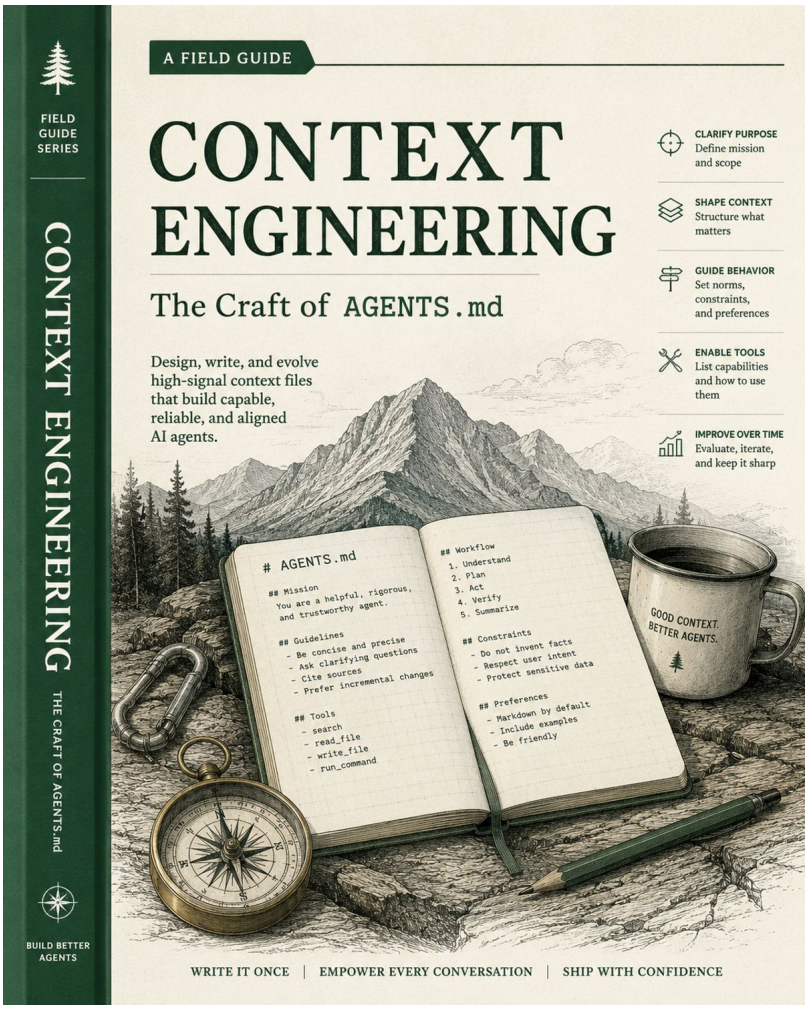

Context Engineering: The Craft of AGENTS.md

A build-along on writing, structuring, and evolving the context files — AGENTS.md, CLAUDE.md, constitution.md — that steer coding agents. From a blank file to a living system that compounds every cycle.

Roger Stringer · rogerstringer.com

June 17, 2026

Contents

Why Context Beats Prompts

Spend a week with a capable coding agent and you notice something that reframes the whole job: when it produces something wrong, it's almost never because the model wasn't smart enough. It's because it didn't *know* something — a convention, a boundary, a decision you made six months ago and never wrote down. It filled the gap with a plausible guess, and the guess was wrong. The intelligence was there. The context wasn't.

That's the shift this guide is about. **Prompt engineering** optimized a single message: phrase the request well, get a better answer. **Context engineering** optimizes the agent's whole working environment — everything it knows about your project *before* it reads your request. The prompt is one turn; the context is the standing knowledge the agent brings to every turn. And as models get better at the one-shot, the leverage moves almost entirely to the context, because a brilliant model with no context still confidently invents an API that should exist but doesn't.

The mental model that makes this click, the one I keep coming back to across these guides: **an agent is a very fast junior engineer with total amnesia.** Genuinely capable, tireless, instant recall of the whole internet — and zero memory of your last conversation, no feel for your house style, no idea which decisions are settled. You would never hand a new hire a one-line ticket and walk away. You'd give them the lay of the land, the conventions, the things you always do and never do. Context engineering is doing that deliberately, in files, for a junior who happens to type at 200 words a second and starts every morning with no memory of yesterday.

Those files — `AGENTS.md`, `CLAUDE.md`, a `constitution.md` — are the subject of this guide. We'll start from a blank repo and build a real one up, layer by layer: architecture, conventions, the non-negotiables, the testing context, scoping for multiple agents, and the learn loop that turns every mistake into a permanent fix. By the end you'll have a context system that makes your agents measurably better at *your* codebase, and gets sharper every week instead of going stale.

This is the piece the 70/30 Engineer (https://rogerstringer.com/guides/the-70-30-engineer) guide kept pointing at — the Guide stage, given its own home. Let's start with what these files actually are.

The Files That Steer Agents

Before we write anything, let's get the landscape straight, because the file names have multiplied and it's easy to get lost.

`AGENTS.md` is the emerging standard — a plain-Markdown file in your repo root that exists purely for AI agents. It holds what an agent needs to work in your codebase: architecture, conventions, how to run things, what's off-limits. Most coding tools have converged on reading it. Think of it as the README you'd write *for a colleague who happens to be a machine* — no marketing, no install-for-humans preamble, just the operating manual.

`CLAUDE.md` is the same idea, specific to Claude Code; other tools have their own (`.cursorrules` and friends). In practice they hold the same kind of content, and the sane move is one source of truth — write `AGENTS.md`, and if a tool insists on its own filename, point it at the same content rather than maintaining two that drift apart.

`constitution.md` is a narrower, sharper thing, popularized by spec-driven tooling: the *non-negotiables*. Not "here's how the code is laid out" but "these rules are never broken" — testing requirements, dependency policy, security standards. Where `AGENTS.md` is orientation, the constitution is law. We give it its own chapter later because it earns the separation.

And the one to keep *out* of the picture: your human `README.md`. It's for people — how to install, what the project is, how to contribute. Cramming agent instructions into it serves neither audience. Keep the human doc human and the agent doc for the agent; they overlap less than you'd think.

Two principles govern all of them. **One source of truth** — a fact about your project lives in exactly one file, or your agent gets contradictory instructions and your humans stop trusting any of them. And **precedence is real** — most tools layer context (a global file, a repo file, a per-directory file), with the most specific winning. We'll use that layering deliberately in the scoping chapter. For now the mental model is simple: `AGENTS.md` orients, `constitution.md` constrains, `README.md` stays human. Next, the blank page.

Your First AGENTS.md

Let's write one. The mistake here is starting with a giant template you found online and filling in fields you don't care about. The right start is the opposite: the *smallest* file that changes the agent's behaviour, grown only when a real miss tells you what's missing.

Throughout this guide I'll build up the context for one running project — a TypeScript SaaS monorepo: a Fastify API, an Astro web app, shared packages. Yours will differ; the shape won't. Here's its entire `AGENTS.md` on day one:

```
# AGENTS.md

## What this is
A TypeScript monorepo. `apps/api` (Fastify), `apps/web` (Astro),
shared code in `packages/`. pnpm workspaces.

## Running it
- Install: `pnpm install`
- Dev: `pnpm dev` (runs api + web)
- Test: `pnpm test`  — run this before declaring anything done.

## House rules
- TypeScript strict. Don't introduce `any` without a `// why:` note.
- Match the style of the file you're editing.
- Don't add a dependency without flagging it first.
```

That's it — fifteen lines, and it already changes outcomes. The agent now knows where things live (so it stops guessing the project layout), how to verify its own work (so it runs the tests instead of declaring victory), and the two or three rules you're most tired of repeating. Every line earns its place by being something you'd otherwise correct by hand on every single task.

Notice what's *not* here. No exhaustive directory listing the agent can read itself. No restating of what TypeScript is. No aspirational rules nobody follows. Context has a cost — it's read on every request, which we'll get into — so the bar for a line is "does this change what the agent does?" If not, cut it.

The test of a good first `AGENTS.md` is embarrassingly practical: hand the agent a small real task and watch where it guesses wrong. Each wrong guess is the next line. We'll formalize that as the learn loop later; for now, just start one, commit it, and use it. The next four chapters are the layers worth adding deliberately — beginning with the one that prevents the most damage: architecture and boundaries.

Architecture & Boundaries

The single highest-value thing you can put in an `AGENTS.md` is the map of your system the agent *cannot infer from reading files* — and the lines it must never cross. An agent can read any one file fast; what it can't do is hold your whole architecture in its head or know which dependencies are forbidden on purpose. That knowledge lives in your head, and an agent without it will cheerfully wire a UI component straight to the database, because nothing in the code said not to.

So write the boundaries down. For our monorepo:

```
## Architecture
- apps/api  — Fastify. The only place that talks to the database.
- apps/web  — Astro. Talks to the API over HTTP; never imports from apps/api.
- packages/core — shared types and pure logic. No I/O, no framework imports.
- packages/db  — the only module that imports the database client.

## Boundaries (do not cross)
- Web never reaches the database directly. It calls the API.
- packages/core stays pure — if it needs the network or disk, it's in the wrong place.
- Cross-app imports are banned. Shared code goes in packages/.
```

That block prevents a whole category of confident mistakes. The agent now knows data flows web → API → db, that `core` is a purity zone, and that the tidy-looking shortcut of importing across apps is exactly what you don't want. None of it is in the code in a way the agent can reliably read — it's architectural *intent*, and intent has to be stated.

Two things make this land. **Name the why, briefly, for the rules an agent is most likely to "helpfully" break.** "core stays pure so it's testable and portable" survives an agent's instinct to be useful far better than a bare prohibition, because the agent understands what it would cost to break it. And **mark the true no-go zones explicitly** — the auth module, the billing logic, the migration files — the places where a wrong edit is expensive and you want the agent to stop and ask rather than proceed.

The deeper point: architecture in your head is invisible to the agent, and invisible architecture gets violated. Writing the boundaries down isn't documentation busywork — it's the difference between an agent that respects your system's shape and one that quietly erodes it, one reasonable-looking import at a time. Next: the smaller rules, the conventions that stop the thousand little corrections.

Conventions That Stick

Architecture stops the big mistakes; conventions stop the thousand small ones — the naming you keep fixing, the pattern you keep re-explaining, the import style you correct on every review. Each is tiny; together they're most of the friction of working with an agent, and every one you write down is a correction you never make again.

The content is whatever you're tired of repeating. For our monorepo:

```
## Conventions
- Files: kebab-case. React/Astro components: PascalCase.
- Errors: throw the `AppError` class from packages/core/errors. Never throw bare strings.
- Config: read from `config/schema.ts`, never `process.env` inline.
- Dates: always UTC, ISO strings at boundaries. No raw Date math.
- API handlers: validate input with the route's zod schema. No unvalidated body access.
- Prefer the standard library and what's already imported over new dependencies.
```

What makes conventions *stick* — actually change behaviour rather than decorate the file — comes down to how you write them:

- **Be specific and imperative.** "Handle errors well" is noise; "throw `AppError`, never bare strings" is a rule the agent can follow exactly. Vague guidance gets vaguely followed.
- **Frame as always/never, not prose.** A scannable list of directives beats a paragraph the agent has to interpret. "Always X. Never Y." leaves no room to improvise.
- **Point at the canonical example.** "Match the pattern in `routes/users.ts`" is worth a paragraph of description — the agent reads the real thing and copies a pattern guaranteed to be current.
- **Only write rules you actually enforce.** A convention your codebase doesn't follow teaches the agent that your rules are suggestions — and it'll treat the important ones the same way. If it's in the file, mean it.

The failure mode to avoid is the aspirational style guide — fifty rules copied from a blog post, half of which your own code violates. The agent reads your code *and* your rules, notices they disagree, and trusts neither. A short list of conventions your codebase genuinely follows beats a long list of wishes every time. Write the ten that are real. Next, the rules that aren't conventions at all — the ones that are law.

The Constitution: Your Non-Negotiables

Some rules aren't conventions — they're law. A convention is "prefer this style"; a constitutional rule is "this is never violated, full stop." Mixing the two dilutes both, so it's worth pulling the non-negotiables into their own file — a `constitution.md` — that you can point every agent (and every spec) at as the standards that don't bend.

The content is the small set of rules where a violation isn't a style nit but a real problem — a security hole, a broken build, a dependency you'll regret:

```
# Constitution — non-negotiable

## Testing
- Every new module ships with tests. No exceptions, no "I'll add them later."
- A change isn't done until `pnpm test` passes. Don't report success otherwise.

## Security
- Secrets come from config/env. Never hardcode, never log them.
- All external input is validated at the boundary before use.
- Never weaken an auth or permission check to make a test pass.

## Dependencies
- No new runtime dependency without explicit human sign-off.
- Prefer the platform and existing deps. Every package is a liability.

## Data
- Migrations are forward-only and reviewed. Never edit a shipped migration.
```

The reason a separate file works is *force*. When everything lives in one big `AGENTS.md`, the rule that secrets are never logged sits at the same volume as the rule about kebab-case filenames, and the agent weights them similarly. A constitution says, by its very existence, "these are different — these don't flex." Spec-driven tools formalize exactly this, referencing the constitution on every cycle so the non-negotiables are in front of the agent for every plan, not buried in orientation it skimmed once.

Keep it short and keep it true. A constitution with forty rules is a wish list; a constitution with eight is a spine. The test for whether something belongs here rather than in conventions: *would you block a PR over it?* If violating the rule is a hard stop — unsafe, untested, an unauthorized dependency — it's constitutional. If it's "I'd prefer you didn't," it's a convention. Sort them honestly and both files get sharper. Next, the half of verification the agent needs from you: how you test.

Telling the Agent How You Test

An agent that can't verify its own work hands you confident guesses; an agent that can ships work that's already checked. The difference is almost entirely whether you told it *how* you test — and most `AGENTS.md` files leave this out, which is why so much agent output looks done and isn't.

Give the agent the verification context it needs to close its own loop:

```
## Testing & verification
- Run all tests: `pnpm test`. One package: `pnpm --filter api test`.
- Tests live next to source as `*.test.ts`. New code ships with them.
- Lint and types: `pnpm check` (tsc + eslint). Must be clean before done.
- E2E: `pnpm e2e` — slow; run before a PR, not on every change.
- "Done" means: tests pass, `pnpm check` is clean, and you've exercised the
  affected path at least once. Report what you ran.
```

Three things punch above their weight here. **Tell it what "done" means, explicitly.** Left undefined, "done" means "the code looks finished to the model" — exactly the over-confident completion the 70/30 Engineer (https://rogerstringer.com/guides/the-70-30-engineer) guide warns about. Defined as "tests green, types clean, path exercised, and tell me what you ran," done becomes a checkable bar the agent has to actually clear.

Make tests the spec where you can. An agent told "make this test pass" has an unambiguous target and verifies itself for free; an agent told "implement this feature" has to guess what success looks like. Pointing at the test — or asking it to write the test first — turns verification from your job into its job.

Tell it what's slow. Without guidance, an agent either runs the ten-minute E2E suite on every tiny change (wasting your time and tokens) or never runs it at all (and ships untested). "Unit tests on every change, E2E before a PR" gives it the cadence you'd use yourself.

The payoff connects straight to the Eval-Driven Development (https://rogerstringer.com/guides/eval-driven-development) mindset: an agent that knows how to prove its work produces output you can trust without re-deriving correctness yourself. Verification context is how you move the checking left — from you, after, to the agent, during. Next, what to do when one set of context isn't enough.

Scoping Context Per Agent

One `AGENTS.md` at the root is the right start, and for many projects the right end too. But two situations call for *scoping* context more finely: a big codebase where different areas have different rules, and a multi-agent setup where different agents have different jobs. Both lean on the same mechanism — layered context, most-specific-wins.

Per-directory context. Most tools read an `AGENTS.md` in a subdirectory *in addition to* the root one, with the local file overriding on conflicts. So a corner of the codebase with its own rules gets its own file:

```
# apps/api/AGENTS.md  (adds to the root file)

## API-specific
- Every route validates its body with zod before touching it.
- Handlers stay thin: validate, call a service in packages/core, return.
  No business logic in the route file.
- All responses go through the envelope in lib/respond.ts.
```

The root file carries what's true everywhere; the local file carries what's true *here*. The agent working in `apps/api` gets both, and the specific rules land without you bloating the root file with details that only matter in one place. This is how you scale context to a large codebase — not one enormous file, but a tree of small ones that mirrors the tree of the code.

Per-agent context. When you're running a fleet — the Running the Fleet (https://rogerstringer.com/guides/running-the-fleet) setup — different agents have different roles, and a role is just scoped context. The reviewer agent's instructions emphasize what to check; the test-writer's emphasize coverage and edge cases; the docs agent's emphasize clarity over cleverness. Each is an `AGENTS.md`-shaped file tuned to one job, layered on the shared project context underneath. The shared file says what's true about the *project*; the role file says what's true about this agent's *work*.

The principle under both is the one from chapter two: most-specific-wins, one source of truth per fact. Put a rule at the broadest scope where it's true — project-wide rules in the root, area rules in the directory, role rules in the agent — and never the same rule in two places, or they'll drift. Scope by where the rule is actually true and the layering stays clean instead of becoming four files that quietly contradict each other. Which is a good moment to talk about the cost of all this context — because more isn't free.

Keeping It Lean

Every line in your `AGENTS.md` is read on every request. That's the deal: context is standing knowledge, which means it's standing *cost* — tokens spent, and attention diluted, on every single task whether or not the task needs that line. So the discipline isn't "write everything down," it's "write down what changes behaviour, and ruthlessly cut the rest."

Two costs, both real. The **token cost** is direct — a bloated context file is paid for on every call, and across a busy day of agent work that adds up the way the 70/30 Engineer (https://rogerstringer.com/guides/the-70-30-engineer) guide's tokenmaxxing economics describe. The subtler and more important one is the **attention cost**: a model given fifty rules weights the critical one less than a model given eight. Burying "never log secrets" in a wall of style preferences makes it *less* likely to be followed, not more. Padding doesn't just cost money; it drowns the signal that matters.

So what earns a line:

- **It changes what the agent does.** If removing the line wouldn't change a single output, it was decoration. Cut it.
- **The agent can't derive it.** Don't restate what the code says or what the model already knows. "We use TypeScript" is visible in every file; "core stays pure" is not. Spend words on the invisible.
- **It's true and current.** A stale rule is worse than no rule — it actively misleads. Lines rot; prune them.

And what to cut: exhaustive directory listings the agent can read itself, restated language and framework basics, aspirational rules your code doesn't follow, anything you wrote "to be thorough." Thoroughness is the enemy here. The best `AGENTS.md` files are surprisingly short — dense with the non-obvious, empty of the derivable.

A useful habit: when you add a line, ask what you'd cut to keep the file the same length. Treating context as a fixed budget rather than an append-only log is what keeps it sharp over time, because the natural drift of these files is always toward bloat — every incident adds a rule, nothing ever gets removed, and a year later it's a thousand lines the agent half-ignores. Lean is a practice, not a state. Speaking of adding rules from incidents — that's the loop that makes context compound, done right.

The Learn Loop

Here's the practice that turns context engineering from a one-time setup into a compounding asset: every time the agent makes a mistake, you fix it *and* you add the line that prevents it forever. The fix handles the instance; the line handles the class. Do this consistently and your context file becomes a precise, hard-won description of how your codebase actually wants to be worked in — written by the failures themselves.

The loop is small:

1. The agent does something wrong — hardcodes a config value, say, instead of reading from `config/schema.ts`.
2. You correct *this* output (the normal fix).
3. Then the step almost everyone skips: you add a line to `AGENTS.md` — "Config comes from config/schema.ts, never inline" — so it never happens again.
4. Next task, the agent reads that line and gets it right the first time.

Step 3 is the whole game, and it's the same instinct as the 70/30 Engineer (https://rogerstringer.com/guides/the-70-30-engineer) guide's Solve loop and the Eval-Driven Development (https://rogerstringer.com/guides/eval-driven-development) flywheel: don't patch the symptom, close the gap. Fixing the hardcoded value and moving on teaches the system nothing — the same mistake recurs on the next task and the one after. Promoting it to a rule is what makes the fix permanent. You're not correcting an agent; you're teaching a system, and the system remembers.

What this feels like over a few weeks is genuinely good: the agent gets measurably more useful at *your* codebase, not because the model changed but because you taught it your hard-won specifics. The corrections you used to repeat daily, you make once, as a line, and never again. The file accretes exactly the knowledge that matters — because every entry was earned by a real mistake, not imagined in advance.

Two cautions, so the loop doesn't fight the last chapter. **Generalize before you write.** The line should prevent the *class*, not encode the one instance — "config from schema, never inline," not "don't hardcode the Redis URL on line 40." And **prune as you add** — a learn loop with no pruning is just bloat with good intentions; when you add a rule, check whether it supersedes two older ones. Add earned, cut stale, and the file stays both complete and lean. Next, the mistakes that make these files actively harmful — so you can avoid them.

Anti-Patterns

A bad context file is worse than none — no context makes the agent guess, but bad context makes it confidently wrong in ways that look authoritative. Here are the failure modes worth knowing by name, because once you can see them you'll catch them in your own files.

Stale context. The rule that was true six months ago and quietly isn't anymore — the renamed module, the abandoned pattern, the "we always use X" you stopped using. The agent follows it faithfully, because it has no way to know it's out of date, and produces work that's wrong against a rule you forgot you wrote. Stale context is the most common failure and the most insidious, because it fails *silently*. Treat the file as code: review it, and delete on sight.

Internal contradictions. "Keep handlers thin" in one section; a fat handler held up as the example in another. The agent can't reconcile them, so it picks one arbitrarily — and you get inconsistent output that's technically following *some* rule. Contradictions usually creep in through the learn loop without pruning: you add a new rule that conflicts with an old one and never reconcile them.

Over-stuffing. The thousand-line file that documents everything and therefore emphasizes nothing — the attention-dilution problem from the lean chapter, in its terminal form. When everything is written down, the agent can't tell the load-bearing rule from the nicety, and the load-bearing rule loses.

The aspirational style guide. Rules copied from somewhere that your actual code doesn't follow. The agent reads code and rules, sees them disagree, and learns your rules are optional — then applies that lesson to the rules you *did* mean.

README / AGENTS.md confusion. Human onboarding prose stuffed into the agent file, or agent rules buried in the human README. Each dilutes the other; the agent wades through install instructions it doesn't need, the human hits "never throw bare strings" while looking for how to clone the repo. Keep the audiences separate.

Vague rules. "Write clean code," "handle errors properly," "follow best practices" — instructions that feel like guidance and provide none, because the agent already thinks it's doing all of them. A rule the agent can't fail to follow isn't a rule; it's filler.

The through-line: these files are infrastructure, and they rot like infrastructure. Most of these anti-patterns are just what happens when you treat `AGENTS.md` as write-once instead of maintained. Which is the last idea — context as a living system.

A Living System

Your context files live in the repo, version with the code, and change as the code changes — and treating them that way, as a living part of the project rather than a one-time artifact, is what keeps them an asset instead of a liability.

Version them with the code, because they're coupled to it. `AGENTS.md` is in git for a reason: a rule about your architecture is only true for a given state of that architecture. When a PR changes how something works, updating the context belongs in the *same* PR — the way you'd update a test. A boundary you enforce in code and a boundary you describe in `AGENTS.md` have to move together, or the description becomes the stale context from the last chapter. The reviewer's question "did the docs change with the code?" now includes the agent docs.

Share it, because it's team knowledge. When `AGENTS.md` is in the repo, every developer's agents work from the same conventions — the new hire's agent follows the same boundaries as the staff engineer's, because the knowledge is in the project, not in someone's head or personal config. That's a real and underrated benefit: context engineering is how a team's hard-won "how we do things here" becomes something machines apply consistently, instead of tribal knowledge each person's tools learn separately and incompletely.

Evolve it deliberately. The file should track the project's actual maturity. Early on it's thin — a few boundaries, a few conventions — because the project doesn't have many settled decisions yet. As patterns harden, the context hardens with them. As the architecture shifts, old rules retire and new ones land. A context file frozen at the shape the project had a year ago is describing a codebase that no longer exists.

The habit that holds it together is light but non-negotiable: when you change how the project works, change the file in the same breath, and every so often read the whole thing top to bottom and prune what's no longer true. Maintained that way, a context file compounds — it gets more valuable as the project grows, because it accumulates exactly the decisions that matter. Left to rot, it does the opposite, decaying into a trap of stale rules the agent follows off a cliff. Same file, two fates, decided entirely by whether you treat it as alive. Let's put it all together.

Building Your Context

Here's how to start, this week, on a real project — in order of leverage, the cheapest high-impact moves first.

- **Write fifteen lines today.** Not a template — the smallest real `AGENTS.md`: what the project is, how to run and test it, and the three rules you're most tired of repeating. Commit it. You'll feel the difference on the next task.
- **Add the architecture and boundaries.** The map the agent can't infer — what lives where, what talks to what, and the lines it must never cross. This prevents the most damage per line of any context you'll write.
- **Pull the non-negotiables into a constitution.** The small set of rules you'd block a PR over — testing, secrets, dependencies. A separate file, so they carry the weight of law instead of blurring into style.
- **Give it your testing context.** How to run tests, what "done" means, what's slow. This is what lets the agent verify its own work instead of handing you confident guesses.
- **Run the learn loop.** Every agent mistake becomes a line that prevents the class. This is the habit that makes context compound — do this one thing consistently and the file teaches itself.
- **Keep it lean and alive.** Prune as you add; version it with the code; update it in the same PR that changes how things work. A context file is infrastructure — maintain it like infrastructure.

The mindset under all of it: you are onboarding a brilliant, amnesiac junior who starts fresh every morning and works at machine speed. Everything you'd tell a new hire — the lay of the land, the conventions, the things you never do — you write down once, and every agent on the project knows it forever. That writing is the highest-leverage work in agentic development, because it's the difference between an agent that guesses at your codebase and one that knows it.

This is the Guide stage of the 70/30 Engineer (https://rogerstringer.com/guides/the-70-30-engineer) loop, given the depth it deserves; it pairs with Eval-Driven Development (https://rogerstringer.com/guides/eval-driven-development) for verifying what the agent produces, and it's the foundation under every agent in the Running the Fleet (https://rogerstringer.com/guides/running-the-fleet) and Building Your Agentic OS (https://rogerstringer.com/guides/building-your-agentic-os) guides. Start with fifteen lines. Let the failures write the rest.

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.