Build vs. Buy in the AI Era

A decision framework for founders drowning in "should we build this with AI or pay for it" — when to build, when to buy, when to wait, and how the economics changed once agents could write the code.

Roger Stringer · rogerstringer.com

June 19, 2026

Contents

The Question Every Founder Is Now Asking

"Should we build this or buy it?" is the oldest question in technology decisions, and for decades it had reasonably settled answers: build what's core to your business, buy the commodity stuff, and weigh the cost of engineering against the cost of a subscription. Then AI coding agents made building *feel* nearly free — and the whole calculation got scrambled.

Here's the new shape of the question, and why it's genuinely harder. A founder looks at a SaaS tool costing $2,000 a month and thinks: *I could just have an AI agent build that in a weekend.* Sometimes that's true. Often it's a trap. The old build-vs-buy math assumed building was expensive and slow, which kept the boundary clear. Now that generating the first version is cheap and fast, the boundary has moved — but not to where most people think, because the *generating* was never the expensive part of building.

This guide is a decision framework for founders, technical leaders, and anyone making the call in an environment where "just build it with AI" is always on the table and frequently wrong. We'll work through:

- **What actually changed** — and, crucially, what didn't (the part that matters most).
- **The true cost of build** and **the true cost of buy**, both fuller than the sticker price.
- **The third option** people forget: wait.
- **A decision framework** — the questions that sort build from buy from wait.
- **Core vs. context, the AI-build trap, reversibility**, and worked examples that run real decisions through the framework.

The punchline I'll keep returning to, because it's the key the whole AI-era confusion turns on: **AI made *generation* cheap, but generation was never the expensive part of building — owning, maintaining, securing, and verifying software was, and AI didn't make those cheaper at all.** The 70/30 Engineer (https://rogerstringer.com/guides/the-70-30-engineer) guide makes this case for code; this guide applies it to the build-vs-buy decision. Get that one idea right and most of these decisions get clearer. Let's start with what changed.

What Actually Changed

To decide well in the AI era, you have to be precise about what AI actually changed — because the popular version ("AI makes building cheap, so build everything") is wrong in a way that leads to expensive mistakes. Here's the accurate version.

What got cheaper: generation. Producing the first working version of a piece of software — the code, the initial feature — genuinely got faster and cheaper with capable AI agents. A thing that took a developer two weeks might now take two days. That's real, and it's the part everyone sees in the demo. If building were *only* generation, "build everything" would follow.

What didn't get cheaper: everything else about owning software. And this is the whole point. The 70/30 Engineer (https://rogerstringer.com/guides/the-70-30-engineer) guide's central insight applies directly: generation is the 70% that got cheap; the 30% that's the actual job — verifying it's correct, securing it, maintaining it as the world changes, fixing it at 2am, owning the consequences when it breaks — did *not* get cheaper. If anything that work grew, because now there's more software flowing into it. When you "build" a tool, you're not signing up for the weekend of generation; you're signing up for *years* of the 30%. AI discounted the cheap part and left the expensive part exactly as expensive.

The Jevons trap, at the decision level. The 70/30 guide describes how cheaper generation leads to *more* code, not less work — the Jevons paradox. The same logic warps build-vs-buy: because building feels cheap, founders build far more than they used to, accumulating a sprawl of in-house tools that each need owning, securing, and maintaining forever. The cheapness of generation seduces you into a maintenance burden whose true cost is invisible at decision time and crushing later.

So what actually changed for the decision? The boundary moved, modestly: some things that were clearly "buy" because building was too expensive are now genuinely worth building, because the *generation* cost dropped enough to tip them. But far fewer than the hype suggests — because the decision was never really about generation cost. It was always about *total cost of ownership* and *strategic fit*, and AI barely touched the first and didn't touch the second.

The reframe that makes the rest of this guide work: **stop asking "can AI build this?" (the answer is usually yes, and it's the wrong question) and start asking "do I want to *own* this for the next five years?"** That question AI didn't change at all — and it's the one that actually decides correctly. Let's price both sides of it honestly, starting with the true cost of build.

The True Cost of Build

When you decide to build, what are you actually committing to? Almost never just the building — and the gap between the cost people imagine (the generation) and the cost they incur (everything after) is where build-vs-buy decisions go wrong. Let's price it honestly.

The first version is the down payment, not the price. Generating the initial tool, AI-assisted or not, is the cheap, visible, exciting part. But software isn't a thing you make once; it's a thing you *keep*. The real costs arrive after the demo:

- **Maintenance, forever.** Dependencies need updating, security patches applying, and the thing breaks when an API it relies on changes or a platform shifts under it. A tool you built is a tool you maintain for as long as you use it — and "forever" is a long multiplier on even a small monthly effort.
- **The 30% that doesn't scale.** Per the 70/30 Engineer (https://rogerstringer.com/guides/the-70-30-engineer) guide: every feature you build needs verifying, securing, and owning, and that work didn't get cheaper. The more you build, the more of this you carry. A pile of AI-generated in-house tools is a pile of 30%-debt accruing quietly.
- **The edge cases and the long tail.** The demo handles the happy path. Real use surfaces the 100 edge cases a mature SaaS already solved over years — the timezone bug, the weird input, the scale problem, the integration nobody anticipated. Building the 80% is a weekend; the last 20% is most of the actual work, and it's invisible when you decide.
- **Opportunity cost — the big one.** Every hour spent building and maintaining a tool is an hour not spent on the thing only *you* can build: your actual product, your differentiation. Usually the largest and least-counted cost. A founder who builds their own analytics dashboard saved a subscription and spent the scarcest resource they have — their team's attention — on a solved problem.
- **The security and reliability burden.** When you build it, you own its security holes, its outages, its data handling. A bought tool's vendor carries that (and is often far better at it); a built tool makes it yours, including the liabilities the Technical Due Diligence (https://rogerstringer.com/guides/technical-due-diligence) and Auth (https://rogerstringer.com/guides/auth-done-right) guides catalogue.

When building is genuinely worth all that: when the thing is *core* to your business (next chapter), when no good off-the-shelf option exists, when owning it is a real strategic advantage, or when the long-run economics genuinely favor it at your scale. Building is the right call often — just far less often than "AI makes it cheap" implies, and never for the reason that generation got cheap.

The honest mental model: **building isn't buying a tool, it's adopting a dependent for life.** Price the lifetime, not the birth. Now the other side — because buying has hidden costs too.

The True Cost of Buy

Buying has its own costs the sticker price hides, and a fair decision prices them as honestly as it prices building — otherwise you'll over-correct into a sprawl of subscriptions that's its own kind of expensive. Here's what "buy" actually costs.

- **The subscription, compounding.** The obvious one, but reckon with it over time and at scale: $500/month is $6,000/year, and per-seat pricing means it grows with your headcount or usage — sometimes faster than your revenue. A tool that's cheap at 5 users can be brutal at 500. Model the cost at the scale you're heading toward, not today's.
- **Lock-in and switching costs.** Once your data and workflows live in a vendor's tool, leaving is expensive — migration, retraining, rebuilding integrations. That lock-in is also leverage *they* have over you: price increases, feature removals, an acquisition that changes everything. You're renting on their terms, and the terms can change. (The reversibility chapter is about exactly this.)
- **Fit — the 80% problem in reverse.** A bought tool does what its vendor decided it should do, which is rarely *exactly* what you need. You bend your process to fit the tool, or live without the 20% you actually wanted. For a commodity need that's fine; for something close to your core, the misfit is a real, recurring tax.
- **The integration tax.** Tools don't exist alone; they have to talk to your other tools and your data. Wiring a bought tool into your stack — and keeping those integrations working as both ends change — is ongoing work the subscription price doesn't include. "Buy" is rarely as turnkey as it looks.
- **Dependency on someone else's roadmap and survival.** You're betting the vendor keeps the product good, keeps it alive, and doesn't get acquired and gutted. Build a critical workflow on a tool from a shaky startup and you've taken on a real risk — the vendor-concentration risk the Technical Due Diligence (https://rogerstringer.com/guides/technical-due-diligence) guide flags.

When buying is clearly right: for commodity needs (the Auth (https://rogerstringer.com/guides/auth-done-right) guide's "reach for a provider" line, email, payments, analytics), when the vendor does it far better than you ever would, when speed-to-value matters more than control, and when it's *not* core to your differentiation. For most of what most companies need, buying is the correct default — you don't build your own Stripe.

The honest summary of both sides: **build trades money-later for control-and-burden; buy trades control-and-money-forever for speed-and-someone-else's-burden.** Neither is free, and the right choice depends on the specific thing. So the rest of the guide is about deciding well — starting with an option people forget exists.

The Third Option: Wait

Build-vs-buy is framed as a binary, which is the first mistake — because there's a third option that's frequently the best one and almost never on the whiteboard: **wait.** Don't build it, don't buy it, not yet.

Why waiting is so often right. Most "we need a tool for X" urges are weaker than they feel in the moment. Before committing money (buy) or your scarcest attention (build), ask whether you need to solve this *now* at all:

- **The need might not be real yet.** Founders constantly tool up for problems they anticipate rather than have. Building or buying analytics before you have enough users to analyze, a fancy CRM before you have enough customers to manage, a workflow tool for a process you do twice a month — these are solutions shopping for a problem. Waiting until the pain is real means you'll know what you actually need, instead of guessing.
- **The manual version is often enough for surprisingly long.** A spreadsheet, a shared doc, a manual process, a cheap no-code stopgap — the unglamorous interim solution frequently carries you further than you'd think, at near-zero cost and zero lock-in. "Do it by hand until it hurts" is genuinely good advice; the hurt tells you when (and exactly what) to build or buy.
- **The market is moving fast.** In a fast-moving category — and AI tooling is the fastest — waiting six months often means a dramatically better option exists, cheaper or more capable, that didn't when you first felt the itch. Committing early in a churning market can lock you into the worst version of a tool that's about to get much better. Patience is a strategy.
- **Waiting preserves optionality and information.** Every month you wait, you learn more about your real need, and the options improve. Deciding now spends information you don't have yet; waiting buys it cheaply.

When *not* to wait: when the pain is real and present and costing you now, when the lack of the tool is actively blocking the business, or when a clearly-right cheap option exists and dithering is just procrastination. Waiting is a deliberate choice, not an excuse to avoid deciding — the failure mode is *drifting* (never deciding) versus *waiting* (consciously deferring with a trigger for when to revisit).

The discipline this adds to the framework: **before build-or-buy, always ask "or neither, yet?"** The cheapest, most reversible, most information-preserving option is frequently to not solve the problem with software at all, for now. The First 90 Days (https://rogerstringer.com/guides/the-first-90-days) guide's "understand before you act" applies here too — the urge to tool up is often the urge to feel productive before you've understood the real need. With three options on the table, let's build the framework that chooses between them.

A Decision Framework

Here's the framework — a set of questions that sort build from buy from wait. Run a decision through these, in order, and the answer usually falls out. None is a single deciding factor; together they triangulate the right call.

1. Is it core to your differentiation, or context? The most important question (its own chapter next). If this thing is part of *why customers choose you* — your actual edge — lean build. If it's necessary plumbing that every company has and no customer cares about, lean buy. Build your moat; rent the commodity.

2. Do you actually need it now? The wait question. If the pain isn't real and present, the answer might be "neither, yet." Don't build or buy for anticipated needs; solve real ones.

3. Does a good off-the-shelf option exist? If a mature, well-fit tool already solves this well, the bar for building is high — you'd be rebuilding solved work. If the options are all poor-fit or absent, building moves up the list. Survey the market honestly before assuming you must build.

4. What's the true total cost each way — over years, at your future scale? Not the sticker price. Build = generation + maintenance-forever + the 30% + opportunity cost. Buy = subscription-at-scale + lock-in + integration + fit-tax. Model both at where you're *heading*, not where you are.

5. Can you actually own it if you build it? Be honest about capacity. Building means committing to maintain, secure, and support it indefinitely. If your team can't credibly own another system — the Auth (https://rogerstringer.com/guides/auth-done-right) guide's hard question about whether you can own security long-term — then "build" is a fantasy that becomes orphaned, rotting code. AI generating it doesn't change whether you can *own* it.

6. How reversible is the decision? Favor the choice you can undo cheaply (its own chapter). A reversible decision can be made fast and corrected; an irreversible one deserves much more deliberation. Prefer options that don't trap you.

How to use it: these aren't a scorecard to total up — they're a sequence that usually surfaces a clear answer. Core + a poor market + you can own it ⁉build. Context + a good option exists ⁉buy. Need isn't real yet ⁉wait. The questions interact: something core that you *can't* currently own might mean buy-for-now-and-revisit; something context-y with no good option might mean a cheap stopgap while you wait for the market.

The meta-point, and the through-line of the whole guide: **the framework deliberately doesn't include "can AI build it?"** — because that question, the one the AI era foregrounds, is the least useful input to a good decision. The useful questions are about ownership, fit, strategic value, timing, and reversibility — exactly the ones AI didn't change. The single most important of them gets its own chapter: core vs. context.

Core vs. Context

If you take one principle from this guide, take this one, because it decides more build-vs-buy calls correctly than any other: **build what's core to your business; buy what's context.** It's an old idea (Geoffrey Moore's core-vs-context), and the AI era makes it *more* important, not less.

Core is the thing customers choose you for — your differentiation, your edge, the work that *is* the business. For a fintech, the risk engine. For a design tool, the canvas. For your product, whatever makes it yours rather than a clone. Core is where building is worth every cost in the previous chapters, because owning and controlling it *is* the point — it's your moat, and you don't rent your moat. You want full control, the ability to evolve it freely, and no dependency on a vendor's roadmap for the thing that defines you.

Context is everything else — the necessary work that keeps the business running but isn't *why* anyone picks you. Auth, payments, email, analytics, the CRM, the helpdesk, the deploy pipeline. Real, needed, but commodity: a hundred companies have the same need, mature tools solve it well, and building your own gains you nothing a customer will ever notice. Context is where buying wins, because the only thing building it buys you is a maintenance burden on a solved problem and attention stolen from your core.

Why AI sharpens this rather than blurring it. The seductive AI-era error is "building is cheap now, so build the context stuff too" — build your own auth, your own analytics, because an agent can generate them in a weekend. But the core-vs-context logic was *never* about generation cost; it was about where your finite attention and ownership capacity should go. AI made generating context-software cheaper, but it didn't make *owning* it cheaper or make it any more strategically valuable. Building your own auth is exactly as much of a distraction from your core as it was before — you've just made the distraction faster to start and no less costly to keep. The cheapness is a trap precisely because it tempts you to spend ownership capacity on things that will never differentiate you.

The sharp test: *Will a customer ever choose us, or choose us over a competitor, because we built this ourselves?* If yes, it's plausibly core — consider building. If no — if it's invisible plumbing they'd never know or care was custom — it's context, and you should almost certainly buy it and spend the saved attention on the thing they *will* choose you for.

Most of what a company needs is context. That's not a knock; it's the reason buying is the right default and building is the deliberate exception reserved for your edge. Keep your scarce build capacity for the moat. Which brings us to the specific AI-era trap that violates all of this — "we'll just have AI build it."

The AI-Build Trap

There's a specific, seductive line of reasoning the AI era produced, and it's wrong often enough — and expensively enough — to deserve its own chapter: **"we don't need to buy that, we'll just have AI build it."** It sounds like leverage. It's frequently a trap, and naming exactly how it goes wrong helps you avoid it.

The trap, step by step. A founder sees a tool, balks at the price, and reasons: AI can generate something like this fast and nearly free, so why pay? They have an agent build a version over a weekend. It demos well. They cancel the subscription, feeling clever. Then:

- **The 80/20 reveals itself.** The weekend build did the obvious 80%. The missing 20% — the edge cases, the scale, the integrations, the polish the mature tool accumulated over *years* of real customers hitting real problems — turns out to be most of the actual value, and now it's their problem to build, one painful gap at a time.
- **The maintenance burden lands.** The thing they generated is now a system they own forever (the build-cost chapter), accruing the 30% of verification, security, and upkeep that AI didn't make cheaper. The "free" tool has a permanent staffing cost they didn't price.
- **It pulls focus from the core.** Every hour now spent shoring up their home-grown version of a solved, commodity problem is an hour stolen from the thing only they can build (core vs. context). They've redirected their scarcest resource — attention — toward something a customer will never value.
- **The ownership question was never asked.** "Can AI build it?" got answered (yes); "can we *own* it for five years?" never got asked, and that's the one that mattered.

When "have AI build it" genuinely is the right call. It's not always a trap — the skill is telling the difference. Building with AI is genuinely right when the thing is **core** (build your moat, and AI helping you build it faster is pure upside), when it's **small and bounded** (a simple internal script, a glue utility — little to maintain, no commodity tool fits exactly), or when **no good option exists** and you'd have to build regardless. The trap is specifically using "AI can build it" to justify building **context** — commodity software a mature vendor already does better — because the generation got cheap. That's the case where cheap generation seduces you into an expensive ownership mistake.

The discipline that defuses the trap: when you catch yourself thinking "we'll just have AI build it instead of buying," stop and run the actual framework — *is this core or context? do we need it now? can we own it for years? what's the true lifetime cost?* The AI-build impulse skips straight past every question that matters to the one that doesn't ("is it possible?"). Make it answer the real questions, and the genuine build cases survive while the traps reveal themselves. Most of the time, for context software, the boring answer — just buy the tool — is the right one, and the weekend is better spent on your product. Speaking of correcting course, the next idea is what makes a wrong call survivable: reversibility.

Reversibility & Switching Costs

A principle that quietly improves build-vs-buy decisions more than any amount of analysis: **favor the choice you can undo cheaply.** Most of these decisions are made under real uncertainty — you don't fully know your future scale, needs, or the market — and in uncertainty, reversibility is worth a lot, because it lets you decide fast and correct cheaply instead of agonizing toward a guess.

Reversible vs. irreversible decisions deserve different rigor. The framing (Bezos's one-way vs. two-way doors) is exactly right here. A *reversible* decision — one you can back out of cheaply — should be made quickly; if it's wrong, you fix it. An *irreversible* one — expensive or impossible to undo — deserves much more deliberation, because you'll live with the mistake. Most build-vs-buy calls sit somewhere on this spectrum, and where they sit should change how hard you think and which way you lean.

What makes each side reversible or not:

- **Buying is often more reversible than it looks — if you choose for it.** A tool you can leave (your data is exportable, the integration is shallow, the workflow isn't deeply wired in) is a cheap, two-way-door decision: try it, switch if it disappoints. But buying *can* be deeply irreversible — the vendor whose proprietary format traps your data, whose tool your whole operation grows around. The reversibility isn't fixed; it depends on how you adopt. Prefer tools with clean exits, keep your data portable, and don't wire a vendor so deep you can't leave.
- **Building is reversible at the start, then increasingly not.** Generating a first version is cheap to abandon. But the longer a built tool lives — the more it accretes, the more your processes depend on it, the more institutional knowledge sinks into it — the more irreversible it becomes. A home-grown system five years deep is a one-way door you walked through without noticing, the key-person-risk trap the Technical Due Diligence (https://rogerstringer.com/guides/technical-due-diligence) guide catalogues, now self-inflicted.

How this sharpens the decision:

- **When uncertain, prefer the more reversible option** — usually a cheap, low-lock-in buy, or a manual stopgap (the wait chapter), precisely because it preserves your ability to change your mind once you've learned more.
- **For irreversible decisions, slow down and deliberate** — a deep build of something core, or adopting a vendor you'll be wedded to, is a one-way door worth real thought.
- **Engineer reversibility into whichever you choose.** Keep data portable, integrations shallow, dependencies loose, so even a "committed" decision retains an exit. The most expensive decisions are the ones you can't walk back; design to keep walking-back possible.

The synthesis: **in an uncertain, fast-moving environment — which the AI era absolutely is — reversibility is a feature you should actively value and protect.** It lets you make most calls quickly and cheaply, reserve your deliberation for the genuinely irreversible, and stay adaptable as you learn. Now let's run some real decisions through everything we've built.

Worked Examples

Frameworks click when you see them run, so let's put real build-vs-buy decisions through the questions and watch how they land. Each is a common founder situation; the point isn't the specific verdict but how the framework *gets* there.

"Should we build our own authentication?" Run it: Core or context? *Context* — nobody chooses you for your login. Good option exists? *Yes, several excellent providers* (and even rolling your own is a known quantity, per the Auth (https://rogerstringer.com/guides/auth-done-right) guide). Can we own it for years? *Maybe, but security is a forever-burden few small teams should take on.* Reversible? *A provider with portable users is fairly reversible.* !**Buy** (or, if you specifically want control and can own it, the careful roll-your-own from the Auth guide — but never build it just because AI can). The AI-build impulse here is a classic trap: auth is the canonical context.

"Should we build the core thing our product actually does?" The risk engine, the canvas, the matching algorithm — whatever *is* the product. Core or context? *Core, definitionally.* Good option exists? *If one did, you'd have no product.* Worth owning? *Owning it IS the business.* !**Build**, and use AI to build it faster — pure upside, because you'd own it regardless and speed on your moat is leverage, not a trap. Easy call; the framework just confirms the obvious.

"Should we build an internal admin dashboard?" The murky middle. Core or context? *Context, but* close to your specific operations, so off-the-shelf fits poorly. Good option? *Generic admin tools exist but don't quite fit; a low-code option might.* Need it now? *Yes, the team is hurting.* Own it? *It's small and bounded — plausibly yes.* !**Lean build (small, AI-assisted) or low-code**, because it's bounded, poorly served by commodity tools, and close to your real workflows — one of the genuine "have AI build it" cases, *because* it's small and ill-fit by buying, not because generation is cheap.

"Should we build a CRM?" Core or context? *Context.* Good options? *Many mature ones.* Need it now? *Maybe — do you have enough customers to need one, or would a spreadsheet do?* !**Wait, then buy.** Use the spreadsheet until it hurts (the wait chapter), then buy the well-fit tool. Building your own CRM is almost never right; it's a deeply solved, deeply commodity problem, and an AI-built one is a maintenance sink masquerading as a saving.

"Should we build a tool for a process we do twice a month?" Need it now? *Barely.* !**Wait / do it manually.** The cheapest, most reversible option for a low-frequency pain is no tool at all. Don't build or buy for a problem that isn't costing you much.

The pattern across all of them: the framework keeps steering away from "can AI build it?" toward *core-vs-context, real-need, true-cost, ownership, reversibility* — and those questions produce sensible answers the AI-build excitement would have gotten wrong. Notice how often the verdict is **buy** or **wait**: that's not anti-building, it's the honest distribution once you price ownership instead of generation. Build is the deliberate exception for your core, not the default for everything AI can generate. The last chapter is about making the call and standing behind it.

Making & Owning the Call

You've got the framework; the last thing is the discipline of actually deciding — with conviction, in writing, and revisited as reality changes — because a good framework still requires someone to make the call and own it.

Decide, don't drift. The worst outcome isn't choosing wrong; it's *never choosing* — the need lingers, half-built tools rot, three overlapping subscriptions accrete, and nobody ever actually decided anything. Run the framework, reach a verdict (build, buy, or wait), and commit to it. Even "wait" is a real decision when it's conscious and has a trigger for revisiting; drifting is the failure. A leader's job here is to *end* the deliberation with a call, not keep it open forever.

Write down why. The single highest-leverage habit: record the decision and the reasoning — core or context, the true costs you weighed, the reversibility, what you're betting on. A short paragraph. This does two things: it forces clarity (you'll catch a weak decision while writing the rationale), and it gives you something to *check against later* when circumstances change. It's the same own-the-decision discipline the 70/30 Engineer (https://rogerstringer.com/guides/the-70-30-engineer) guide demands for code, applied to a strategic choice — a decision you can't explain is one you haven't really made.

Revisit when the facts change — because they will. These decisions aren't permanent, and the AI era makes the ground move fast: a tool you built may be better bought now that a great option appeared; a tool you bought may be worth building once it became core to your differentiation; a "wait" reaches its trigger when the pain finally arrives. Set a revisit point for the consequential ones ("reassess in a year, or when we hit X scale"), and *actually* revisit. The written rationale is what makes revisiting honest — you can see what you assumed and check whether it still holds. Changing your mind when the facts change isn't flip-flopping; it's the only sane way to operate in a fast-moving environment, and reversibility (the chapter before last) is what makes it cheap.

Own the outcome. Whichever you chose, you own what follows — the build you have to maintain, the vendor you bet on, the wait that may have cost you. Owning it means no "the AI made it look easy" excuses (the build-trap chapter) and no blaming the vendor for a dependency you chose. The accountability is the same one that runs through every guide here: you made the call, you stand behind it, you correct it when you're wrong.

That's the whole framework: **AI made generation cheap, but the build-vs-buy decision was never about generation — it's about ownership, fit, strategic value, timing, and reversibility, none of which AI changed.** Ask "do we want to own this for years?" not "can AI build it?"; build your core and buy your context; wait when the need isn't real; favor reversible choices; and decide, write it down, and own it. Do that and you'll navigate the "just build it with AI" era with judgment instead of hype — which is exactly the scarce skill the whole 70/30 Engineer (https://rogerstringer.com/guides/the-70-30-engineer) thesis says now matters most. For the leadership context around these calls, the Fractional CTO Field Guide (https://rogerstringer.com/guides/the-fractional-cto-field-guide) and First 90 Days (https://rogerstringer.com/guides/the-first-90-days) guides are the companions; for assessing technology you're buying or inheriting, Technical Due Diligence (https://rogerstringer.com/guides/technical-due-diligence). Now go make the call.

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.