What is agentic engineering?

I use the term **agentic engineering** to describe the practice of developing software with the assistance of coding agents.

What are **coding agents**? They're agents that can both write and execute code. Popular examples include Claude Code, OpenAI Codex, and Gemini CLI.

What's an **agent**? Clearly defining that term is a challenge that has frustrated AI researchers since at least the 1990s but the definition I've come to accept, at least in the field of Large Language Models (LLMs) like GPT-5 and Gemini and Claude, is this one:

Agents run tools in a loop to achieve a goal

The "agent" is software that calls an LLM with your prompt and passes it a set of tool definitions, then calls any tools that the LLM requests and feeds the results back into the LLM.

For coding agents, those tools include one that can execute code.

You prompt the coding agent to define a goal. The agent then generates and executes code in a loop until that goal has been met.

Code execution is the defining capability that makes agentic engineering possible. Without the ability to directly run the code, anything output by an LLM is of limited value. With code execution, these agents can start iterating towards software that demonstrably works.

Agentic engineering

Now that we have software that can write working code, what is there left for us humans to do?

The answer is *so much stuff*.

Writing code has never been the sole activity of a software engineer. The craft has always been figuring out *what* code to write. Any given software problem has dozens of potential solutions, each with their own tradeoffs. Our job is to navigate those options and find the ones that are the best fit for our unique set of circumstances and requirements.

Getting great results out of coding agents is a deep subject in its own right, especially now as the field continues to evolve at a bewildering rate.

We need to provide our coding agents with the tools they need to solve our problems, specify those problems in the right level of detail, and verify and iterate on the results until we are confident they address our problems in a robust and credible way.

LLMs don't learn from their past mistakes, but coding agents can, provided we deliberately update our instructions and tool harnesses to account for what we learn along the way.

Used effectively, coding agents can help us be much more ambitious with the projects we take on. Agentic engineering should help us produce more, better quality code that solves more impactful problems.

Isn't this just vibe coding?

The term "vibe coding" was coined by Andrej Karpathy in February 2025 - coincidentally just three weeks prior to the original release of Claude Code - to describe prompting LLMs to write code while you "forget that the code even exists".

Some people extend that definition to cover any time an LLM is used to produce code at all, but I think that's a mistake. Vibe coding is more useful in its original definition - we need a term to describe unreviewed, prototype-quality LLM-generated code that distinguishes it from code that the author has brought up to a production ready standard.

About this guide

Just like the field it attempts to cover, *Agentic Engineering Patterns* is very much a work in progress. My goal is to identify and describe patterns for working with these tools that demonstrably get results, and that are unlikely to become outdated as the tools advance.

I'll continue adding more chapters as new techniques emerge. No chapter should be considered finished. I'll be updating existing chapters as our understanding of these patterns evolves.

Writing code is cheap now →

Chapters in this guide

Writing code is cheap now

The biggest challenge in adopting agentic engineering practices is getting comfortable with the consequences of the fact that *writing code is cheap now*.

Code has always been expensive. Producing a few hundred lines of clean, tested code takes most software developers a full day or more. Many of our engineering habits, at both the macro and micro level, are built around this core constraint.

At the macro level we spend a great deal of time designing, estimating and planning out projects, to ensure that our expensive coding time is spent as efficiently as possible. Product feature ideas are evaluated in terms of how much value they can provide *in exchange for that time* - a feature needs to earn its development costs many times over to be worthwhile!

At the micro level we make hundreds of decisions a day predicated on available time and anticipated tradeoffs. Should I refactor that function to be slightly more elegant if it adds an extra hour of coding time? How about writing documentation? Is it worth adding a test for this edge case? Can I justify building a debug interface for this?

Coding agents dramatically drop the cost of typing code into the computer, which disrupts *so many* of our existing personal and organizational intuitions about which trade-offs make sense.

The ability to run parallel agents makes this even harder to evaluate, since one human engineer can now be implementing, refactoring, testing and documenting code in multiple places at the same time.

Good code still has a cost

Delivering new code has dropped in price to almost free... but delivering *good* code remains significantly more expensive than that.

Here's what I mean by "good code":

- The code works. It does what it's meant to do, without bugs.
- We *know the code works*. We've taken steps to confirm to ourselves and to others that the code is fit for purpose.
- It solves the right problem.
- It handles error cases gracefully and predictably: it doesn't just consider the happy path. Errors should provide enough information to help future maintainers understand what went wrong.

- It’s simple and minimal - it does only what’s needed, in a way that both humans and machines can understand now and maintain in the future.
- It's protected by tests. The tests show that it works now and act as a regression suite to avoid it quietly breaking in the future.
- It's documented at an appropriate level, and that documentation reflects the current state of the system - if the code changes an existing behavior the existing documentation needs to be updated to match.
- The design affords future changes. It's important to maintain YAGNI - code with added complexity to anticipate future changes that may never come is often bad code - but it's also important not to write code that makes future changes much harder than they should be.
- All of the other relevant "ilities" - accessibility, testability, reliability, security, maintainability, observability, scalability, usability - the non-functional quality measures that are appropriate for the particular class of software being developed.

Coding agent tools can help with most of this, but there is still a substantial burden on the developer driving those tools to ensure that the produced code is good code for the subset of good that's needed for the current project.

We need to build new habits

The challenge is to develop new personal and organizational habits that respond to the affordances and opportunities of agentic engineering.

These best practices are still being figured out across our industry. I'm still figuring them out myself.

For now I think the best we can do is to second guess ourselves: any time our instinct says "don't build that, it's not worth the time" fire off a prompt anyway, in an asynchronous agent session where the worst that can happen is you check ten minutes later and find that it wasn't worth the tokens.

← What is agentic engineering? Hoard things you know how to do →