Roll Your Own Coding Agent

Build a minimal coding agent from scratch — the tool-calling loop, context, and the Guide/Generate/Verify/Solve cycle in real code — so you understand exactly what's under Claude Code and Cursor.

Roger Stringer · rogerstringer.com

June 20, 2026

Contents

What's Actually Under Claude Code

The first time you watch Claude Code edit five files, run your tests, read the failures, and fix its own mistake, it feels like magic. It isn't. Underneath, a coding agent is three things, and you can hold all of them in your head at once: **a model, a set of tools, and a loop that runs between them.** That's it. The rest is polish.

Here's the whole idea in one breath. You send the model a request and a list of tools it's allowed to use. The model can't touch your files directly — it can only *ask*: "call `read_file` with this path," "call `run_command` with these args." Your code runs the tool, sends the result back, and the model decides what to do next. Loop that until the model says it's done. The "agent" is the loop. The intelligence is rented from the model. The *power* — the ability to actually change your world — comes entirely from the tools you hand it.

Understanding this from the inside is worth an afternoon, for a reason that pays off every day after: once you've built the loop yourself, the real tools stop being mysterious. You'll know why context matters so much (you're literally choosing what goes in the next message), why verification is a separate step (the loop has to *run* the tests to see them), why approvals exist (every tool call is your code deciding whether to proceed). The 70/30 Engineer (https://rogerstringer.com/guides/the-70-30-engineer) guide's AC/DC loop — Guide, Generate, Verify, Solve — stops being an abstraction once you've coded the thing that does it.

So we're going to build one. A small but genuinely working coding agent in TypeScript — call it **minicode** — that reads files, writes files, runs commands, and loops until a task is done. By the end it'll be a couple hundred lines you fully understand, and Claude Code will look less like a wizard and more like a very well-built version of the thing you just made.

No framework, no agent library. Just the model API and a loop. Let's start with the loop, because it's the whole skeleton.

The Core Loop

Before any API calls, let's get the shape of the thing in pseudocode, because everything else hangs off it. An agent loop is this:

```
messages = [ the user's task ]
loop:
    response = model(messages, tools)
    add response to messages
    if response asks to use tools:
        for each tool call:
            result = run the tool
            add result to messages
        continue loop          # let the model react to the results
    else:
        done — the model gave a final answer
```

Read it twice, because that's genuinely the entire architecture of every coding agent you've used. The model never acts on the world. It emits *intentions* — "I want to call `read_file`" — and your loop is what turns intention into action and feeds the consequence back. Generate, then act, then observe, then repeat.

Three things are worth noticing now, because they shape every decision later:

The conversation accumulates. `messages` grows every turn — the task, the model's tool requests, your tool results, all of it. That growing list *is* the agent's working memory. There's no hidden state; what the model knows on turn five is exactly what's in that array. Context engineering is just deciding what goes in it.

The model drives, your code gates. The model decides *what* to do, but your loop decides whether to actually do it. That seam — between the model asking and your code running — is where approvals, sandboxing, and safety live. There's a human-shaped gap in the loop, on purpose.

The exit condition is the model giving up its turn. The loop ends when the model responds *without* asking for a tool — it's said its piece and has nothing left to run. Until then it keeps working, which is exactly what makes it feel autonomous.

That's the skeleton. Now we wire in the one part we don't write ourselves: the model.

Talking to the Model

Time for the one piece we rent instead of build: the model. We'll use Anthropic's API and the messages endpoint. Install the SDK and set a key:

```
pnpm add @anthropic-ai/sdk
export ANTHROPIC_API_KEY=sk-...
```

The simplest possible call — no tools yet, just text in, text out:

```
import Anthropic from "@anthropic-ai/sdk";

const client = new Anthropic();

const res = await client.messages.create({
  model: "claude-sonnet-4-6",
  max_tokens: 1024,
  system: "You are a terse, precise coding assistant.",
  messages: [{ role: "user", content: "Say hello in one word." }],
});

console.log(res.content[0].type === "text" ? res.content[0].text : "");
```

Three things in that call matter for everything ahead. **`system`** is the standing instruction — the agent's personality and rules, sent every turn; this is where the Guide-stage context from Context Engineering (https://rogerstringer.com/guides/context-engineering-the-craft-of-agents-md) will eventually live. **`messages`** is the conversation array from our loop — the growing memory. And **`content` is a list of blocks**, not a string: a response can hold a text block and — once we add tools — `tool_use` blocks too.

That last detail trips people up, so hold onto it: the model doesn't reply with a string, it replies with an *array of typed blocks*, and you handle each by its `type`. We just took `content[0].text`; in a moment we'll loop over every block.

A real agent keeps appending to `messages`: you send the array, get a response, push the response on, and send it again next turn. The model is stateless — it remembers nothing between calls — so the array is the entire memory. Every clever thing an agent does to manage context is, underneath, just deciding what stays in that list and what gets dropped.

Right now this model can talk but it can't *do* anything — it has no hands. Tools are the hands. That's next, and it's the heart of the whole thing.

Tool Calling

This is the chapter that turns a chatbot into an agent. A **tool** is a function you describe to the model — a name, a description, and a schema for its inputs. You're not giving the model your code; you're giving it a *menu* and letting it order.

Here's a tool definition. Note it's pure description — no implementation yet:

```
const toolDefs = [
  {
    name: "read_file",
    description: "Read a file from the working directory and return its contents.",
    input_schema: {
      type: "object",
      properties: { path: { type: "string", description: "Path relative to cwd" } },
      required: ["path"],
    },
  },
];
```

Pass `toolDefs` to `messages.create`, and the model gains a new move: instead of (or alongside) text, it can emit a `tool_use` block — "I want to call `read_file` with `{ path: 'src/index.ts' }`." The response looks like this:

```
// res.content might be:
[
  { type: "text", text: "Let me check that file." },
  { type: "tool_use", id: "toolu_01", name: "read_file", input: { path: "src/
index.ts" } },
]
```

The model has *asked*. It hasn't read anything — it can't. Now your code does the call-and-return dance:

1. See the `tool_use` block, run the actual function (read the file).
2. Send the result back as a `tool_result` block, matched to the request by `tool_use_id`:

```
const toolResult = {
  role: "user",
  content: [{
    type: "tool_result",
    tool_use_id: "toolu_01",
    content: "<the file's contents>",
  }],
};
```

1. Append both the model's request and your result to `messages`, and loop. The model now *sees* the file contents and decides its next move.

That round trip — model asks via `tool_use`, you answer via `tool_result`, both go into the history — is the entire mechanism. Every capability a coding agent has is just another entry on that tool menu. So let's give it the three tools that make it a *coding* agent.

Your First Tools

A coding agent needs surprisingly few tools to be useful. Three, really: read a file, write a file, run a command. With those it can explore a repo, change it, and check its work — which is most of what coding is. Let's implement them.

```
import { readFile, writeFile } from "node:fs/promises";
import { execSync } from "node:child_process";

const toolImpls: Record<string, (input: any) => Promise<string>> = {
  async read_file({ path }) {
    return await readFile(path, "utf8");
  },

  async write_file({ path, content }) {
    await writeFile(path, content, "utf8");
    return `Wrote ${content.length} bytes to ${path}`;
  },

  async run_command({ command }) {
    try {
      return execSync(command, { encoding: "utf8", timeout: 30_000 });
    } catch (err: any) {
      // The model needs to SEE failures, not have them thrown away.
      return `Command failed:\n${err.stdout ?? ""}\n${err.stderr ?? err.message}`;
    }
  },
};
```

A few choices in there matter more than they look:

Tools always return a string the model can read. The model's whole world is text in the conversation. A tool that succeeds returns useful output; a tool that *fails* returns the error as text — it does not throw. That `catch` in `run_command` is doing essential work: when the tests fail, the agent needs to read the failure to fix it. Swallow the error or crash the loop, and you've blinded the agent at the exact moment it needs to see.

Outputs should be honest and bounded. Return what actually happened — real stdout, real stderr, the byte count. In a production version you'd also cap huge outputs (a 10,000-line log will blow your context window), but keep it simple for now.

These three compose into everything. "Run the tests" is `run_command`. "Fix the bug" is `read_file` then `write_file`. "Explore the project" is `run_command("ls -R")` then `read_file`. You don't need a tool per task — you need a small set of primitives the model can combine, exactly like a shell.

We now have a model that can ask and tools that can act. Time to wire them into the loop from chapter two and watch it actually work.

The Agent Loop in Real Code

Everything connects here. We have the loop skeleton, a model that emits `tool_use`, and three tools. Let's assemble the whole agent — this is the heart of minicode, and it's shorter than you'd expect:

```
import Anthropic from "@anthropic-ai/sdk";
const client = new Anthropic();

async function runAgent(task: string) {
  const messages: Anthropic.MessageParam[] = [
    { role: "user", content: task },
  ];

  while (true) {
    const res = await client.messages.create({
      model: "claude-sonnet-4-6",
      max_tokens: 4096,
      system: SYSTEM_PROMPT,
      tools: toolDefs,
      messages,
    });

    // Record what the model said/asked.
    messages.push({ role: "assistant", content: res.content });

    // Print any text the model emitted.
    for (const block of res.content) {
      if (block.type === "text") console.log(block.text);
    }

    // If it didn't ask for tools, it's done.
    if (res.stop_reason !== "tool_use") break;

    // Run every requested tool, collect the results.
    const results: Anthropic.ToolResultBlockParam[] = [];
    for (const block of res.content) {
      if (block.type !== "tool_use") continue;
      console.log(`!${block.name}(${JSON.stringify(block.input)})`);
      const out = await toolImpls[block.name](block.input);
      results.push({ type: "tool_result", tool_use_id: block.id, content: out });
    }

    // Feed the results back; loop so the model can react.
    messages.push({ role: "user", content: results });
  }
}
```

That's a working coding agent. Hand it `runAgent("Add a /health route to src/server.ts that returns 200 OK")` and watch: it'll `read_file` the server, `write_file` the new route, maybe `run_command` the tests, and narrate as it goes. The loop drives, the model decides, the tools act, and the conversation array carries the memory between turns.

Trace one cycle against the code to make it concrete: the model emits a `text` block ("Let me read the server") and a `tool_use` block (`read_file`); we push that, see `stop_reason` is `tool_use`, run the read, push the result as a `tool_result`; loop; now the model *sees the file* and emits a `write_file` call; and so on, until it answers with text and no tool call, and we break.

It works — but it's a blunt instrument. It knows nothing about your project, follows no rules, and you're trusting it completely. The next chapters sharpen it: context, guidance, verification, and the safety gate. Start with what it knows.

Context & Memory

Our agent starts every task knowing nothing but the task string. A real coding agent knows about your *project* — and now that you've built the loop, you can see exactly what "knowing" means: it's whatever you put in the `messages` array and the `system` prompt before the model runs. Context isn't magic. It's data you choose to include.

There are two levers, and you're already holding both.

The system prompt is standing knowledge. It's sent every turn, so it's where project-wide truth goes — the architecture, the conventions, the rules. The obvious move: read an `AGENTS.md` from the repo and drop it in.

```
import { readFileSync, existsSync } from "node:fs";

const projectContext = existsSync("AGENTS.md")
  ? readFileSync("AGENTS.md", "utf8")
  : "";

const SYSTEM_PROMPT = `You are minicode, a coding agent.
Work in small steps. Read before you write. Run tests to verify.

${projectContext ? "# Project context\n" + projectContext : ""}`;
```

That's the Context Engineering (https://rogerstringer.com/guides/context-engineering-the-craft-of-agents-md) guide seen from the engine room: `AGENTS.md` is powerful precisely because it's text that rides in the system prompt on *every single turn*. You built the thing that reads it, so you understand first-hand why a sharp `AGENTS.md` changes outcomes and a bloated one costs you — every line is in every request.

The messages array is working memory. Everything the agent has read and done this task lives there, and it grows fast — a few file reads and you're carrying thousands of tokens of history. That's fine until it isn't: long tasks will overflow the context window. The real tools manage this by *compacting* — summarizing old turns, dropping stale file contents, keeping the recent and relevant. You don't need that yet, but now you can see why it exists: the memory is a finite array, and someone has to decide what stays.

The lesson the loop teaches that no explanation can: an agent doesn't "have" context, it's *given* context, every turn, by code you control. Which means the quality of the agent is largely the quality of what you feed it — the Guide stage, again. So let's make the guidance sharp.

Guiding the Agent

The same model, in the same loop, can be a careful engineer or a reckless one — and the difference is the system prompt. This is the Guide stage of the AC/DC loop, except now you're writing the actual string that does it, which makes the principle concrete in a way reading about it never does.

A naive system prompt ("You are a coding agent") gets you a model that guesses at everything. A good one encodes how you want it to *work*:

```
const SYSTEM_PROMPT = `You are minicode, a careful coding agent.

# How you work
- Read the relevant files before changing anything. Never edit blind.
- Make the smallest change that does the job. One step at a time.
- After any code change, run the tests with run_command and read the output.
- If a tool fails, read the error and adjust — don't repeat the same call.

# What you never do
- Don't invent file paths or APIs. If unsure, read or list first.
- Don't add dependencies. Use what's already in the project.
- Don't claim something works without running it.

${projectContext}`;
```

Every line there is steering you can feel. "Read before you write" stops the agent editing files it's hallucinating. "Run the tests after changes" wires verification into the agent's own habits instead of leaving it to you. "Don't claim it works without running it" attacks the confident-completion problem the [70/30 Engineer](https://rogerstringer.com/guides/the-70-30-engineer) guide warns about — directly, in a sentence.

Two things you'll notice once you start tuning this:

The system prompt is a behaviour dial, and small changes move it a lot. Add one line — "prefer editing existing files over creating new ones" — and the agent's whole style shifts. This is the cheapest, highest-leverage place to improve your agent, exactly as the Guide stage promises.

You can't prompt your way out of missing tools or context. If the agent keeps guessing wrong, the fix is usually a better `AGENTS.md` or a tool it's missing (like `list_files`), not more stern instructions. Guidance shapes *how* it works with what it has; it can't conjure what isn't there.

Guidance tells the agent to verify. But telling isn't enough — the loop has to actually run the check and react to it. That's the Verify stage, and it's where good agents separate from demos.

Verifying & Looping

The single thing that makes a coding agent trustworthy rather than impressive is that it *checks its own work* — and you've already built the machinery for it. `run_command` runs the tests; the tool result feeds the failure back; the loop lets the model react. Verification isn't a feature you add; it's the loop doing what it already does, pointed at a test command.

Watch the Verify/Solve cycle fall out of the loop naturally. Give minicode "fix the failing test in `cart.test.ts`" and it runs like this:

```
!run_command("pnpm test cart.test.ts")
   FAIL: expected total 30, got 25  (discount applied twice)
!read_file("src/cart.ts")
!write_file("src/cart.ts", <fix>)
!run_command("pnpm test cart.test.ts")
   PASS
!"Fixed: the discount was applied in both addItem and checkout. Removed the duplicate."
```

Nothing new in the code made that happen — it's the chapter-six loop plus a system prompt that says "run the tests after changes." The agent generated a fix, *verified* it by running the suite, read the result, and only stopped once the check passed. The test output going back into the conversation as a `tool_result` is the entire trick: the agent can see whether it succeeded, so it doesn't have to guess.

This is why, in the 70/30 Engineer (https://rogerstringer.com/guides/the-70-30-engineer) framing, verification is the new bottleneck — and building the loop shows you why precisely. The model can generate a fix in one turn, but *confirming* the fix requires actually running something and feeding the result back, which is a whole extra cycle the agent (or you) has to drive. Generation is one block; verification is a round trip.

Two upgrades make verification real rather than hopeful:

Make tests the agent's definition of done. A system-prompt line — "a task is complete only when the relevant tests pass, and you've run them this turn" — turns 'looks done' into 'proven done.' The agent stops lying to itself because you defined the bar as a command it has to run.

Let failures drive, but cap the loop. A failing test feeds back and the agent tries again — good. But an agent that *can't* fix something will loop forever, burning tokens. Add a turn limit (say, 25 iterations) and bail with the last state — the same circuit-breaker instinct the Running the Fleet (https://rogerstringer.com/guides/running-the-fleet) guide builds for multi-agent systems.

The agent can now check itself. But it's still doing whatever it wants to your filesystem the moment it decides to. Before you let it run unattended, you need a hand on the cord.

Approvals & Safety

Your agent will, given the chance, run any command the model emits — including `rm -rf` if it talks itself into it. Building the loop made the danger legible: there's one exact line where intention becomes action — where your code calls `toolImpls[block.name]` — and that line is the only place safety can live. Everything protective hangs off that seam.

The first and most important guard is an **approval gate** on the tools that can hurt you:

```
const DANGEROUS = new Set(["write_file", "run_command"]);

for (const block of res.content) {
  if (block.type !== "tool_use") continue;

  if (DANGEROUS.has(block.name)) {
    console.log(`Agent wants: ${block.name}(${JSON.stringify(block.input)})`);
    const ok = await confirm("Allow? (y/n) ");
    if (!ok) {
      results.push({
        type: "tool_result",
        tool_use_id: block.id,
        content: "Denied by user.",
        is_error: true,
      });
      continue;
    }
  }
  const out = await toolImpls[block.name](block.input);
  results.push({ type: "tool_result", tool_use_id: block.id, content: out });
}
```

Notice the denial is just another `tool_result` — you tell the model "no" in its own language, and it adapts (tries a different approach, or asks why). The human isn't outside the loop bolting on safety; the human is *a participant in the loop*, exactly the [70/30 Engineer](https://rogerstringer.com/guides/the-70-30-engineer) "keep a hand on the wheel" idea, now a literal `if` statement.

The layers worth stacking, cheapest first:

- **Approve the irreversible.** Reads are safe; writes and commands aren't. Gate the ones that change the world, auto-allow the ones that don't. That alone removes most of the risk.
- **Scope the blast radius.** Refuse paths outside the working directory, block obviously destructive commands, and — for real autonomy — run the whole thing in a container or VM, so a bad call is contained by the OS, not by your good intentions. (The [Running the Fleet](https://rogerstringer.com/guides/running-the-fleet) guide makes the same point: a scoped personality is not a scoped blast radius; you need a real sandbox.)
- **Never feed it secrets it doesn't need.** The agent's world is the context you give it. Don't put production credentials in reach of a process that improvises.

The mental model the loop hands you: an agent is exactly as dangerous as its tools and exactly as safe as the gate in front of them. You wrote the gate, so you decide. Which means you can now safely let it do something bigger — like delegate.

Subagents & Delegation

Here's a move that feels advanced and is almost trivial once you have the loop: an agent can *call another agent*. Make `runAgent` itself a tool, and your single agent becomes a tiny fleet.

Why bother? Because context is finite (chapter seven) and focus beats breadth. When the main agent hits a self-contained sub-problem — "summarize what these twelve files do," "work out how this library's API is used" — you don't want all that exploration cluttering the main conversation. You want a fresh agent to go do it, come back with just the answer, and leave its mess behind.

That's a tool:

```
// definition
{
  name: "delegate",
  description: "Spawn a sub-agent to handle a focused subtask and return only its
result.",
  input_schema: {
    type: "object",
    properties: { task: { type: "string" } },
    required: ["task"],
  },
}

// implementation
async delegate({ task }) {
  return await runAgent(task);   // a fresh loop, its own message history
}
```

The sub-agent runs its own loop with its own clean `messages` array, burns through whatever exploration it needs, and returns one string. The parent's context stays uncluttered — it sent a sentence and got an answer, not the twelve files. You've just built the `delegate_task` primitive the Running the Fleet (https://rogerstringer.com/guides/running-the-fleet) guide describes, from scratch, in a handful of lines.

And now you can *see* the distinction that guide draws between two kinds of delegation, because you'd implement them differently. A `delegate` the parent waits on — fork, get the answer, continue — is a function call; it's for sub-answers the parent needs to keep working. A durable task queue that outlives the parent, that another agent picks up later, is a different shape entirely — a board, not a function call. The fleet guide is the second shape; this is the first. Same idea, scaled.

This is the seed of everything in the Agentic OS (https://rogerstringer.com/guides/building-your-agentic-os) and Fleet guides: specialists, coordination, one brain spawning many heads. It all grows from "the loop can call the loop." You don't need a framework to have a multi-agent system — you need a `delegate` tool and the nerve to recurse.

Which brings us to the real payoff of building all this yourself.

What You Learned by Building It

You built a coding agent. It's a couple hundred lines — a loop, a model call, three tools, a system prompt, an approval gate, and a `delegate` that recurses. It is not a toy: point it at a real repo with a real `AGENTS.md` and it will read, edit, test, and fix actual code. But the running agent isn't the prize. The understanding is.

Here's what you can now *see* that was invisible before:

- **An agent is a loop, a model, and tools.** Every coding agent you'll ever use is a more-polished version of what you just wrote. The magic was always machinery.
- **The model only ever asks; your code acts.** That seam is where everything important lives — safety, approvals, sandboxing, the human in the loop. You can't reason about agent safety until you've seen there's exactly one line where intention becomes action.
- **Context is a choice you make every turn.** `AGENTS.md` in the system prompt, file contents in the messages array — the agent knows what you feed it and nothing more. Context Engineering (https://rogerstringer.com/guides/context-engineering-the-craft-of-agents-md) stops being abstract once you've written the `readFileSync` that does it.
- **Verification is a round trip, which is why it's the bottleneck.** Generating a fix is one block; proving it works is a whole extra cycle the loop has to drive. The 70/30 Engineer (https://rogerstringer.com/guides/the-70-30-engineer) guide's whole thesis is visible in your own loop.
- **A fleet is just the loop calling the loop.** Agentic OS (https://rogerstringer.com/guides/building-your-agentic-os) and Running the Fleet (https://rogerstringer.com/guides/running-the-fleet) are this, scaled, with coordination on top.

The practical reason this matters: you'll spend the next decade working *with* these tools, and the engineers who understand the loop will get far more out of them than the ones who treat them as wizardry. You'll write better `AGENTS.md` files because you know how they're consumed. You'll trust the right things and gate the dangerous ones because you know where the seam is. You'll debug a misbehaving agent by asking "what's in its context?" instead of re-rolling the dice. Understanding the machine is leverage.

Should you build your own agent for real work? Mostly no — Claude Code and its kin are better-built, better-tested, and maintained by people who do only this. Build your own to *understand*, then use the good ones to *ship*, the same way you'd read the source of a framework you depend on. The point was never to compete with the real tools. It was to make them stop being magic.

From here: Context Engineering (https://rogerstringer.com/guides/context-engineering-the-craft-of-agents-md) to feed your agents well, the 70/30 Engineer (https://rogerstringer.com/guides/the-70-30-engineer) for the discipline of working with them, and Building Your Agentic OS (https://rogerstringer.com/guides/building-your-agentic-os) when you're ready to give one a memory and a life of its own. You know what's under the hood now. Go drive.

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.