Building with AI

The Vibes Don't Scale

Vibe coding works great for the first few PRs. Then you do it two hundred more times and the codebase quietly stops making sense. The fix isn't better prompts — it's building a machine that compounds.

Paul Stack
13 Apr 2026 • 6 min read

[Expanded from a LinkedIn post I wrote]

I keep seeing posts about how AI writes terrible code. Hallucinated APIs, ignored conventions, architectures that make no sense. And the conclusion is always "AI isn't ready for real engineering work."

I don't think that's right. I think the code is bad because the process is bad.

Vibe coding works great for the first few PRs. You describe what you want, the model writes it, tests pass, ship it. Then you do that two hundred more times and one morning you open the codebase and realize you can't tell which decisions were intentional and which were the model filling in blanks because nobody asked it to do anything specific. Nothing is broken exactly, it just drifted. The architecture softened and patterns that existed for a reason got quietly replaced by whatever the model felt like doing that day.

That's the actual risk. Not the hallucination that blows up immediately, you catch that in five minutes. It's the six months of code that works fine, passes every test, and slowly stops making sense as a whole. Same big ball of mud that engineering teams have always built when they ship without architecture, except now it happens in weeks instead of years because agents produce code so much faster. I've been there, it's painful, so I stopped doing it that way.

Systems thinking matters more now, not less

There's a tempting narrative that AI makes software architecture less important. Model can write code in seconds, why bother with upfront design? Just generate, test, iterate. We hear this referred to as AI Slop...

That gets the direction exactly backwards. When code is cheap to produce, the decisions about *what code to write* become the bottleneck. Architecture, domain

boundaries, conventions, the constraints that keep a codebase coherent, those matter more when you're shipping hundreds of commits a week, not less.

This isn't a new idea. Two books that have shaped how I think about this are *Thinking in Systems* by Donella Meadows and *Domain-Driven Design* by Eric Evans. Timeless, both of them, and both argue the same fundamental point: the structure of a system determines its behavior more than the quality of any individual component. A well-architected system with mediocre code outperforms a poorly-architected one with brilliant code: EVERY TIME! That was true when humans wrote everything and it's even more true now.

There's a deeper version of this worth naming. Even before AI, how many developers truly understood what their system was doing beneath the abstractions? Not the visible behavior but the actual mechanics under the covers. Most of us were already operating with a partial map. AI widens that gap dramatically. Agents writing code at speed across a codebase, and the distance between what's running and what any single person understands grows fast. That gap was uncomfortable before. But now it's dangerous because nobody is going to slow down and read every line when the agent is shipping ten PRs a day.

When an agent writes code without understanding the system it's contributing to, it optimizes locally. This solves the immediate problem in front of it, but local optimization in a complex system is how you get drift. Each change reasonable in isolation, overall direction random, because nobody is steering.

The fix isn't better prompts. It's giving agents the same structural context a senior engineer carries in their head. Which domain does this change belong to? What are the boundaries? What conventions exist and why? What constraints aren't in the code but shape every decision? An agent with that context writes fundamentally different code than one working from a one-line description.

Breaking down the issue keeps the scope clear

One pattern I noticed early: agents do better work when the scope is specific and the problem is well-understood before they start coding. It sounds obvious but the temptation with AI is to skip straight to implementation because it's so fast.

Some people will argue the solution isn't more upfront design — it's refactoring. Let the agent write, clean it up after. There's truth in that. But refactoring requires someone making decisions about what the design should actually be, and if you didn't have that clarity before the code was written you probably don't have it after either. What I've found works is baking the refactoring mindset into the planning phase rather than deferring it.

The lifecycle I've settled on forces a deliberate breakdown before any code gets written. Agent reads the issue, explores the codebase, classifies the problem.

Then it produces a versioned implementation plan — not a vague description, a step-by-step breakdown with specific files, testing strategy, risks identified. That plan gets adversarially reviewed: another pass specifically trying to find weaknesses, missing edge cases, scope creep. If the review finds critical problems, plan gets revised. Human gives feedback and the cycle repeats until the plan is clean.

Only then does implementation start.

The reason this matters isn't just quality: it's scope control. Without a plan, agents tend to solve adjacent problems they encounter along the way. They'll spot something that looks wrong in a nearby file and fix it. Refactor a function they're calling because it could be "better." Each micro-decision individually defensible, collectively devastating. Now your PR does six things instead of one and the reviewer can't tell what was intentional versus improvised.

A clear plan gives the implementation a contract. Code review checks the code against the plan it signed up to deliver. Drift becomes visible immediately because there's something specific to drift from.

Same principle behind good issue decomposition in any engineering process, AI or not. Break work down until each piece has a single clear purpose, hold the implementation to that purpose. The difference with AI is you have to be more explicit about it. The agent doesn't carry the implicit judgment of "this isn't what we're doing right now" that an experienced engineer has.

Learning from the process is the compounding advantage

The part most people skip is the feedback loop.

When a plan gets torn apart in adversarial review, that's information. When code review catches a convention violation, that's information. When the same architectural mistake shows up three times in a month, that's a pattern. Teams that capture those patterns and feed them back into the process get better over time. Teams that don't are just running the same mediocre workflow on repeat, wondering why the models aren't improving.

In practice this means a few things. The conventions file in your repo should be a living document that evolves every time you discover a new class of mistake. Planning constraints should get more specific as you learn what a good plan looks like for your particular codebase. Review criteria should reflect actual failure modes you've seen, not a generic checklist you wrote once.

Every time I hit a non-obvious problem during a session, e.g. something that wasted time, caused a wrong approach, revealed a gap in the conventions, I capture it. Not as a reactive rule ("don't do X") but as a positive convention ("when you see Y, the approach is Z"). Over months, this accumulates into a body of institutional knowledge the agents can read. First week, the process is clunky. Catches obvious problems. Six months in, the conventions are specific enough and the planning constraints tight enough that agents produce code that looks like it was written by someone who's been on the team for a year.

That compounding effect is the real argument for building the machine. Not that the first PR is better, it might not be, honestly. It's that the five hundredth PR is dramatically better because the process learned from the first four hundred and ninety-nine.

The prompt is the least interesting part

The thing I find strange about the discourse around AI code quality is how much it focuses on prompting. People share prompts, compare prompts, debate prompting strategies. Prompting matters, obviously. But it's maybe 5% of what determines whether the code is good.

The other 95% is everything the agent already knows before it sees the prompt. The architecture document. The domain conventions. Planning constraints. Review criteria. History of what worked and what didn't. The scope boundary of the specific task. Testing strategy. The adversarial dimensions the plan was challenged against etc etc etc.

All of that is context, and that context is a system design problem, not a prompt engineering problem.

If your AI workflow is "paste issue into chat, hope for the best," the code will be bad. Not because the model is bad. Because you gave it nothing to work with. You skipped the architecture, the breakdown, the plan, the review, the conventions, the learning loop, and then blamed the model for the output.

The models are capable of producing excellent code. But they need the same things a junior engineer needs: clear scope, understood conventions, a solid plan, someone checking the work. The difference is you can encode all of that into a process that runs every time, automatically, and gets better with every iteration.

Build the machine. The vibes don't scale.