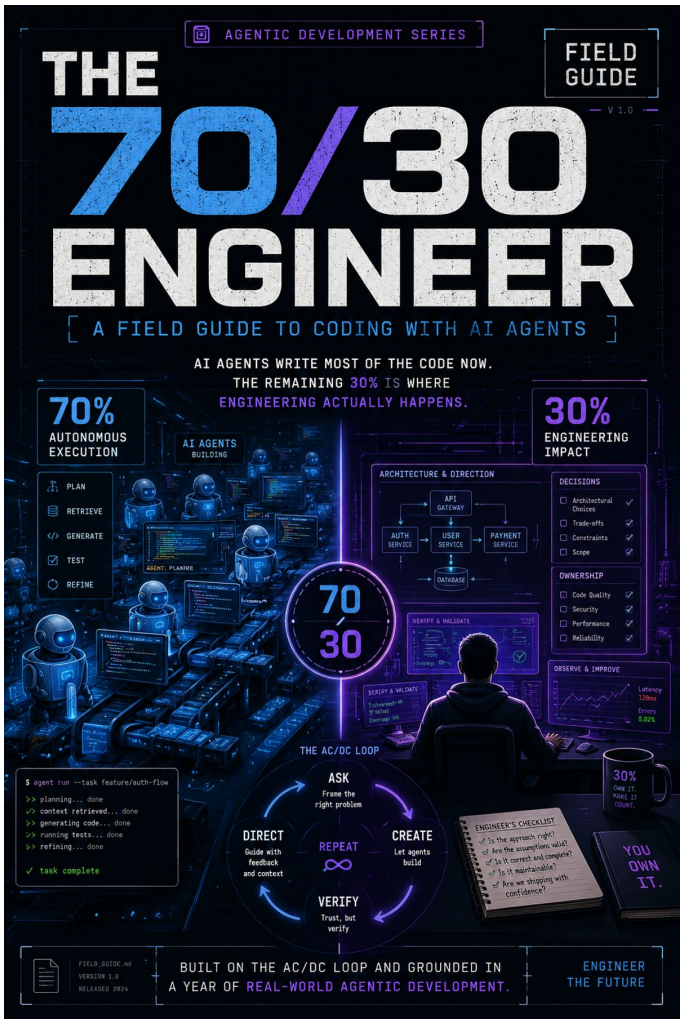

The 70/30 Engineer

The machine does about 70% of the typing now. That part is real, and it's not going back in the box. But the work that's left — guiding the agent, verifying what it produces, and owning the result — is the other 30%, and it turns out that 30% is the entire job.

This is a field guide to working that way on purpose: a real, adoptable workflow for coding with agents, built on the AC/DC loop (Guide, Generate, Verify, Solve), grounded in what a year of agentic development actually taught the industry — the wins, the data, and the failure modes nobody likes to talk about. It's for the engineer who wants to be faster without shipping 1.7x more bugs to prove it.

Roger Stringer · rogerstringer.com

June 23, 2026

Contents

The 70/30 Inversion 3
What a Year of Agentic Coding Taught Us 4
The AC/DC Loop 5
Guide: Context Engineering and Specs 6
Generate: Delegating Well 8
Verify: The New Bottleneck 10
Solve: Close the Loop 12
Putting It Together: One Feature, Whole Loop 13
Owning the 30% 15
Don't Outsource Mastery 16
The Economics of Agentic Coding 17
Becoming a 70/30 Engineer 18
About Roger 19

The 70/30 Inversion

Here's the shift nobody quite prepared us for. A couple of years ago, writing software meant writing software — you sat down, you typed the code, the typing *was* the job. Today, if you're working with a capable coding agent, the machine produces most of the actual lines. Call it 70%. Some days it's more.

The natural panic is: if the machine writes the code, what's left for me? And the answer, once you've worked this way for a while, is the most important thing I can tell you: **the 30% that's left is the entire job.**

Think about what got removed. The agent took over the part that was always the most mechanical — translating a clear intention into syntax. What it did *not* take over is knowing what to build, deciding how it should fit the system, telling whether what came back is actually correct, and standing behind it when it ships. That work didn't shrink. If anything it grew, because now you're doing it at the speed of a machine that can generate a thousand lines while you're refilling your coffee.

So the "70/30" in the title is a deliberate inversion. The 70% — the generation — is the part everyone markets, demos, and gets excited about. It's also the easy part now, and the part where your personal leverage is lowest, because the model is doing it. The 30% — guiding, verifying, owning — is smaller in volume and almost invisible in a demo, and it is where all the difficulty and all the value have quietly concentrated.

The engineers I see thriving in this new world aren't the ones chasing more of the 70%, trying to get the agent to one-shot ever-bigger features while they watch. They're the ones who got very, very good at the 30%. They write specifications a machine can't misread. They've built a verification habit that catches the agent's confident mistakes before anyone else sees them. And they treat every line that ships under their name as theirs, regardless of who typed it.

This guide is about becoming that engineer. It's a real, adoptable workflow — not a hype reel and not a doom essay. We'll build it on a simple loop called AC/DC (Guide, Generate, Verify, Solve), we'll ground every claim in what a year of agentic coding actually taught the industry, and we'll be honest about the failure modes, because pretending they don't exist is exactly how teams end up shipping more bugs faster and calling it progress.

The machine can have the 70%. The 30% is where the engineering lives.

What a Year of Agentic Coding Taught Us

Before we build a workflow, let's be honest about what we actually know, because this field runs on hype in both directions and a working engineer can't afford either flavor.

Start with the most humbling study so far. In 2025, METR ran a proper randomized controlled trial — the gold-standard kind — on experienced open-source developers working in repositories they knew well. The developers expected AI to speed them up by about 24%. Instead, they were roughly 19% *slower* with the tools than without. The truly unsettling part: even after finishing, they still believed AI had made them about 20% faster. The perception gap was the headline finding, not the slowdown itself.

Now, the honest footnote: METR themselves walked it back in early 2026, saying their follow-up methodology was unreliable — partly because developers who love AI increasingly refused to participate in no-AI conditions at all — and concluding, essentially, "we don't actually know." So don't take "AI makes you slower" as settled science. Take the *perception gap* as the durable lesson: we are demonstrably bad at judging our own productivity while using these tools. It feels fast. Feeling fast and being fast are different measurements, and the difference is exactly the review-and-rework time that the next chapters are about.

Then there's the quality data, which is less ambiguous. In December 2025, CodeRabbit analyzed 470 real-world pull requests and found AI-generated code carried about 1.7x more issues overall than human-written code — with security vulnerabilities running 1.5 to 2x higher (improper password handling, insecure object references, and cross-site scripting nearly tripled at 2.74x), logic and correctness issues up around 75%, and performance problems showing up almost 8x as often. An academic study the same year, across half a million Python and Java samples, pointed the same direction: AI code tends to be simpler and more repetitive but more prone to certain high-risk patterns.

None of that means "don't use agents." It means the speed is real *and* the defects are real, and the gap between shipping fast and shipping well is precisely the discipline most teams skip. The acceleration is free; the safety is not.

The adoption numbers tell the rest of the story. Stack Overflow's 2025 survey found something like 84% of developers using or planning to use AI tools — while 46% actively distrust their accuracy, and positive sentiment had actually slipped from the 70%+ of prior years toward 60%. Read that twice: near-universal adoption, widespread distrust, cooling enthusiasm. That's not a fad and it's not a love affair. It's a tool people have decided they can't not use, while remaining wary of it. Which is the correct posture.

And the practitioners who've gone deepest tend to land in the same nuanced place. The ClickHouse team, after a year of running agents against an enormous C++ codebase, summed it up about as well as anyone: the agents work, but they don't work for everything, and the line between "use the agent" and "don't bother" moved repeatedly over the year. They found the wins came from small, tightly specified tasks, not from waving a hand at a vague problem and hoping.

That's the whole evidence base in one breath: huge adoption, genuine speed, measurably more defects, a treacherous perception gap, and seasoned teams insisting on tight scoping and human judgment. Everything else in this guide is a response to those facts.

The AC/DC Loop

If you only take one mental model from this guide, make it this one. It comes from a framing called the Agent Centric Development Cycle — AC/DC — laid out by Manish Kapur in The New Stack, and it's the cleanest way I've seen to think about agentic work. Four stages: **Guide, Generate, Verify, Solve.**

Most of the noise in this industry is about exactly one of those four: Generate. That's the stage where the agent produces code, and it's what every demo shows and every vendor sells. But the framework's real insight is that the whole thing stands or falls on the *other three*. If your Guide stage is weak, the agent starts from wrong assumptions and confidently builds the wrong thing. If your Verify stage is weak, errors compound invisibly until they surface in production. If your Solve stage is weak, you accumulate a growing pile of known problems with no system for actually fixing them.

Here's the loop in plain terms:

- **Guide** — you give the agent boundaries, not just a prompt. The task, yes, but also the architecture it has to fit, the conventions it has to follow, the constraints that live in your head and nowhere else. This is the first and highest-leverage layer of control.
- **Generate** — the agent writes the code. This is the 70%. It's also, increasingly, the part you should worry about least, because it's the part the model is genuinely good at.
- **Verify** — you establish that what came back actually does what it should, safely. Not a glance. A real, repeatable function of the workflow, happening both while the agent works and after it thinks it's done.
- **Solve** — when verification finds problems (it will), you remediate them systematically, re-check the fixes, and feed what you learned back into the Guide stage so the next cycle starts smarter.

Notice that three of the four stages are the 30% from the last chapter — the guiding, the verifying, the owning-and-closing. AC/DC is basically a map of where your attention goes once the machine is doing the typing. Generation sits in the middle, flanked on both sides by the work that actually determines whether you ship something good.

The counterintuitive part, and the part worth sitting with: as the models get better at Generate, the other three stages get *more* important, not less. A more capable agent will produce more code, faster, across more of your system — which means more output flowing into Verify and more decisions flowing out of Guide. Better generation raises the stakes on everything around it.

The next four chapters take the stages one at a time. They're the spine of the whole workflow, so it's worth getting each one into your hands.

Guide: Context Engineering and Specs

Guide is the stage with the highest return on effort, and it's the one people skip. The instinct is to type a quick prompt and let the model figure out the rest. The discipline is to realize that almost every bad agent output traces back to thin guidance — the agent didn't fail, it just filled the gaps you left with plausible guesses.

The practice that's taken over in 2026 has a name: **spec-driven development.** The idea is simple and a little old-fashioned: don't prompt the agent to write code, give it a specification and let it implement. There's a whole ecosystem now — GitHub's Spec-Kit (which integrates with basically every agent, Claude Code and Cursor included), plus OpenSpec, Kiro, and others — but the tooling matters less than the flow, which goes:

1. **Spec** — write down what you're building and why, in terms of behavior and outcomes, before any code. What the feature does, the rules it must obey, what's explicitly out of scope. Resist naming the tech stack here; that's the plan's job.
2. **Plan** — turn the spec into a technical approach: architecture, data model, contracts, the constraints from your project's "constitution" (more on that in a second). Commit it alongside the spec.
3. **Tasks** — decompose the plan into atomic, independently shippable units, each with one objective, the files it touches, and an acceptance check. A good task list reads like a checklist a competent junior could execute without asking you questions.
4. **Implement** — *now* the agent generates, one task at a time, each ending in a verification step.

The single best argument for working this way I've seen put as a rule of thumb: an extra hour spent writing a clear spec saves about three days of agent thrash and three weeks of code review. That ratio matches my own experience closely. The thrash — the agent confidently going down the wrong road, you yanking it back, it going down a slightly different wrong road — is almost always a spec problem wearing a prompt costume.

Here's that flow on one small, real feature — rate-limiting a public API endpoint — kept deliberately short so you can see the shape. We'll carry this same feature all the way through the loop in a later chapter, so it's worth meeting it here.

Spec (behavior and rules, no tech):

```
# Rate limiting on the public API
Unauthenticated requests to /api/* are capped per client.
- Limit: 100 requests / minute / IP.
- Over the limit returns HTTP 429 with a Retry-After header.
- The health check (/api/health) is exempt.
- Out of scope: per-user limits, billing tiers.
```

Plan (how, with the constraints from your constitution):

```
# Plan
- Token-bucket per IP, stored in Redis (already running; no new deps).
- Middleware in api/middleware/ratelimit.ts, applied before routing.
- Limit + window come from env, not inline — per constitution.
- 429 body uses the standard AppError envelope.
```

Tasks (atomic, independently shippable, each with a check):

```
1. Add ratelimit config to the env schema + .env.example.
   'schema validates, tests pass
2. Implement token-bucket helper in lib/ratelimit.ts.
   'unit tests cover limit / refill / exempt
```

```
3. Wire the middleware into the request pipeline.
   'integration test: 101st request in a minute returns 429
```

That's the whole feature — and notice what you'd actually hand the agent: not "add rate limiting," but task 2, with its acceptance check, on its own. Each task reads like a ticket a competent junior could execute without coming back to ask what you meant.

The other half of Guide is **context engineering**, which is the successor to prompt engineering. Prompt engineering optimized a single human-to-model exchange. Context engineering optimizes the agent's whole working environment: what it knows about your codebase before it writes a line. The mechanism most agents converged on is a persistent context file — `AGENTS.md` or `CLAUDE.md` — that lives in the repo and exists purely for the agent. It holds the things that would clutter a human README: architectural boundaries, naming conventions, the testing approach, the "we always do it this way and never that way" rules. Spec-Kit and similar tools formalize a `constitution.md` for exactly the non-negotiables — testing conventions, dependency policies, design-system standards — captured once and referenced every cycle.

Concretely, an `AGENTS.md` doesn't need to be long to earn its keep — it needs to be the stuff you're tired of repeating:

```
# AGENTS.md

## Architecture
- Monorepo. Apps in apps/, shared code in packages/. Never import across apps directly.
- API is Fastify; web is Astro. No Express, no Next.

## Conventions
- TypeScript strict. No `any` without a `// why:` comment.
- Config comes from the env via config/schema.ts — never read process.env inline.
- Errors use the AppError envelope in lib/errors.ts.

## Testing
- Vitest. Every new module ships with tests. Run `pnpm test` before declaring done.

## Never
- Add a dependency without flagging it first. Prefer the stdlib or what's already here.
- Touch auth/ or billing/ without an explicit go-ahead.
```

Every line in there is a mistake you'd otherwise correct by hand on every cycle. Write it once; the agent reads it every time. (That `config comes from the env` line is going to earn its keep a few chapters from now — watch for it.)

The mental model that makes all of this click: **an agent is, effectively, a very fast junior engineer.** Brilliant recall, genuinely helpful, tireless — and with zero memory of your last conversation, no instinct for your house style, and a tendency to confidently invent an API that should exist but doesn't. You would never hand a new junior a one-line ticket and walk away. You'd give them context, conventions, a clear spec, and a defined first task. Guide is just doing that, deliberately, for a junior who happens to type at 200 words a second.

Get this stage right and the other three get dramatically easier. Get it wrong and no amount of model intelligence saves you.

Generate: Delegating Well

Generate is the 70% — the part the agent does — so this is the shortest of the four stage chapters, which is itself the point. Your job here isn't to generate. It's to delegate well and stay in the director's chair.

The most reliable lesson from teams who've done this at scale: **small, tightly specified tasks beat big vague ones, every time.** The ClickHouse team, working a massive C++ codebase, found the wins came from handing the agent small, over-specified pieces of work — not from describing a sprawling feature and hoping. This matches the spec-driven flow from the last chapter: you already broke the plan into atomic tasks, so feed them one at a time. An agent given a crisp task with a clear acceptance check behaves like a sharp junior. The same agent given "build the billing system" behaves like a junior who's had three coffees and read half a Wikipedia article.

A few habits that make the Generate stage actually work:

- **Use plan mode before act mode.** Most serious agents will let you have them lay out their intended approach before they touch a file. Read it. This is your cheapest possible intervention point — catching a wrong assumption in the plan costs you a sentence; catching it after the agent has written four files costs you a cleanup.
- **Checkpoint before big moves.** Have the agent pause and confirm before anything destructive or far-reaching. "Work autonomously, but stop and check before you change the schema" is a reasonable standing instruction.
- **Keep tasks independently shippable.** If a task can land on its own, a mistake in it is contained. Giant interdependent changes are where agent errors metastasize.
- **Run agents in parallel when the work is genuinely independent.** Git worktrees and multiple agent sessions let you have one agent on a feature while another grinds through a refactor. This is real leverage — but only if you have the verification capacity to review what comes back, which is the constraint, not the generation.

Those last two habits are concrete moves, not vibes. Parallel work is one command — separate trees, separate branches, no agents stepping on each other:

```
# one agent on the rate-limit feature, another on an unrelated refactor
git worktree add ../app-rate-limit  -b feature/rate-limit
git worktree add ../app-auth-cleanup -b refactor/auth-cleanup
```

And the checkpoint is one standing instruction worth pinning in your agent config or `AGENTS.md`, so you don't retype it every session:

```
Work autonomously through the assigned task, but:
- show me your plan before writing any files,
- stop and confirm before changing the schema, touching auth/, or adding a dependency,
- end every task by running the tests and reporting what passed.
```

One short block, and it converts the agent from a thing that surprises you into a thing that checks in at exactly the moments a surprise would be expensive — the plan before it commits to an approach, the pause before anything you can't easily undo.

And the meta-skill: **know what not to delegate.** There's a useful three-level way to think about agent involvement — from copy-pasting snippets out of a chat window, to in-editor assistance, to full agentic delegation — and the right level genuinely differs by task. Gnarly concurrency, subtle performance-critical paths, anything where you can't easily verify correctness, the core architectural decisions that everything else hangs off — these often belong to you, or at minimum belong in a much tighter human-in-the-loop.

The agent is a fast junior; some decisions you don't hand to a junior no matter how fast they are.

Generate is where the speed comes from, and it's genuinely thrilling the first time you watch an agent knock out an afternoon's work in four minutes. But notice that everything that made it go well happened in the *previous* chapter, and everything that determines whether you can trust it happens in the *next* one. The generation itself, increasingly, takes care of itself.

Verify: The New Bottleneck

Verification used to be a late step. You wrote code, a teammate reviewed it, the pipeline checked it, and problems got caught before they grew too big to understand. That whole arrangement was built around a human pace of writing — a few hundred lines, shaped slowly, reviewed by someone who could hold the change in their head.

Agents broke that arrangement. Now a single task can return thousands of lines across multiple files in one reasoning loop. The volume of code needing understanding shot up; the human capacity to understand it did not. So verification stopped being a checkpoint at the end and became the actual bottleneck of the whole process. This is the central structural fact of agentic development: **generation is cheap, verification is expensive, and the expensive thing is now the constraint.**

Which means Verify has to be treated as a first-class function, not a courtesy glance at a pull request. A few principles:

Verify in two places. Once while the agent is still working — catching a wrong turn mid-flight so it can correct — and again after it believes it's done, to test whether the output actually satisfies the requirements. The first loop steers; the second loop judges. Skipping the in-flight loop is how you end up reviewing a beautifully complete implementation of the wrong thing.

Make verification repeatable and explainable, not vibes. Deterministic analysis, security scanning, complexity checks, and tests produce *evidence* — what was checked, what passed, what failed, and why. That evidence is what lets you actually trust the code and, in a team, what lets you be accountable for it. "I read it and it looked fine" is not verification; the CodeRabbit data — 1.7x more issues, security vulnerabilities up to nearly 3x — is a precise measurement of what "looked fine" misses.

Use automated review as a quality gate on every change. This is the single highest-leverage move, and it's why the AI-code-review category exists. An automated reviewer that runs on every PR catches the predictable, measurable failure modes — the insecure object reference, the unhandled error path, the hardcoded credential — at machine speed, so your human attention goes to the things only a human can judge: is this the right design? does it fit the system? Run your SAST and security linters automatically; centralize the things AI gets wrong most (credential handling especially) so the agent physically can't freelance them.

Build an AI-aware review checklist. Because AI fails in *patterned* ways, you can target them. Are the error paths actually covered? Are the concurrency primitives correct? Are configuration values validated? Are passwords handled through the approved helper and not reinvented inline? These questions aim straight at where the data says agents are weakest.

Don't leave that checklist in prose — make it literal and drop it into your PR template, so it runs on every change instead of only the ones you remember to scrutinize:

```
## AI-aware review checklist
- [ ] Error paths covered — every fallible call handles or propagates failure
- [ ] No swallowed exceptions or empty catch blocks
- [ ] Concurrency: shared state guarded, no dropped awaits, no obvious races
- [ ] Config values validated and read from env, never hardcoded
- [ ] Secrets go through the approved helper — none inline, none logged
- [ ] Inputs validated at the boundary (injection, XSS, insecure object refs)
- [ ] New dependencies are real, maintained, and actually necessary
- [ ] Tests assert behavior, not merely that the code runs
```

Every line on it maps to a failure mode the data says agents hit more often than humans do — it's not generic hygiene, it's aimed.

Lean on tests as executable specification. A task that ends with passing tests it didn't get to write itself is

far more trustworthy than one that didn't. Tests are verification that runs forever, for free, on every future change.

And keep the compounding math in view, because it's the quiet killer. A step that's 95% reliable sounds great until you chain ten of them — 0.95 to the tenth is about 60%. At 90% per step it's barely a third. Agentic workflows are exactly these chains of probabilistic steps, which is why an impressive demo and a system that survives production are separated entirely by the verification and checkpoints you build between the steps. The demo skips them. You don't get to.

Solve: Close the Loop

Verification is only useful if it leads to action. Solve is the stage that closes the loop — and it's the one that quietly determines whether your whole system gets better over time or just generates an ever-growing list of problems you'll "get to later."

Here's the failure mode Solve exists to prevent: you stand up great verification, it starts finding real issues, and … the issues pile up. Every detection without a matching remediation is just a more efficient way to grow a backlog. A verification layer with no Solve attached doesn't make your software better; it makes you better-informed about how it's getting worse.

So Solve is a systematic loop, not a someday list:

1. **Remediate** — when verification flags something, fix it now, while the context is hot and the change is small. An agent is often excellent at this part: hand it the specific finding, the relevant code, and the acceptance criterion, and let it propose the fix.
2. **Re-check** — run the fix back through the same verification that caught the problem. A fix you didn't re-verify is a hope, not a fix.
3. **Learn** — and this is the step almost everyone drops — feed what you learned back into the Guide stage. If the agent keeps making the same mistake, that's not the agent's fault anymore; it's a missing line in your `AGENTS.md` or a gap in your spec. The whole cycle is supposed to spiral inward, each pass starting from better context than the last.

That third step is what turns AC/DC from a checklist into a flywheel. The first time the agent hardcodes a config value, you fix it and add "config values come from the environment, never inline" to your constitution. The next cycle, it doesn't make that mistake, because you taught the *system*, not just patched the *instance*. Teams that do this well find their guidance files slowly accreting into a precise, hard-won description of how their codebase actually wants to be worked in — and their agents getting measurably more useful as a direct result.

There's an efficiency dividend here too, and it's a genuinely surprising one. A controlled study from Sonar found that agents working in higher-quality codebases used meaningfully fewer resources — on the order of 7% fewer input tokens, 8% fewer output tokens, 11% less reasoning effort, and 34% fewer re-reads of files — than the same agents in messier code. Clean code isn't just nicer for humans anymore; it's cheaper and more reliable for the machine, because the agent spends less effort figuring out what's going on. Solve, done properly, keeps your codebase in the state where the agent works best. The discipline pays for itself twice.

Generate gets the applause. Solve is where the compounding happens — quietly, cycle after cycle, in the direction of a codebase and a set of guardrails that make every future task a little easier than the last.

Putting It Together: One Feature, Whole Loop

We've walked the four stages one at a time. Now watch them run as a loop on a single real feature — the rate-limiting spec from the Guide chapter — because the balance this whole guide is about only really shows up in motion. I'll tag each move **[70%]** (the agent's) or **[30%]** (yours) so the split is impossible to miss.

Guide — [30%], ten minutes. This part you already saw: the spec, the plan, the three tasks, and an `AGENTS.md` that already says *config comes from the env* and *errors use the AppError envelope.* Nothing's been generated yet — and this is where the feature is already half-won. Mind the ratio: ten minutes here to save the three days of thrash the last few chapters kept promising.

Generate, task 2 — [70%], four minutes. You hand the agent exactly one task: *implement the token-bucket helper in `lib/ratelimit.ts`, with unit tests covering limit / refill / exempt.* Plan mode first:

```
> Plan: a pure token-bucket class, millisecond-precision refill, no I/O in the
  helper (Redis wiring lives in the middleware, task 3). Tests: limit, refill, exempt.
Proceed? (y/n)
```

That plan is right — I/O kept out of the helper, exactly the seam you'd want — so you let it run. **[30%]:** reading that plan and approving it *was* the entire intervention. It writes the class and the tests; `pnpm test` goes green; you read the diff — small, focused, correct. Done.

Generate, task 3 — [70%], then the interesting part. Same move: the middleware that wires the bucket to Redis and applies it before routing. Plan looks fine, agent writes it, tests pass.

Verify — [30%], and this is where the job actually lives. Tests passing is necessary, not sufficient. You run the AI-aware checklist against the middleware, and one line earns its entire existence:

> - Error paths covered — every fallible call handles or propagates failure

The middleware calls Redis on every request — and if Redis is unreachable, the call throws, the middleware throws, and *every API request 500s.* The tests didn't catch it because they assumed Redis was up. A rate limiter that takes down the whole API when its store hiccups is worse than no rate limiter at all. Nothing about the code *looked* wrong; the demo would've been flawless. The checklist caught what "looked fine" missed — the CodeRabbit gap, in miniature.

And the fix is a judgment call that's yours, not the agent's: should the limiter **fail open** (let traffic through when Redis is down) or **fail closed** (block it)? For a public API, fail open with a logged warning — availability beats perfect enforcement here. The agent can't make that call; it doesn't know your risk posture. That's the 30% in its purest form.

Solve — [70%] doing it, [30%] deciding it. You hand the finding back as a crisp task: *if the Redis call fails, log a warning and allow the request; add a test simulating Redis down.* The agent — genuinely good at this once it's told — writes the try/catch and the test. You re-check: green, including the new Redis-down case. Then the step almost everyone skips. You add one line to `AGENTS.md`:

```
## Never
- Middleware backed by an external store (Redis, etc.) must fail open and log —
  never block or error requests because the store is down.
```

Now it isn't a bug you fixed; it's a mistake the system won't make again. The next limiter, the next cache, the next feature flag — the agent reads that line and gets it right the first time. The instance became a rule.

Tally the cycle. The agent generated essentially all of the code — the bucket, the middleware, two rounds of tests, the fix. Call it the 70%, and it was genuinely fast. Your share was smaller in volume and decisive in value: the ten-minute spec that made the tasks cleanly delegable, the plan you approved, the one checklist line that caught a production outage waiting to happen, the fail-open judgment the agent couldn't make, and the rule you wrote so it compounds. That's the 30% — and reading it back, notice that none of it was typing code. It was deciding what to build, catching what was wrong, and owning what shipped.

That's the whole balance, on one small feature. Run that loop a hundred times and the shape holds: the machine gets faster at the 70, your 30 gets sharper and more valuable, and your `AGENTS.md` quietly fills with everything you've taught it. The rest of the guide is about protecting that 30% — owning it, and not letting it rust.

Owning the 30%

Here's a sentence that is going to come up in postmortems for the rest of the decade, and you do not want to be the one saying it: *"The AI wrote that part."*

It's not a defense. It has never been a defense. If the code shipped under your name, in your pull request, into your product, it's yours — every line of it, regardless of which intelligence, carbon or silicon, did the typing. The 30% that's actually your job has ownership sitting right at its center. You are responsible for 100% of what you ship, and the percentage the agent generated has no bearing on that whatsoever.

This sounds obvious stated plainly, and yet the industry spent 2025 learning it the hard way. There's a genuinely strange new problem of AI coding agents reaching for dependencies that don't quite exist, or pulling in packages that nobody actually owns or maintains — the agent confidently `imports` something plausible, and now there's a supply-chain liability in your tree that no human ever consciously chose. Security teams spent the year pointing out that AI ships code faster than traditional security processes were built to inspect, and the CodeRabbit numbers put hard figures on the exposure — security vulnerabilities running up to nearly 3x more common in AI-generated code. None of that is the agent's problem. It's yours, because ownership doesn't transfer to the tool.

What ownership looks like in practice:

- **You read what you ship.** Not skim — read, at least the parts that matter, with enough attention to actually catch the confident mistake. If the volume is too high to read, the volume is too high to ship; slow the generation down to match your verification capacity.
- **You can explain every meaningful decision in the change.** If you can't say why the code does what it does, you haven't reviewed it, you've witnessed it.
- **You own the dependencies.** Every new package the agent introduces is a decision you're making, even if you didn't realize you were making it. Check what got pulled in.
- **You're accountable for the security posture**, the data handling, the edge cases. "The model didn't think of that" describes a gap in your verification, not an excuse for the gap.

The deeper point is about professional identity. The agent changed *how* you produce code; it did not change what it means to be the engineer responsible for it. A structural engineer who used a new modeling tool still signs the drawings. The tool made them faster; it didn't make them less accountable. Same here. The signature is still yours, and the signature is the job.

This is also, frankly, where the human stays valuable in a way no model has touched. Anyone can generate code now. The person who will *stand behind* it — who owns the outcome, carries the accountability, and can be trusted to have actually checked — is exactly as scarce and exactly as valuable as they ever were. Maybe more, because there's so much more unowned code in the world to stand in contrast to.

Don't Outsource Mastery

There's a trap built into all of this, and it's a quiet one because it feels fine right up until it doesn't. If the agent does the hard parts, you stop practicing the hard parts. And the skill you stop practicing is the exact skill you need to supervise the agent. The autopilot flies the plane beautifully — until the day it can't, and the pilot who hasn't hand-flown in two years is suddenly the only thing between everyone and the ground.

That analogy isn't decoration. Aviation documented the over-trust-in-automation problem decades before AI coding existed: humans systematically over-trust automated systems, relax their own vigilance, and miss the warning signs because the automation seemed confident. It's very likely part of the mechanism behind that perception gap from earlier — the work *feels* easier and faster partly because the agent's confidence partially bypasses your own internal quality gate. The errors slip through not because you couldn't catch them, but because you stopped looking as hard.

So the rule is: **don't outsource mastery.** Lean on the agent for leverage, absolutely. But a developer who only knows how to *operate* the agent — who can prompt it but couldn't write or even fully evaluate the code themselves — is capped at whatever the tool happens to do well, and is helpless exactly when it goes wrong. The agent is a fast junior; a junior with no senior to review them is how you get the 1.7x-more-bugs outcome shipped straight to production.

This cuts hardest for two groups. **Juniors**, who are being handed tools that do the very exercises they most need to struggle through to build judgment — and judgment is the thing you can't prompt your way to. And **seniors**, who risk letting hard-won instincts atrophy because the agent makes it so easy not to use them. Some seasoned engineers have been blunt about this: there's a real strain of "I don't want to maintain AI-generated code" among experts, and even Linus Torvalds has pushed back hard on the "99% of code will be AI" triumphalism. They're not being luddites. They're pointing at the maintenance and comprehension burden that arrives later, after the demo glow fades.

Which leads to the most underrated skill of the agentic era: **knowing when not to use the agent.** The honest framing that's emerging is that the mature engineer vibe-codes the prototypes and the throwaway scripts — where speed is everything and the blast radius is zero — and spec-drives everything that actually ships. And for some work, you still just write it yourself: the subtle algorithm where you need to understand every branch, the security-critical path, the performance-sensitive core, the gnarly bug where the act of writing the fix by hand is how you understand the problem. Human-written code still wins anywhere comprehension *is* the deliverable.

Stay sharp enough to catch the agent. Write enough by hand that your judgment doesn't rust. Treat the tool as leverage for skills you have, not a replacement for skills you're letting go. The 70/30 engineer isn't the one who delegates the most — it's the one who still could do the 70% themselves, and chooses where it's worth their time.

The Economics of Agentic Coding

Let's talk about money, because the economics of agentic coding are weirder than they first look and they'll shape how you work whether you think about them or not.

The naive view is that AI makes code cheaper, full stop. Generation does get cheaper. But two things complicate the picture, and a fractional-CTO lens makes them impossible to ignore.

First, **the bill is real and it's growing fast.** "Tokenmaxxing" became a word in 2026 for a reason — teams discovering that running capable agents across real workloads burns tokens, and tokens are dollars. An agent that re-reads your whole codebase on every task, thrashes through a vague spec, or loops on a problem it can't verify is expensive in a way a junior developer's salary makes you feel monthly but agent usage makes you feel by the hour. The cost moved from headcount to compute, and compute scales with how *sloppily* you run the loop.

Which connects to the surprising part: **code quality is now an efficiency variable, not just a maintainability one.** That Sonar study is worth repeating here in economic terms — agents in cleaner codebases used roughly 7–8% fewer tokens and 11% less reasoning effort, and re-read files about a third less often, than the same agents in messy ones. Clean code literally costs less to operate on, because the agent spends fewer tokens figuring out what's going on. Technical debt used to be a slow tax paid by future humans. Now it's also a metered tax paid in real time, every time an agent touches the confusing part of your system. The Guide-and-Solve discipline from earlier isn't just about correctness; it's a direct line item.

Then there's the paradox that catches teams off guard: **cheaper code generation does not reduce total spend on engineering — it increases the work.** This is the old Jevons paradox, the one where making a resource cheaper to use causes you to use much more of it. When code is cheap to produce, you produce vastly more of it — more features, more experiments, more services — and every one of those still needs to be reviewed, verified, secured, owned, and maintained. The generation got cheaper; the *verification and ownership* didn't, and now there's far more of it. The bottleneck didn't disappear. It moved to the 30%, and the 30% doesn't scale just because the 70% got cheap.

For anyone making technical decisions for a business, that reframes the whole proposition. The pitch "AI will cut our engineering costs" is half-true in a dangerous way. Generation costs drop; verification, review, security, and maintenance costs rise to meet the increased volume; and the net only comes out ahead if you've built the discipline to keep the verification side efficient. A team that adopts agents without adopting AC/DC doesn't get cheaper software — it gets more software of lower quality, faster, with a metered bill attached and a maintenance burden compounding behind it. That's not a saving. That's a liability with good demo energy.

The engineers and the organizations that win the economics are the same ones that win the quality: tight guidance, ruthless verification, clean codebases that are cheap to operate on, and clear ownership of what ships. The discipline isn't a tax on the speed. It's the thing that turns the speed into actual value instead of an expensive pile of code nobody can stand behind.

Becoming a 70/30 Engineer

So how do you actually become this engineer? Not in theory — starting Monday, in your real codebase, with your real deadlines.

The shift is captured well in three words that keep coming up as the new operating model: **delegate, review, own.** The old model was *write*. The role moved from creator to curator — less time producing foundational code by hand, more time directing a portfolio of agent work, setting its boundaries, and judging its output. That's not a smaller job. It's a more senior one, applied to more surface area than you could ever have covered by typing. Systems thinking over syntax.

If you want a concrete path, work the AC/DC loop in order of leverage:

- **Start with Guide, because it's the cheapest win.** Before you do anything else, write an `AGENTS.md` (or `CLAUDE.md`) for your project — conventions, architecture, the rules you're tired of repeating. Then, for your next real feature, write a short spec before you prompt. Just those two habits will change your hit rate more than any model upgrade.
- **Tighten your Generate habits.** Break work into small tasks. Use plan mode. Checkpoint before big moves. Stop asking for the whole feature in one breath.
- **Make Verify non-negotiable.** Put an automated review gate on every PR. Have agents write tests. Build the AI-aware checklist — error paths, concurrency, config validation, credentials — and actually run it. This is where you buy back the safety the speed spent.
- **Close with Solve.** When you fix something the agent got wrong, add a line to your guidance so it never makes that mistake again. Watch your context files turn into a flywheel.

And hold onto the two things that keep this honest. **Own everything you ship** — the signature is still yours. And **don't outsource the mastery** that lets you catch the agent when it's confidently wrong; keep your hands in the code enough that your judgment stays sharp. The 70/30 engineer isn't the person who delegates the most. It's the person who could still do the 70% themselves and has gotten so good at the 30% that their judgment, taste, and accountability are the scarce, valuable thing in a world drowning in cheap code.

The honest end state, the one the best practitioners seem to be settling into: vibe-code the prototypes, spec-drive everything that ships, write the truly hard parts by hand, and verify all of it like you mean it. Let the machine have the 70%. Get great at the 30%. That's the whole craft now, and it's a genuinely good time to be the kind of engineer who takes the craft seriously.

If you want to go deeper on the team-level version of this — leading whole engineering orgs where humans and agents work side by side — that's its own discipline, and I get into it in the agentic-teams chapter of the Fractional CTO field guide. And if you want to see the Guide-and-Verify discipline applied to a real stack, the Datastar and Directus field guides are where the theory here meets actual shipping code.

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 work as a fractional CTO through **Data McFly** (https://datamcfly.com) — helping founders and teams set technical direction, build AI-powered workflows, and actually ship the hard parts. The 70/30 split in this guide is the same one I run client work on; here's what it looks like as a method (https://datamcfly.com/the-70-30-method). If you want someone in your corner who's done it before, book a free 30-minute call (https://datamcfly.com/how-it-works) — no pitch, no pressure.

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.