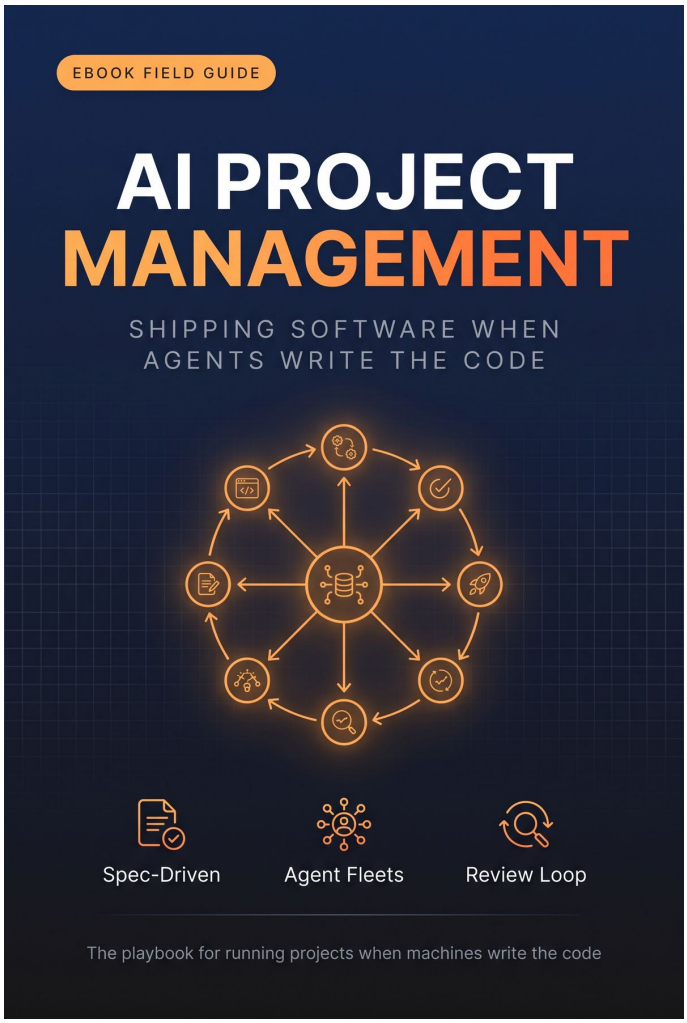

AI Project Management: Shipping Software When Agents Write the Code

Something strange happened to my job this year. The typing got cheap. An agent can turn a paragraph of intent into a working pull request while I refill my coffee, and yet the projects did not get faster. Not really. More code shipped, more PRs merged, and somehow the same features took roughly the same number of weeks to actually land.

That gap is what this guide is about. When machines write the code, the scarce thing is no longer the writing. It's the specifying, the slicing-up, and the reviewing. Project management stopped being about handing out work and started being about feeding agents instructions clear enough that their output is worth keeping, then catching the parts that aren't.

I've been running this way for a while now: specs before code, work cut into agent-sized pieces, a small fleet on separate worktrees, and a review loop that assumes the machine got something subtly wrong. This is the playbook I wish someone had handed me. It covers spec-driven development, decomposition, acceptance criteria an agent can test against, parallelism without chaos, the review loop, governance for code you didn't write, and the handful of metrics that tell you whether any of it is actually working.

It's written for engineers and tech leads who already use agents to write code and want to run real projects with them, not demos.

This is a living document and will be updated as the tools and the workflows keep evolving.

Roger Stringer · rogerstringer.com

July 24, 2026

Contents

The Bottleneck Moved

For most of my career, the constraint on shipping software was fingers on keyboards. You could have a perfect plan and a clean spec, and it still took weeks because someone had to sit there and write every line. Estimation, staffing, sprints, all of it existed to manage one scarce resource: engineering time spent typing.

That resource stopped being scarce. I can describe a feature in a paragraph and have a working pull request back before I've finished my coffee. The first time it happened I felt like I'd been handed a cheat code. Then I looked at the calendar a month later, and the projects were landing at about the same pace they always had.

This is the thing nobody warns you about, and the data backs it up hard. Teams leaning on AI complete more tasks and merge far more pull requests, one study put the PR number near double, and yet system-level delivery stays roughly flat. Individual output goes up. Shipped, trustworthy software does not follow at the same rate.

Here's why. If typing was 40% of the real work, and you make typing nearly free, you've removed 40% at best. The other 60% is still sitting there, and now it's under more pressure than ever. Specifying what you actually want. Breaking it into pieces that fit. Reviewing output you didn't write and can't fully trust. Those didn't shrink. They grew, because the machine can now generate wrong code much faster than it used to.

So the job changed. Managing an AI-assisted project is not about assigning work to people and tracking their progress. It's about three things that used to be background noise and are now the whole game: writing intent down clearly enough that an agent can act on it, cutting work into pieces an agent can finish in one go, and running a review loop that assumes the output is subtly broken until proven otherwise.

I want to be blunt about the failure mode, because I lived it. You get excited, you point the agent at a fuzzy request, it produces a lot of plausible code, and you merge it because it looks right and the tests pass. Three weeks later you're paying it back with interest. The reported numbers are ugly: pull requests with AI-assisted code carry around 1.7 times more issues than human-written ones, and teams see technical debt climb 30 to 40% within six months of going all in. That's a process problem.

The rest of this guide is the process. We'll start where every good project now starts, which is not the code. It's the spec.

Spec Before Code

The single highest-leverage habit I've picked up is boring to describe and quietly enormous in practice: write the spec before you let the agent write the code.

There's a name for the failure it prevents. People call it vibe coding, and I don't mean that as an insult, I do it too when I'm exploring. You type "add login," the model picks reasonable defaults, and out comes something. The problem is that "reasonable defaults" almost never match what you actually had in your head. The model didn't know you wanted magic links instead of passwords, or that sessions had to survive a server restart, or that this had to reuse the existing user table. It couldn't know. You never said.

Spec-driven development flips the order. You write down what the system should do, in enough detail that the intent is unambiguous, and that document becomes the source of truth. Not the code. The code is derived from the spec, and when they disagree, the spec wins and the code gets fixed. In 2026 this stopped being a niche idea and became the default. Every serious tool shipped a flavor of it: GitHub's Spec Kit with its specify, plan, and tasks commands, AWS Kiro, the BMAD method with its cast of planning agents, Google's Antigravity. They differ in ceremony but agree on the core move.

Why does this work so well with agents specifically? Two reasons. First, intent drift. A vague prompt leaves a hundred small decisions to the model, and it will make all hundred, quietly, and you'll discover them during review. A spec makes those decisions up front where they're cheap. Second, context decay. Once a codebase grows past what the agent can hold in its head, it starts forgetting earlier decisions and contradicting itself. A written spec is external memory the agent can re-read every time, so decision number one still holds when it's working on decision number fifty.

You don't need heavy tooling to get most of the benefit. A spec can be a markdown file in the repo. I like a simple shape: what we're building and why, the key decisions and constraints, what's explicitly out of scope, and how we'll know it works. That last part matters enough that it gets its own chapter later. The out-of-scope section is underrated. Agents love to be helpful and will happily refactor three files you never mentioned. Telling it what not to touch is as valuable as telling it what to build.

Here's the part that sells it. Writing a real spec for a feature takes me thirty to sixty minutes. Skipping it and letting the agent improvise costs me two to four hours of cleanup, review churn, and rework. That's not a close call. The spec is the cheapest hour you'll spend on the whole feature.

A spec is a document about the whole feature, though. An agent doesn't work on a whole feature well. It works on pieces. So the next move is turning that spec into work small enough for a machine to actually finish. That's decomposition, and it's where a lot of projects quietly fall apart.

Try this

1. Take the next feature you were about to hand an agent as a one-line prompt. Stop, and instead write a spec first (there's a fill-in template in the toolkit at the end of this guide).
2. Force yourself to fill the **out-of-scope** section with at least three things the agent should *not* touch. This is the part that prevents the most rework.
3. Now hand the agent the spec instead of the one-liner, and note where its output lands closer to what you actually wanted. That delta is the spec earning its hour back.

Decomposition: Agent-Sized Work

Ask an agent to "build the billing system" and you'll get a demo. It'll look impressive for about ten minutes, until you try to use it and find the edges are made of wet cardboard. Ask it to "add a function that, given a subscription and a proration date, returns the amount owed in cents, with tests," and you'll get something you can actually keep.

The difference is size. Agents are genuinely good at small, well-bounded tasks and genuinely bad at large fuzzy ones, and the gap is wider than it is for humans. Decomposition, taking a big goal and cutting it into pieces the machine can finish, is the least glamorous skill in this whole discipline and probably the most important. It's the thing that separates a coding agent from an expensive autocomplete.

My rule of thumb: every task should be small enough to implement, test, and verify in a single focused session. If I can't picture the agent finishing it and me checking it in one sitting, it's too big and gets split. Each piece should be three things. Specific, so the agent knows exactly what "done" looks like. Achievable with the tools it actually has. And ordered, so later pieces build on earlier ones instead of colliding with them.

That ordering point is worth slowing down on. The naive version of decomposition is a flat checklist, and flat checklists lie, because they pretend everything is independent when it isn't. The honest version is a dependency graph. Some tasks have to happen before others. Some don't depend on each other at all and can run in parallel. Once you see the work as a graph instead of a list, two things pop out: the critical path that actually determines how fast the feature lands, and the independent branches you can hand to different agents at the same time. We'll use that second insight hard when we get to parallelism.

There's a brownfield tax I have to mention, because it'll bite you if you don't plan for it. Agents shine on greenfield work, the reported gains run 35 to 40% on clean new code. Drop the same agent into a large, messy, existing codebase and the gain collapses to 10% or less. Decomposition is your main lever against this. In a gnarly codebase, smaller tasks with tighter boundaries and explicit "here are the three files involved, touch nothing else" scoping are the difference between help and havoc.

One more thing I've learned the hard way. Decomposition is where you should be spending your senior brain, not your junior one. Cutting the work well requires understanding the system, anticipating the interactions, and knowing where the bodies are buried. That's exactly the judgment the agent doesn't have. Hand it well-shaped tasks and it flies. Hand it a fuzzy mountain and it produces a fuzzy mountain of code.

But "specific" and "small" still aren't enough on their own. A task can be small and clear and still produce something wrong, because you never pinned down what correct actually means. That's the job of acceptance criteria, and if you write them the way most people do, the agent can't use them.

Try this

1. Take a spec from the last chapter and decompose it into tasks, applying one hard filter: could the agent finish this *and* could you verify it in a single sitting? If not, split it again.
2. Draw the dependencies, even as a rough sketch. Mark which tasks truly depend on others and which don't. The ones that don't are your parallel branches for later; the chain is your critical path.
3. For any task that touches existing, messy code, add an explicit scope line: "these files only, touch nothing else." That one line is your main defense against the brownfield tax.

Acceptance Criteria an Agent Can Test

Most acceptance criteria are wishes wearing a suit. "Notifications should be fast." "The UI should feel responsive." "Handle errors gracefully." A human reviewer squints at those and fills in the gaps from context. An agent can't. Worse, it will confidently declare all three satisfied, because from its point of view they are. It did something with notifications, the UI does render, and there's a try-catch in there somewhere.

The fix is to make every criterion a concrete, checkable fact. Not "notifications should be fast" but "the notification appears in the sidebar within two seconds of the comment being saved." Not "handle errors gracefully" but "if the payment API returns a 402, show the retry banner and do not clear the cart." The test is simple: could you write an automated check that returns pass or fail against this sentence? If not, it isn't acceptance criteria yet, it's a vibe.

This matters more with agents than it ever did with people, because of what happens next. When your criteria are concrete, the agent can write tests against them, often before writing the implementation. You're not just describing the target, you're handing the machine a way to check its own work. That closes a loop that used to require you standing over its shoulder. I'll routinely tell an agent to write the failing test first, from the criteria, then make it pass. The criteria become executable, and "done" stops being a matter of opinion.

I keep two layers, and I'd encourage you to as well. The first is per-task acceptance criteria, which answer "did we build the right thing for this specific piece?" Those change with every task. The second is a standing Definition of Done, the bar every task clears no matter what: tests pass, no linter errors, no secrets committed, the change is behind a flag if it touches a live path, docs updated if the public interface moved. The Definition of Done is your ambient quality floor. The acceptance criteria are the specific target on top of it. An agent needs both spelled out, because it will not infer your standards from vibes the way a teammate who's been around for a year would.

Here's a subtle trap. Agents are eager to please, and eagerness plus vague criteria equals scope creep. Ask for a function and give loose criteria, and you might get the function plus a refactor of two neighboring files plus a new helper module you didn't want. Tight criteria, plus an explicit statement of what's out of scope, keep the machine on the rails. Constraints aren't the enemy of a good agent. They're what let it run without a babysitter.

When your criteria are this concrete, something nice happens at the project level. A task's status stops being a guess. It's either meeting its checks or it isn't. That's the raw material for actually running the board, where the spec and the pieces and the criteria turn into a flow of work you can watch move. That's next.

Try this

1. Take the fuzziest acceptance criterion you've got and rewrite it until you could hand it to someone as an automated test. "Fast" becomes "within 2 seconds"; "gracefully" becomes a specific status code and a specific UI response.
2. Tell the agent to write the failing test *first*, straight from that criterion, then implement until it's green. Feel how "done" stops being an argument.
3. Write your standing Definition of Done once (the toolkit has a starter) and paste it into every task. If you find yourself re-explaining the same standard twice, it belongs in the DoD, not the task.

Running the Board

I resisted this chapter for a while, because "running the board" sounds like project-manager theater, the stuff engineers roll their eyes at. But the board is where all the earlier pieces become a real workflow instead of a pile of good intentions, and with agents in the mix the board does something new. The board is now the queue the agents pull from.

Start with the flow of a single feature. You have a spec. You decompose it into tasks, each with concrete criteria. Now those tasks need to live somewhere an agent can pick them up: issues, a task file, whatever your stack uses. The move that pays off is writing each issue as a self-contained brief. Not "implement caching," but the actual context: the goal, the relevant files, the constraints, the acceptance criteria, and the out-of-scope note. This is the same discipline as the spec, one level down. A well-written issue is one an agent can execute without you re-explaining the whole system every time, and that reusability is the point.

The frameworks formalize this and it's worth seeing the shape even if you never adopt one wholesale. BMAD, for example, runs a little cast: an analyst and a product manager shape the requirements, an architect makes the technical decisions, and a scrum-master agent turns all that into hyper-detailed story files that carry full context down to the developer agent. You may not want that much ceremony, I usually don't, but the insight underneath is solid. The plan is not separate from the work. The plan, written densely enough, is the work, because it's what the agent consumes.

Two habits keep my board honest. First, one task in flight per agent, not five. It's tempting to fan out wide, and we will fan out in the next chapter, but each stream needs its own clean context or the quality craters. Second, the board reflects reality, not hope. A task is not "done" because the agent said it finished. It's done when it has met its criteria and passed review. I keep a hard column between "agent produced something" and "verified," because collapsing those two is how bad code sneaks into main.

That gap between generated and verified is the whole ballgame, and it's why the next two chapters exist. But before we get to catching mistakes, there's a multiplier we've been circling. Once work is cut into independent, well-specified pieces sitting on a board, you don't have to run them one at a time. You can run a fleet. Done carelessly that's a recipe for chaos. Done well it's the biggest speedup on offer.

Parallelism Without Chaos

The real unlock, the thing that actually moves a project's pace instead of just its PR count, is parallelism. One agent working through tasks in sequence is a faster typist. Several agents working independent branches of the dependency graph at once is a different kind of leverage. This is where you stop being an implementer and start being an orchestrator, and honestly it took me a while to get comfortable with the shift.

The mechanism that makes it sane is the git worktree, and it's become the standard for a reason. A worktree gives each agent its own checkout of the repo on its own branch, in its own directory. They can all work at the same time without stepping on each other's files, because they're literally not in the same files. No merge chaos mid-flight, no two agents fighting over the same buffer. When each finishes, its branch comes back through review like any other. If you take one practical thing from this chapter, it's this: give every parallel agent its own worktree. It's the cheap trick that makes the whole pattern hold together.

But parallelism is only safe when the work is genuinely independent, and this is where the earlier chapters pay off. Remember the dependency graph from decomposition? That graph is your fan-out plan. The independent branches, the tasks that don't rely on each other, are exactly what you can run concurrently. The critical path, where each step needs the last, stays sequential no matter how many agents you own. Trying to parallelize a chain just produces agents building on top of work that doesn't exist yet, and you'll spend more time reconciling than you saved. So the honest sequence is: decompose, find the independent branches, fan those out, keep the chain in order.

There's a ceiling here, and it's you. Every agent you run is output you have to review, and review does not parallelize the way generation does. I can spin up six agents in six worktrees in a minute. I cannot review six streams of unfamiliar code in a minute. So I match the fan-out to my actual review capacity, not to how many agents I can technically launch. A fleet that generates faster than you can verify is a debt machine. The right number of parallel agents is the number whose output you can actually keep up with, and for most work that's a small number, not twenty.

The orchestrator mindset is the adjustment that matters most. Your job is no longer to write the code. It's to keep the whole system coherent: the specs current, the tasks well-shaped, the branches flowing back and getting merged, the agents unblocked. You're conducting, not playing. And the most important part of conducting, the part that decides whether all this speed produces software you'd actually ship, is what happens when the branches come back. That's the review loop, and it's now the real cost center of the whole operation.

Try this

1. Set up a git worktree for a branch (`git worktree add ../feature-x feature-x`) and run an agent in it while another agent works your main checkout. Watch them not collide, because they're in different directories.
2. From your dependency sketch, pick only the genuinely independent branches to fan out. Resist parallelizing the critical path. That just makes agents build on work that isn't there yet.
3. Before you launch the fleet, set a hard cap: how many streams of unfamiliar code can *you* actually review well today? Run that many, not one more. The bottleneck is your eyes, not the agents.

The Review Loop

Here's the uncomfortable truth of this whole way of working: when the machine writes the code, review stops being a formality and becomes the main event. The generation is cheap and fast. The reviewing is where your time actually goes now, and if you don't design for that, the speed you gained on the front end quietly leaks back out.

The numbers make it concrete. Senior engineers in 2026 report spending 20 to 35% more time on code review, largely from AI-assisted pull requests coming in faster and looser. That's the hidden cost behind the productivity paradox from the first chapter. All those extra merged PRs have to be read by someone, and reading unfamiliar code is slower than writing familiar code. If your process treats review as an afterthought, this is exactly where it jams.

I think of a single task's lifecycle as a loop, not a line. Plan, do, assess, review. The agent plans its approach, does the work, assesses its own output against the acceptance criteria, and then a human reviews. When review finds a problem, the task doesn't fail, it routes back into the loop with specific feedback and comes around again. The key is that self-assessment step in the middle. Because you wrote concrete, testable criteria back in chapter four, the agent can check a lot of its own work before it ever reaches you. That's how you keep the human review step from drowning: you make sure the machine has already caught the mechanical failures, so your attention goes to the things only a human catches.

And those things are specific. I've learned to review AI code differently than human code, because it fails differently. A human junior writes code that's naive but honest, you can usually see what they were thinking. An agent writes code that's confident and plausible and occasionally, quietly, wrong in a way that reads perfectly. It invents an API that doesn't exist but looks like it should. It handles the happy path beautifully and silently drops an edge case. It writes a test that asserts the bug. So I read AI output assuming it's subtly broken until I've confirmed otherwise, and I aim my attention at the seams: the boundaries, the error paths, the places it touches the parts of the system it couldn't see.

A few habits keep the loop fast. Small PRs, because a diff you can hold in your head is one you can actually review, and huge agent-generated diffs are where bad code hides. Feedback that's specific and criteria-shaped, so the next pass through the loop is a real fix and not another guess. And a firm rule that review is not a rubber stamp on green tests, because passing tests only prove the agent satisfied the tests, some of which the agent wrote. Green is necessary. It is nowhere near sufficient.

This all works fine when you're reviewing a feature you understand. It gets harder, and higher-stakes, when the code touches things where a subtle mistake becomes an incident. Auth, payments, infrastructure, anything security-sensitive. For those, ordinary review isn't enough, and you need an extra layer of governance around code you didn't write. That's next.

Try this

1. On the next agent PR, review it against a different instinct: assume it's subtly wrong until proven otherwise, and spend your attention on the seams, error paths, boundaries, anything touching parts of the system the agent couldn't see. Don't start at the happy path; it's almost always fine.
2. If a diff is too big to hold in your head, send it back to be split before you review it. Big agent diffs are where bad code hides.
3. Once this week, deliberately distrust green tests: find a passing test the agent wrote and check whether it actually asserts the *right* behavior, or just asserts what the code happens to do.

Governance for Code You Didn't Write

There's a category of change where "looks right and tests pass" is not good enough, and if you're running agents at any real scale you will hit it constantly. Authentication. Payment flows. Cryptography. Infrastructure and deploy config. Anything touching customer data or a live execution path. A subtle bug here becomes an incident, a breach, or a bill. And this is precisely where AI-generated code is weakest.

The research on this is genuinely alarming, and I think everyone running agents should sit with it for a second. Studies consistently find that 30 to 40% of AI-generated code snippets contain at least one known class of security vulnerability. The volume of new security findings from AI code has jumped roughly tenfold from late 2024. Code duplication has risen fourfold. None of this means agents are useless. It means agents ship vulnerabilities at scale and speed, and a process that doesn't account for that is going to ship them straight to production.

So I run tiered gates. Most changes go through the normal review loop from the last chapter, and that's fine. But when a change touches the sensitive categories, the bar goes up automatically: mandatory human review by someone who knows that subsystem, security-focused scanning, and a much higher burden of proof before it merges. The trigger is the blast radius, not the size of the diff. A three-line change to how sessions are signed gets more scrutiny than a three-hundred-line change to a settings page. The point is that "an agent wrote it" plus "it touches auth" should light up differently than ordinary work, and that difference should be built into the process, not left to whoever happens to be reviewing that day.

The other habit worth adopting is provenance. For code an agent produced, especially anything that got merged with real autonomy, keep a record of how it came to exist: which model, what prompt and context, when, what validation it passed, who approved it. This sounds bureaucratic until the first time something breaks in production and you're trying to understand where a weird piece of code came from and why. With human code you can go ask the person. With agent code, the provenance record is the person. It's how you trace a bad pattern back to its source, and how you learn whether a particular prompt or setup keeps producing the same class of mistake so you can fix it upstream.

I want to be clear that this is the same instinct that makes us gate production deploys and require review on human PRs, applied honestly to a source of code that is faster, tireless, and measurably more likely to introduce a vulnerability without noticing. Governance is what lets you say yes to the speed without betting the company on it.

All of this, the specs, the decomposition, the reviewing, the gates, raises an obvious question. Is it working? Are you actually shipping better software faster, or just generating more of it and feeling busy? Answering that honestly turns out to be harder than it sounds, because the obvious metrics lie now. That's the next chapter.

Measuring What's Real

The most dangerous dashboard in 2026 is the one that shows your team crushing it. More commits, more pull requests, more tasks closed, all trending up and to the right. It feels like winning. And it can be completely disconnected from whether you're actually delivering, because AI made the vanity metrics trivially easy to inflate while leaving the real ones untouched.

This is the paradox from chapter one, now as a measurement problem. When 30 to 70% of your committed code is machine-generated, the old throughput numbers stop meaning what they used to. Deployment frequency and lead time, the classic DORA metrics, get gamed by sheer volume. Studies show AI adoption bumping throughput a modest 2 to 18% while change failure rates climb, so you can post better-looking speed numbers and ship more breakage at the same time. If you only watch output, you'll conclude everything's great right up until the incidents tell you otherwise.

So I don't throw DORA out, I treat it as a floor and add two things on top. The first is attribution. Track what fraction of your code is AI-generated, and then use that as a lens on every other metric. Look at rework and churn for AI code versus human code separately. Compare failure rates on AI-heavy PRs against human-only ones. Without that split, every number is an average that hides the story. The interesting question is never "how are we doing," it's "how is the AI-generated part doing compared to the rest," and you can't answer that if you don't tag it.

The second is quality signals, because that's where the AI tax actually shows up. Rework rate is the one I watch hardest, and the industry numbers are sobering: only about 7% of teams keep rework under 2%, and AI code generation pushes that number up if you're not paying attention. Review time is another, since a rising review burden is the direct symptom of the bottleneck moving. Rollback frequency, defect rates, PR size, technical-debt indicators. These are the metrics that tell you whether all that generated code is an asset or a liability, and they're exactly the ones a volume dashboard ignores.

My favorite signal is the softest and I think the truest. Ask your engineers what percentage of their week went to work that needed deep focus and original thought. The whole promise of agents is that they absorb the rote work and free people up for the hard, creative parts. If that number is going up, the tools are doing their job. If it's going down, your people aren't being liberated by AI, they're being captured by it, stuck babysitting and cleaning up after machines. No throughput chart will tell you that. The question does.

The uncomfortable finding underneath all of this is that the returns don't live in the tools. They live in the system around the tools: the clarity of the workflow, the quality of the platform, the alignment of the team. Which is really what this whole guide has been about. So let's put the pieces together into something you can actually run.

Your Agentic PM Loop

Let me pull the threads into one loop, because that's what this really is. Not a collection of tips, a repeatable cycle you run over and over, tightening it each time.

It goes like this. Start with a spec, because the intent has to be written down before the machine can act on it well. Decompose the spec into agent-sized tasks, small enough to finish and verify in one sitting, arranged as a dependency graph rather than a flat list. Give each task concrete acceptance criteria the agent can test itself against, sitting on top of a standing Definition of Done. Load those tasks onto a board as self-contained briefs. Fan the independent branches out across agents in separate worktrees, kept to a number you can actually review. Run each task through the plan-do-assess-review loop, reading the output as if it's subtly broken until proven otherwise. Gate the sensitive changes harder and keep provenance on what the machines produced. And measure the real signals, attribution and rework and deep-focus time, not the vanity volume. Then feed what you learned back into the next spec. That's the loop.

None of the individual pieces are exotic. What's new is that the center of gravity moved. The old project management put people at the center and managed their time. This one puts specification and verification at the center and treats generation as the cheap, fast, slightly untrustworthy step in between. Get comfortable with that inversion and everything else follows.

Here's where it gets fun, and where this connects to the rest of what I write about. This whole loop can run on infrastructure you own. The specs and tasks live in a system like Directus. The orchestration, the routing, the notifications, the glue between the board and the agents, runs in something like n8n. The agents themselves reach your tools through MCP. Wire those together and the loop stops being a manual ritual and becomes a system: work comes in, gets specced, gets decomposed, gets handed to agents, comes back for review, and the state of the whole thing lives on a box you control. That's the agentic stack, and running a project on top of it is exactly this loop, automated as far as you're comfortable automating it.

I'll leave you with the mindset more than the mechanics, because the tools will keep changing and the mindset won't. Cheap generation is not the same as cheap delivery. The machine writing the code was never the hard part, we just didn't notice because it used to be bundled with all the parts that are hard. Specification. Decomposition. Judgment. Review. Those are still yours. Agents made them more valuable, not less. Manage those well and the agents will make you genuinely faster. Manage them badly and the agents will help you produce more work that never ships. The loop is how you stay on the right side of that line.

Toolkit: The Feature Spec Template

The guide made the case in prose. The rest of it is the toolkit, the templates and checklists I actually use, stripped down so you can lift them straight into your own work. Start here, with the artifact everything else hangs off: the feature spec.

This is the markdown file from the "Spec Before Code" chapter, as a fill-in. Drop it in the repo (I keep these in `/specs/<feature>.md`), fill the brackets, and it becomes the source of truth the agent derives code from. Thirty to sixty minutes here saves the two-to-four hours of cleanup that improvising costs.

```
# Spec: [feature name]

## What & why
[One paragraph: what we're building and the problem it solves.
If you can't say why in a sentence, you're not ready to build it.]

## Key decisions & constraints
The decisions you want made on purpose, not defaulted by the model:
- [e.g. "magic-link auth, not passwords"]
- [e.g. "reuse the existing `users` table — do not create a new one"]
- [e.g. "sessions must survive a server restart"]
- [tech constraints, patterns to follow, things it must integrate with]

## Out of scope (read this twice)
The most valuable section. What the agent should NOT touch or build:
- [e.g. "do not refactor the notification system"]
- [e.g. "no new dependencies without flagging first"]
- [e.g. "billing changes are a separate spec"]

## How we'll know it works
The feature-level success conditions (per-task criteria live on the tasks):
- [concrete, checkable outcome]
- [concrete, checkable outcome]

## Open questions
[Anything genuinely undecided. Better named here than guessed by the agent.]
```

Two notes from using this a lot. The **out-of-scope** section is the one people skip and the one that prevents the most rework, an agent left to its own devices will help you in ways you didn't ask for, and this is where you stop it. And keep the spec in the repo, not a doc tool, so the agent can re-read it on every task. That's the "external memory" that beats context decay. When the code and the spec disagree, the spec wins and the code gets fixed.

Toolkit: The Self-Contained Issue Brief

A spec covers a whole feature. An agent executes a single task, and it executes best when the task carries its own context so you're not re-explaining the system every time. This is the self-contained issue brief from the "Running the Board" chapter, as a template. Each decomposed task becomes one of these, on whatever your board is (GitHub issues, a task file, a kanban card).

```
# Task: [specific, single-sitting piece of work]

## Goal
[One or two sentences. What this task delivers — not the whole feature,
just this piece.]

## Context
- Spec: [link to the feature spec this belongs to]
- Relevant files: [the actual paths the agent should look at]
- Depends on: [task IDs that must be done first, or "nothing"]

## Scope
- Touch: [the files/modules this task is allowed to change]
- Do NOT touch: [explicit boundaries — especially in messy codebases]

## Acceptance criteria (each one testable)
- [ ] [concrete pass/fail fact, e.g. "POST /x with a 402 shows the retry
  banner and does not clear the cart"]
- [ ] [concrete pass/fail fact]
- [ ] Tests written against the above and passing

## Definition of Done
[Reference your standing DoD — next chapter — so every task clears the
same quality floor without restating it.]
```

The test of a good brief is simple: **could an agent execute it without you re-explaining the system?** If it needs the surrounding context to make sense, put that context in the brief. This is the same discipline as the spec, one level down, and the payoff is reusability. A well-written brief is one you can hand to an agent cold, which is exactly what makes the board a queue the fleet can pull from instead of a list you have to narrate.

One habit worth keeping: the `Depends on` line is what turns your flat list back into the dependency graph from the decomposition chapter. Fill it honestly and the independent tasks, the ones with no dependencies, reveal themselves as your parallel branches.

Toolkit: The Definition of Done Checklist

Per-task acceptance criteria change with every task. The Definition of Done doesn't. It's the standing bar every task clears no matter what, the ambient quality floor from the "Acceptance Criteria" chapter. Write it once, paste it into every task brief, and stop re-explaining your standards to a machine that won't infer them.

Here's a starter. Cut what doesn't apply, add what your stack demands, and keep it short enough that it actually gets read.

```
# Definition of Done

Every task clears all of these before it's "done" — not "the agent
said so," but verified:

## Correctness
- [ ] All acceptance criteria for the task are met
- [ ] Tests written for the new behavior, and the whole suite passes
- [ ] Tests assert the *right* behavior, not just what the code does

## Quality
- [ ] No linter or type errors
- [ ] No dead code, no debug logging, no commented-out blocks left behind
- [ ] Diff is small enough to review in one sitting (split it if not)
- [ ] Stayed in scope — no unrequested refactors of neighboring files

## Safety
- [ ] No secrets, keys, or credentials committed
- [ ] Change is behind a flag if it touches a live path
- [ ] No new dependency added without being flagged for review
- [ ] Inputs validated; error paths handled, not just the happy path

## Traceability
- [ ] Docs / README updated if the public interface changed
- [ ] For agent-written code on a sensitive path: provenance recorded
  (see the governance chapter)
```

The reason this exists as a separate, standing artifact is the thing that makes agents different from teammates: a human who's been on the team a year absorbs these standards by osmosis. An agent never does. It will produce work that's locally plausible and quietly below your bar every single time, unless the bar is written down and attached to the work. The DoD is that bar, made explicit. Your quality floor, enforced by paste rather than hope.

Rule of thumb: the moment you catch yourself giving the same piece of feedback on a second task, it doesn't belong in the feedback, it belongs in the Definition of Done.

Toolkit: The Sensitive-Change Gate & Metrics That Matter

Two of the guide's ideas are really operating rules you set once and run against: which changes get extra scrutiny, and which numbers you actually trust. Here they are as reference cards.

The sensitive-change gate

Most changes go through the normal review loop. Some get a higher bar automatically, triggered by **blast radius, not diff size**, a three-line change to how sessions are signed outranks a three-hundred-line settings page. Flag a task for the elevated gate if it touches any of these:

- **Authentication / authorization**, login, sessions, permissions, tokens
- **Payments / billing**, anything that moves money or changes what's charged
- **Cryptography**, signing, encryption, hashing, secret handling
- **Infrastructure / deploy config**, anything that changes how prod runs
- **Customer data**, anything that reads, writes, or exposes PII
- **Any live execution path** where a subtle bug is an incident, not a ticket

When the gate trips, the bar goes up: **mandatory human review by someone who knows that subsystem**, security-focused scanning, and a higher burden of proof before merge. And for agent-written code on these paths, record **provenance**, which model, what prompt/context, when, what validation it passed, who approved. With human code you can ask the author later; with agent code, the provenance record *is* the author.

The metrics that actually tell you something

The volume dashboard (commits, PRs, tasks closed) is the one that lies now, AI makes it trivial to inflate while delivery stays flat. Treat DORA as a floor and watch these instead:

Attribution (the lens that makes everything else readable)

- What fraction of merged code is AI-generated? (Tag it, or every other number is a misleading average.)
- Rework/churn on AI code vs. human code, tracked *separately*
- Change-failure rate on AI-heavy PRs vs. human-only PRs

Quality signals (where the AI tax actually shows up)

- **Rework rate**, the one to watch hardest; under 2% is elite and rare
- Review time / review burden, rising review load is the bottleneck moving, made visible
- Rollback frequency, defect rate, PR size, technical-debt trend

The truest signal (and the softest)

- Ask your engineers: what percentage of your week went to work needing deep focus and original thought? Up means the agents are absorbing the rote work as promised. Down means your people are being captured by the machines, not freed by them, stuck babysitting and cleaning up. No throughput chart will surface that. The question will.

Run both cards on a cadence (the gate per task, the metrics per sprint or month) and you get the thing the whole guide is really after: the speed of cheap generation without quietly betting the company on code nobody truly reviewed.

About Roger

I'm Roger Stringer. I build things, break them, and write up what I learned so you don't have to learn it the hard way. These Field Guides come straight out of that work.

Working on something bigger? I take on a handful of fractional CTO (https://rogerstringer.com/guides/the-fractional-cto-field-guide) engagements, helping founders and teams set technical direction, build AI-powered workflows, and actually ship the hard parts. If you're wrestling with the kind of problem this guide covers and want someone in your corner who's done it before, that's exactly what I help with. Drop me a line. (mailto:roger.stringer@hey.com)

And if a guide helped, got something wrong, or you just want to compare notes, I'd love to hear from you:

- **Email**: roger.stringer@hey.com (mailto:roger.stringer@hey.com)
- **X**: @freekrai (https://x.com/freekrai)
- **GitHub**: github.com/freekrai (https://github.com/freekrai)
- **LinkedIn**: linkedin.com/in/rogerstringer (https://www.linkedin.com/in/rogerstringer)

New guides go up as I hit problems worth documenting. Follow along wherever suits you.