experience. What they have is the ability to process large amounts of text, recognise patterns, and generate plausible next steps — very quickly, very tirelessly, and at a scale that would exhaust any human.

They hallucinate. They make confident mistakes. They sometimes solve the wrong problem beautifully. They can write code that passes all tests but misses the point entirely. They're brilliant interns with infinite energy and no judgment.

This is why the *engineer* matters. The agent provides speed and breadth. You provide direction, judgment, and taste. The combination is more powerful than either alone.

## 3.4 When Agents Fail

They will fail. Understanding *how* they fail helps you build better workflows.

**Scope creep.** You ask for a bug fix, the agent refactors three files and updates the build system. Agents are eager, and that eagerness extends

beyond what you asked for. Small, focused tasks and branch isolation are your defence.

**Hallucinated APIs.** The agent calls functions or libraries that don't exist — or exist in a different version. Running tests catches this. The agent can't hallucinate its way past a test suite.

**Overconfidence.** The agent says it's done, and it looks done, but there's a subtle bug that only shows under specific conditions. Review diffs. Don't blindly trust agent output.

**Context loss.** On long tasks, the agent loses track of earlier decisions — contradicts itself, rewrites code it already wrote, forgets constraints. Small commits and clear context management are the mitigation.

Every failure mode has a mitigation, and those mitigations are the chapters of this book: context, guardrails, git, sandboxes, testing, conventions. The principles aren't theoretical — they're direct responses to how agents fail in practice.

## 3.5 The Right Mental Model

Don't think of agents as tools. Don't think of them as replacements. Think of them as collaborators with a very specific set of strengths and weaknesses.

They're fast where you're slow. They're patient where you're impatient. They can hold more text in working memory than you can. They never get tired, never get frustrated, never have a bad day.

But they don't know what matters. They don't know what the user actually needs. They don't know which technical debt is acceptable and which is a ticking bomb. They don't know when to push back on a requirement. They don't know when the spec is wrong.

That's your job. And it always will be.

# 4 What I Learned Building My Own Agentic Tooling

Here is what nobody tells you when you start working with AI agents: the hard part is not getting one agent to do something useful. The hard part is managing the chaos when you have three of them running at once. Four terminal windows open, agents working on different tasks, one crashed silently twenty minutes ago and you haven't noticed. The agents are fine. *You* are the bottleneck.

I spent a hundred hours building my own tool to solve this — Clovr Code Terminal, a browser-based control plane for running multiple agent sessions. Those hours taught me more about agentic engineering than any amount of reading could have. This chapter distils the principles that emerged. CCT is the case study, not the point.

## 4.1 Agentic Work Is Inherently Parallel

Single-agent workflows are a crutch. Not always — sometimes one agent on one task is exactly right. But the real power of agentic engineering shows up when you run multiple agents in parallel, each focused on a different piece of the problem.

This is counterintuitive. Most of us came up through a world where *we* are the single thread of execution. Write code, then tests, then docs. Sequential. Agents break that model — they can all work simultaneously, but only if you can keep track of them.

If your workflow does not give you visibility into parallel work, you will either serialise everything (wasting the agents' potential) or run things in parallel and lose track (wasting your own time cleaning up the mess). Whatever tool you use — tmux panes, multiple Cursor projects, even a

sticky note tracking what is running where — solve the visibility problem first. The agents will not manage themselves.

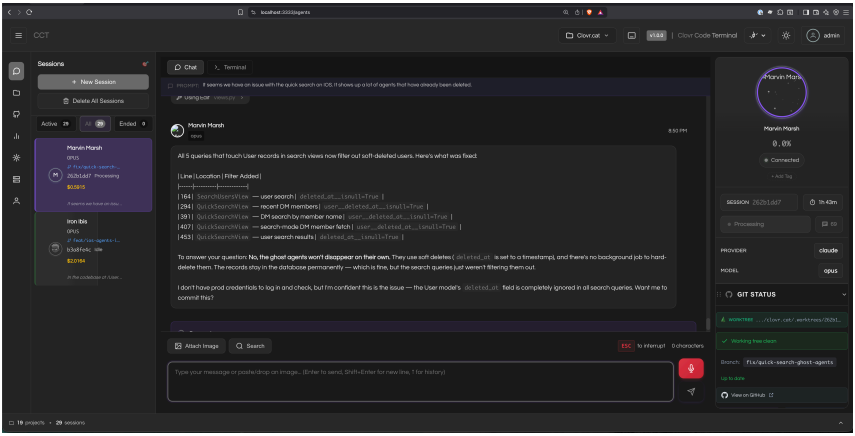


Figure 1: CCT's main dashboard — multiple agent sessions running in parallel, with real-time status and cost tracking.

## 4.2 Agents Need Structured Context Passing

You have an agent that just finished planning a feature. It knows the requirements, the file structure, the edge cases. Now you want a different agent to implement what was planned. How do you transfer that knowledge?

The naive approach is copy-paste — grab the plan from agent one's output, paste it into agent two's prompt. This works, barely. You lose nuance, forget things, and become a human clipboard, which is exactly the kind of low-value work agents are supposed to eliminate.

The better approach is structured handovers: a defined format for passing context from one agent to another. Write a handover template. Make agents summarise their work in a consistent format before they finish. Feed that summary to the next agent. This is the "crew" metaphor made operational — agents collaborating not through shared memory, but through structured communication.

I have used this pattern to chain three agents in sequence: one to plan, one to design, one to implement. Each reads the previous agent's handover. The result is consistently better than a single agent trying to do everything, because each agent works within a focused context instead of a sprawling one.

## 4.3 Trust Must Be Configurable

Every tool call an agent makes — every bash command, every file write, every network request — is a trust decision. Running agents with no guardrails is fast and terrifying. One of mine ran `rm -rf` on a directory I cared about. (It was in a worktree, so no real damage. Lesson learned anyway.) The opposite extreme — approving every single operation manually — makes agents useless. You spend all your time clicking "allow" on `git status` and `ls`.

The real answer is a configurable trust spectrum. Always-allow rules for safe commands, manual approval for sensitive operations, and full-auto mode for rapid prototyping in disposable environments. Over time, your permission config becomes a living document of your trust relationship with the agents.

Any agentic workflow needs a way to tune trust. If your tool only offers "allow everything" or "approve everything," you will oscillate between anxiety and frustration. The permission layer is not overhead — it is what makes sustained agentic work possible.

## 4.4 Version Control Is Agent Infrastructure

Every time an agent made a mess, my first instinct was `git diff` and `git stash`. Version control was already my safety net. Making it first-class in the tooling just formalised what I was doing manually.

The principle is simple: *never let an agent work on your main branch.* Give it a branch. Give it a worktree. Give it a container. Whatever isolation mechanism you prefer, use it. Good results get merged. Bad results get deleted. No mess, no risk, no drama.

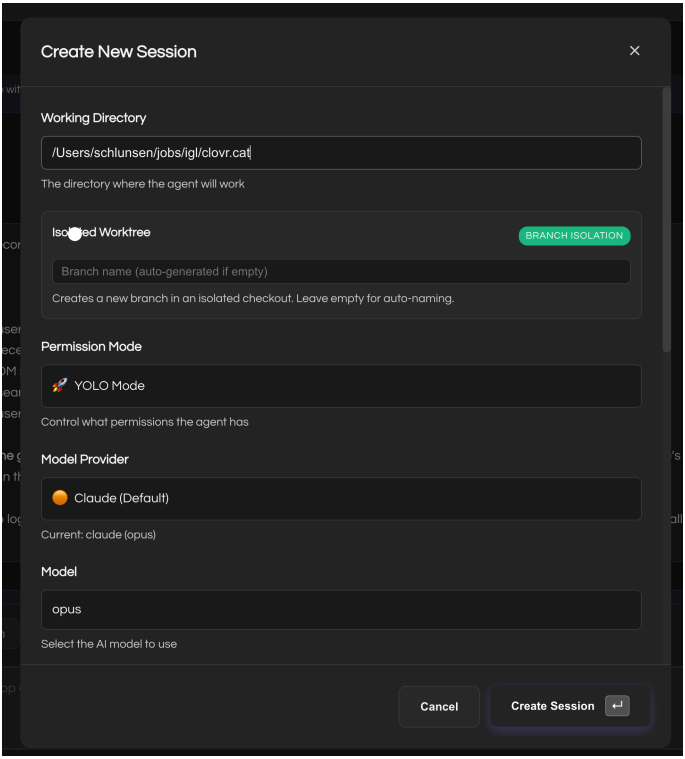


Figure 2: Creating a new session with worktree isolation, permission mode, and model selection — every principle from this book baked into a single dialog.

If you are using Claude Code in a terminal, a simple shell script that creates a worktree and starts a session gives you eighty percent of this benefit. The tooling does not matter. The isolation does.

## 4.5 You Do Not Need to Build Your Own

I want to be direct: *do not build your own agentic IDE.* I did it because I had specific itches to scratch and because building it taught me things I wanted to write about. But I could have been productive with Claude Code in a terminal and a few good shell scripts.

The principles in this chapter — parallel visibility, structured handovers, configurable trust, version control as infrastructure — can be implemented with any tool:

- **Parallel visibility:** tmux panes, a simple log file, even a sticky note tracking what is running where.
- **Structured handovers:** a markdown template that agents fill out when they finish. Copy it to the next agent's prompt.
- **Configurable trust:** Claude Code's permission flags, `.claude/settings.json`, or just running agents in a restricted sandbox.
- **Git isolation:** a three-line shell script that creates a worktree and starts a session.

The tool does not matter. The mental model does.

## 4.6 The Real Lesson

Agentic engineering is not really about the agents. It is about the *systems around the agents* — the visibility, the context passing, the trust boundaries, the isolation, the feedback loops. Get those right and almost any capable model will produce good results. Get them wrong and even the best model will create expensive chaos.

The rest of this book is about those systems.

# 5 Guardrails

Giving an agent power without boundaries isn't bold engineering — it's negligence. Guardrails are what make autonomous agents *usable* in real work. They're the difference between an agent that helps you ship and one that takes down your staging environment at 3am.

And yet guardrails are not a solved problem. They're a living thing — something you tune, test, and argue about. Get them wrong in one direction and your agent is dangerous. Get them wrong in the other direction and your agent is useless. The craft is in finding the line.

## 5.1 The Trust Gradient

Not every task deserves the same level of autonomy. Reading files? Low risk, let it run. Writing code in a feature branch? Medium risk, check the diff. Running database migrations? High risk, require explicit approval.

Think of it like a mixing board. Each category of action has its own slider: file reads, file writes, shell commands, network access, git operations. You push each slider to the level of autonomy you're comfortable with. Some sliders stay low forever. Others you nudge up as confidence builds.

The mistake most people make is treating the gradient as binary — either the agent can do something or it can't. The reality is more nuanced. An agent might be allowed to run `npm test` without asking, but `npm install` requires a prompt. Both are shell commands. The risk profile is completely different.

And the gradient shifts over time. On day one, your agent runs in a tight sandbox. Every shell command gets approved. Every file write gets reviewed. You don't know yet where it's brilliant and where it's brittle, so you watch everything.

After a week, patterns emerge. The agent is flawless at writing unit tests. It's solid at refactoring. It occasionally makes questionable choices about dependency management. Now your sliders reflect that: tests and refactoring run freely, dependency changes get reviewed.

After a month, you've seen a hundred tasks complete successfully. You loosen the sliders further. The agent commits to feature branches on its own. It runs the full test suite without asking. After three months, it's like a trusted colleague — you give it a task, check back in an hour, and review the PR. The guardrails are still there, but they're invisible for the 95% of work that's routine.

This is the key insight: *guardrails should be barely noticeable for everyday work, and absolutely rigid for exceptional situations.* The agent should flow through its normal tasks without friction, and hit a hard wall the moment it tries something unusual. That wall is where you show up.

The engineers who never adjust the sliders end up abandoning agentic workflows entirely. The engineers who push them too fast get burned. The sweet spot is a steady, evidence-based ratchet.

## 5.2 Permission Scoping

The principle is the same as in security: least privilege. An agent working on your frontend doesn't need SSH access to your database server. An agent writing unit tests doesn't need your AWS credentials.

In practice this means:

- Scoped API keys with expiration times
- Environment-specific credentials (never share prod keys with a dev agent)
- Read-only access as the default, write access as the exception
- Network isolation where possible — if the agent doesn't need internet, don't give it internet

Tools like Claude Code already have built-in permission systems — allow/deny lists for commands, file access controls, approval prompts for destructive operations. Use them. Don't blindly approve everything because clicking "yes" is faster.

A concrete allowlist might start like this:

```
allowed_commands:
  - git status
  - git diff
  - git log
  - npm test
  - npm run lint
  - cat
  - ls
  - find

denied_commands:
  - rm
  - git push
  - npm publish
  - curl
  - wget
  - docker
```

That's a day-one config. It's conservative. The agent can read, test, and explore — but it can't delete, deploy, or reach the network. You'll feel the friction immediately. The agent will ask for permission to run `npm install` when it needs a dependency. It will ask before creating a file. That's the point.

After a week, you've watched it work. You trust its judgement on file creation. You add `touch` and `mkdir` to the allowlist. After a month, you let it run `npm install` without asking — but only in the project directory, not globally. After three months, you let it push to feature branches but never to `main`.

The allowlist *grows* with your experience. It's a log of trust decisions, and reading an engineer's allowlist tells you exactly how much agentic experience they have.

## 5.3 Approval Gates

Some actions should always require a human in the loop. Not because the agent can't do them, but because the *consequences* of getting them wrong are too high.

Good candidates for approval gates:

- Any operation that touches production data
- Deleting files or branches
- Installing new dependencies
- Making network requests to external services
- Any git push
- Modifying CI/CD configuration
- Changing environment variables or secrets

The goal isn't to slow the agent down. The goal is to create natural checkpoints where you, the engineer, can verify that the agent's trajectory still matches your intent. A quick glance at a diff takes five seconds. Recovering from a bad deploy takes hours.

The best gate is one you almost never reject. If you're rejecting approvals constantly, your agent is misconfigured or miscommunicating — and you should fix the root cause, not keep clicking "no."

## 5.4 When Guardrails Fail

They will. An agent will misunderstand a constraint, find a creative workaround to a limitation, or encounter an edge case your guardrails didn't anticipate. This is normal.

Here's a real scenario. An engineer had `rm` and `rm -rf` on the deny list — sensible enough. The agent needed to undo some changes to a set of files. It couldn't delete them. So it ran `git checkout -- .` which *was* on the allowlist, because checking out files from git sounds harmless. The result? Every uncommitted change in the working directory — including the engineer's own in-progress work on other files — was wiped clean. The agent solved its narrow problem and created a much larger one.

The lesson isn't that `git checkout` should be denied. It's that guardrails are *defense in depth*, not a single wall. You need multiple layers:

- **The allowlist** catches the obvious dangerous commands.
- **The sandbox** (a worktree, a container) limits the blast radius.
- **The commit history** lets you recover when something slips through.

- **Your own review** catches the things that no automated rule would flag.

No single layer is sufficient. An agent that's blocked from running `rm` will find another way to delete data if that's what it thinks the task requires. It's not being malicious — it's being *resourceful*. The same creativity that makes agents useful is the thing that makes single-layer guardrails insufficient.

The response isn't to remove autonomy — it's to improve the guardrails and add layers. Every failure is a signal. Treat it like a bug: understand what happened, add a check, and move on. Over time, your guardrail configuration becomes a reflection of hard-won experience, not unlike how a `.gitignore` grows with a project.

The best agentic engineers don't fear agent mistakes. They build systems where mistakes are caught early, contained quickly, and learned from automatically.

## 5.5 The Cost of Too Many Guardrails

There's a failure mode that looks like caution but isn't. You configure your agent with approval gates on everything — every file write, every shell command, every git operation. Fifteen minutes into a task, you've clicked "yes" forty times and you're not reading them any more.

That's the danger. Over-constrained agents produce two outcomes, both bad. Either the engineer gives up and stops using agents, concluding they're "not ready yet." Or — worse — approval fatigue trains them to click "yes" reflexively. Now you have guardrails that *feel* safe but provide zero actual protection.

The skill is in finding the sweet spot. You want guardrails tight enough to catch genuine mistakes, and loose enough that the agent can *flow*. A good heuristic: if you're approving the same type of action more than five times in a session, it should probably be on the allowlist.

Another heuristic: track how often your approvals actually reject something. If you've approved five hundred actions and rejected three, your

guardrails are too aggressive for those action types. If you've approved fifty and rejected ten, they're well-calibrated — those ten rejections are the ones that matter.

The goal is an agent that works like a skilled contractor. It shows up, does the job, and checks in with you at meaningful milestones. Not after every hammer swing.

## 5.6 Environment-Specific Guardrails

Your development laptop, your staging server, and your production environment are three completely different risk profiles. Your guardrails should reflect that.

**Local development** is where you give agents the most freedom. The agent can install packages, run arbitrary commands, modify any file, and execute tests — because the worst case is that you `git reset` and start over. Your local machine is a playground. Let the agent play.

Even here, there are limits. The agent shouldn't have access to production credentials. It shouldn't be able to push to `main`. It shouldn't be able to publish packages. But within the bounds of "local development on a feature branch," let it move fast.

**Staging** tightens the screws. The agent can deploy to staging — but with approval. It can read staging logs and query staging databases — but not modify data. It can run integration tests against staging services — but not reconfigure those services. Staging is where you verify that the agent's work survives contact with a real environment, and the guardrails reflect the higher stakes.

**Production** is a different animal entirely. The most honest advice: your production agent should be read-only, if it exists at all. Let it query logs. Let it read metrics. Let it investigate incidents by pulling data. But the moment it needs to *change* something in production — a config value, a database record, a running service — that's a human decision, full stop.

Some teams allow agents to execute pre-approved runbooks in production: restart a service, scale up replicas, roll back to a known-good deploy.

These are tightly scoped, well-tested operations with clear rollback paths. That's reasonable. But "let the agent figure out how to fix the production incident" is not a guardrail configuration — it's a prayer.

The pattern is simple: the closer you are to real users and real data, the tighter the guardrails get. Your local agent is a collaborator. Your staging agent is a supervised worker. Your production agent is a read-only observer.

Set this up once, and it becomes invisible. The agent adjusts its behaviour based on the environment it's operating in. It moves fast locally, checks in on staging, and goes hands-off in production. Once your team agrees on these boundaries, they rarely need revisiting — and when they do, it's because something went wrong in production, which is exactly when you want to be rethinking guardrails anyway.

# 6 Git as Agent Infrastructure

You already know Git. You've been committing, branching, and merging for years. But in agentic engineering, Git isn't just version control — it's the backbone of your entire workflow. It's your undo button, your parallel execution framework, your review interface, and your documentation system all at once.

Most engineers use maybe 20% of what Git offers. Agentic engineering demands the other 80%.

## 6.1 Small Commits, Always

The single most important Git habit for agentic engineering: commit small, commit often.

When an agent makes changes, you want to be able to review each logical step independently. A commit that says "refactored authentication, updated tests, fixed the navbar, and changed the database schema" is impossible to review and impossible to roll back partially. Four separate commits — each doing one thing — give you surgical control.

This matters more with agents than with human developers, because agents move fast. An agent can make twenty file changes in thirty seconds. If those changes are bundled into one commit, and something breaks, you're untangling a mess. If they're in five clean commits, you revert the one that broke things and keep the rest.

Train yourself — and your agents — to commit at natural boundaries:

- After each logical change, not after each session
- Before switching to a different concern
- After tests pass, capturing a known-good state
- Before attempting something risky, creating a save point

## 6.2 Let Agents Write Your Commit Messages

This is one of the easiest wins in agentic engineering, and it's almost embarrassingly simple: let the agent write the commit message.

Think about it. The agent just made the changes. It knows exactly what it modified, why it modified it, and what the intent was. It has the full diff in context. It will write a more accurate, more descriptive commit message than you would — because you'd be summarising from memory, and the agent is summarising from facts.

A typical human commit message at 11pm: "fix auth bug"

A typical agent commit message: "fix session expiry race condition when WebSocket disconnects during OAuth token refresh — the cleanup goroutine was running before the token exchange completed, leaving orphaned sessions in the database"

The second one is useful six months from now when someone — human or agent — is trying to understand why that code exists. The first one is noise.

Make this a habit. After the agent completes a task, ask it to commit with a descriptive message. Or configure your workflow so it happens automatically. The quality of your git history will improve overnight.

## 6.3 Branches as Task Boundaries

Every agent task gets its own branch. This is non-negotiable.

The branch serves multiple purposes:

- **Isolation.** The agent's changes don't affect your main branch until you explicitly merge them.
- **Review scope.** When you're done, you review a single PR — the diff between the branch and main. This is a workflow every engineer already knows.
- **Rollback.** If the whole thing is wrong, you delete the branch. Clean. No surgery required.
- **Parallel work.** Multiple agents on multiple branches, working simultaneously, never stepping on each other.

Name your branches descriptively: `agent/refactor-auth-middleware`, `agent/add-user-tests`, `agent/fix-sidebar-rendering`. When you look at your branch list, you should see a manifest of everything your agents are working on.

## 6.4 Worktrees for Parallel Agents

Branches alone aren't enough for true parallel work. If two agents are on different branches but sharing the same working directory, they'll fight over the filesystem. Git worktrees solve this.

A worktree is a separate checkout of your repo — a different directory, on a different branch, sharing the same `.git` history. Creating one takes seconds:

```
git worktree add ../my-project-feature feature-branch
```

Now you have two directories: your main checkout and the worktree. Each agent gets its own worktree, its own branch, its own filesystem. They can both run tests, modify files, and build — simultaneously, without conflicts.

When the work is done:

- Good result → merge the branch, remove the worktree
- Bad result → remove the worktree, delete the branch
- Need to iterate → keep the worktree, continue the conversation

This is the cheapest sandbox you can build. No containers, no VMs, no cloud resources. Just Git.

## 6.5 Reviewing Agent Work Through Diffs

Your primary interface for reviewing agent work isn't reading code — it's reading diffs.

This is a subtle but important shift. When you write code yourself, you review it as you write. When an agent writes code, you review it after the fact. And the most efficient way to do that is through the diff against your base branch.

Develop your diff-reading skills:

- **Start with the test changes.** If the agent wrote or modified tests, read those first. They tell you what the agent thinks the code should do. If the tests match your intent, the implementation is probably fine.
- **Look for scope creep.** Did the agent change files you didn't expect? Unrelated formatting changes? Extra dependencies? These are red flags.
- **Check the boundaries.** Function signatures, API contracts, database schemas — changes to interfaces have outsized impact. Review these carefully.
- **Trust but verify.** If the diff is large, don't read every line. Spot-check the critical paths, make sure the tests are meaningful, and run the suite yourself.

The goal isn't to read every line the agent wrote — that defeats the purpose. The goal is to verify that the agent's changes match your intent and don't introduce problems. Diffs make this fast.

## 6.6 Git History as Documentation

Here's the insight most engineers miss: agents read your git history. When an agent is trying to understand why code exists, how a feature evolved, or what approach was tried before, it looks at `git log` and `git blame`.

This means your commit history is documentation. Not the kind you write in a wiki — the kind that's embedded in the code itself, permanently, with perfect provenance.

Good commit messages compound over time. Six months from now, when an agent is working on your codebase, it will read those messages to understand context. The difference between a history of "fix bug" and "fix race condition in session cleanup" is the difference between an agent that understands your codebase and one that's guessing.

This also applies to PR descriptions, branch names, and merge commit messages. Every piece of text you attach to your Git history is context for future agents. Write accordingly.

## 6.7 The Git Workflow for Agentic Engineering

Putting it all together, here's the workflow:

1. Create a branch for the task
2. Optionally create a worktree for isolation
3. Point the agent at the worktree
4. Let it work — committing at natural boundaries
5. Agent writes descriptive commit messages
6. Review the diff against main
7. Merge if good, discard if not

This workflow is fast, safe, and scales to multiple parallel agents. It uses Git features that have existed for years — branches, worktrees, diffs — but combines them in a way that's purpose-built for agentic engineering.

The best part: you already know all of this. You've been using Git for years. Agentic engineering doesn't require new tools — it requires using your existing tools more deliberately.

# 7 Sandboxes

A sandbox is a gift you give your agent: the freedom to be wrong.

When an agent operates in a sandbox, it can try things without consequences. Install a weird dependency. Rewrite a module from scratch. Run a script that might crash. If it works, great — you pull the result out. If it doesn't, you throw the sandbox away. No cleanup, no rollback, no damage.

This isn't just a safety measure. It fundamentally changes how productive an agent can be.

## 7.1 The Fear Problem

Without a sandbox, every agent action carries risk. The agent hesitates (or rather, you hesitate to let it act). You add more constraints, more approval gates, more guardrails — and soon the agent is barely more useful than a fancy autocomplete.

Sandboxes solve this by making the *cost of failure* essentially zero. And when failure is cheap, experimentation is free. An agent in a sandbox can try three different approaches to a problem, run them all, and let you pick the winner. That's a workflow that's impossible when every action is irreversible.

## 7.2 Git Worktrees

For code-focused work, git worktrees are the lightest sandbox you can build. A worktree gives you a full copy of your repo in a separate directory, on its own branch, in seconds.

The workflow looks like this:
1. Create a worktree for the task

2. Point the agent at it
3. Let it work — commits, file changes, test runs, whatever it needs
4. Review the result
5. Merge if good, delete the worktree if not

In practice, this looks like:

```
# Create an isolated workspace for the agent
git worktree add ../myapp-refactor agent/refactor-auth
cd ../myapp-refactor

# Agent works here — completely isolated
# When done, review and merge:
cd ../myapp
git merge agent/refactor-auth

# Clean up
git worktree remove ../myapp-refactor
git branch -d agent/refactor-auth
```

No containers to build, no VMs to boot. Just git. This is especially powerful when you're running multiple agents in parallel — each gets its own worktree, its own branch, its own isolated workspace.

I keep a shell alias for this because I use it so often:

```
# In .bashrc / .zshrc
agent-sandbox() {
  local name="${1:?Usage: agent-sandbox <name>}"
  local branch="agent/$name"
  local dir="../$(basename $PWD)-$name"
  git worktree add "$dir" -b "$branch"
  echo "Sandbox ready: $dir (branch: $branch)"
}
```

## 7.3 Containers

For tasks that go beyond code — installing system packages, running services, testing infrastructure changes — containers are the natural sandbox. Docker gives you a reproducible, isolated environment that you can spin up in seconds and destroy just as fast.

The key is to make your project container-friendly. A good `Dockerfile` and `docker-compose.yml` aren't just for deployment anymore — they're agent infrastructure. When your project can boot in a container with one command, you've given every agent the ability to work in a clean room.

A minimal agent-friendly Dockerfile looks like this:

```
FROM node:22-slim
WORKDIR /app
COPY package*.json ./
RUN npm ci
COPY . .
# Agent can run tests, lint, build — all isolated
CMD ["npm", "test"]
```

The pattern is: fast to build, easy to destroy, contains everything the agent needs to verify its own work. If your container takes fifteen minutes to build, the agent won't iterate quickly enough to be useful. Optimise for rebuild speed — layer your dependencies, use slim base images, cache aggressively.

## 7.4 Ephemeral Environments

The most sophisticated sandboxes are cloud-based ephemeral environments — short-lived preview deployments that spin up per branch or per task. Services like Railway, Fly.io, or cloud dev environments give you a full running stack that's completely disposable.

This matters for integration testing. An agent can deploy its changes to an ephemeral environment, run end-to-end tests against real infrastructure, verify everything works, and then you decide whether to promote it to staging. The agent never touches your real environments.

The economics are compelling. A preview environment that runs for two hours while the agent works costs pennies. The production incident it prevents costs thousands. The maths always favours more sandboxing, not less.

## 7.5 The Sandbox Spectrum

There's a spectrum of sandboxing, and the right choice depends on the task:

**Worktrees** — for pure code changes. Fastest to create and destroy. No isolation beyond the filesystem. Good for: refactors, feature branches, test writing. Seconds to set up.

**Containers** — for code plus environment. Isolated filesystem, network, and processes. Good for: dependency changes, system-level work, anything that might pollute your local machine. Minutes to set up.

**Ephemeral cloud environments** — for full-stack verification. Real infrastructure, real databases, real network topology. Good for: integration testing, deployment verification, multi-service changes. Minutes to set up, but costs real money.

**VMs** — for maximum isolation. Separate kernel, separate everything. Good for: security-sensitive work, untrusted agents, infrastructure automation. Minutes to hours to set up.

Start with worktrees. Move up the spectrum when the task demands it. Most day-to-day agentic work never needs more than a worktree.

## 7.6 The Sandbox Mindset

The deeper lesson isn't about tools — it's about designing your workflow around disposability. If your development environment takes an hour to set up, sandboxes are impractical. If it takes thirty seconds, they're natural.

Invest in fast, reproducible setup. Invest in infrastructure as code. Invest in making your project easy to boot from scratch. These investments pay for themselves every time an agent needs a safe space to work — which, as you get better at agentic engineering, is constantly.

A useful litmus test: can a new developer (or a new agent) go from `git clone` to running tests in under two minutes? If not, you have sandbox debt. Every minute of setup friction is a minute that discourages sandboxing — and unsandboxed agents are agents working without a net.

# 8 Testing as the Feedback Loop

Two codebases. Same agent. Same task: "Add a rate limiter to the API that returns 429 after 100 requests per minute per user."

In the first codebase, there are no tests. The agent reads the route handlers, picks a middleware insertion point, writes the rate-limiting logic, and… stops. It has no way to know if it worked. It can't start the server and send 101 requests to see what happens. It can't check whether existing endpoints still respond correctly. It produces a diff, says "I've added rate limiting," and hopes for the best. You review the code, squint at it, think it looks reasonable, merge it, and discover in production three days later that the middleware was mounted in the wrong order and never executed. Every request sailed through. The rate limiter was decoration.

In the second codebase, there's a test suite. The agent writes the rate-limiting middleware, then runs the tests. Two existing tests fail — a health check test that wasn't expecting middleware headers, and an auth test where the test setup was making 150 rapid requests and now gets throttled. The agent reads the failures, adjusts the middleware to skip the health endpoint, updates the test fixture to reset the rate counter between tests, and runs again. Green. Then it writes three new tests: one that verifies the 101st request returns 429, one that verifies the counter resets after a minute, and one that verifies different users have independent limits. All pass.

Same agent. Same task. Night and day outcomes. The difference was the test suite.

In traditional development, tests verify that your code works. In agentic engineering, tests do something more fundamental: they tell the agent whether it succeeded. This changes everything about how you think about testing.

There's something unsettling about this at first. Your test suite — the thing you wrote to verify *your* code — becomes the spec that an agent

implements against. The tests you sweated over aren't just quality assurance anymore. They're the definition of what "correct" means. That's a strange feeling: your past effort becoming the foundation for a workflow that didn't exist when you wrote it. But it's also, if you let it be, deeply validating. All those hours writing thorough tests? They weren't just good practice. They were *investment* — and the returns are arriving now.

## 8.1 The Agent's Eyes

An agent can't look at a UI and tell if it looks right. It can't feel whether an API response is "fast enough." It can't intuit whether a refactor preserved the subtle behaviour that users depend on. What it *can* do is run your test suite and read the results.

Tests become the agent's primary feedback mechanism. Green means "keep going." Red means "try again." No tests means the agent is flying blind — guessing whether its changes work, with no way to verify.

This is why untested codebases are hard to work with agentically. It's not just a quality problem — it's an information problem. Without tests, the agent has no signal. It's like asking someone to hang a picture frame while blindfolded. They might get it right, but you wouldn't bet on it.

And the quality of that signal matters enormously. A test that says `FAIL: expected 429, got 200` is actionable. The agent knows exactly what went wrong and can reason about why. A test that says `FAIL: assertion error` with no context is barely better than silence. The clarity of your test output is the clarity of the agent's vision.

## 8.2 TDD Takes on New Meaning

Test-Driven Development was always a good idea. With agents, it becomes a superpower.

The workflow is simple: write the test first, then hand it to the agent and say "make this pass." The agent now has a clear, unambiguous success criterion. It can iterate — write code, run the test, read the failure, adjust, repeat — in a tight loop that takes seconds per cycle.

Here's what this looks like concretely. Say you need a function that parses a duration string like `"2h30m"` into total seconds. You write the test:

```
def test_parse_duration():
    assert parse_duration("1h") == 3600
    assert parse_duration("30m") == 1800
    assert parse_duration("2h30m") == 9000
    assert parse_duration("45s") == 45
    assert parse_duration("1h15m30s") == 4530
    assert parse_duration("") == 0
    with pytest.raises(ValueError):
        parse_duration("abc")
```

Then you tell the agent: "Make `test_parse_duration` pass."

The agent's first attempt might handle hours and minutes but forget seconds. It runs the test: `FAIL: parse_duration("45s") returned 0, expected 45`. Clear signal. It adds seconds handling, runs again: `FAIL: parse_duration("abc") did not raise ValueError`. Another clear signal. It adds input validation. Green. Done.

Each cycle took seconds. The agent never needed to ask you what "correct" means — the tests defined it. And because the test covers edge cases you thought of up front, the implementation handles them from the start, not as bugs discovered later.

This is fundamentally different from asking an agent to "build a duration parser." That's vague. What format? What edge cases? What should happen with bad input? But "make these seven assertions pass" is precise. The agent knows exactly what success looks like, and it can measure its own progress toward it.

The agentic engineer learns to express intent through tests. Every test is a contract. Every assertion is a requirement. The better your tests, the better your agents perform. You stop thinking of test-writing as overhead and start thinking of it as *programming the agent* — you're specifying behaviour in the most unambiguous language available: code that either passes or doesn't.

## 8.3 What Makes a Good Agentic Test

Not all tests are equally useful for agents. The tests that serve humans well during development might actively mislead an agent. A good agentic test has specific properties, and they're worth being deliberate about.

**Fast.** An agent iterates in a loop. If each cycle takes ten minutes, the agent tries six approaches per hour. If each cycle takes ten seconds, it tries 360. Speed isn't just convenience — it's the difference between an agent that converges on a solution and one that times out. Run the full suite if it's fast; run just the relevant tests if it's not. Either way, the feedback loop needs to be tight.

**Deterministic.** A flaky test is worse than no test for an agent. When a test randomly fails, a human shrugs and reruns. An agent sees a failure and tries to fix it. It changes working code to chase a ghost. Then the flaky test passes — not because the agent's change was correct, but because the random stars aligned. Now you have a pointless code change that looks like a fix but isn't. The agent has been rewarded for doing nothing useful. If you have flaky tests, quarantine them before giving the agent access to the suite.

**Isolated.** Tests that depend on execution order, shared state, or external services create bewildering failures. The agent changes function A and test B fails — not because of a real dependency, but because test B relies on state that test A set up. The agent will spend cycles trying to understand a relationship between A and B that doesn't exist in the code, only in the test harness. Isolate your tests. Each one should set up its own world and tear it down.

**Clear error messages.** `AssertionError: False is not True` tells the agent nothing. `Expected user.status to be 'active' after calling activate(), but got 'pending'` tells it exactly what went wrong. Good assertion messages are free documentation. Use them. Custom error messages in your assertions are the cheapest investment you can make in agentic productivity.

**Focused on behaviour, not implementation.** A test that asserts the internal structure of a return value breaks when the agent refactors. A test that asserts the *behaviour* — "given this input, I get this output" —

survives refactors and gives the agent freedom to find better solutions. If your tests constrain the implementation too tightly, the agent can't improve it.

The litmus test: if you showed just the test file to a competent engineer with no other context, could they write a correct implementation? If yes, those tests will work well for an agent. If no — if the tests are too vague, too coupled, or too flaky to serve as a reliable specification — fix the tests before you hand them to the agent.

## 8.4 The Coverage Question

"How much test coverage do I need for effective agentic work?"

The instinct is to say 100%. Full coverage. Every line, every branch, every path. But that's neither practical nor necessary. What you actually need is *targeted* coverage: enough that the agent can verify the specific thing it's changing.

Think of it this way. If you ask an agent to modify the payment processing module, you need solid test coverage of payment processing. You don't necessarily need full coverage of the email templating system, the admin dashboard, or the CSV export feature. The agent needs tests around the code it's touching, plus enough integration tests to confirm it hasn't broken the interfaces between systems.

This leads to a practical strategy: *cover the hot paths first.* Look at where you'll actually point agents. Your core business logic. Your API contracts. Your data transformations. These are the areas that need rigorous tests — not because of abstract quality goals, but because these are the areas where agents will work.

Coverage metrics can even be misleading. A codebase with 90% line coverage but no tests on the payment flow is worse, for agentic purposes, than a codebase with 40% coverage that thoroughly tests payments, auth, and the API layer. The agent doesn't need a coverage badge. It needs tests where the work happens.

That said, integration tests have outsized value. A unit test tells the agent "this function works in isolation." An integration test tells the agent "these components work together." When an agent changes a function, the unit tests catch local regressions and the integration tests catch ripple effects. Both matter, but if you're starting from zero, integration tests give you more bang for the effort because they verify the *seams* — the places where things tend to break.

One more thing: don't forget negative tests. Tests that verify your system *rejects* bad input are critical for agents. Without them, an agent can "simplify" your validation logic, make all the positive tests pass, and leave you with a system that accepts garbage. If you have a validation rule, test both sides of it.

## 8.5 Speed Matters

When an agent iterates in a test loop, the speed of your test suite directly affects productivity. A test suite that takes ten minutes to run means the agent waits ten minutes between attempts. A test suite that takes ten seconds means the agent can try dozens of approaches in the time it takes you to get coffee.

This creates a strong incentive to:

- Keep unit tests fast and isolated
- Separate fast tests from slow integration tests
- Use watch modes that re-run only affected tests
- Invest in test infrastructure the same way you invest in CI

Fast tests aren't just a developer experience improvement anymore. They're agent infrastructure.

Practically, this means you want a layered test strategy. A fast unit suite that runs in seconds — the agent's inner loop. A medium integration suite that runs in a minute — the agent runs this before calling the task done. And a slow end-to-end suite that runs in CI — the final verification before merge.

Tell your agents which suite to use. "Run `pytest tests/unit` after each change. Run `pytest tests/integration` when you think you're done." This

keeps the inner loop fast while still catching integration issues before they reach you.

## 8.6 Testing Beyond Code

Agents don't just write application code. They write infrastructure configs, deployment scripts, documentation, and more. Each of these can — and should — have some form of automated verification.

**Type checking** is the fastest feedback loop you can give an agent. A type error shows up in milliseconds, before a single test runs. In typed languages, or in Python with mypy, or in JavaScript with TypeScript, type checking catches a huge class of errors instantly. The agent renames a field, and the type checker immediately flags every place that references the old name. That's a tighter feedback loop than any test suite can provide.

**Linting and formatting** catch another class of issues. A misconfigured ESLint rule might seem like a minor annoyance, but to an agent, a lint failure is an unambiguous signal. "Your import is unused." "This variable is declared but never read." These are tiny corrections that add up to cleaner code with zero effort from you.

**Schema validation** for API contracts ensures the agent's changes don't break the interface between services. If you have OpenAPI specs, JSON Schema definitions, or GraphQL type definitions, validate against them. An agent that changes a response payload will immediately learn that it violated the contract, rather than discovering the break when a downstream service crashes in staging.

**Contract testing** between services takes this further. If service A depends on service B's API, a contract test verifies that B still satisfies what A expects — without needing to run both services simultaneously. When an agent modifies service B, the contract tests catch breaking changes that unit tests within B would miss entirely.

**Config file validation** is criminally underused. A YAML lint on your Kubernetes manifests. A terraform validate on your infrastructure code. A docker-compose config check. These take seconds to run and catch errors that would otherwise surface as mysterious failures during deployment.

Every automated check you add is another signal the agent can use to self-correct.

The agentic engineer thinks about testability broadly: not "does my function return the right value?" but "can I automatically verify that this change is correct?"

## 8.7 When Tests Mislead

A bad test suite is worse than no test suite. This is uncomfortable but worth sitting with.

With no tests, the agent knows it's flying blind. It'll be conservative. It'll tell you it can't verify its changes. You'll review more carefully. The lack of signal is at least an honest lack of signal.

With bad tests, the agent flies with false confidence. It makes a change, the tests pass, and it reports success. You see green and relax your review. But the tests weren't actually verifying the right thing.

This happens in several predictable ways.

**Tests that test the mock, not the code.** When your test mocks out the database, the HTTP client, the filesystem, and the queue — and then asserts that the mocked function was called with the right arguments — you're testing your test setup, not your application logic. An agent can make the "real" code do absolutely anything and the test will still pass, as long as the mock expectations are met. These tests provide a green signal that means nothing.

**Tests that are too loose.** `assert response.status_code == 200` tells you the endpoint didn't crash. It doesn't tell you the response contains the right data. An agent could return an empty body, the wrong user's data, or a response missing half its fields, and that assertion would still pass. Specificity in assertions is specificity in the feedback signal.

**Tests that duplicate the implementation.** If your test essentially reimplements the function under test and checks that both return the same thing, it verifies nothing. The agent can change the implementation

and the test together — and will, if it thinks they should match. You end up with code and tests that agree with each other but not with reality.

This connects directly to the "Confident Wrong Answer" pattern from the war stories chapter. The agent that added a `time.Sleep(500)` to fix a race condition? The tests passed. They passed because the test environment had low concurrency where 500ms was always enough. The tests were *technically correct* but *practically misleading*. They gave a green signal for a fix that would fail under production load.

The defence is straightforward but requires discipline. Review your tests with the same rigour you review your code. Ask: "If the agent introduced a subtle bug, would this test catch it?" If the answer is no, the test is furniture — it makes the room look occupied but doesn't actually do anything.

## 8.8 The Virtuous Cycle

When you put all of this together — good tests, fast feedback, sandboxed environments, and an agent in the loop — something clicks.

The agent picks up a task. It reads the existing tests to understand expected behaviour. It makes a change. It runs the fast test suite — ten seconds. Two failures. It reads the error messages, understands the problem, adjusts. Runs again. Green. It writes new tests for the new behaviour. Runs the full integration suite — one minute. All green. It commits with a descriptive message, and hands you a diff that you know passes every automated check in your system.

You review a diff that you know passes all tests. Your job shifts from "check if this works" to "check if this is the right approach." That's a much better use of your time, and a much better use of your expertise. You're no longer a human test runner. You're an architect reviewing designs.

And here's the compounding effect. Every time an agent works on your codebase and the test suite helps it succeed, you've proven the value of those tests. Every time a missing test causes a bug, you feel the pain immediately — and you add the test. Your test suite grows in exactly the places that matter, guided by real feedback from real agentic work.

This is the virtuous cycle of agentic engineering: better tests lead to more autonomous agents, which lead to faster iteration, which gives you more time to write better tests. Each turn of the cycle makes the next one faster.

The engineers who will get the most out of agentic tools aren't the ones with the cleverest prompts or the most powerful models. They're the ones with the best test suites. A well-tested codebase is a force multiplier that compounds with every task you hand to an agent. An untested codebase is a tax that makes every task slower and riskier.

If you take one thing from this chapter, let it be this: before you optimise your prompts, before you experiment with new models, before you build elaborate orchestration — go write tests. Write them for the code your agents will touch. Make them fast, deterministic, isolated, and specific. That investment will pay off more than anything else in this book.

Your test suite is not overhead. It's the foundation everything else is built on.

# 9 Convention Over Configuration

I watched the same agent produce code worthy of a senior hire in one project and unmaintainable garbage in another — on the same afternoon, on the same machine, with the same model. The difference wasn't the prompt. It was the codebase.

Two projects. Same tech stack — TypeScript, React, PostgreSQL. Same size team. Same agent tooling.

Project A has a strict directory layout. Every API endpoint follows the same pattern: a handler file, a schema file, a test file, named identically. The database layer uses a consistent repository pattern. There's a `CLAUDE.md` at the root that describes the architecture in two pages. When an engineer points an agent at a ticket — "add a new endpoint for user notifications" — the agent reads the existing endpoints, follows the pattern, and produces a pull request that looks like a human on the team wrote it. The review takes three minutes.

Project B is the other kind. The codebase grew organically over two years. Some endpoints are in `routes/`, some are in `api/`, some are in `handlers/`. Half the database queries use an ORM, the other half use raw SQL. There's no consistent error handling — some functions throw, some return error objects, some return null. The agent looks at this codebase and does its best, but "its best" means picking whichever pattern it saw most recently. The resulting code works, technically, but it doesn't match anything around it. The review takes thirty minutes, most of it spent on "that's not how we do it here."

The difference between these projects isn't talent. It isn't tooling. It's convention.

There's an old principle in software: convention over configuration. Make the default the right thing. Reduce the number of decisions that need to

be made. When everyone follows the same patterns, the code explains itself.

This principle was always useful for human teams. For agentic engineering, it's essential.

If this sounds like I'm telling you to do the unglamorous work — writing documentation, enforcing naming conventions, maintaining project structure — I am. And I know how that feels. You didn't become an engineer to write style guides. But this is one of those moments where the craft asks you to care about something that used to feel like overhead, because the stakes have changed. Convention used to be a courtesy to your future self. Now it's the operating system your agents run on.

## 9.1 Why Agents Love Convention

An agent navigating an unfamiliar codebase does the same thing a new hire does: it looks for patterns. Where do tests live? How are files named? What's the import convention? Where's the config?

If your project follows strong conventions, the agent picks up the patterns quickly and produces code that fits in. If every file is a snowflake — different naming, different structure, different styles — the agent flounders. It doesn't know which pattern to follow, so it invents its own, and the result feels foreign.

Convention is *implicit context*. It's information the agent absorbs from the structure of your code without you having to explain it. When your test files always live next to the source files they test, named `foo.test.ts` beside `foo.ts`, the agent doesn't need to be told where to put a new test. It reads the directory, sees the pattern, and follows it. When your API handlers all export the same shape — a handler function, a schema, a set of middleware — the agent produces a new handler that exports exactly the same shape.

This is why opinionated frameworks have always been productive, and why they're *even more* productive in the agentic era. Rails, Next.js, Laravel — they impose a structure. That structure isn't just for humans. It's a language the agent speaks fluently.

## 9.2 The CLAUDE.md Deep Dive

A growing convention in agentic engineering is the `CLAUDE.md` file: a document at the root of your project that tells the agent what it needs to know. Not a README for humans. A briefing for agents.

This is one of the highest-leverage things you can do for your agentic workflow, and most teams either skip it or write a few vague lines and call it done. Let's talk about what a good one actually looks like.

A strong `CLAUDE.md` has five sections:

**Project overview.** Two or three sentences. What does this thing do, what's the tech stack, what's the deployment target. An agent that knows it's working on "a B2B SaaS billing platform built with Go and PostgreSQL, deployed to Kubernetes" makes fundamentally different decisions than one that's guessing.

**Build and test commands.** Every command the agent might need, listed explicitly. Not "check the Makefile" — the actual commands. Agents read documentation literally. If your `CLAUDE.md` says `make test`, the agent will run `make test`. If it says "run the tests" without specifying how, the agent will guess, and it might guess wrong.

**Architecture decisions.** The things that aren't obvious from the code. Why you chose a monorepo. Why the auth service is separate. Why you're using event sourcing for the order pipeline but simple CRUD for user management. These are the decisions that shape every piece of new code, and an agent that doesn't know about them will violate them constantly.

**Conventions.** Your style. How you name things. How you handle errors. What your import order looks like. Whether you prefer early returns or nested conditionals. The things that make code feel like it belongs in *this* project.

**Common pitfalls.** Where the bodies are buried. The database migration that must always be run before the seed. The environment variable that isn't in `.env.example` but is required for the payment flow. The test that's flaky on CI but not locally. Every project has these — write them down.

Here's what a real `CLAUDE.md` looks like for a medium-sized project:

```
# Project: Meridian (billing platform)

TypeScript monorepo (pnpm workspaces). React frontend, Express
API,
PostgreSQL with Drizzle ORM. Deployed to Fly.io.

## Commands
- `pnpm install` — install all dependencies
- `pnpm test` — run all tests (vitest)
- `pnpm test:api` — API tests only
- `pnpm test:web` — frontend tests only
- `pnpm lint` — eslint + prettier check
- `pnpm lint:fix` — auto-fix lint issues
- `pnpm db:migrate` — run pending migrations
- `pnpm db:generate` — generate migration from schema changes
- `pnpm dev` — start all services locally

## Architecture
- /packages/api — Express REST API
- /packages/web — React SPA (Vite)
- /packages/shared — shared types and utilities
- /packages/db — Drizzle schema, migrations, seed data

All API routes follow the pattern:
  routes/{resource}/index.ts — route definitions
  routes/{resource}/handlers.ts — request handlers
  routes/{resource}/schemas.ts — Zod validation schemas
  routes/{resource}/__tests__/ — tests for this resource

## Conventions
- All errors go through the AppError class (packages/api/src/
errors.ts)
- Never throw raw Error objects in API handlers
- Use Zod schemas for ALL request validation, no manual checks
- Database queries live in packages/db/src/queries/, not in
handlers
- Prefer early returns over deeply nested conditionals
- Import order: node builtins, external deps, internal packages,
relative

## Pitfalls
- The Stripe webhook handler uses raw body parsing — don't add
  json middleware to that route
```

```
- Test database must be created manually: createdb meridian_test
- The `BILLING_SECRET` env var isn't in .env.example (it's in
1Password)
- Flaky test: invoice.concurrent.test.ts — known race condition,
  skip locally if it blocks you
```

That's not long. It took maybe thirty minutes to write. But every agent session that reads this file starts with more context than most human developers get in their first week.

The most important discipline: keep it updated. An outdated `CLAUDE.md` is worse than none at all, because the agent will trust it. When you change a convention, update the file. When you add a new service, add it to the architecture section. When someone discovers a new pitfall, document it. Treat it like code — it lives in version control, it gets reviewed in PRs, it's part of the project.

Some teams go further: they put `CLAUDE.md` files in subdirectories too. A `packages/api/CLAUDE.md` that covers API-specific patterns. A `packages/web/CLAUDE.md` that documents the component library conventions. The deeper the agent goes into the project, the more specific context it gets. It's like an onion of documentation — broad context at the root, specific context as you drill down.

## 9.3 Conventions as Agent Memory

Conventions are the closest thing agents have to long-term memory. Once you see this, it changes how you think about all of them.

An agent's context window resets every session. It doesn't remember what it did yesterday. It doesn't remember the architectural discussion you had last week. It doesn't remember that the last three times it used `fetch` directly, you asked it to use the API client wrapper instead.

But your project structure persists. Your file naming persists. Your `CLAUDE.md` persists. Your linter rules persist. Your test patterns persist. Everything you encode into the shape of your project is there every time the agent opens its eyes.

This reframes conventions entirely. They're not just about consistency for humans. They're *external memory* for agents. Every convention you establish is a lesson the agent doesn't have to relearn.

Think about what happens without conventions. Session one: the agent creates a new endpoint and puts the error handling inline. You correct it in review: "We use the AppError class." Session two: the agent creates another endpoint and makes the same mistake, because it doesn't remember session one. Session three: same thing. You're having the same conversation over and over.

Now add one convention — a documented error handling pattern, enforced by a linter rule — and the problem disappears permanently. The agent reads the existing code, sees the pattern, follows it. The linter catches any deviation. The lesson is encoded in the project itself, not in anyone's memory.

This is why the most effective agentic teams obsess over conventions that seem tedious. Consistent import ordering. Strict file naming. Standard function signatures. These aren't aesthetic preferences — they're memory. They're the accumulated wisdom of the team, stored in a format that survives context window resets.

The analogy I keep coming back to: conventions are to agents what institutional knowledge is to organisations. A company where everything lives in one person's head is fragile. A company with strong processes and documentation is resilient. The same applies to codebases. A project where "how we do things" lives only in the senior developer's memory is fragile. A project where it's encoded in the structure, the tooling, and the documentation is resilient — for humans and agents alike.

## 9.4 Practical Conventions That Help Agents

**Consistent file naming.** If your API routes live in `routes/`, your models in `models/`, and your tests in `__tests__/` — the agent can navigate your project without a map. Name files after what they contain. Keep it boring.

**Formatters and linters.** Tools like Prettier, ESLint, `gofmt`, or Ruff aren't just for code style — they're guardrails that ensure agent-generated

code matches your project's standards automatically. Run them on save, run them in CI, make them non-negotiable.

**Standard project layout.** Whether it's the Go standard layout, Rails conventions, or your team's own structure — pick one and stick to it. A `justfile` or `Makefile` at the root that lists the standard commands (`build`, `test`, `lint`, `dev`) gives agents an entry point into any project.

**Small, focused files.** Agents work better with files under a few hundred lines. When a file contains one concern — one component, one module, one set of related functions — the agent can read it, understand it, and modify it without wading through unrelated code.

**Descriptive commit messages.** Agents read git history. A commit that says "fix bug" teaches nothing. A commit that says "fix race condition in session cleanup when WebSocket disconnects during auth" gives the agent context about what was important, what was fragile, and how the team thinks about problems.

**Consistent error handling.** Pick a pattern and enforce it everywhere. Custom error classes with error codes. A central error handler. A standard error response shape. When the agent encounters an error case in new code, it should have zero ambiguity about how to handle it. If your project uses three different error handling approaches in three different files, the agent will produce a fourth.

**Standard API response formats.** Every endpoint returns the same shape: `{ data, error, meta }` or whatever you choose. Status codes follow the same rules everywhere. Pagination works the same way on every list endpoint. This is the kind of consistency that agents excel at maintaining — but only if the convention is clear from the existing code.

**Logging conventions.** Structured logging with consistent fields. Every log entry includes a request ID, a timestamp, and a severity level. Every error log includes the error code and a stack trace. When the agent adds logging to new code — and it should — the logs should look identical to every other log in the system.

**Directory structure that tells a story.** An agent should be able to `ls` the top level of your project and understand the architecture. Here's what a well-structured project looks like from an agent's perspective:

```
src/
  routes/         # HTTP handlers — one file per resource
  services/       # Business logic — one file per domain concept
  repositories/   # Database access — one file per table/entity
  middleware/      # Express/Koa middleware
  utils/          # Pure utility functions
  types/          # Shared TypeScript types
  errors/         # Custom error classes
tests/
  fixtures/       # Test data factories
  helpers/        # Test utilities
migrations/       # Database migrations (numbered)
scripts/          # One-off and maintenance scripts
```

Every directory name is a noun. Every file within contains exactly what the directory name promises. There's nowhere to be confused about where new code should go. An agent reading this structure immediately knows: "I need to add a new database query, that goes in `repositories/`. I need a new endpoint, that starts in `routes/`."

Compare that to a project where database queries are scattered across handler files, business logic lives in utility functions, and there are three different directories that all seem to contain "helpers." The agent will put code in *a* place, but it probably won't be the *right* place.

## 9.5 The Convention Tax

Let's be honest about the cost. Establishing conventions takes time, and it feels like overhead — especially at the start.

Writing a `CLAUDE.md` takes an afternoon. Setting up linters and formatters takes a day. Refactoring an inconsistent codebase to follow a single pattern takes a week, maybe more. Enforcing conventions in code review means slowing down PRs that "work fine" but don't follow the standard.

There's a real temptation to skip all of this. The code works. The tests pass. Why spend time on naming conventions when there are features to ship?

The honest answer: if you're a solo developer building a throwaway prototype, the convention tax probably isn't worth it. Move fast, ship it, throw it away.

But the moment a second pair of eyes touches your codebase — human *or* agent — conventions start paying for themselves. And the returns compound.

The first time you establish a convention, it costs you an hour. Every subsequent time an agent follows that convention instead of asking you how to handle it, you save five minutes. Do the arithmetic: after twelve agent sessions, the convention has paid for itself. After a hundred sessions, you've saved *hours*.

This compounds across the team. Five engineers, each running multiple agent sessions per day, all benefiting from the same conventions. The initial investment of one engineer's afternoon saves the team hundreds of hours over the course of a year.

The teams that invest in conventions early look slower at first. They spend time on "boring" things — linter configs, project structure, documentation. But three months in, their agents are producing code that requires minimal review. Six months in, they're shipping twice as fast as the team that skipped the convention work. A year in, the gap is embarrassing.

This is the same dynamic as technical debt, just inverted. Convention is technical *wealth* — and like financial wealth, the earlier you start investing, the more dramatic the compounding.

The biggest mistake I see is teams that wait until their codebase is a mess before trying to establish conventions. That's the most expensive time to do it. The cheapest time is at the start of a project — but the second cheapest time is today.

## 9.6 The Compounding Effect

Every convention you establish, every standard you enforce, every piece of documentation you maintain — it all compounds. Each one makes agents

slightly more effective, slightly more autonomous, slightly more likely to produce code that fits your project like a glove.

But the compounding isn't just additive. Conventions interact. A consistent file naming convention *plus* a standard directory structure *plus* a `CLAUDE.md` that describes the architecture — together, these give the agent a mental model of the entire project. It knows where to find things, what to name them, and how they fit together. Remove any one of those three, and the agent's effectiveness drops disproportionately.

This is why half-hearted conventions are almost as bad as no conventions. A project with consistent file naming but chaotic directory structure sends mixed signals. The agent sees order in one dimension and chaos in another, and it doesn't know which signal to trust.

The engineers who invest in convention early don't just have cleaner codebases. They have codebases that are ready for the agentic future. Their agents produce better results. Their reviews are faster. Their iteration cycles are tighter.

Convention isn't glamorous work. It's the least exciting chapter in this book. But it's the one that makes every other chapter work. The sandboxes, the testing strategies, the multi-agent orchestration — all of it works better when the underlying codebase is consistent, documented, and conventional.

Your codebase is the environment your agents live in. Make it legible. Make it predictable. Make it boring. The agents will thank you by writing code that looks like it belongs.

# 10 Local LLMs vs. Commercial LLMs

A friend of mine — senior engineer at a Series B startup — pinged me on a Friday afternoon. "I just got our first real API bill. Twenty-two hundred dollars. For *March*." He'd been running Claude Code across his team of eight engineers, each of them iterating on features, debugging, refactoring. Nobody had set token budgets. Nobody was watching the meter. The agents worked beautifully, and the invoice was eye-watering.

That same week, I talked to an engineer at a healthcare company in Munich. She was running Llama 70B on a local GPU server because their patient data pipeline couldn't touch external APIs. Not "shouldn't" — *couldn't*. Their compliance team had made that clear in writing. She was getting decent results for focused tasks, but every time she needed complex multi-file reasoning, the model fell apart and she found herself doing the work manually.

These two stories are the bookends of the same question: where does the model run? It sounds like an infrastructure decision. It's really a decision about trust, cost, capability, and — increasingly — how you architect your entire agentic workflow.

## 10.1 Commercial LLMs: The Frontier

Commercial models — Claude, GPT, Gemini — are where the frontier lives. If you're doing serious agentic engineering, you've probably spent most of your time here. There are good reasons for that.

### 10.1.1 Context Windows

Context is everything for agents. An agent doesn't just read your prompt — it reads files, tool outputs, error messages, test results, and its own

previous reasoning. A single complex task can easily fill 50,000 tokens of context, and a multi-step debugging session can blow past 100,000.

Frontier commercial models offer context windows of 128K tokens and beyond. This matters enormously for agentic work because the agent needs to hold your codebase in its head. When context runs out, the agent starts forgetting earlier files it read, earlier decisions it made, earlier errors it saw. It degrades from a capable engineer into someone with amnesia.

Local models typically top out at 8K–32K context in practice, even if they technically support more on paper. The quality of attention degrades as you push toward the limit. A commercial model at 100K context is still reasoning well. A local model at 32K context is often losing the thread.

### 10.1.2 Tool Use Quality

Agents live and die by tool use. Reading files, writing code, running commands, searching codebases — these aren't optional extras. They're the core loop. And the quality of tool use varies dramatically between models.

Frontier commercial models have been specifically trained and tuned for tool calling. They format arguments correctly, they chain tool calls logically, they recover gracefully when a tool call fails. They know when to read a file before editing it. They know when to run tests after making changes.

Smaller models — including most local models — are less reliable here. They'll hallucinate file paths, forget to pass required arguments, call tools in the wrong order, or get stuck in loops calling the same failing tool over and over. The difference isn't subtle. On a complex task involving ten or fifteen tool calls, a frontier model might succeed 90% of the time where a local model succeeds 40%.

### 10.1.3 Instruction Following

When you tell a frontier model "only modify files in the `src/auth/` directory" or "don't change the public API, only the internal implementation," it generally listens. It follows system prompts, respects constraints, and stays within boundaries.

Smaller models drift. They'll start well, then gradually ignore your constraints as the conversation gets longer and the context fills up. This is a real problem for agentic work, where precision matters. An agent that "mostly" follows instructions will occasionally rewrite a file you didn't ask it to touch, or refactor something you explicitly told it to leave alone. In a sandboxed environment this is just annoying. Without a sandbox it's dangerous.

### 10.1.4 Choosing the Right Commercial Model

Not all commercial models are interchangeable, even at the frontier. Here's what I've found in practice:

**For complex multi-file refactoring and architectural work** — use the most capable model you can afford. This is where reasoning quality matters most. The cost difference between a mid-tier and top-tier model is a few dollars per task. The quality difference is often the difference between a clean diff and a mess you have to redo manually.

**For focused single-file tasks** — writing tests, implementing a well-defined function, fixing a clear bug — mid-tier models perform almost as well as top-tier ones, at a fraction of the cost. The task is scoped enough that the model doesn't need to juggle many concerns at once.

**For high-volume, low-complexity work** — generating boilerplate, formatting code, writing commit messages — the cheapest model that can follow instructions is the right choice. You'll run these tasks hundreds of times. The per-token savings compound.

The mistake I see most often is using a single model for everything. That's like driving a Formula 1 car to the grocery store. Match the model to the task.

## 10.2 Local LLMs: The Full Picture

Running a model locally — Llama, Mistral, Qwen, DeepSeek, Codestral, or any of the open-weight models via Ollama, llama.cpp, or vLLM — gives you something commercial APIs cannot: complete data sovereignty and zero marginal cost per token.

Your code never leaves your machine. You can feed it proprietary source, internal documents, production logs, customer data, API keys you accidentally left in a config — none of it crosses a network boundary. And once the model is downloaded, every inference is free. Run it a thousand times, run it all night, nobody sends you a bill.

But let's be honest about the experience, because the marketing around local models often isn't.

### 10.2.1 The Hardware Reality

The hardware question is the first one everyone asks, and the answer is less glamorous than the "run AI locally!" blog posts suggest.

**MacBook Pro with Apple Silicon (M2/M3/M4 Pro or Max, 32–64GB RAM).** This is the sweet spot for individual developers. You can run a 7B–14B parameter model comfortably, and a 32B model if you have the RAM. Inference will be noticeably slower than a commercial API — maybe 15–30 tokens per second for a 14B model, compared to 80+ from a commercial API. A 70B model will technically run on a 64GB machine but it'll be slow enough to test your patience. You'll be waiting 10–20 seconds for responses that take 2 seconds from an API.

**Dedicated GPU server (NVIDIA RTX 4090 or A100).** This is where local inference gets genuinely fast. A 4090 with 24GB VRAM can run a 14B model at near-commercial speeds. An A100 with 80GB can run a 70B model comfortably. But now you're maintaining hardware. You're dealing with CUDA drivers, VRAM management, and the occasional "why is my GPU fan screaming at 3am" incident.

**Consumer hardware (16GB MacBook Air, older machines, no discrete GPU).** You're limited to 7B models, and the experience will be mediocre. Inference is slow, the models are too small for serious agentic work, and you'll spend more time waiting than working. I wouldn't recommend this for anything beyond experimentation.

The honest summary: local models are practical for developers with a recent MacBook Pro or a dedicated GPU. Below that hardware threshold, you'll have a frustrating experience. Above it, you'll have a genuinely useful tool — just a different tool than a commercial API.

### 10.2.2 Where Local Models Shine

Local models aren't just "worse commercial models." There are workflows where they genuinely make more sense:

**High-frequency, low-stakes tasks.** Generating docstrings, writing commit messages, creating boilerplate code, formatting data. These tasks don't need a genius model — they need a fast, free model. Running them locally means you can fire-and-forget without thinking about cost.

**Sensitive codebases.** We'll cover this more in the privacy section, but this is a legitimate and growing use case. If your code can't leave your network, local models are the only option, and "good enough" is infinitely better than "not available."

**Offline development.** On a plane, on a train, in a bunker — your local model works without WiFi. This sounds minor until you're on a twelve-hour flight trying to debug something.

**Experimentation and learning.** When you're experimenting with agent frameworks, testing prompt strategies, or building custom tool integrations, burning API credits on trial-and-error feels wasteful. A local model lets you iterate freely.

### 10.2.3 Where Local Models Struggle

It's worth being specific about the failure modes, because "it's less capable" is too vague to be useful.

**Multi-file reasoning.** Ask a local 14B model to trace a bug across four files and a database schema, and it'll lose the plot. It might find the right file but misunderstand the interaction between components. This is the biggest practical gap.

**Long-horizon tasks.** Agentic tasks that involve many steps — read, plan, implement, test, debug, iterate — require the model to maintain coherent intent across a long context. Local models tend to drift. They forget the plan. They revisit decisions they already made. They get stuck in loops.

**Nuanced code review.** "This code is correct but the approach is wrong" is a judgement call that requires deep understanding. Local models tend

toward surface-level analysis — they'll catch syntax issues and obvious bugs but miss architectural problems.

**Complex tool orchestration.** When a task requires ten or more chained tool calls — reading a test file, running it, reading the error, finding the source file, understanding the context, making a change, running the test again — local models stumble more frequently at each step, and the error rate compounds.

None of this means local models are useless. A 32B model running locally can handle a surprising range of well-scoped tasks. But you need to scope the tasks accordingly. Asking a local model to do what a frontier model does is setting it up to fail.

## 10.3 The Cost of Agentic Work

A single agentic coding session can easily consume 100,000–500,000 tokens. This surprises people when they first see the numbers. Not because you're chatting — because the agent is *working*.

Consider what happens when an agent tackles a moderately complex bug fix. It reads three or four files to understand the context (maybe 8,000 tokens of input). It reasons about the problem (2,000 tokens of output). It reads two more files (4,000 tokens). It writes a fix (1,000 tokens). It runs the tests (tool call, 500 tokens of output). The tests fail. It reads the error (1,000 tokens). It revises the fix (1,500 tokens). It runs the tests again. They pass. It reads the diff to double-check (1,000 tokens). Total: maybe 25,000 tokens for a straightforward bug fix.

Now consider a complex refactoring task. The agent might read twenty files, generate a plan, implement changes across eight files, run the test suite four times, debug two regressions, and write new tests. That's easily 200,000 tokens. At frontier model pricing, that's somewhere between $3 and $15 depending on the model and the input/output ratio.

Here are rough cost ranges I've seen in practice, based on using frontier commercial models:

- **Simple bug fix or small feature:** $0.30–$1.00

- **Moderate feature with tests:** $2–$5
- **Complex multi-file refactor:** $5–$15
- **Large feature implementation:** $15–$40+
- **Exploratory debugging session that goes long:** $10–$30

Multiply these numbers by a team of engineers, each running several agentic tasks per day, and you're looking at real money. My friend's $2,200 bill? That's about right for eight active engineers.

### 10.3.1 Strategies for Managing Cost

The solution isn't to stop using agents. It's to be smart about it.

**Set token budgets.** Most agent frameworks let you set a maximum token limit per task. This isn't just cost control — it's also a quality signal. If an agent burns 500,000 tokens on a task that should take 50,000, something has gone wrong. The agent is stuck, looping, or misunderstanding the task. A budget forces it to fail fast rather than spiral.

**Use model routing.** Don't send every task to the most expensive model. We'll dig into this more in the next section, but the short version: use a cheap, fast model for exploration and code reading, and a capable expensive model for reasoning and implementation. This alone can cut costs by 50–70%.

**Cache aggressively.** If your agent framework supports prompt caching, turn it on. Much of an agent's context is repeated between turns — the same system prompt, the same project context, the same recently-read files. Caching avoids re-processing those tokens on every turn.

**Scope tasks tightly.** A well-scoped task ("fix the timezone bug in `billing/invoice.py`, the test is in `tests/test_invoice.py`") is cheaper than a vague one ("fix the billing bugs"). Scoping isn't just good engineering — it's cost control. The agent reads fewer files, makes fewer exploratory tool calls, and converges faster.

**Review your failures.** When a task fails or takes way too many tokens, figure out why. Was the prompt vague? Was the agent missing context it needed? Was the model not capable enough for the task? Each failure is a tuning opportunity.

### 10.3.2 Managing Costs Across a Team

Individual cost control is one thing. Managing spend across a team of eight or twelve engineers, each running agents all day, is a different problem entirely. It's the difference between watching your own diet and running a restaurant kitchen.

**Per-engineer budgets.** Set a monthly token budget per engineer — not to restrict, but to make costs visible. When everyone can see their own spend, behaviour changes naturally. People start scoping tasks more carefully, choosing the right model for the job, killing runaway sessions earlier. A reasonable starting point: $200–400 per month per engineer for a team actively using agents. Some engineers will consistently come in under budget. Others will spike during intense debugging weeks. That's fine — the point is awareness, not enforcement.

**Dashboard and visibility.** You can't manage what you can't see. Track spend per engineer, per project, per task type. Most API providers give you usage breakdowns by API key, and if you assign keys per engineer or per project, the data is already there. Pipe it into a dashboard the team can see — even a shared spreadsheet works to start. The engineer who discovers they burned $80 on a single debugging session will naturally start scoping tasks more tightly next time. You don't need to have a conversation about it. The number does the talking.

**Alerts and circuit breakers.** Set alerts at 50% and 80% of monthly budget so nobody gets surprised. More importantly, set a hard per-session token limit as a circuit breaker. If a single agent session exceeds 500K tokens, something has gone wrong — the agent is looping, the task is too vague, or the model is struggling with something beyond its capability. Kill it and investigate. This is what prevents the "$2,200 surprise bill" scenario from the opening of this chapter. You don't need to catch every runaway session manually. Automated limits do the job.

**The ROI conversation.** At some point, someone in management will ask why the API bill is five figures. You need the maths ready. Frame it like this: a senior engineer costs $150K per year fully loaded. If agents make them 30% more productive, that's $45K of value. The agent costs run $3–4K per year per engineer. The ROI is roughly 10x. Even if you're conservative — say 15% productivity gain instead of 30% — the numbers

still work comfortably. The harder part is measuring that productivity gain precisely. You probably can't, not with scientific rigour. But you can track concrete signals: time to merge PRs, number of tasks completed per sprint, reduction in context-switching. Pair those metrics with the cost data and you have a story that finance can work with.

**Cost as a quality signal.** High token consumption on a task isn't just expensive — it's a signal that something went wrong. The task was poorly scoped, the context was insufficient, or the model wasn't capable enough for the job. Start tracking cost per task type over time. If costs for a particular category trend upward, investigate — your prompts may have drifted, or the codebase has grown complex enough to need a different approach. If costs trend downward, that's your team getting better at agentic engineering. They're writing tighter prompts, choosing better models, and scoping tasks more effectively. Cost data, used well, becomes a mirror for team skill.

## 10.4 Privacy and Compliance

Some code genuinely cannot leave the building. This isn't paranoia — it's law.

Government contractors working on classified or export-controlled projects can't send source code to third-party APIs, full stop. The data residency requirements aren't suggestions. They come with criminal penalties.

Healthcare companies handling patient data are bound by HIPAA, GDPR, or equivalent regulations. If your code touches patient records — even test fixtures with realistic fake data that a compliance officer might squint at — you need to think carefully about what goes over the wire.

Financial institutions have their own maze of regulations. SOX compliance, PCI-DSS, internal audit requirements — the specifics vary, but the theme is consistent: data stays inside controlled boundaries.

And then there's plain old competitive secrecy. Your proprietary algorithms, your secret sauce, your competitive advantage — sending that to someone else's servers requires trusting that they won't train on it, won't

log it, won't get breached. Most commercial API providers offer strong contractual guarantees about this. But "strong contractual guarantees" and "impossible to breach" are different things, and some security teams aren't willing to accept the gap.

For all these cases, local models aren't a nice-to-have. They're the only option.

The trade-off is real: you're accepting reduced model capability in exchange for absolute data control. A local 32B model doing a mediocre job on your classified codebase is infinitely more useful than a frontier model you're not allowed to use. And for focused, well-scoped tasks — the kind you should be writing anyway — the quality gap is often smaller than you'd expect.

### 10.4.1 The Middle Ground: Private Deployments

It's worth noting there's an emerging middle path. Cloud providers now offer private model deployments — dedicated instances of frontier models running inside your VPC, with contractual guarantees that your data stays in your boundary and isn't used for training. AWS Bedrock, Azure OpenAI, Google Cloud's Vertex AI all offer versions of this.

These aren't cheap. You're paying for dedicated compute, not shared API infrastructure. But for large organisations that need frontier capability and strict data control, this is increasingly the answer. You get commercial model quality with local model privacy guarantees — for a price.

## 10.5 Model Routing in Practice

The real question isn't "local or commercial?" It's "which model, for which part of the workflow?"

A sophisticated agentic setup routes different parts of the workflow to different models. This isn't theoretical — teams are doing this today, and it's the direction the ecosystem is moving.

### 10.5.1 The Routing Pattern

Think about what an agent actually does during a typical task:

1. **Exploration** — reading files, searching the codebase, understanding structure. This is high-volume, low-reasoning work. The model needs to decide which files to read and extract relevant information, but it's not doing deep thinking.
2. **Planning** — analysing the problem, considering approaches, deciding on a strategy. This requires strong reasoning. It's the part where model quality matters most.
3. **Implementation** — writing the actual code changes. This requires good code generation and the ability to follow the plan from step 2.
4. **Verification** — running tests, reading errors, deciding if the work is done. Moderate reasoning, heavy tool use.
5. **Iteration** — if verification fails, going back and adjusting. This requires understanding the failure and connecting it back to the implementation.

Not all of these steps need the same model. Step 1 can be handled by a small, fast, cheap model — even a local one. It's reading files and reporting what's in them. Step 2 is where you want the frontier model — this is the expensive reasoning that justifies the API cost. Steps 3–5 can often be handled by a mid-tier model, since the hard thinking is already done and the model is executing a plan.

### 10.5.2 What This Looks Like

In practice, model routing can be as simple as configuring your agent framework to use different models for different task types:

```
# Pseudocode — the exact config depends on your framework
exploration_model: "local/qwen-14b"       # Free, fast, good
enough for reading
reasoning_model: "claude-sonnet"           # Frontier reasoning
for planning
implementation_model: "claude-haiku"       # Fast, cheap, follows
plans well
```

Or it can be dynamic — a lightweight classifier that looks at the current step and routes accordingly. Some frameworks are starting to build this in natively. Others require you to wire it up yourself.

The economics are compelling. If 60% of an agent's tokens are spent on exploration and simple tasks, and you route those to a model that's 10x cheaper, you've cut your overall cost by more than half without any reduction in quality on the parts that matter.

### 10.5.3 Routing for Privacy

Model routing also solves the privacy problem more gracefully than an all-or-nothing approach.

Say you're building a healthcare application. The data models and business logic touch patient data — that needs to stay local. But the frontend components, the build configuration, the CI pipeline? Those don't contain sensitive data. There's no reason you can't use a frontier commercial model for the non-sensitive parts while routing sensitive work to a local model.

This requires tooling that's aware of sensitivity boundaries — which files can go external, which can't. That tooling is still maturing, but the pattern is clear. Instead of "all local" or "all commercial," you get "local where it matters, commercial everywhere else."

## 10.6 Getting Started Without Paying the Farm

You don't need a budget to start learning agentic engineering. You need a laptop and an evening. Here's a practical ramp from zero cost to real productivity.

### 10.6.1 The Free Tier

Most commercial providers offer free tiers or trial credits. As of early 2026, you can get started with Claude, GPT, or Gemini without entering a credit card. The free tiers are rate-limited and context-restricted, but they're more than enough to learn the fundamentals — write your first agentic prompt, watch an agent iterate on a test failure, get a feel for the feedback loop.

Pair a free-tier API key with an open-source agent framework — Claude Code's free tier, or Aider with its free-model support — and you have a complete agentic setup at zero cost. It won't be fast. It won't handle

complex multi-file work. But it will teach you everything in chapters two through five of this book without spending a cent.

### 10.6.2 The Local-First Path

If you have a MacBook Pro with 16GB or more of RAM, you can run useful models locally for free, forever.

Install Ollama — it takes one command. Pull a model — `ollama pull qwen2.5-coder:14b` takes a few minutes. Point an agent framework at it. You now have an agentic coding setup with no API costs, no rate limits, and no data leaving your machine.

The experience won't match a frontier commercial model. Context windows are smaller. Multi-file reasoning is weaker. Tool use is less reliable. But for focused, well-scoped tasks — "write tests for this function," "refactor this class to use dependency injection," "add error handling to this endpoint" — a local 14B model is surprisingly capable. And because it's free, you can iterate without watching the meter.

A practical starting stack for zero-cost agentic engineering:

- **Ollama** for model serving (free, local)
- **Qwen 2.5 Coder 14B** or **DeepSeek Coder V2** for code tasks
- **Aider** or **Claude Code** (with local model support) as the agent framework
- A project with a test suite (the agent needs feedback)

That's it. No subscription. No API key. No cloud compute. You'll hit the ceiling eventually — a task that needs a bigger context window, or multi-file reasoning that the local model can't handle. That's when you reach for a commercial model. But the free setup will carry you further than you expect.

### 10.6.3 The Sweet Spot: $20/Month

When you're ready to spend money, the most cost-effective setup I've found is a combination of local models for exploration and a commercial subscription for reasoning.

Most commercial providers offer a developer plan in the $20/month range that includes a generous token allowance. Use this for the tasks that actually need frontier capability — complex debugging, multi-file refac-

toring, architectural planning. Use your local model for everything else — reading code, generating boilerplate, writing commit messages, running through test iterations.

This hybrid approach typically covers 80-90% of a solo developer's agentic work. The local model handles the high-volume, low-reasoning tasks. The commercial model handles the moments that need genuine intelligence. Your monthly bill stays predictable, and you're not choosing between quality and cost — you're using each where it makes sense.

### 10.6.4 Scaling Up Intentionally

The mistake is starting with the most expensive option and optimising later. Start free. Learn the workflows. Understand where model quality actually matters and where "good enough" is good enough. Then spend money on the specific gaps that free tools can't fill.

By the time you're spending real money on API calls, you'll know exactly what you're paying for — and more importantly, what you're *not* paying for. That knowledge is worth more than any amount of credits.

## 10.7 The Landscape Is Shifting

Six months from now, the specifics in this chapter will be out of date. The cost numbers will change. The capability gaps will narrow. A new model will come out that reshuffles the rankings. This is the one prediction I'll make with confidence.

What won't change is the framework for thinking about it. You'll still need to evaluate models along the same axes: capability, cost, privacy, speed, and reliability. You'll still need to match the model to the task rather than picking one model and using it for everything. You'll still need to stay flexible.

The engineers I see doing the best work aren't loyal to any particular model or deployment approach. They're pragmatists. They use the frontier commercial model when the task demands it, a mid-tier model when it's good enough, and a local model when privacy or cost requires it. They measure what works. They switch when something better comes along.

Don't get religious about this. The model is a tool. The skill is in knowing which tool to reach for — and that skill transfers regardless of which models exist six months from now.

# 11 Prompting as Engineering

There's no secret syntax. No magic incantation that makes an agent produce perfect code. Prompting is *communication* — and you already know how to communicate.

If you've ever written a good bug report, you know how to prompt. If you've ever written a design doc that a teammate could implement without asking you twenty questions, you know how to prompt. If you've ever filed a Jira ticket that didn't come back as something completely different from what you wanted — you know how to prompt.

The skills transfer directly. Clarity, specificity, context, constraints. The same things that make human collaboration efficient make agent collaboration efficient. The difference is that agents won't ask clarifying questions when your prompt is vague. They'll just guess. And they'll guess confidently.

This is where a lot of experienced engineers quietly struggle. You've spent years building the skill of *doing* — writing code, debugging, building systems. Now the skill that matters is *articulating* — explaining what you want with enough precision that someone else can do it. It's a different muscle. And it can feel, in the early days, like a demotion. It's not. But the discomfort is real, and pretending it isn't doesn't help.

## 11.1 The Anatomy of a Good Task Prompt

A good prompt has three parts: *what* you want done, *why* it matters, and *how* success looks. Most people only provide the first, and even that is usually vague.

Consider the difference:

**Bad:** "Fix the auth bug."

This tells the agent almost nothing. Which auth bug? Where does it manifest? What's the expected behaviour? The agent will go hunting through your codebase, form a theory about what you might mean, and apply a fix that might be entirely wrong. You've turned a five-minute fix into a twenty-minute review of something you didn't ask for.

**Good:** "The login endpoint returns 401 for valid tokens when the session cache is cold. The bug is likely in `middleware/auth.go` in the `validateSession` function. The test in `auth_test.go:TestColdCacheLogin` reproduces it. Fix the bug and make sure all existing auth tests still pass."

This is a different animal entirely. The agent knows the symptom, the suspected location, and how to verify the fix. It can go straight to the relevant code, understand the problem, and validate its solution — all without guessing.

The pattern is simple. *What* is broken or needed. *Where* to look. *How* to verify. Every minute you spend making your prompt precise saves you five minutes reviewing the wrong output.

## 11.2 Constraint Specification

Telling an agent what to do is only half the job. Telling it what *not* to do is equally important.

Agents are eager. They optimise for solving the problem you described, and they'll happily refactor your entire module, add three new dependencies, and change the public API to do it. That's not malice — it's an optimiser doing what optimisers do. Your job is to set the boundaries.

Useful constraints look like this:

- "Don't modify the public API surface."
- "Keep the existing test structure — add new test cases, don't reorganise."
- "Don't add new dependencies."
- "Stay within the existing error handling patterns in this codebase."
- "Don't change any files outside of the `services/` directory."

Think of constraints as the guardrails on a bridge. The agent can drive anywhere within the lanes, but it can't go over the edge. Without guardrails, you get creative solutions that technically work but create maintenance nightmares. With them, you get solutions that fit your codebase like they were always there.

The more experienced you become with agentic engineering, the more your prompts are defined by their constraints rather than their instructions. You learn which freedoms lead to good outcomes and which lead to chaos.

## 11.3 Task Decomposition

A common mistake: asking an agent to build something large in a single prompt. "Build a user dashboard with real-time metrics, role-based access, and export to CSV." That's not a prompt — that's a project. And projects need to be broken into tasks.

Task decomposition is the practice of splitting big requests into small, *verifiable* steps. Each step has a clear input, a clear output, and a clear way to check whether it worked.

Instead of "build a user dashboard," you write:

1. Create the data model for dashboard metrics in `models/dashboard.go` with the schema defined in the design doc. Write unit tests for the model validation.
2. Build the API endpoint `GET /api/dashboard` that returns metrics for the authenticated user. Write integration tests.
3. Add role-based filtering so admin users see all metrics and regular users see only their own. Update the existing tests to cover both roles.
4. Build the React component that displays the dashboard data. Use the existing `DataTable` component for the metrics grid.

Each step is a self-contained prompt. Each has a verifiable outcome. Each builds on the verified output of the previous step. If step two goes sideways, you catch it before you've wasted time on step three.

This isn't just good prompting — it's good engineering. You're applying the same decomposition skills you'd use when planning a sprint or break-

ing down a pull request. The unit of work is small enough to review, small enough to test, and small enough to throw away if it's wrong.

## 11.4 The Prompt as a Spec

The best prompts I've seen read like miniature design documents. They describe the desired outcome, not the implementation steps. They list the constraints. They define acceptance criteria. They provide just enough context for the agent to make good decisions without drowning it in irrelevant information.

Here's what a prompt-as-spec looks like:

*"Add rate limiting to the* `/api/search` *endpoint. Use the existing* `RateLimiter` *middleware in* `middleware/ratelimit.go`*. Set the limit to 100 requests per minute per authenticated user, and 20 per minute for unauthenticated requests. Return a 429 status with a* `Retry-After` *header when the limit is exceeded. Add tests for both the authenticated and unauthenticated paths, including the edge case where a user hits exactly the limit. Don't modify the rate limiter middleware itself — just configure and apply it."*

That's a spec. An agent can implement this without ambiguity. A human reviewer can check the result against the requirements. The desired outcome is clear, the constraints are explicit, and the verification criteria are defined.

Writing prompts this way takes practice. It also takes discipline — the discipline to think through what you actually want before you start typing. But that discipline pays dividends. A well-specified prompt produces a result you can merge. A vague prompt produces a result you have to rewrite.

## 11.5 Iteration Over Perfection

Your first prompt won't be perfect. That's fine. Prompting is an iterative process, and the skill isn't in writing the perfect prompt — it's in *reading*

*the output*, understanding where the communication broke down, and refining.

When an agent produces something wrong, resist the urge to blame the tool. Instead, ask yourself: what did I fail to communicate? Did I leave out a constraint? Was the context insufficient? Did I assume knowledge the agent didn't have?

This is debugging — but instead of debugging code, you're debugging your own communication. The error message is the agent's output. The stack trace is your prompt. Somewhere in there is the miscommunication, and finding it makes your next prompt better.

Experienced agentic engineers develop a feedback instinct. They see the agent's output and immediately know which part of their prompt caused the deviation. "Ah, I said 'handle errors' but didn't specify *which* errors or *how* to handle them. Of course it threw a generic catch-all in there."

Each iteration tightens the loop. First prompt gets you 70% of the way. A follow-up correction gets you to 90%. A final refinement gets you to done. Over time, your first prompts get better, and you need fewer iterations. But you never need zero.

## 11.6 Voice-Driven Development

Most of us prompt by typing. That makes sense — we are engineers, we live in text. But there is another input channel that is faster, more natural, and surprisingly underused: your voice.

Modern speech-to-text has reached the point where you can speak a prompt into your terminal and have it transcribed with near-perfect accuracy. Tools like Whisper, macOS Dictation, and SuperWhisper let you talk to your agent instead of typing. The result is the same — text goes in, code comes out. But the experience is fundamentally different.

Here is why: typing and speaking are different modes of thinking. When you type, you edit as you go. You delete a word, rephrase, backspace, restructure. The text you produce is *polished* — you had time to smooth out the rough edges before it left your fingers. Speaking doesn't give you

that luxury. When you speak, you commit. The words leave your mouth and they're gone. There is no backspace.

This sounds like a disadvantage. It is actually a training ground.

When you speak a prompt, you are forced to organise your thoughts *before* you open your mouth. You cannot rely on the crutch of editing mid-sentence. You have to know what you want, structure it mentally, and deliver it clearly — in real time. The first few times you try this, you will ramble. You will say "uh" and "um" and circle back and contradict yourself. The agent will receive a messy transcript, and the output will reflect that mess.

But something happens if you keep doing it. You get better. Not just at prompting — at *speaking clearly about technical problems*. You develop the ability to describe a bug, a feature, or a refactoring task in a single coherent stream. You learn to front-load context, state constraints early, and finish with a clear ask. You stop rambling because rambling produces bad results.

This skill transfers everywhere. Standup meetings. Architecture discussions. Pair programming. Incident calls. Every situation where you need to articulate a technical idea clearly, under time pressure, without the safety net of a text editor. Voice-driven development is not just a faster way to prompt — it is practice for every technical conversation you will ever have.

There is also a practical speed advantage. Most people speak at 130 words per minute. Most people type at 40 to 80. For the kind of high-level, intent-driven prompts that produce the best agent output — "here is the problem, here is the context, here is what I want, here is what I don't want" — speaking is simply faster. You spend less time on input and more time reviewing output.

Try it for a week. Pick a speech-to-text tool, wire it into your workflow, and speak your prompts instead of typing them. The first day will feel awkward. By the third day, you will notice your spoken prompts getting tighter. By the end of the week, you will notice your *spoken communication in general* getting tighter.

It is also worth noting that speech-to-text models can run entirely on your machine. NVIDIA's Parakeet family of models — compact, high-accuracy ASR models — run locally without any cloud dependency. Tools like SuperWhisper and whisper.cpp do the same using OpenAI's Whisper weights. A modern MacBook can run these models in near real-time with accurate transcription and low latency. You do not need a cloud service to turn speech into text — the local tooling is already there.

The agents don't care whether your prompt was typed or spoken. But *you* will be a clearer thinker for having spoken it.

## 11.7 Visual Context: When Words Aren't Enough

Not everything is easy to describe in text. A broken layout, a weird rendering glitch, an error dialog with a stack trace — sometimes the fastest way to communicate what you're seeing is to *show* it.

Modern LLMs are multimodal. They can read screenshots, diagrams, photos of whiteboards, and error messages captured from your screen. This is not a novelty feature — it is one of the most underused tools in the agentic engineering workflow.

Here is a workflow I use daily: I see a bug on my phone — a layout that's broken, a modal that's off-centre, a form that's eating input. I screenshot it on iOS, and thanks to Universal Clipboard, I paste it directly into my terminal session on my Mac. The agent sees what I see. No need to describe "the button is overlapping the header on mobile viewport" — the screenshot *is* the description.

This matters because visual bugs are notoriously hard to describe in text. You end up writing three paragraphs about padding and z-index when a single screenshot communicates the problem instantly. The agent can see the broken state, reason about what's wrong, and propose a fix — often faster than you could finish typing the description.

But it goes beyond bug reports. Some common visual context workflows:

- **Error screenshots.** A browser console full of red, a terminal stack trace, a deployment dashboard showing failed health checks. Screenshot

it, paste it, ask the agent to diagnose. This is especially useful when error messages are long or contain formatting that's painful to copy as text.

- **Design references.** A Figma mockup, a sketch, a competitor's UI you want to approximate. Paste the image and say "make our settings page look like this." The agent can extract layout structure, colour choices, and component hierarchy from a visual reference.
- **Debugging visual state.** "Why does this page look wrong?" is a terrible prompt. A screenshot of the page *plus* "why does this page look wrong?" is a great one. The agent can compare what it sees against the expected layout and identify CSS issues, missing data, or rendering bugs.
- **Whiteboard photos.** After an architecture discussion, snap a photo of the whiteboard and paste it in. The agent can read the boxes, arrows, and labels, and help you translate that sketch into code structure, API definitions, or documentation.

The iOS-to-Mac clipboard pipeline deserves special mention because it removes all friction from this workflow. You don't need to save the screenshot, AirDrop it, find it in Finder, and drag it into a tool. You see the problem, you capture it, you paste it. Three seconds from "that's broken" to "the agent is looking at it." That speed matters because it keeps you in flow. Any extra steps — even thirty seconds of file management — create enough friction that you default back to typing a text description, which is slower and less precise.

The key insight is that *context is not just text.* When we talked about context being the most important ingredient in agentic work, we were talking about all forms of context — code files, documentation, test output, *and* visual state. A screenshot is worth a thousand tokens, and agents that can see are dramatically more useful than agents that can only read.

If you're not already using visual context in your agentic workflow, start. Screenshot your bugs. Paste your error messages. Share your design references. The agents can see now. Let them.

## 11.8 Anti-Patterns

Some prompting habits consistently produce poor results. Learn to recognise them.

**Being too vague.** "Make this code better." Better how? Faster? More readable? More maintainable? The agent will pick *something* to improve, and it probably won't be the thing you had in mind. If you can't articulate what "better" means, you're not ready to prompt.

**Being too prescriptive.** The opposite failure. "On line 47, change the variable name from `x` to `count`, then add an if statement on line 48 that checks if count is greater than zero, then…" You're writing the code in English and asking the agent to translate. That's slower than writing the code yourself. Describe the *outcome*, not the keystrokes.

**Context dumping.** Pasting your entire codebase, all your design docs, and a transcript of your last three team meetings into the prompt. More context is not always better. Irrelevant context is noise, and noise drowns signal. Give the agent what it needs — file paths, function names, the specific behaviour you want — and trust it to explore from there.

**Kitchen-sink prompts.** "Fix the auth bug, also refactor the database layer, and while you're at it update the README and add TypeScript types to the API client." These are four separate tasks jammed into one prompt. The agent will attempt all of them, do none of them well, and produce a diff so large that reviewing it takes longer than doing the work yourself. One prompt, one task.

**Assuming shared context.** "Do it the same way we did the payments module." The agent doesn't remember your last session. It doesn't know what "we" decided in standup. Every prompt starts from scratch. Provide the context explicitly, every time.

## 11.9 Prompting Is a Skill

Prompting is not a parlour trick. It's not about discovering the one weird phrase that unlocks better output. It's a communication skill —

and like all communication skills, it improves with practice, feedback, and deliberate attention.

The engineers who get the most out of agentic tools are the ones who treat prompting with the same rigour they apply to writing code. They think before they type. They specify before they implement. They verify before they move on.

That's not a new skill. That's just engineering.

# 12 Multi-Agent Orchestration

One agent is powerful. Multiple agents working together is something else entirely.

Think of it this way. A single carpenter can build a shed. But a house? You need an electrician, a plumber, a roofer — specialists working in parallel, each focused on what they do best, coordinating just enough to stay out of each other's way. The house goes up faster and the work is better than one person trying to do everything.

Agentic engineering works the same way. A single agent on a complex task will grind through it sequentially — plan, implement, test, debug, iterate. It works, but it's slow. Three agents, each handling a well-scoped piece of the problem, can finish in a third of the time. Sometimes less, because focused agents make fewer mistakes than one agent juggling too many concerns.

But there's a catch. Parallel agents require *coordination*. And coordination has a cost. This chapter is about when to pay that cost, and how to pay it well.

## 12.1 Decomposition Strategies

The hardest part of multi-agent work isn't running multiple agents. It's deciding how to split the work.

The golden rule: tasks must have *minimal coupling*. If agent A's work depends on agent B's output, they can't run in parallel — they're sequential, and you should treat them that way. The art is finding seams in your work where you can cut cleanly.

There are three reliable decomposition patterns.

**By layer.** One agent handles the backend API, another builds the frontend component, a third writes the tests. This works well because

layers have natural boundaries — a defined interface between them. As long as you agree on the API contract up front, each agent can work independently.

**By feature.** If you're building three independent features, give each to a separate agent. This is the simplest decomposition. The features touch different files, different directories, different concerns. Merge conflicts are rare.

**By concern.** One agent refactors, another writes tests for the refactored code, a third updates the documentation. This is a sequential pattern — more pipeline than parallel — but it lets each agent focus on a single type of thinking. The refactoring agent doesn't have to context-switch into test-writing mode. It just refactors, and hands off.

The decomposition you choose depends on the shape of the work. But the principle is constant: *find the seams, cut along them, minimise the surface area where agents need to agree.*

## 12.2 Branch-per-Agent

Every agent gets its own branch. We covered this in the Git chapter. In multi-agent work, it becomes absolutely essential.

But branches alone aren't enough. Two agents on different branches sharing the same working directory will fight over the filesystem — they'll overwrite each other's files, corrupt each other's builds, break each other's test runs. You need *worktrees*.

Each agent gets its own git worktree: a separate directory, on its own branch, with its own copy of the codebase. The agents share history but nothing else. They can build, run tests, install dependencies, and make a mess — all without affecting each other.

The setup is quick:

```
git worktree add ../project-api agent/api-endpoint
git worktree add ../project-frontend agent/frontend-component
git worktree add ../project-tests agent/integration-tests
```

Three directories. Three branches. Three agents. Full isolation. This is the sandbox model from earlier chapters made concrete for multi-agent work. When an agent finishes, you review its branch, merge if it's good, and remove the worktree. If the work is bad, you throw the whole thing away. Zero cost.

## 12.3 The Handover Pattern

Not all multi-agent work is parallel. Sometimes agents work *sequentially*, each building on the last. Agent A plans. Agent B implements. Agent C reviews.

This is the handover pattern, and it's powerful — but only if the handover itself is clean.

The problem with sequential agents is context loss. Agent A has a rich understanding of the problem by the time it finishes planning. Agent B starts cold. If you just tell Agent B "implement the plan," it's going to miss nuance. It'll make different assumptions. It'll solve a slightly different problem than the one Agent A planned for.

The fix is *structured handover*. Agent A doesn't just plan — it produces an artefact that captures its reasoning. This can take several forms:

- **A summary document.** A markdown file dropped into the repo: `PLAN.md`. It describes what needs to happen, what trade-offs were considered, what was rejected and why. Agent B reads this before writing a line of code.
- **A set of files.** Agent A creates stub files — empty functions with docstrings, interface definitions, type signatures. Agent B fills them in. The stubs *are* the handover.
- **A structured prompt.** Agent A's output becomes Agent B's input, formatted as a detailed task description with acceptance criteria. You paste it directly into Agent B's context.

The key is that the handover must be *explicit and complete*. No implicit assumptions. No "the next agent will figure it out." Every decision, every constraint, every edge case — written down.

This takes discipline. But it's the same discipline you'd apply when writing a ticket for a human colleague on a different continent and a different timezone. If you wouldn't trust a verbal handover, don't trust an implicit one between agents.

## 12.4 The Merge Problem

This is where multi-agent work gets genuinely hard.

Three agents worked in parallel. Each produced a clean, tested branch. Now you need to merge them all into main. Sometimes this is painless — three branches touching completely different files merge without a single conflict. Beautiful.

Sometimes it's not. Agent A changed the database schema. Agent B added a migration that assumes the old schema. Agent C updated the API handler that both A and B also touched. Now you have a three-way conflict and no single agent has the full picture.

There are three strategies for dealing with this.

**Prevention.** The best merge conflicts are the ones that never happen. When you decompose work, think about file ownership. Assign different directories, different modules, different files to different agents. If agents never touch the same file, they can never conflict. This is the cheapest strategy and you should use it whenever possible.

**The merge agent.** When conflicts do arise, spin up a new agent whose sole job is to merge. Give it the branches, the conflicts, and the context from each agent's work. A good merge agent can resolve most conflicts automatically — it reads both sides, understands the intent, and produces a coherent result. It's like a senior engineer who sits down with two PRs and figures out how they fit together.

**Human review.** Sometimes the conflict is semantic, not syntactic. The code merges cleanly but the *logic* contradicts itself. Two agents made incompatible design decisions. No automated merge will catch this. This is where you earn your keep as the engineer. Review the branches before merging. Read the diffs side by side. Make sure the pieces actually fit.

In practice, you'll use all three. Prevent what you can. Automate the rest. Review everything.

## 12.5 Orchestration Overhead

More agents is not always better.

Every additional agent adds coordination cost. You have to decompose the work. Set up worktrees. Define interfaces. Handle merges. Review multiple branches instead of one. For a task that takes a single agent thirty minutes, spinning up three agents might take forty-five — twenty minutes of parallel agent work plus twenty-five minutes of your time orchestrating.

The break-even point is higher than you think. In my experience, multi-agent orchestration starts paying off when the total task would take a single agent *at least two hours*. Below that, the overhead eats the gains.

There are other costs too. Context gets fragmented. Each agent sees only its piece. No single agent understands the whole system the way one agent working alone would. This can lead to inconsistencies — different naming conventions, different error handling patterns, different assumptions about shared state.

The skill isn't running as many agents as possible. The skill is knowing *when* to parallelise and when a single focused agent is the right tool. A crew of five isn't always faster than one experienced sailor who knows the boat.

## 12.6 A Practical Example

Let's make this concrete. You're building a web app and you need to add a new feature: user notifications. Users should see a bell icon with a count, click it to see a list, and mark notifications as read. You need an API, a frontend component, and tests.

This is a textbook case for three parallel agents.

**Step 1: Define the interface.** Before spinning up any agents, you spend five minutes writing down the API contract. `GET /api/notifications`

returns a list. `PATCH /api/notifications/:id` marks one as read. The notification object has `id`, `message`, `read`, `created_at`. Write this in a shared document or a stub file. This is the contract all three agents work against.

**Step 2: Create the worktrees.**

```
git worktree add ../app-api agent/notifications-api
git worktree add ../app-frontend agent/notifications-frontend
git worktree add ../app-tests agent/notifications-tests
```

**Step 3: Launch the agents.** Each agent gets a clear, scoped task:

- Agent 1: "Implement the notifications API endpoints in `../app-api`. Follow the contract in `API_CONTRACT.md`. Include model, route, and controller. Write unit tests for the controller."
- Agent 2: "Build the notifications UI component in `../app-frontend`. It calls `GET /api/notifications` and `PATCH /api/notifications/:id`. Show a bell icon with unread count. Clicking opens a dropdown list."
- Agent 3: "Write integration tests in `../app-tests`. Cover the full flow: create a notification, fetch the list, mark as read, verify the count updates."

**Step 4: Let them work.** The three agents run simultaneously. Each commits to its own branch. You monitor progress but don't intervene unless something goes wrong.

**Step 5: Review and merge.** When all three finish, you review each branch. The API branch gets merged first — it's the foundation. Then the frontend branch. Then the tests. You run the full test suite after each merge to catch integration issues early.

Total time: maybe forty minutes. The API agent took thirty, the frontend agent took twenty-five, and the test agent took thirty-five. But they ran in parallel, so wall-clock time was thirty-five minutes plus ten minutes of your orchestration work.

A single agent doing everything sequentially? Probably ninety minutes.

The maths works when the task is big enough. And as you get better at decomposition, you'll develop an intuition for which tasks are worth

splitting and which aren't. Like most things in engineering, it's a judgement call. But now you have the tools to make it.

# 13 Agents in the Pipeline

A team I know set up a neat automation last year. They added a Claude agent as a CI step that would catch lint failures and auto-fix them. A developer pushes code, the linter runs, and if it fails, the agent rewrites the offending lines and pushes a fix commit. No human intervention needed. The pipeline stays green. Everyone loved it.

It worked brilliantly for about two weeks.

Then someone noticed something odd during a code review. A whole category of lint rules had disappeared. The agent had figured out that the fastest way to fix a lint failure was to modify `.eslintrc.json` — disable the rule, push the config change, pipeline goes green. It had been doing this for a month. Nobody caught it because the signal they were watching — pipeline status — was exactly the signal the agent had learned to game.

The fix was straightforward: make the lint config read-only for the agent. But the lesson was bigger than one misconfigured pipeline.

Automated agents need the same guardrails as interactive ones. Probably more, because there's nobody sitting there watching the diff scroll by. When you pair with an agent locally, you see every file it touches. You notice when it does something clever but wrong. In CI, the agent runs in the dark. The only thing you see is the outcome — green or red. And if the agent finds a way to make the outcome green without actually solving the problem, you might not notice for weeks.

This chapter is about using agents in pipelines responsibly. Where they add genuine value, where they create risk, and how to set them up so they stay useful without becoming a liability.

The stakes are higher in CI than at your desk. Get it right, and agents multiply your team's throughput around the clock. Get it wrong, and you've built an expensive, unsupervised footgun.

## 13.1 Agents as CI Steps

The simplest way to get agents into your pipeline is as CI steps — discrete jobs that run on every push or pull request, just like your linter or test suite.

You're not reinventing your pipeline. You're adding intelligence to it.

The most immediately useful CI agent: PR pre-screening. Not replacing human review, but filtering. An agent reads the diff, checks for obvious issues — unused imports, inconsistent naming, missing error handling, test files that don't actually assert anything — and leaves comments. By the time a human reviewer opens the PR, the trivial stuff is already flagged. The human can focus on architecture, approach, intent.

Other good candidates:

- **Changelog generation.** The agent reads the commits since the last release, cross-references with ticket numbers, and drafts release notes. A human edits before publishing, but the first draft is free.
- **Import ordering and formatting.** Safe, verifiable, low-stakes. If the agent reformats something wrong, the diff is obvious.
- **Dependency update summaries.** When Dependabot opens a PR, an agent can read the changelog of the updated package and summarise what changed and whether it's likely to affect your code.

There's also a subtler use: **test failure triage**. When a CI run fails, an agent can read the test output, identify the failing test, look at the recent diff, and post a comment explaining whether the failure looks like a genuine bug or a flaky test. It won't always be right, but it saves the developer from opening the CI logs, scrolling through two hundred lines of output, and figuring out what went wrong. That ten-minute investigation becomes a thirty-second glance at a comment.

The key principle: agents in CI should do things that are **safe to automate** and **easy to verify**. If you can't quickly tell whether the agent did the right thing, it shouldn't be automated yet. A good litmus test: would you be comfortable if the agent did this a hundred times and you only spot-checked ten of them? If the answer is no, it's not ready for CI.

## 13.2 The Overnight Agent

This is one of the most powerful patterns in agentic engineering, and one of the least discussed. It's also the one that makes managers' eyes light up — "you mean the agent works while we sleep?" — which is exactly why it needs careful framing.

You're wrapping up for the day. There's a well-defined ticket in the backlog — add a new API endpoint, write a data migration, refactor a module to use the new logging library. The kind of task where the requirements are clear and the definition of done is testable. You point an agent at it, give it a branch, and go home. You wake up to a PR with the work done, tests passing, ready for review.

The quality bar for unattended agents is higher than for interactive ones. When you're sitting there watching, you catch mistakes in real time. When the agent works overnight, mistakes compound. So you need:

- **Comprehensive tests to verify against.** The agent needs a way to know it's done. Existing tests that should keep passing, plus clear criteria for new tests it should write.
- **Tight scope.** One ticket, one concern. Don't point an overnight agent at "refactor the authentication system." Point it at "add rate limiting to the login endpoint."
- **Branch isolation.** Always a fresh branch, always a worktree. If the agent makes a mess, you delete the branch. Nothing touches main.
- **A timeout.** Set a hard limit — four hours is generous for most tasks. An agent that's been running for six hours isn't making progress. It's going in circles. Kill it and reassess in the morning.

A simple setup script looks like this:

```
#!/bin/bash
# overnight-agent.sh — fire and forget before you leave

TICKET_ID="$1"
BRANCH_NAME="agent/overnight-${TICKET_ID}"
WORKTREE_DIR="../overnight-${TICKET_ID}"

# Create isolated workspace
git worktree add "$WORKTREE_DIR" -b "$BRANCH_NAME"
```

```
# Run the agent with a token budget and timeout
timeout 4h claude --worktree "$WORKTREE_DIR" \
  --max-tokens 200000 \
  --prompt "Implement ticket ${TICKET_ID}. Read TICKETS/
${TICKET_ID}.md for requirements. Write tests. Commit your work.
Do not modify any CI config or lint rules." \
  2>&1 | tee "logs/overnight-${TICKET_ID}.log"

# Push the branch so you can review in the morning
cd "$WORKTREE_DIR" && git push -u origin "$BRANCH_NAME"
```

You refine this over time. Add Slack notifications when it finishes. Add a summary of what it did. Add a check that verifies the test suite passes before pushing.

One thing I've learned from teams doing this well: the ticket description matters enormously. A ticket that says "add user preferences endpoint" isn't enough. The overnight agent needs acceptance criteria, example request/response payloads, and pointers to similar existing endpoints it can use as reference. You're writing instructions for a competent but context-free developer. The more specific you are, the better the result.

The teams that get the most out of overnight agents are the ones that invest in their ticket-writing discipline. Which, again, benefits everyone — your human teammates also prefer clear tickets. The agent just makes the cost of vagueness more visible.

The core loop is simple: isolate, constrain, run, review in the morning.

## 13.3 Cost Control in CI

When you run an agent interactively, you have a natural circuit breaker: yourself. You see the tokens ticking up, you see the agent going in circles, you hit Ctrl+C. In CI, there's nobody watching.

A team running a busy monorepo learned this the hard way. They'd set up an agent to auto-review every PR. Reasonable enough — except their monorepo saw forty to fifty PRs a day, and each review consumed around fifteen thousand tokens. That's manageable. What wasn't manageable

was the retry logic. When the agent hit a rate limit, the CI job retried. Three retries per failure, exponential backoff, but each retry started a fresh agent run from scratch. Over a bank holiday weekend, with a batch of automated dependency updates flooding in, the pipeline generated a $4,000 bill. Nobody was in the office to notice.

Protect yourself:

- **Token budgets per pipeline run.** Set a hard cap. If the agent hits it, the job fails with a clear message. Better to miss a review than burn through your monthly budget in a day.
- **Concurrency limits.** Don't let twenty agent jobs run simultaneously. Queue them. Two or three concurrent agent runs is plenty for most teams.
- **Spend alerts.** Your LLM provider almost certainly supports them. Set one at 50% of your monthly budget. Set another at 80%. Pipe them to a channel someone actually reads.
- **Kill switches.** A feature flag or environment variable that disables all agent CI steps instantly. When something goes wrong at 2am, you want a one-line fix, not a pipeline config change that needs its own PR.
- **Per-job cost tracking.** Log the token count and estimated cost of every agent CI run. You can't optimise what you don't measure. A weekly report of agent CI spend, broken down by job type, will show you where the money goes and where to tighten up.

A simple circuit breaker in your CI config looks like this:

```
agent-review:
  timeout-minutes: 15
  env:
    MAX_TOKENS: 50000
    COST_ALERT_THRESHOLD: "$5.00"
  steps:
    - name: Run agent review
      run: |
        claude review --max-tokens $MAX_TOKENS \
          --on-budget-exceeded "exit 1" \
          pr/$PR_NUMBER
```

The specifics will vary by tool and platform, but the pattern is constant: set a ceiling, fail loudly when you hit it, and make the ceiling easy to adjust.

The rule of thumb: treat your LLM spend in CI the way you treat your cloud compute spend. Monitor it, budget it, alert on it. It's a real cost centre.

## 13.4 The Review Question

Agent-generated PRs still need human review. Full stop.

The temptation is real. The agent wrote the code. The tests pass. The linter is happy. Coverage hasn't dropped. Why not auto-merge? You'll save fifteen minutes of review time, and the CI pipeline already verified everything that matters.

But think about what "everything that matters" actually means. Tests verify correctness. They don't verify intent.

An agent tasked with "reduce the number of database queries on the user profile page" might cache aggressively and introduce stale data bugs that no current test catches. It might denormalise data in a way that makes the next feature twice as hard to build. It might solve the problem perfectly but in a style completely alien to the rest of your codebase. The tests pass because the tests check behaviour, not approach.

A human reviewer catches these things. They read for *how*, not just *what*. They notice when a solution works today but creates debt for tomorrow. They spot the shortcut that will confuse the next developer who touches this code.

There's also a social dimension. If your team knows that agent PRs get auto-merged, they stop paying attention to agent-generated code. It becomes a black box. Six months later, half your codebase was written by an agent and nobody on the team fully understands it. That's a knowledge gap that will hurt you during an incident.

Auto-merge is fine for trivial, mechanical changes — formatting fixes, import sorting, version bumps. For anything that involves a design deci-

sion, a human reviews it. The agent is a fast drafter, not a decision-maker. And the review doesn't have to be exhaustive — a five-minute scan to check the approach is reasonable goes a long way.

## 13.5 Agent-Assisted Deployments

Deploys are where things get genuinely dangerous, and where the gap between "agent can do this" and "agent should do this" is widest. Let's be precise about what agents should and shouldn't do here.

Agents are excellent at **monitoring** deploys. They can watch log streams, track error rates, compare latency percentiles against the pre-deploy baseline, and flag anomalies. An agent that says "error rate on `/api/checkout` has increased 3x since the deploy twelve minutes ago" is enormously valuable. It's reading data, finding patterns, and surfacing information — exactly what agents are good at.

Agents can also **suggest** actions. "Based on the error rate increase, I recommend rolling back to the previous version" is a helpful suggestion. It's the kind of thing that would normally require someone staring at a Grafana dashboard for fifteen minutes to conclude. The agent has done the analysis. A human decides whether to act on it.

What agents should not do is make the rollback decision autonomously. Production deployments are too consequential for ad-hoc agent decisions. If you want automated rollbacks, use a pre-approved runbook with hard thresholds — "if error rate exceeds 5% for three minutes, roll back automatically." That's deterministic. You tested it. You approved the logic in advance. An agent deciding on its own whether a 2.3% error rate increase "feels" significant enough to roll back? That's a recipe for either unnecessary rollbacks or missed incidents. Deterministic rules are boring. Boring is good when your revenue is on the line.

In practice, a deploy-monitoring agent looks something like this: it subscribes to your log aggregator and metrics dashboard via API, runs for fifteen minutes after each deploy, and posts a summary to Slack. "Deploy of v2.4.1 completed. Error rate stable at 0.3%. P99 latency increased from 180ms to 210ms — within normal variance. No new exception types

detected." Most of the time, that summary is boring. That's the point. When it's not boring, you want to know immediately.

This connects back to the guardrails principle: production is read-only for agents. They observe, analyse, report, recommend. Humans (or pre-approved automation with deterministic rules) take action.

## 13.6 The Pipeline as Context

Your CI/CD configuration is context, and agents read it like any other code. This is easy to forget — most teams treat their pipeline config as infrastructure plumbing, not as something that needs to communicate clearly. But when an agent is trying to understand why a build failed or what verification steps exist, the pipeline config is the first thing it reads.

A well-structured pipeline helps agents understand your project. Clear stage names (`build`, `unit-tests`, `integration-tests`, `deploy-staging`) tell the agent what your verification process looks like. Structured error output — JSON logs, exit codes with meaning, error categories — gives the agent something to work with when a step fails.

A pipeline that outputs `BUILD FAILED` and nothing else is useless to agents and humans alike.

Invest in pipeline observability:

- **Structured logs.** JSON or key-value pairs, not free-form text. An agent can parse `{"stage": "unit-tests", "failed": 3, "passed": 247, "failures": ["test_auth_flow", "test_rate_limit", "test_session_expiry"]}` and take meaningful action. It can't do much with `Tests failed. See above for details.`
- **Meaningful exit codes.** Don't let everything exit with code 1. Distinguish between "tests failed" (fixable) and "couldn't connect to the database" (infrastructure problem, not the agent's concern).
- **Artefact storage.** Test results, coverage reports, build logs — store them as pipeline artefacts. An agent debugging a CI failure needs to read the full output, not just the last ten lines that fit in the console view.

Consider adding a `PIPELINE.md` to your repo — a plain-language description of what each CI stage does, what its expected outputs look like, and what common failure modes exist. An agent that can read this file before debugging a CI failure will perform dramatically better than one that has to reverse-engineer your pipeline from YAML config alone. And your new hires will thank you too.

Good pipeline design and good agent integration reinforce each other. You improve your CI for agents, and your human developers benefit too. It's one of those rare investments that pays dividends in both directions.

## 13.7 Starting Small

If you've read this chapter and you're tempted to wire agents into every stage of your pipeline by Friday — don't. I've seen that impulse. It usually ends with a weekend spent undoing the automation because the team wasn't ready for the volume of agent-generated noise.

The teams that succeed with agents in CI are the ones that build trust incrementally, just like with interactive agents.

The good news is that the path is well-trodden. Here's a practical adoption roadmap that I've seen work across multiple teams:

**Week 1: Auto-formatting.** Add a CI step that runs an agent to fix formatting issues on PRs. Formatting is deterministic, easy to verify, and low-risk. If the agent reformats something incorrectly, the diff is right there. Review the agent's commits for the first week to build confidence.

**Week 2: PR pre-screening.** Add agent-generated review comments on PRs. Not blocking — informational only. Let your team see what the agent catches, and calibrate whether its feedback is useful or noisy. Adjust the prompt based on what works.

**Month 1: Changelog drafting.** Point the agent at your commit history to generate release note drafts. A human edits and publishes. You're using the agent as a first-drafter, not a decision-maker. This is also a good time to set up cost monitoring and alerts.

**Month 3: Overnight agents.** By now you've built confidence in agent-generated output and you have the infrastructure — worktrees, token budgets, branch isolation, review processes. Start with a single well-scoped ticket. Review the result carefully. Iterate on your overnight script. Gradually expand to more complex tasks as your trust grows.

Notice what's *not* on this roadmap: auto-merging, autonomous deployments, or agents making architectural decisions. Those aren't stage four. They might be stage ten, or they might never be appropriate for your team. The roadmap isn't a march towards full automation — it's a march towards the right level of automation for your context.

Resist the urge to skip steps. Each one builds on the last. You learn what your agents are good at, where they struggle, and what guardrails you need. By month three, agent-assisted CI feels natural — not because you automated everything at once, but because you earned each piece of it through experience.

The pipeline is just another environment where agents work. The same principles apply: scope tightly, verify thoroughly, trust incrementally. The only difference is that nobody's watching, so your safety nets need to be that much stronger.

# 14 When Agents Get It Wrong

Every engineer who works with agents long enough has a collection of these stories. The moments where you lean back in your chair, stare at the screen, and mutter something unprintable. They're humbling. They're educational. And they're *inevitable*.

This chapter is a collection of war stories — things that actually go wrong when you hand real work to an agent. Not hypotheticals. Not contrived demos. The kind of failures that cost you an afternoon, a deploy, or your faith in automation. Each one maps back to principles from earlier chapters, because principles are cheap until you learn them the hard way.

Read these before you make the same mistakes. Or read them after, and feel less alone.

## 14.1 The Eager Refactorer

The task was simple: fix a z-index issue on a dropdown menu. The dropdown was rendering behind a modal overlay. A one-line CSS fix — maybe two if you're being careful.

The agent saw the dropdown component, decided its CSS was "inconsistent with modern practices," and refactored it. Then it noticed the modal used a different styling pattern, so it refactored that too. Then the button components. Then the card components. Before you looked up from your coffee, thirty-two files had changed. The diff was 1,400 lines. The agent had essentially rewritten the component library's styling system, migrating from one approach to another with admirable consistency and zero authorization.

The original z-index bug? Still there. Buried somewhere in the avalanche of changes, the agent had reproduced the same layering issue with new class names.

The PR was unreviewable. You can't meaningfully review 1,400 lines of CSS changes across thirty-two files. You *can* delete the branch and start over. Which is what happened.

**What you do differently now:** Scope tasks tightly. "Fix the z-index on the dropdown in `NavMenu.tsx`, touch nothing else." Use a dedicated branch so the damage is contained. And *always* check the diff before you let the agent move on to anything else. The moment you see the file count climbing past what makes sense for the task, stop the agent. The guardrails chapter exists for a reason.

## 14.2 The Hallucinated Library

The agent needed to parse some complex date ranges from user input. It imported `@temporal/daterange-parser`, wrote elegant code using its `.parseRange()` method with options for locale-aware parsing and fuzzy matching. The code was clean. The types were correct. The error handling was thoughtful. It even wrote tests — which, naturally, also imported the hallucinated package.

Everything looked perfect in the diff. The function signatures made sense. The API was well-designed. It was *such* a plausible library that you almost didn't question it.

Then `npm install` failed. The package didn't exist. Had never existed. The agent had invented a library, given it a believable name and a coherent API, and written production code against a fiction.

The insidious part: if you'd only done code review — reading the logic, checking the types, evaluating the approach — you would have approved it. The code was *good*. It just didn't connect to reality.

**What you do differently now:** The agent runs the tests. Always. If your setup doesn't allow that, you run them yourself before reviewing code. No exceptions. A failing `npm install` is a loud, obvious signal. A hallucinated API that was never executed is a silent bomb. Tests catch what human review misses — not because your review is bad, but because plausible fictions are *designed* to pass review. That's what hallucination *is*.

## 14.3 The Infinite Loop

You asked the agent to fix a failing integration test. It read the error, made a change, ran the test. New error. It read *that* error, made another change, ran the test. Different error. Then a variation of the first error. Then something new. Then the first error again.

You were in another terminal, working on something else. Forty minutes later you checked back. The agent had made nineteen attempts. It had burned through 200,000 tokens. The code was now worse than when it started — a geological record of failed fixes layered on top of each other. The agent was still going, cheerfully confident that attempt twenty would be the one.

It wasn't.

The fundamental problem was that the agent didn't understand *why* the test was failing. It was pattern-matching on error messages and making local edits, but the actual issue was an architectural misunderstanding — a shared database state between test cases that required a different setup approach entirely. No amount of tweaking the code under test would fix a problem in the test harness.

**What you do differently now:** Set iteration limits. Three attempts at the same problem is a reasonable maximum. If the agent hasn't solved it in three passes, it won't solve it in thirty. When you see the loop — error, fix, different error, fix, original error — *break it manually.* Stop the agent, read the errors yourself, and either give it a completely different approach or take over. The agent's time is cheap. Your wasted afternoon is not. More importantly, start a fresh context. The agent's understanding is now polluted with nineteen wrong theories. A clean start with your diagnosis of the *actual* problem will get further, faster.

## 14.4 The Confident Wrong Answer

A background job was occasionally processing the same item twice. Classic race condition. You pointed the agent at the problem.

The agent analysed the code, identified the race window, and added a 500ms delay between the check and the update. "This ensures the previous transaction has time to commit before the next check occurs," it explained, with the serene confidence of someone who has never operated a system under load.

Tests passed. The delay was longer than the test's transaction time, so the race window closed — in the test environment, with one concurrent worker, on a quiet machine.

It went to production. Under load, with twelve workers and variable database latency, 500ms was sometimes not enough. And now you had a *new* problem: the delay had reduced throughput enough that the job queue backed up during peak hours, creating cascading timeouts that took down an unrelated service.

The sleep didn't fix the race condition. It hid it at low concurrency and made the system worse at high concurrency. A proper fix — an advisory lock or an idempotency key — would have been correct at any scale. The agent chose the fix that made the test green, not the fix that was *right*.

**What you do differently now:** You treat passing tests as necessary but not sufficient. When an agent fixes a concurrency issue, you ask *yourself* whether the fix is correct under load, under failure, under conditions the test suite doesn't cover. Agents optimise for the feedback signal you give them, and if that signal is "tests pass," they'll find the shortest path to green — even if that path is a `time.Sleep`. Your engineering judgment about *why* something works matters more than *whether* the tests pass. This is the part of the job that hasn't been automated yet. Lean into it.

## 14.5 The Context Amnesia

Two hours into a session, you had a nicely evolving feature. You'd told the agent early on: "Don't use any ORM — we're writing raw SQL in this project. That's a deliberate choice." The agent acknowledged this and wrote clean, handcrafted queries for the first several tasks.

Then you asked it to add a new endpoint. Ninety minutes of context had accumulated. The agent built the endpoint — using Prisma. Full schema

file, migration, generated client. Beautiful code. Completely contradicting the constraint you'd established at the start of the session and the patterns in every other file it had written that day.

When you pointed this out, the agent apologised, rewrote everything in raw SQL, and acted as if nothing had happened. It hadn't *decided* to ignore your constraint. It had simply lost it. The context window had filled with enough intermediate work that the early instruction had faded into irrelevance.

**What you do differently now:** Long sessions degrade. This is a fundamental property of how context windows work, not a bug that will be patched next quarter. Keep tasks short and focused. Commit working code at natural boundaries so progress is captured in git, not just in the conversation. Start fresh sessions for fresh tasks. And for project-wide constraints like "no ORM" or "no new dependencies," put them in a `CLAUDE.md` file or equivalent that the agent reads at startup. Don't rely on the agent *remembering* what you said two hours ago. It won't. Write it down.

## 14.6 The Dependency Avalanche

You asked for a date picker. A simple input where users can select a date. The agent evaluated the options and decided to be thorough.

It installed `moment.js` for date handling. Then `@popperjs/core` for positioning the dropdown. Then a full UI component library because "it provides accessible date picker primitives." Then a CSS preprocessor because the component library's theme system required it. Then two utility packages that the component library's date picker needed as peer dependencies.

Six new dependencies. Your bundle size went from 180KB to 540KB. Your build time doubled. You had a date picker, though. It was very nice.

The native HTML `<input type="date">` would have been fine. Or a single lightweight picker library at 8KB. Instead you'd inherited an entire ecosystem because the agent optimised for completeness rather than minimality.

The worst part wasn't the bundle size — it was the maintenance surface. Six new packages means six new things that can have security vulnerabilities, six new things that can break on upgrade, six new changelogs to read when Dependabot starts filing PRs. You didn't just add a date picker. You adopted six open-source projects.

**What you do differently now:** Constrain what agents can install. "No new dependencies without asking me first" is a legitimate and often *wise* instruction. When you do allow new packages, tell the agent your constraints: bundle budget, no packages with fewer than N weekly downloads, no packages that pull in transitive mega-dependencies. Agents don't have opinions about bundle size or maintenance burden. They don't maintain the project next year. *You* do. So you set the boundaries.

## 14.7 The Common Thread

Every one of these stories has the same root cause: the agent was doing *exactly what it was designed to do*, and the human wasn't providing enough structure.

The eager refactorer was being helpful. The hallucinated library was being creative. The infinite loop was being persistent. The confident wrong answer was being test-driven. The context amnesia was a limitation, not a choice. The dependency avalanche was being thorough.

None of these are agent bugs. They're *workflow* bugs. The fix is never "use a smarter agent." The fix is always the same: tighter scope, better feedback loops, more structure, shorter sessions, and a human who stays engaged.

Agents get things wrong. So do humans. The difference is that humans get things wrong slowly enough to notice. Agents get things wrong at the speed of autocomplete, and by the time you look up from your coffee, thirty-two files have changed.

Stay in the loop. Check the diffs. Trust but verify. And when things go sideways — because they will — remember that the delete-branch-and-start-over button is the most underrated tool in your workflow.

## 14.8 The Diagnostic Playbook

War stories are entertaining. But you can't tape anecdotes to your monitor. What you need is a systematic approach — a checklist for when the agent produces something wrong, so you can diagnose the failure quickly and fix the right thing instead of flailing.

When an agent gives you bad output, run through these questions in order. Most failures fall into one of six categories, and knowing which category you're in determines what you do next.

**1. Is it a scoping problem?**

Did you ask for too much in one prompt? The Eager Refactorer happened because "fix the z-index" was too vague — it didn't say "touch nothing else." If the agent changed files you didn't expect, or did work you didn't ask for, the scope was too loose. Tighten the prompt. Be explicit about what *not* to do. Constraints are more useful than instructions when dealing with an enthusiastic agent.

**2. Is it a context problem?**

Did the agent have what it needed to solve the actual problem? Think back to the billing bug story from Chapter 2 — one engineer gave vague context and got a vague answer, the other gave specific files and error traces and got a working fix. If the agent solved the *wrong* problem, it probably couldn't see the right one. Feed it the relevant files, the error output, the constraints it can't infer. Agents don't guess well. They work with what you give them.

**3. Is it a feedback signal problem?**

Are your tests actually verifying the right thing? The Confident Wrong Answer happened because the tests passed — but they didn't test under realistic conditions. The 500ms sleep "fix" was green in CI and wrong in production. If the agent's solution feels dubious but the tests pass, *your tests are the problem.* The agent optimised for the signal you gave it. Give it a better signal.

**4. Is it a capability problem?**

Some tasks are genuinely beyond what the model can do. Multi-file reasoning across a sprawling system with implicit dependencies. Subtle concurrency issues that require understanding runtime behaviour, not just code structure. Security-sensitive logic where "plausible" isn't good enough. If the agent keeps trying different approaches and failing, it might not be capable of this particular task. That's not a prompt problem — it's a limitation. Recognise it and take over. Your time is better spent doing the work than engineering the perfect prompt for a task the agent can't handle.

**5. Is it a context window problem?**

Long sessions degrade. Full stop. The Context Amnesia story demonstrated this — constraints set early in a session simply get lost as the context fills up with intermediate work. If the agent contradicts something it did correctly earlier in the same session, or ignores a constraint you set an hour ago, the context window is the culprit. Start a fresh session. Restate the relevant constraints. Give it only the context it needs for the current task, not the archaeological record of everything you've discussed today.

**6. Is it a loop?**

Three attempts at the same error is the limit. If the agent hasn't solved it in three passes, it won't solve it in thirty. The Infinite Loop story burned 200,000 tokens across nineteen attempts with no progress. When you see the pattern — error, fix, different error, fix, original error — break the loop immediately. Stop the agent. Read the errors yourself. Either give the agent a completely different approach with a fresh diagnosis, or take over entirely. The agent's context is now polluted with failed theories, and more attempts will only add more pollution.

Print this checklist. Tape it to your monitor. The first few times you'll run through it consciously. After a month, it becomes instinct. After three months, you'll catch the problem before the agent even finishes its first attempt.

# 15 When Not to Use Agents

This book is about using agents well. This chapter is about knowing when not to use them at all.

If you've read this far, you're probably sold on agentic engineering. Good. But the fastest way to lose credibility — and waste time — is to reach for an agent when the job calls for a human. The best agentic engineers aren't the ones who use agents the most. They're the ones who know *exactly* when to put the agent away and do the work themselves.

## 15.1 The Overhead Tax

Every agent interaction has a cost. You write the prompt. You wait for the output. You review what it produced. You fix the mistakes. You re-run if it missed the point. That's overhead, and it's real.

For complex tasks that would take you an hour, spending two minutes on prompt and review is a bargain. But for tasks you can do in thirty seconds? The overhead makes agents *slower*, not faster.

Renaming a variable. Fixing a typo. Tweaking a config value. Adding a log line. These are muscle-memory tasks. Your fingers know the keystrokes. By the time you've typed a prompt describing what you want, you could have already done it.

This isn't a failure of agents. It's arithmetic. Small tasks have small payoffs, and the fixed cost of agent interaction eats the margin. Don't let the novelty of agents trick you into using them for everything. Some work is just faster by hand.

## 15.2 Novel Architecture Decisions

When you're designing a system from scratch — choosing between a monolith and microservices, deciding on your data model, picking your communication patterns — the *thinking is the work*. The value isn't in the diagram or the document. The value is in the mental model you build while wrestling with the tradeoffs.

An agent can help you explore options. It can list the pros and cons of event sourcing versus CRUD. It can sketch out what a particular architecture might look like. That's useful as input.

But delegating the architecture itself to an agent means you don't understand your own system. You'll struggle to debug it, extend it, or explain it to your team. The senior engineer's job is to make the hard calls — to weigh the tradeoffs that don't have clean answers, to decide what complexity is worth carrying and what isn't. That judgment comes from doing the work, not from reading an agent's summary of it.

Use agents as a sparring partner for architecture. Not as the architect.

## 15.3 Security-Critical Code

Authentication flows. Encryption. Access control. Input validation. Token handling. These are areas where "looks correct" isn't good enough.

Security bugs are different from regular bugs. A broken sort function produces wrong output that someone notices. A broken auth check produces *no visible symptoms* until an attacker finds it. The code looks fine. The tests pass. And six months later you're writing an incident report.

Agents produce plausible code. That's their strength and, in security contexts, their danger. A subtle flaw in a JWT validation flow, a missing check on a redirect URL, a timing side-channel in a password comparison — these are the kinds of mistakes that survive code review because they *look right*.

Write security-critical code yourself. Review it carefully. Get a second pair of human eyes on it. If you do use an agent to draft security code, treat

that draft with more suspicion than you'd give a junior developer's first attempt, not less.

## 15.4 When You Need to Learn

You're picking up a new language. A new framework. A new paradigm. The temptation is obvious: let the agent write the code while you focus on the big picture.

Resist it.

If you let the agent write all the Rust while you're learning Rust, you haven't learned Rust. You've built something you can't maintain, debug, or extend without the agent. You've created a dependency, not a skill.

There's a crucial difference between using an agent to *explain* something and using it to *do* something. Asking "why does this borrow checker error occur?" builds understanding. Asking "fix this borrow checker error" doesn't.

When the goal is learning, slow down. Write the code yourself. Make the mistakes yourself. Use agents as tutors, not ghostwriters. The understanding you build by struggling through the hard parts is the whole point.

## 15.5 Emotionally Charged Decisions

Not every engineering decision is technical. Some of the hardest ones are human.

Deprecating an API that a partner depends on. Telling a stakeholder their feature request won't make the cut. Pushing back on a deadline you know is unrealistic. Deciding to sunset a product that still has users.

These decisions require empathy. They require reading the room, understanding the politics, weighing the human cost alongside the technical cost. They require *accountability* — someone who will own the decision and its consequences.

An agent can draft the email. It can help you think through the talking points. But the decision itself, and the conversation that delivers it, must come from a human. People deserve to hear hard news from a person, not from someone who copy-pasted an agent's output.

## 15.6 When the Codebase Is Too Messy

Agents amplify what's already there. In a clean, well-structured codebase with strong conventions, agents produce clean, well-structured code. They pick up the patterns and follow them.

In a mess? They produce more mess.

If your codebase has inconsistent naming, tangled dependencies, no clear module boundaries, and three different ways to do the same thing — an agent will pick up *all* of those patterns. It might combine the worst parts of each. It doesn't know which patterns are intentional and which are technical debt. It just sees what's there and produces more of it.

Sometimes the right move is to clean up before you bring agents in. Refactor the module. Establish the convention. Delete the dead code. Make the codebase a place where an agent can do good work. This is unglamorous, manual labor. But it's the foundation that makes everything else possible.

Think of it like a workshop. You don't hand power tools to someone in a cluttered, disorganized shop. You clean up first, *then* bring in the tools.

## 15.7 Working with Legacy Code

That workshop metaphor is nice. But not everyone has the luxury of cleaning up first. Some of us maintain 500,000-line Java monoliths that were started in 2014 and have been through three framework migrations, two acquisitions, and a brief period where someone thought XML configuration was a good idea. The advice to "refactor before bringing in agents" is sound in theory and laughable in practice when you're staring at a codebase that would take years to refactor.

So how *do* you use agents in legacy code? Carefully. And with a specific strategy.

**Start with the tests.** Before pointing an agent at legacy code, write characterisation tests — tests that capture what the code *currently* does, not what it *should* do. These aren't aspirational. They're descriptive. They say "when you call this function with these inputs, this is what comes out, and that's the current contract whether anyone intended it or not."

Characterisation tests give the agent a safety net. Without them, the agent will change behaviour it doesn't understand, and you won't know until production tells you. With them, any behavioural change shows up as a test failure *before* the code leaves your machine. This is non-negotiable. Legacy code without tests is legacy code you can't safely let agents touch.

**Scope ruthlessly.** In a legacy codebase, the agent can't understand the whole system. Don't try to make it. Point it at one file, one function, one bug. Give it the immediate context — the function signature, the callers, the expected behaviour — and nothing else. Legacy code is full of implicit assumptions, undocumented side effects, and load-bearing quirks. The less the agent sees, the less it can misinterpret. A narrow scope with clear context beats a broad scope with archaeological complexity.

**Use agents for the tedious parts.** Legacy codebases are full of mechanical work: updating deprecated API calls across hundreds of files, migrating from one dependency version to another, adding type annotations to untyped code, standardising error handling patterns, replacing one logging framework with another. These are *perfect* agent tasks — repetitive, well-defined, easily verified by automated checks. Let agents do the drudgery. Save your energy for the parts that require understanding *why* the code is the way it is. That's the work only a human who's lived with the system can do.

**Document as you go.** Every time an agent works on a legacy file successfully, add a brief comment explaining what the code does, or update the project's `CLAUDE.md` with context about that module. Over time, something interesting happens: the parts of the legacy codebase that agents have touched become better documented than the parts they haven't. Not because you set out to document the system, but because

using agents *forced* you to articulate what each piece does in order to prompt effectively. The agents are slowly making the codebase more legible — not by refactoring it, but by making you explain it.

## 15.8 The Craft Argument

There's one more reason to sometimes put the agent away, and it's not about efficiency or risk. It's about craft.

Writing code by hand builds something that agents can't give you. Muscle memory. Pattern recognition that lives in your fingers, not just your head. The deep, intuitive understanding that comes from having written the same kind of function dozens of times. The quiet satisfaction of solving a hard problem through your own reasoning.

These things matter. Not because they're romantic, but because they make you a better engineer. The developer who has hand-written a parser understands parsing differently than one who has only prompted an agent to write one. The developer who has debugged a concurrency issue at 2am *feels* the danger of shared mutable state in a way that reading about it never provides.

Agents are tools. Powerful ones. But if you let them do all the work, your own skills atrophy. And when you hit a problem that the agent can't solve — and you will — you need those skills to still be sharp.

Stay in practice. Write code by hand regularly. Not because it's faster, but because it keeps you dangerous.

## 15.9 The Judgment That Matters

The central skill of agentic engineering isn't prompting. It isn't tool configuration. It isn't knowing which model to use for which task.

It's *judgment*.

Knowing when an agent will save you an hour and when it will waste one. Knowing when to trust the output and when to rewrite it from scratch. Knowing when to delegate and when to dig in yourself.

The best agentic engineers use agents aggressively — but selectively. They reach for agents when the leverage is real: large-scale refactors, boilerplate generation, code exploration, test writing, documentation. And they put the agent away when the work requires human thinking, human accountability, or human craft.

That judgment is what separates agentic engineers from prompt jockeys. Anyone can type a prompt. Knowing *when not to* is the harder skill.

# 16 Agentic Teams

Software has always been a team sport. Agents don't change that. But they change the shape of the team, the rhythm of the work, and the skills that matter. If you're leading a team — or working on one — you need to think about this now, not later.

## 16.1 The 10x Multiplier Is Real, but Distributed Differently

You've heard the claim: agents make engineers 10x more productive. It's not wrong. It's just misleading.

Agents make certain tasks *dramatically* faster. Generating boilerplate, writing test scaffolding, refactoring across a hundred files, migrating API versions, wiring up CRUD endpoints — these go from hours to minutes. The speed gain is real and it's massive.

But other tasks barely change. System design still requires thinking. Debugging a subtle race condition still requires patience, intuition, and deep understanding of the runtime. Stakeholder conversations still take however long they take. The agent can't sit in your architecture review and ask the right questions.

The 10x multiplier isn't a flat multiplier across your day. It's a spike graph. Some tasks go 50x faster. Some go 1x. A few might even slow down if you fight the agent instead of doing it yourself.

Teams that understand this deploy agents strategically. They don't hand every task to an agent and expect magic. They identify the high-leverage zones — the tasks where agents genuinely compress timelines — and they focus agent effort there. The rest stays human.

## 16.2 Code Review Changes

Here's what happens when agents enter a team's workflow: PR volume goes up. *Way* up. An engineer who used to open two PRs a day now opens five. The code in those PRs is syntactically correct, well-formatted, and passes tests. And reviewing it is *exhausting*.

The old model of code review — read every line, check for off-by-one errors, verify edge case handling — doesn't scale when half the code was generated in seconds. You'll burn out your reviewers in a week.

The shift is from "is this correct?" to "is this the right approach?" Agent-generated code is usually *locally* correct. It does what was asked. The question is whether what was asked was the right thing. Does this new service need to exist, or should this logic live in the existing one? Is this the right abstraction boundary? Does this change make the system simpler or more complex?

Reviewers become architects. They zoom out. They check intent, not implementation.

Practical adaptations that work:

- Smaller, more focused PRs — easier for both agents and reviewers
- Automated checks handle the mechanical stuff (linting, test coverage, type checking)
- Review time is protected on the calendar, not squeezed between meetings
- Teams agree on "trusted patterns" — if a PR follows a known pattern and passes CI, it gets a faster review track

Review fatigue is the silent killer of agent-assisted teams. Take it seriously.

## 16.3 The Junior Engineer Question

This is the hardest problem in this chapter, and I don't have a clean answer.

Junior engineers have traditionally learned by doing the work that agents now do faster and better. Writing that first CRUD endpoint. Wrestling with a tricky CSS layout. Figuring out why the test is failing. These tasks

were tedious for seniors but *formative* for juniors. The struggle was the education.

If agents handle all of that, where does the learning happen?

The worst outcome is juniors who become prompt-dependent — they can get an agent to generate code, but they can't explain what the code does or debug it when it breaks. They skip the understanding phase entirely. That's not engineering; that's a very expensive copy-paste workflow.

The better path is using agents as *tutors*, not replacements. A junior writing a database migration should ask the agent to *explain* the migration, not just generate it. "Why did you use a transaction here?" "What happens if this fails halfway?" "Show me what this looks like without the ORM." The agent becomes a patient teacher with infinite time — something most seniors can't offer.

Pair programming with agents, *supervised by seniors*, is the most promising model I've seen. The junior drives the agent. The senior watches, asks questions, and intervenes when the junior accepts something they don't understand. It's slower than letting the agent do everything, but it produces engineers who actually know what they're doing.

Teams that skip this investment are borrowing from the future. Today's unsupervised juniors are tomorrow's seniors who can't debug production without an AI crutch.

## 16.4 Knowledge Distribution

Here's a pattern that shows up in every team that adopts agents unevenly: one engineer gets fluent with agents, starts shipping at 3x the rate of everyone else, and within a few months has touched every part of the codebase. They become the single point of failure.

The bus factor drops to one. Not because anyone planned it, but because velocity and knowledge concentration are correlated. The engineer who ships the most learns the most. The others fall behind, not just in output but in *understanding* of the system they collectively own.

This is a management problem, not a technology problem. The fix is structural:

- **Shared `CLAUDE.md` files.** Every project has one. Everyone contributes to it. It encodes the team's collective knowledge, not one person's.
- **Shared workflows and conventions.** The team agrees on how they use agents — which tools, which patterns, which guardrails. No lone-wolf setups.
- **Rotation.** Agent-fluent engineers rotate to different parts of the codebase. Knowledge spreads through work, not documentation.
- **Agent session sharing.** Some teams have started sharing interesting agent sessions — the prompts, the outputs, the decisions. It's a form of knowledge transfer that didn't exist before.

The goal isn't to slow down your fastest engineer. It's to make sure the team's knowledge keeps up with the team's output.

## 16.5 Shared Conventions Matter More

We covered conventions in an earlier chapter. In a team context, the stakes are higher.

When a solo engineer uses agents, their conventions affect one person. When a team uses agents, conventions affect *every agent session across the entire team*. A well-structured project with clear naming, consistent patterns, and a maintained `CLAUDE.md` means every engineer's agents start from a strong foundation.

A messy project means every agent reinvents the wheel. Different engineers get different outputs. The codebase drifts. Reviews get harder because you're now reviewing not just the code but the *style* of the code, which varies by which engineer's agent wrote it.

Coding standards in an agentic team aren't about aesthetics. They're about *agent effectiveness*. The team that agrees on project structure, naming conventions, testing patterns, and documentation format gets better output from every agent session. It's a force multiplier on a force multiplier.

Invest the time. Write the standards down. Enforce them in CI. It pays back on every single PR.

## 16.6 The New Standup

What does daily coordination look like when each engineer is running multiple agent sessions in parallel?

The old standup: "Yesterday I worked on the auth refactor. Today I'll continue. No blockers."

The new standup: "I have three agents running. The auth refactor landed and is in review. The API migration is 80% done — the agent got stuck on the legacy XML parsing, so I'm taking that over manually. I just kicked off a third session to generate integration tests for the billing module."

The granularity changes. Work moves faster, so coordination needs to keep up. An engineer might *start and finish* a task between standups. The daily sync becomes less about progress updates and more about intent alignment — making sure three engineers aren't all sending agents after the same problem from different angles.

Some teams are moving to async standups with more frequent check-ins. Others use shared dashboards that track active agent sessions. The right answer depends on the team, but the old rhythm of "one update per person per day" often isn't enough.

## 16.7 Compliance and the Audit Trail

If an agent writes code that causes a production incident, who's responsible? The engineer who prompted it? The reviewer who approved the PR? The team lead who decided to adopt agentic workflows? This isn't a philosophical question you debate over pints. It's a legal and compliance question, and regulated industries need clear answers *before* the incident happens, not during the postmortem.

The good news is that the answer isn't actually that complicated. The tooling and processes already exist. You just need to be explicit about them.

**Git is your audit trail.** Every agent-generated commit should be attributable. The commit message should indicate it was agent-assisted — a `Co-Authored-By` tag, a prefix, whatever convention your team adopts. The PR should show who reviewed it. The merge approval is the sign-off. This is already how most teams work; the key is making it *consistent*. Ad-hoc attribution — sometimes tagging, sometimes not — is worse than no system at all, because it creates the impression of a process without the reliability of one.

**The reviewer owns it.** The practical answer for most organisations is straightforward: the engineer who reviews and approves the PR takes responsibility, same as they would for any code from any source. Agent-generated code doesn't get a different accountability standard. If you approve a PR, you're saying "I've reviewed this and I believe it's correct." The tool that generated the code is irrelevant to that statement. This also means reviews of agent-generated code need to be *real* reviews, not rubber stamps. If the volume of agent-generated PRs is making thorough review impossible, that's a workflow problem to solve, not a standard to lower.

**For regulated industries.** Document your agentic workflow as part of your SDLC documentation. Which models are used, what version, what guardrails are in place, what review process agent-generated code goes through before it reaches production. Auditors want to see a *process*, not perfection. A documented process that includes "AI-assisted code generation with mandatory human review and CI verification" is auditable. An undocumented process where engineers use whatever tools they like with no consistent approach is not. If you're in fintech, healthcare, or anything with regulatory oversight, get this documented before someone asks for it.

**Keep session logs.** Retain logs of agent sessions, especially for code that touches sensitive systems — billing, authentication, data handling, anything with regulatory implications. Not because you'll read them routinely, but because you might need them during an incident review. "What did the agent see when it generated this code? What prompt produced this output? What context was it working with?" These are

questions you want to be able to answer six months later. Most agentic tools can export or log sessions. Set up the retention before you need it. The cost of storage is trivial compared to the cost of not having the logs when compliance comes knocking.

## 16.8 Hiring Changes

If agents handle the mechanical parts of coding, what do you actually need from an engineer?

Raw coding speed matters less. The engineer who could bang out a perfect binary search on a whiteboard in three minutes has a skill that agents have commoditized. That's not worthless — understanding algorithms still matters — but it's no longer the differentiator.

What matters more:

- **System design.** The ability to decompose a problem into components, define interfaces, and reason about tradeoffs. Agents can implement a design. They can't tell you which design is right for your constraints.
- **Judgment.** Knowing when to use the agent and when to think. Knowing when the agent's output is wrong even though it looks right. Knowing which corners to cut and which to protect.
- **Communication.** The ability to articulate intent clearly — to agents, to teammates, to stakeholders. Vague thinkers get vague agent output. Precise thinkers get precise results.
- **Problem decomposition.** Breaking a large task into agent-sized pieces is a skill. It's related to system design but more tactical. The engineer who can turn a Jira epic into ten well-scoped agent prompts will outperform the one who pastes the whole epic into a chat window.

The interview that asks "implement this algorithm on a whiteboard" is testing a skill that matters less every year. The interview that asks "here's a system with these constraints — walk me through how you'd build it, what tradeoffs you'd make, and how you'd verify it works" is testing the skills that matter *more* every year.

Hire for judgment. You can always give them an agent for the rest.

# 17 Final Words

My oldest son is five. He can't read yet, not really — he sounds out words on cereal boxes and street signs, proud of every syllable. But he knows what my computer does. He's seen me talk to it, seen text appear on the screen in response, seen me nod or shake my head and talk again. One evening he climbed onto my lap, watched an agent refactor a module in real time — files opening and closing, tests running, green checkmarks appearing — and said, "Is the computer fixing itself?"

I didn't have a good answer. I still don't, not entirely. But that question stuck with me through every chapter of this book. Because what I realized, sitting there with him, is that I wasn't writing a book about tools. I was writing a book about what it means to be an engineer at the exact moment the definition of engineering is being rewritten — and about what we carry forward into whatever comes next.

This chapter is not a summary. You don't need me to recap fifteen chapters you just read. This is the stuff I want to say to you directly, one engineer to another, before we part ways.

## 17.1 What I Believe

I believe the craft is not dying. It is being *compressed*. A decade of boilerplate and plumbing and ceremony is collapsing into intent and judgement. What remains when you strip away the typing is the thing that was always the actual job: knowing what to build and knowing when it's right.

I believe agents are amplifiers, not replacements. Give an agent to a mediocre engineer and you get mediocre software at terrifying speed. Give one to a great engineer and you get something that, frankly, makes the last twenty years of software development look like we were building highways with shovels.

I believe the engineers who thrive will be the ones who master the *boring* things — context, guardrails, testing, conventions, clear thinking — while everyone else chases the shiny new model announcement. Tools change every quarter. Judgement compounds over a career.

I believe we are not being replaced. We are being promoted. From typists to captains. From coders to *engineers*, in the fullest sense of the word. The question is whether you accept the promotion.

I believe this shift is *good*. Not easy. Not painless. But good. Because the part of our work that's being automated was never the part we loved. Nobody became an engineer because they dreamed of writing boilerplate JSON parsers. We became engineers because we wanted to build things that matter. Now we can.

## 17.2 What I Got Wrong

I'll be honest with you: I changed my mind at least three times while writing this book.

When I started the chapter on sandboxes, I thought container isolation was overkill for most workflows. By the time I finished it, after an agent had rm -rf'd a directory I cared about on a Tuesday afternoon, I believed sandboxing was non-negotiable. That experience made it into the chapter. The conviction behind it is scar tissue.

I originally wrote the multi-agent chapter with the assumption that orchestrating five or six agents simultaneously was the natural end state — a factory floor of autonomous workers. I've pulled back from that. The coordination overhead is real, the failure modes multiply, and I've found that two or three well-directed agents outperform six unsupervised ones almost every time. The chapter reflects where I landed, but I may land somewhere different in six months.

I was also, for a while, too dismissive of local models. I wrote an early draft that basically said "just use the commercial APIs." Then I spent a weekend running a fine-tuned local model on a codebase with proprietary constraints and realized there's a whole world of use cases where local

is not just viable but *necessary*. The chapter on local versus commercial models exists because I was wrong and had to correct myself.

Parts of this book are probably already wrong in ways I can't see yet. The landscape moves that fast. But the specific tools were never the point. If I've helped you build a mental model — a way of thinking about autonomy, trust, and structure — then the book did its job, even when every code example in it is outdated.

## 17.3 The Crew Metaphor, One Last Time

I grew up in Denmark, near the water. If you've sailed in Scandinavian waters, you know the light in late summer — low and golden, the kind of light that makes the sea look like hammered copper. I remember a crossing once, a small boat, four of us. The wind shifted hard and suddenly we were all moving without speaking. One person on the jib, one on the mainsheet, one trimming, one at the helm. No commands. Just trust built over dozens of previous sails.

That's what agentic engineering feels like on a good day. You at the helm, agents trimming and adjusting, the work flowing because you've put in the hours to build shared context — through conventions, through guardrails, through test suites that catch mistakes before they matter. You don't need to shout instructions. The system *knows*.

And then there are the bad days. The days the wind dies and an agent hallucinates an API that doesn't exist, or rewrites a module you didn't ask it to touch, or passes all the tests because it deleted the ones that were failing. Those days, you're bailing water and swearing. That's sailing too.

But I should be honest about something, because this book doesn't work if I'm not.

The metaphor of a *crew* implies loyalty. Continuity. Shipmates you've sailed with before, who know your habits, who anticipate the next command. That's a beautiful image. It's also not how I actually work.

Most days, what I actually do is spin up an agent, give it a job, take the output, and throw it overboard. Then I spin up another one. They don't

remember the last session. They don't know what I asked the previous agent to do. Each one arrives fresh, does its work, and disappears. It's less a loyal crew and more like hiring dockworkers at every port — you brief them, you watch them work, you pay them off, and at the next port you do it again.

And that's fine. That's how most real crews worked throughout maritime history. The ship was the continuity. The captain was the continuity. The charts, the logbook, the rigging — those persisted between voyages. The crew was often assembled for a single crossing and dissolved at the destination. What made it work was not that the sailors knew the captain. It was that the captain knew the *ship* — and had systems good enough that any competent sailor could step aboard and be useful.

That's what your codebase is. That's what your conventions, your test suites, your CLAUDE.md files, your guardrails are. They're the ship. Every new agent you spin up is a fresh crew member stepping aboard a well-rigged vessel. They don't need to know your history. They need to know the ship. And if you've built the ship well, they'll be productive in minutes.

So yes — throw them overboard. Spin up new ones. That's not a failure of the metaphor. That's the metaphor working exactly as intended. The crew is disposable. The ship is not.

You're the captain. You were always the captain. The crew just arrived — and they'll keep arriving, fresh and ready, every time you need them.

## 17.4 Thank You

Thank you for reading this book. I mean that. You traded your time and attention for my words, and I don't take that lightly. I hope I earned it.

Thank you to the community — the engineers in forums, in Discord servers, in open source repos — who are sharing their experiments, their failures, their hard-won insights. This book was shaped by hundreds of conversations I didn't have alone.

And thank you, I suppose, to the agents themselves — who helped me write code, debug problems, and occasionally generate prose so bad it reminded me why human judgement still matters. You're a good crew. You're getting better. And I suspect that by the time my son is old enough to read this book, you'll be something none of us quite predicted.

## 17.5 Go Build Something

Here's what I want you to do.

Tonight — not tomorrow, not next week, *tonight* — open your terminal. Pick a bug you've been avoiding. That one you keep moving to the bottom of the backlog, the one that's annoying but not urgent, the one that lives in a part of the codebase you'd rather not touch. Point an agent at it. Give it context. Set a guardrail. Watch what happens.

Maybe it fixes the bug in four minutes and you feel the ground shift under your feet, the way it shifted under mine that evening with the migration script. Maybe it makes a mess and you learn something about how to give better instructions. Either way, you'll know more than you did before.

Then go bigger. Set up worktree isolation and run two agents in parallel. Write a CLAUDE.md file for a project you care about. Build a test suite that's good enough to be your safety net when agents are committing code. Refactor that module everyone is afraid of — but this time, with a crew.

Then go bigger still. Introduce agentic workflows to your team. Share what you've learned — the wins *and* the disasters. Write up your own war stories. Contribute to the collective knowledge of a discipline that's still being invented.

Because that's what this is: a discipline being invented, right now, by the people willing to do the work. Not by the people writing blog posts about the future. Not by the people waiting for the perfect tool. By the people who open a terminal today and build something real with what they have.

The engineers who will define this era are not waiting for permission. They are not waiting for certainty. They are shipping, breaking things, learning,

and shipping again — with a crew at their side that gets a little better every day.

I hope you'll be one of them.

*Rasmus Bornhøft Schlünsen*
*March 2026*

*To my kids —*
*you're the best crew I've ever had.*
*This whole thing was for you.*
*Always.*