Build a self-improving AI system with Claude — the 14-step guide

You commented AGENTS. Here's the whole thing — the 14 steps that turn "I want an AI that does the work" into a real agent that runs itself, reaches for the right tools, handles its own failures, and hands you something you can actually ship.

The thing nobody tells you

Everyone obsesses over the model. Which one's smartest, which one's cheapest, which one topped the leaderboard this week. Then they wire it up, watch it flail on the first real task, and decide "AI isn't there yet."

The model was never the hard part. A good model with no system around it is a genius locked in an empty room — no hands, no memory, no way to check its own work. What makes an agent useful is everything *around* the model: the tools it can reach, the permissions it can't cross, the credentials it never actually sees, the way it watches its own output and fixes what broke.

That's the whole game. And with Claude's Managed Agents — Anthropic runs the agent loop and hands you a sandboxed workspace per run — you get to build that system without hosting the plumbing yourself. Here's how I do it, step by step.

What it looks like when it works

You hand the thing a task once. It picks the right tool for each step, calls an API without ever touching the API key, hits an error, reads the error,

patches its own approach, keeps going — and when it's done, you get one clean diff to review instead of a pile of half-finished guesses.

You watched the whole run in a trace view. You never typed a password into a prompt. Nothing touched production without your say-so. That's the difference between a demo and a system.

Step 1 — Define the agent (write its system prompt)

Start with who it is, not what model it uses. The **agent** is a saved, versioned config — a name, a system prompt, its tools, its skills. The system prompt is where you write the job: what it's for, how it should behave, what "done" looks like, what it must never do.

Write it like you're briefing a new hire who's sharp but has zero context on your world. Be concrete. "You review pull requests for security bugs and report every finding with a severity" beats "you are a helpful assistant." This config is reusable — you write it once and every run points at it.

Step 2 — Pick the model

Now you pick the brain. This is one line on the agent, not the whole project. Default to a strong model for real work; drop to a faster, cheaper one for simple, high-volume steps.

The reason this is step 2 and not step 1: the model is a swappable part. You can change it later without rebuilding anything else. Getting the system right around it is what actually moves the needle.

Step 3 — Give it skills + tools

An agent with no tools can only talk. Give it hands. The built-in toolset covers the essentials — running commands, reading and writing files, searching, fetching web pages. Turn on the full set, then switch off anything this particular job has no business touching.

Skills are the other half: packaged know-how the agent loads only when the task calls for it — building a spreadsheet, editing a slide deck, writing a doc. You attach the skill; the agent reaches for it when it's relevant and ignores it when it isn't. Tools are what it can *do*; skills are how it knows to do the specialized stuff well.

Step 4 — Set permissions (least privilege)

Here's where most people get lazy and later regret it. Every tool can run one of two ways: it just goes, or it pauses and asks you first.

Auto-run the safe, reversible stuff — reading files, searching. Put a confirmation gate on anything that's hard to undo: deleting data, pushing code, sending a message, hitting an external API that costs money. When a gated tool comes up, the agent stops and waits for your yes or no. Give it exactly the access the job needs and not one notch more. A narrow agent is a safe agent.

Step 5 — Configure the environment (network allowlist)

The agent's tools run inside a sandboxed workspace, and that workspace has a door to the internet. Decide how open it is.

Default is wide-open egress, which is fine for a lot of jobs. But if you want control — and for anything touching real systems, you do — switch to a locked-down mode where nothing gets out unless you allow it. List the exact hosts the job needs to reach and nothing else can be contacted. It's the same instinct as least-privilege on tools, applied to the network: the agent can only phone the handful of places you named.

Step 6 — Preinstall the packages it needs

If the job needs specific libraries — a data-analysis run needs its data stack, a scraper needs its parser — set the environment up to have them ready. Don't make the agent burn its first few steps installing dependencies every single run, and don't leave it stuck when the network's locked down and it can't fetch them.

Get the workspace prepped so the moment the agent starts, everything it reaches for is already there. Boring step. Saves you a lot of failed runs.

Step 7 — Create a session

The agent is the reusable blueprint. A **session** is one actual run of it. You create a session pointing at your agent plus the environment it runs in, and that spins up a fresh, isolated workspace for this specific job.

Every session is its own container — its own files, its own state, cleaned up when it's done. Same agent, a hundred sessions, none of them stepping on each other. This is the unit of "do the thing once."

Step 8 — Load it with resources (files, repos, data)

An agent working blind is guessing. Give it the material. You attach **resources** to the session at startup — a data file to analyze, a code repo to work in, a reference doc to follow. They get mounted into the workspace before the agent takes its first step.

This is the difference between "write me a function" and "here's the actual repo, fix the actual bug." Load in what the job is actually about. Real context in, real work out.

Step 9 — Wire credentials through a vault

This is the step that separates a toy from something you'd trust near real systems, so read it twice.

Your agent will need secrets — an API key, a token. **Do not paste them into the prompt.** Anything in a prompt gets stored in the run's history and can be read back later. That's how keys leak.

Instead, put the secret in a **vault**. The workspace only ever sees an opaque placeholder — a meaningless stand-in. The real secret gets swapped in *at the moment the request leaves the sandbox*, on its way to the one service you allowed, and never before. Encrypted at rest, invisible inside the workspace, substituted only at egress. So even if the agent goes sideways or someone tries to trick it, there's no key sitting there to steal — the code running in the box literally cannot read it. That's the whole point: the agent uses the credential without ever holding it.

Step 10 — Kick off the task

Everything's wired. Now you actually start it — you send the agent the task as a message and it goes to work.

One tip that matters more than it looks: **spell out the full job up front, in one clear brief.** Goal, constraints, what done looks like. A well-specified kickoff gets you a far better run than dribbling instructions in one at a time and hoping it stitches them together. Say the whole thing, then let it work.

Step 11 — Handle events (watch its actions; steer or stop it)

A running agent streams a live feed of what it's doing — every message, every tool call, every result. You're not flying blind and you're not locked out. You watch that feed.

If it's heading the wrong way, you can send a **steer** ("actually, focus on the auth module") or an **interrupt** that stops it cleanly and hands control back to you. When it hits one of those permission gates from step 4, this feed is where the "can I run this?" shows up and where you answer. You're the pilot with a hand near the controls, not a passenger.

Step 12 — Turn on observability (session tracing)

You want a record, not just a live glance. Every run has a full **trace** — the complete play-by-play of what the agent thought, what it called, what came back, what it spent. The moment you start a session, grab the link to its trace view and keep it.

When a run does something surprising — and eventually one will — this is where you go to see exactly what happened, step by step, instead of

guessing. You can't improve a system you can't see. Observability is how the thing becomes debuggable, and debuggable is how it gets better over time.

Step 13 — Let it self-recover (read failures, patch, continue)

This is the "self-improving" part people mean when they say it. A good agent doesn't just die on the first error. It reads the failure — the test that broke, the command that errored — figures out what went wrong, adjusts, and keeps going. The tools give it eyes on its own output; a clear "here's what done looks like" gives it a target to keep checking itself against.

Your job is to *set this up*, not to babysit it: give it a way to verify its own work (run the tests, check the output), tell it to keep going until the work actually passes, and let it loop. The recovery is the agent's job. Building the conditions where recovery is possible is yours.

Step 14 — Review the diff/output before you ship

The agent finishing is not the same as the work being right. Last step is always yours: look at what it produced before it goes anywhere real.

When it wrote code, read the diff like you'd review a pull request from a junior — every change, against the plan. When it produced a file or a report, open it and check it. This is why steps 4, 5, and 9 mattered: nothing hit production on its own, so you get a calm final review instead of an emergency cleanup. Approve it, then ship it. The human sign-off is a feature, not a delay.

The bigger picture

Notice what actually made this work. Not one killer prompt and not the fanciest model — a *system*. Tools so it can act. Permissions so it can't overreach. A vault so secrets never leak. A network allowlist so it only phones home where you allowed. A live feed and a trace so you can see everything. A way to check its own work so it recovers instead of quitting. And a human at the end.

That's the real skill, and it's the thing I do all day at Axionox — building production AI systems for clients, not no-code duct tape. The models keep getting better on their own. The part that's actually worth learning is the system you build around them, because that's what turns a smart model into something you'd trust to run real work.

Once you've built one agent this way, you stop asking "which model is best" and start asking "what does the system around it need." That shift is the whole thing.

Your next step

Don't try to automate your whole job tonight. Pick one small, boring, repetitive task you already know the steps to — a report you pull every week, a check you run by hand, a file you reformat over and over.

Build an agent for just that. Walk the 14 steps once, on something low-stakes where a mistake costs you nothing. Watch it in the trace. See it hit an error and fix itself. That's the moment it clicks — and after that, you'll look at every repetitive task differently.

The cost of not starting isn't dramatic. It's just that every week you keep doing by hand the work a system like this would already be doing for you.

Built by Saad Bhatti — Axionox. I build production AI systems for a living (axionox.com). Follow for more like this.