Agent Skills: A Field Guide to the Third Pillar

Your agent can write code. But does it know how your team cuts a release? Can it run your incident playbook the same way twice, or does it improvise something a little different every time? That gap, between raw capability and a repeatable way of doing one specific job, is exactly what skills fill. A skill is procedural memory you write down once: a packaged, reusable how-to that the agent loads when it's relevant and runs the same way every time.

This is the third leg of a trilogy with [Agent Memory](/guides/agent-memory-field-guide) and [The Agent's Self](/guides/agent-self-personality-identity), the three pillars from [Building Your Agentic OS](/guides/building-your-agentic-os). Identity is who the agent is, memory is what it knows, skills are how it does things. We start with what a skill really is, and what it isn't, then build one from a plain folder and a single file. We dig into the two halves of the craft that actually matter: writing a description that makes the agent reach for the skill at the right moment, and

writing a body that makes it succeed once it does. We cover progressive disclosure (why the whole skill isn't sitting in context all the time), how to tell a skill apart from a memory or a tool, and how to version and share skills across a fleet without letting them rot.

By the end you'll be able to take a capable, general-purpose agent and turn it into a specialist that does your specific jobs your specific way, on demand, every time.

This is a living document and will be updated as the tools and patterns evolve.

Roger Stringer · rogerstringer.com

July 1, 2026

Contents

The Third Pillar

If you've read the other two guides in this series, you already have two-thirds of the picture. An agent needs an identity so it knows who it is, and a memory so it knows what's happened. This guide is about the part that's left, and it's the part that actually gets work done: skills.

The who, the what, the how

Here's the split that ties the three pillars together. Identity is the *who*. Memory is the *what*. Skills are the *how*. An agent with a strong self and a good memory can still flail the moment you ask it to do something with a specific procedure, because knowing who you are and what you know doesn't tell you the steps. Skills are where the steps live.

Think about a new hire who's genuinely smart. They have judgment (identity) and they're picking up context about your company (memory). But the first time they cut a release, you walk them through it, because there's a way your team does it that they couldn't have guessed. Write that walkthrough down and hand it to the next hire, and you've made a skill.

What capability misses

A capable model can write code, summarize a document, query a database. That's raw ability, and it's real. What it doesn't have is *your* way of doing a specific job. It doesn't know that your releases need a changelog entry, a version bump, a tag, and a particular Slack message in that order. Left to itself it'll invent something plausible, and plausible-but-different every time is exactly the problem.

Skills close that gap. A skill takes a job the agent could fumble its way through and turns it into something it does correctly and consistently, the way you'd do it yourself.

Procedural memory needs its own home

You might wonder why this isn't just memory, or just part of the agent's constitution. It's a fair question, and the answer is about fit. The constitution is small and always loaded; you can't cram every procedure into it without bloating the thing that defines the agent's character. Memory is for facts that accumulate, not step-by-step methods you want run precisely. Procedures are their own kind of knowledge, and they deserve their own structure: named, packaged, loaded only when the job comes up.

That structure is what the rest of this guide builds. We'll start by pinning down what a skill actually is, because the word gets thrown around loosely and the precise version is more useful.

What a Skill Actually Is

"Skill" is one of those words that's drifted into meaning almost anything, so let's be precise. A skill, in the sense that matters here, is a packaged, reusable procedure that an agent can discover, load, and run when a task calls for it. Concretely, it's usually a folder with a single instruction file at its root and whatever supporting files that procedure needs.

That's it. The power is in the packaging, not the magic.

A worked example

Say you have a release process. A skill for it is a folder, maybe called `cut-release`, containing a file that explains: when to use this (the agent is asked to ship or release), and the exact steps (update the changelog, bump the version, run the build, tag the commit, post the announcement). Drop that folder where your agent can find it, and now "cut a release" is something it does your way instead of guessing.

The procedure went from living in your head, or in a wiki nobody reads, to living somewhere the agent will actually pick it up at the moment it's relevant.

What a skill is not

Three quick contrasts, because skills are easiest to understand by what they're adjacent to.

A skill is not a **prompt**. A prompt is what you type for one task, gone when the task ends. A skill persists and is reused across many tasks without you retyping anything.

A skill is not a **memory**. Memory is facts the agent accumulates: "this project uses Bun," "the user prefers terse commits." A skill is a method: "here is how to do X, start to finish." Facts versus procedures. (We'll draw that line more carefully in a later chapter, because it trips people up.)

A skill is not a **tool**. A tool is a capability the agent invokes, like "run a shell command" or "search the web." A skill is instructions that often *use* tools to carry out a procedure. The tool is the hammer; the skill is the how-to for building the chair.

Why files

Notice a skill is just files. That's deliberate, and it's the same instinct that makes file-based memory and a file-based constitution the right starting point. Files are readable, you can keep them in git, you can diff a change, and you can hand the folder to a teammate. A skill you can open and read is a skill you can trust and fix. We'll lean on that all the way through. Next, let's open one up and look at its anatomy.

Anatomy of a Skill

Let's build one. A skill has a small, predictable shape, and once you've seen it, you'll spot the structure everywhere.

The instruction file

At the root of the skill folder is a single instruction file, conventionally something like `SKILL.md`. It has two jobs, and they're worth keeping mentally separate because the rest of this guide hinges on the distinction.

The top of the file is **metadata**: a name and, most importantly, a description. The description is short, and it answers one question: when should the agent reach for this skill? The body below is the **instructions**: the actual procedure, in as much detail as the job needs.

Here's a skinny example for a code-review skill:

```
---
name: review-pr
description: Review a pull request for bugs and clarity. Use when asked
  to review, critique, or check a PR or a diff before it merges.
---

# Reviewing a PR

1. Read the diff in full before commenting on anything.
2. Flag correctness bugs first, then clarity and maintainability.
3. For each finding, give the file and line and a concrete suggestion.
4. End with a one-line verdict: approve, approve-with-nits, or request-changes.
```

That's a complete, working skill. A name, a description that says when to use it, and a body that says how.

Bundled resources

Many skills need more than prose. A skill folder can carry whatever the procedure depends on:

- **Scripts** the agent runs as part of the steps (a build script, a deploy helper).
- **Templates** it fills in (a PR description template, a report skeleton).
- **Reference files** it consults (a style guide, a checklist, an example of a good output).

The skill body points at these by relative path, and they travel with the folder. This is what turns a skill from "some instructions" into a self-contained unit you can drop anywhere and have it just work.

Keep one skill to one job

The most useful discipline early on: one skill, one job. A `cut-release` skill cuts releases. It does not also review code and answer support tickets. Narrow skills are easier to describe (so they trigger reliably), easier to maintain, and easier for the agent to pick the right one from. When a skill starts trying to do three things, that's usually a sign it should be three skills.

With the shape clear, the interesting part is how the agent decides to use a skill at all without reading every one it has. That's progressive disclosure, and it's next.

Progressive Disclosure

Here's a question that sounds trivial and isn't: if an agent has fifty skills, how does it use the right one without all fifty sitting in its context the whole time? Loading every skill in full would blow the context budget and bury the model in instructions for jobs it isn't doing. The answer is the single most important idea in skill design, and it has a name: progressive disclosure.

Two layers, loaded at different times

Remember the split from the last chapter, metadata versus body. Progressive disclosure uses that split as a loading strategy.

The **descriptions** are always available. A short line for each skill, naming what it's for, sits in the agent's context all the time. This is cheap, a sentence per skill, and it's enough for the agent to know what's on offer.

The **body** loads only when needed. When a task comes in that matches a skill's description, the agent opens that skill and pulls the full instructions into context, just for that job. The other forty-nine stay closed.

So the agent is always aware of everything it *can* do, but only ever reading the details of the thing it's actually doing. It's the table of contents versus the chapter. You keep the table of contents in front of you; you only flip to the chapter you need.

Why this is the whole game

Progressive disclosure is what lets a skill library scale. Without it, every skill you add taxes every single task, and you'd hit a ceiling fast. With it, you can have a hundred skills and pay almost nothing until one is relevant. It's the difference between a reference library, where having more books costs you nothing until you open one, and being handed all hundred books open to read at once.

It also changes how you write skills, which is the part people miss. Because the description is doing the work of deciding whether the body ever gets read, the description carries enormous weight. A skill with a vague description is a skill the agent never opens, no matter how good the instructions inside are. And a skill body can be as long and detailed as the job genuinely needs, because that length only costs anything on the occasions it's actually used.

That sets up the two halves of the craft. Writing a description good enough that the agent reaches for the skill at the right moment, and writing a body good enough that it succeeds once it does. The next two chapters take those one at a time, starting with the description.

Writing a Skill That Triggers

A skill that never gets used is worthless, no matter how good its instructions are. And whether a skill gets used comes down almost entirely to its description, because that's the only part the agent sees when it's deciding. So this is where a lot of the craft lives: writing a description that fires at the right moment and stays quiet the rest of the time.

Describe when, not just what

The instinct is to describe what the skill does. "Reviews pull requests." That's not wrong, but it's weak, because it doesn't tell the agent *when* the moment has arrived. The stronger move is to describe the trigger: the situations, phrasings, and tasks that should make the agent reach for it.

Compare:

- Weak: "A skill for releases."
- Strong: "Use when the user asks to ship, cut, publish, or release a new version, or mentions bumping the version or tagging a build."

The second one is a net. It names the words and situations that should catch, so the agent recognizes the moment even when the user phrases it sideways. You're not writing documentation for a human; you're writing a recognizer for a model.

Over-triggering and under-triggering

There are two ways a description fails, and they pull in opposite directions.

Under-triggering is when the skill is too narrowly described and the agent doesn't realize a task qualifies. The fix is to widen the description: add the synonyms, the adjacent phrasings, the related situations.

Over-triggering is when the description is so broad the agent reaches for the skill when it shouldn't, dragging an irrelevant procedure into the middle of an unrelated task. The fix is to narrow and add boundaries: say what it's *not* for.

You tune between these by watching real behavior. If a skill keeps getting skipped when it should fire, broaden. If it keeps barging in, tighten. It's a dial, not a one-time setting.

Name it like you'd search for it

The skill's name is part of discoverability too. Name it the way someone would describe the job, `cut-release` rather than `rel-mgr-v2`. Clear, specific, job-shaped names help both the agent and the human maintaining the library.

Get the description right and the agent opens the skill at the right time. Now it needs to actually succeed, which is about the body.

Writing a Skill That Works

The description gets the skill opened. The body is what makes it succeed. And the body has its own craft, which is mostly about hitting the right altitude: detailed enough to be reliable, loose enough not to fight the model's own competence.

Write steps, not an essay

The body is a procedure, so write it like one. Clear, ordered steps beat flowing paragraphs, because the agent is going to execute them, not admire them. Where order matters, number the steps. Where a thing must be checked, make it a checklist item. Where there's a decision, state the rule for resolving it.

The release skill says "update the changelog, then bump the version, then build, then tag, then announce," because that order is load-bearing. Spell out what's load-bearing and leave the rest to the model.

Find the right altitude

This is the judgment call. Too vague, and the skill adds nothing the model didn't already know ("review the code carefully"). Too rigid, and you've over-specified things the model handles fine on its own, making the skill brittle and bossy. The sweet spot is to be precise about the parts that are *specific to you*, the parts the model couldn't guess, and trust it on the general competence.

A good test: for each instruction, ask "would a smart person who'd never seen our setup know to do this?" If yes, you can probably cut it. If no, keep it, that's exactly the value the skill adds.

Bundle what the steps need

If a step runs a script, ship the script in the folder and point at it. If a step produces a document, include the template. If "good" has a specific shape, include an example of a good output and tell the agent to match it. Showing one concrete example of the target usually does more than three paragraphs describing it.

Make the deterministic parts deterministic

Some steps shouldn't be left to interpretation at all. If a value has to be computed a certain way, or a file has to be formatted exactly, don't describe it in prose and hope, give the agent a script to run and have it use the output. Push the parts that must be exact into code, and let the prose handle the judgment. That mix, deterministic where it counts and flexible where it helps, is what separates a skill that mostly works from one you can rely on.

With a skill that triggers and works, the natural next question is how it relates to the other things in your agent's head. When is something a skill, versus a memory, versus a tool?

Skills, Memory, and Tools

Once you have all three pillars plus a tool layer, things start blurring at the edges. People cram facts into skills, procedures into memory, and capabilities into prompts, and the whole thing gets muddy. A few clean lines keep it sharp.

Three questions, three homes

Each layer answers a different question, and the question tells you where a thing belongs.

- **A tool** answers "what *can* I do?" It's a capability: run a command, query the database, send an email. Raw ability, invoked.
- **A memory** answers "what do I *know*?" It's a fact that accumulates: this repo uses Bun, the last deploy failed on a missing env var, the user likes short replies.
- **A skill** answers "how do I *do* this job?" It's a procedure: the ordered, specific method for accomplishing a particular task.

When you're not sure where something goes, ask which question it answers. A fact is a memory. A method is a skill. A capability is a tool.

The blur cases

A couple of cases genuinely sit on the line, and they're worth thinking through.

"Our deploy command is `bun run deploy:prod`" feels like a fact, so memory. But "how we deploy to production" (pre-checks, the command, the smoke test, the rollback plan) is a procedure, so a skill. The single fact can live in memory; the method is a skill that might *reference* that fact.

"Search the codebase" is a tool (a capability the agent calls). But "how we investigate a production incident," which uses search along with reading logs and checking recent commits in a particular order, is a skill that orchestrates tools. The tool is a verb; the skill is the play that strings verbs together.

Why keeping them separate pays off

It's tempting to not care and dump everything into one big instructions file. The reason to resist is that each layer has different rules. Memory grows and decays and gets retrieved by relevance. Skills are versioned like code and loaded by progressive disclosure. Tools are gated and invoked. Mix them and you lose the properties that make each one work: your "memory" stops decaying, your "skill" stops being discoverable, your procedures bloat the always-loaded context.

Keep facts in memory, methods in skills, capabilities in tools, and each layer can do its job well. Speaking of versioning like code, that's the next thing skills demand, because a skill you never revise slowly goes stale.

Versioning and Evolving Skills

A skill is code, near enough. It's a piece of behavior you wrote down, it'll be wrong sometimes, and it'll need to change as the thing it describes changes. So treat it the way you treat code: keep it in version control, improve it from real use, and review the changes. The alternative is a skill that slowly drifts out of sync with reality and quietly starts doing the wrong thing.

Keep skills in git

This falls out of skills being files. Put the skill folders in a repo and you get history, diffs, blame, and review for free. When a skill changes, you can see exactly what changed and why. When a new version makes the agent behave worse, you can roll it back. When a teammate edits a shared skill, it goes through a pull request like anything else. None of this is exotic; it's just refusing to treat skills as a special snowflake exempt from the practices that keep code sane.

Improve from real runs

The best source of skill improvements is watching the skill actually run. The agent skipped a step, or misread an instruction, or produced output in slightly the wrong shape. Each of those is a concrete edit: tighten the wording, add the missing step, include an example of the right shape. Over a few rounds this is how a skill goes from "works most of the time" to "works," and it's a much faster path than trying to write the perfect skill up front. Ship a decent first version, then let reality sand it down.

Let the agent help

Capable agents are surprisingly good at maintaining their own skills. An agent that just struggled through a procedure can often tell you precisely where the skill was unclear, and draft the fix. Some setups let the agent update a skill on the fly when it learns something mid-task. That's powerful, and it earns the same caveat as letting memory rewrite identity: review the changes. Agent-proposed edits are a great first draft, not an unreviewed merge to your procedures.

Don't let them rot

The failure that creeps up on you is the stale skill: the release process changed, but the `cut-release` skill still describes the old one, and now the agent confidently does it the wrong way. Stale skills are worse than missing ones, because the agent trusts them. Prune and update deliberately. When a process changes, changing its skill is part of the change, not a someday cleanup. A small library of current skills beats a big library where you're not sure which ones still tell the truth.

Versioning matters even more once skills are shared, because now your edits ripple out to everyone using them. That's the move from one agent to a fleet.

Skills Across a Fleet

One agent with a handful of skills is useful. Where skills really start paying off is when they're shared: a library that many agents draw from, and packaged units you can hand to your whole team. This is the same territory as Running the Fleet (/guides/running-the-fleet), seen from the skills angle.

A skill library

The natural next step from a few skills is a library: a collection of skill folders that any of your agents can pull from. Because skills are self-contained folders, this is mostly an organizational question, not a technical one. Keep them in one place, name them well, and let agents discover the relevant ones through their descriptions (progressive disclosure does the work at any scale).

The library becomes a real asset over time. It's the written-down, runnable version of how your team does things, and every skill you add makes every agent that bit more capable without any of them getting heavier, since they only load what they use.

Shared versus per-agent

Not every skill belongs to everyone. Some are genuinely shared: "how we cut a release" is the same procedure whoever's running it. Others are specific to one agent's role: a reviewer agent has skills a writer agent has no use for. The clean arrangement mirrors the one for identity. A shared core library that every agent can reach, plus per-agent skills that belong to a particular specialist. Shared skills keep the fleet consistent; per-agent skills keep each one good at its actual job.

Skills as distributable units

Because a skill is just a folder, it's portable in a way that's genuinely useful: you can package a skill (or a whole library) as a repository and hand it to someone, and they get the exact procedure you built, scripts and templates and all. This is how good practice spreads. One person works out the right way to do a fiddly job, writes it as a skill, and now everyone's agents can do it that way. It's the same idea as packaging an agent's whole identity as a distributable profile, applied to know-how instead of character.

Mind the shared blast radius

The flip side of sharing is that an edit to a shared skill changes behavior for everyone using it. That raises the stakes on the versioning from the last chapter. Review changes to shared skills carefully, because a careless edit doesn't just affect your agent, it ripples across the fleet. The discipline scales with the reach. A skill only you use, you can edit casually; a skill the whole team depends on deserves the same care as shared code.

That's the whole arc, from one skill in a folder to a library powering a fleet. The last chapter covers where it goes wrong, and how to check before you trust it.

Pitfalls and a Checklist

Skills are leverage, and like any leverage they can amplify a mistake as easily as a good idea. Here are the ways they go wrong, and a checklist to run before you lean on one.

The skill that never fires

The most common disappointment: you write a careful, detailed skill, and the agent never uses it. Almost always the body is fine and the description is the problem, too vague or too narrow for the agent to recognize the moment. If a skill isn't earning its keep, look at the description first. The instructions inside don't matter if the door never opens.

The skill that fights the model

The opposite failure: an over-specified skill that micromanages things the model already does well, making it rigid and worse at the job than it would've been with no skill at all. Skills should add your specific knowledge, not re-teach general competence. If a skill is mostly telling the model how to do things any capable model already knows, trim it down to the parts that are actually yours.

Skill bloat

Left unmanaged, a skill library grows and grows, and a sprawling pile of overlapping, half-current skills is harder to use than a small sharp set. More skills is not the goal. Consolidate overlapping ones, delete the dead ones, and keep each skill narrow. The library should feel curated, not accumulated.

The stale skill

Worth repeating because it's so quiet: a skill that describes a process that's since changed will have the agent confidently doing the old, wrong thing. Updating a skill is part of changing the process it describes, not a cleanup task for later.

Wrong layer

Sometimes the thing you wrote as a skill should've been a tool (a capability you keep re-describing in prose) or a memory (a fact you embedded in a procedure). If a "skill" is really one fact, move it to memory. If it's really a capability you invoke, it may want to be a tool. Right knowledge, wrong home.

The pre-trust checklist

Before you rely on a skill:

- **It triggers.** You've watched the agent actually reach for it on a real, naturally-phrased task, not just confirmed it exists.
- **The description says when.** It names the situations and phrasings that should fire it, and ideally what it's not for.
- **The body is the right altitude.** Precise on what's specific to you, quiet on what the model already knows.
- **It's self-contained.** Scripts, templates, and examples the steps need ship in the folder.
- **Deterministic parts are code.** Anything that must be exact is a script the agent runs, not prose it interprets.

- **It's one job.** The skill does one thing; it isn't three skills in a trench coat.
- **It's versioned and current.** In git, reviewed, and matching the process it describes today.

The payoff

Get this right and your agent stops being a clever generalist that improvises every job and becomes a specialist that does *your* jobs *your* way, on demand, the same every time. Paired with a memory (/guides/agent-memory-field-guide) that compounds and an identity (/guides/agent-self-personality-identity) that stays consistent, skills are the third pillar that turns a capable model into a reliable teammate. That's the whole point of the trilogy: who it is, what it knows, and now, how it works.