# Authoring Agent Skills: A Software-Engineering Approach

Giuseppe Destefanis
Department of Computer Science, University College London
g.destefanis@ucl.ac.uk

May 2026

**Abstract**

Agent Skills are an emerging way to extend large language model agents with reusable procedural knowledge that the agent loads on demand. Anthropic introduced Agent Skills and published the format as an open specification supported across several agent tools. This note argues that a skill is a software artefact and that its construction should follow software-engineering principles, with qualifications: single responsibility, separation of interface from implementation, low coupling, and economy in a shared token budget, together with behavioural evaluation in place of deterministic testing. Using Claude Code as the reference implementation, it describes how a skill is structured, how its contents are loaded in stages, and how to write the description on which selection depends. It places skills against the other mechanisms a developer can use to shape agent behaviour, like project memory files, slash commands, subagents, external tool connections, and hooks, and gives a rule for choosing between them based on who decides that a mechanism runs and what guarantee it provides. It then sets out an evaluation-driven authoring process, a set of patterns and faults commonly encountered in authoring, and the trust question raised by using skills from third parties. We illustrate the comparison drawn in UML class style, the loading model, the anatomy of a skill, the relative position of each mechanism, and the points at which skills and hooks act during a session.

## Note on sources

This note draws on Anthropic's official documentation[1] for Agent Skills, Claude Code skills, hooks, and skill authoring best practices, together with the open Agent Skills specification.

## 1 Introduction

A skill packages a workflow, a set of conventions, domain facts, or a sequence of steps that an agent should follow for a particular kind of task, so that the agent behaves as a specialist without the developer repeating the same guidance in every session. A characteristic property of a skill is that the agent can decide when to apply it, by matching the description, and that decision is probabilistic. There is no compiler or type system confirming that the right skill fired. Authoring choices therefore determine whether a skill is selected at all and whether its instructions are followed once loaded.

Anthropic introduced Agent Skills and published them as a portable format. The directory structure, the YAML frontmatter, and the staged loading model are an open specification supported across a range of agent tools [1]. A skill written against the core format can usually be read by another tool that supports the specification, though tool-specific fields and behaviours may not transfer. This note uses Claude Code as the reference implementation, because it exposes the

[1] https://www.anthropic.com/

surrounding mechanisms in full, but the structure of a skill and the authoring principles apply wherever the format is supported.

**A skill is a software artefact**, and the argument of this note rests on that claim, so it is worth stating the grounds. A skill is made of ordinary files: a `SKILL.md` file of instructions, plus any scripts and reference documents it bundles. These files can be kept under version control, like the rest of a project's code. It has an interface, the description, and an implementation, the body and any bundled scripts. It is composed with other units: it bundles resources, calls tools, and can invoke other skills. It is read and executed by a machine, it is maintained as the surrounding system changes, and it can fail, by not being selected or by being followed incorrectly. These are the properties by which software is identified, so the methods used to build software apply to skills, and this note treats authoring as a design activity. Section 2 develops the consequences.

A developer assembling a real project faces a further problem that the per-skill documentation does not address directly. Several mechanisms shape agent behaviour: project memory files, slash commands, subagents, external tool connections, and hooks. They look similar on the surface and differ in what they guarantee. Choosing the wrong one is a common source of unreliable behaviour, because a requirement that must hold every time is written as advisory prose in a place that only sometimes takes effect.

This note has four aims. The first is to argue that software-engineering principles apply to skills, and to set out the comparison drawn in UML class style and its limits. The second is to set out the structure of a skill, its staged loading model, and the way a description governs selection. The third is to position skills against the other mechanisms on the same surface and give a rule for choosing between them. The fourth is to describe an authoring process based on evaluation and iteration, together with common patterns and faults. The examples use Claude Code, and the diagrams use a generic skill that does not describe any specific deployment.

## 2 Skills as software artefacts

The introduction set out the grounds for treating a skill as a software artefact: ordinary files that can be versioned, an interface and an implementation, composition with other units, and the ways it can fail. Many of the disciplines that apply to ordinary software therefore apply to skills, some directly and some in a modified form, and this note treats authoring as a design activity.

Several principles transfer, though each for a reason specific to how skills work rather than by analogy alone. A skill should have a single responsibility, one coherent capability, because the description that advertises it is matched against a task: a skill scoped to one class of task is selected more reliably, while one that does several things has a diffuse description that matches less well. The separation of interface from implementation is built into the format. The metadata that advertises a skill, its name and description, is the only part the selector reads, and the body is loaded only after selection, so the separation of interface and implementation is one of timing as well as principle. Cohesion applies in its usual sense: a skill, and each reference file within it, groups content that serves a single concern, which keeps each unit focused and, for a skill, sharpens the description that selects it. Coupling applies between skills: a skill that relies on the internal behaviour or output format of another skill, or on a shared resource, is coupled to it, so that a change to one can break the other. Such dependencies are kept few and made explicit, shown as composition and a use relation in Figure 1, which is low coupling in the usual sense of limiting how far a change propagates. The context window is a shared resource with a fixed budget, so a skill is written to spend tokens only where they earn their place, which is resource economy under a hard limit. The name is part of the metadata the selector matches, so a consistent, descriptive name aids selection as well as readability.

Testing transfers only in a qualified form, and it is worth being exact about why. A skill cannot be unit-tested the way a function can, because it has no behaviour in isolation: it is inert text until a model reads it, so what is exercised is the model together with the skill on a task, which is

integration rather than unit testing. The result is non-deterministic, so a skill is judged by a pass rate over several runs rather than a single pass or fail, and most outputs have no exact expected value, so the check is a rubric or a judge rather than an equality assertion. This is behavioural evaluation, closer to evaluating a machine-learning component than to deterministic testing, and Section 9 sets out the process. The exception is a bundled script, which is ordinary code and is unit-tested in the ordinary way.

Figure 1 draws a skill in UML class style. The top compartment holds the skill name. The interface compartment is the description, the contract a caller reads to decide whether to use the skill. The implementation compartment is the body together with the bundled files. Composition links the skill to the resources it bundles, and a dependency links it to the external tools or other skills it calls.

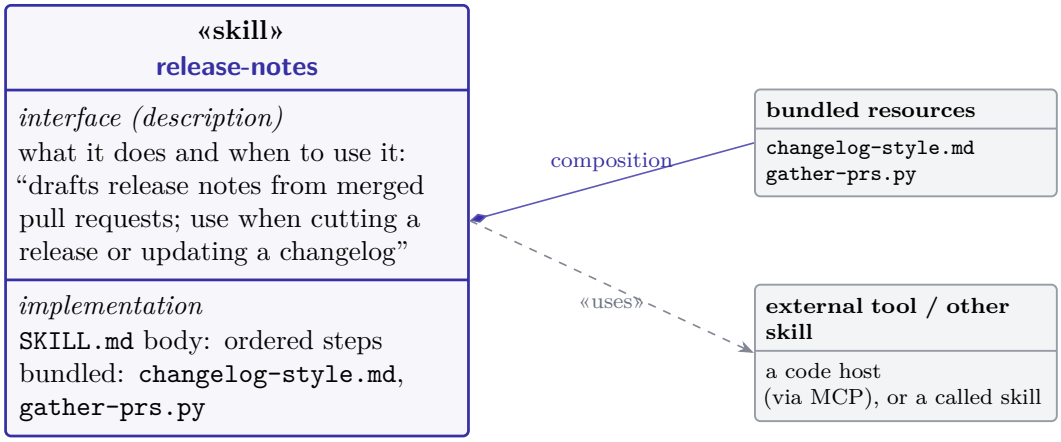


**Figure 1:** A skill drawn in UML class style. The name, interface, and implementation map onto the skill name, the description, and the body with its bundled files. The notation captures structure only: a skill is module-like with a static form, and the model selects it by matching its description, where a method call is dispatched deterministically on its signature.

The comparison is a guide to structure. A skill is module-like in behaviour: a single static unit, like a class that holds only static members, and each skill is independent of the others. Its invocation is probabilistic: the model selects a skill at runtime by matching its description, while a method call is dispatched deterministically on its signature. The static form and the description-based selection are what an author should keep in mind when reading the diagram.

That last difference is the strongest reason to take the design seriously. In conventional code a type checker and a linker catch a mis-wired call before it runs. A skill has no such guarantee: a poorly named or poorly scoped skill fails quietly, either by not being selected or by being selected and then ignored. The contract and the evaluations, that is the description and the behavioural tests, carry the weight that a type system would otherwise carry. Clear interfaces, low coupling, and executable evaluations matter more here than in code a compiler can check.

Where a skill governs a task, the agent's behaviour on that task turns substantially on the quality of the skill. A skill that is selected reliably and followed faithfully raises the floor on the agent's behaviour for that task; one that is not lowers it, often without a visible error. Skill quality is therefore a determinant of system behaviour, which is the reason to architect skills with the care given to the modules of a conventional system. The rest of this note sets out that structure in detail.

## 3 The skill as a unit

A skill is a directory on the filesystem that contains a single `SKILL.md` file and, optionally, supporting files such as reference documents, scripts, and templates. The `SKILL.md` file has two parts: YAML frontmatter holding a `name` and a `description`, and a markdown body holding the

instructions. A minimal skill is one `SKILL.md` with no other files. A larger skill bundles reference material and code alongside it [2]. In Claude Code, the reference implementation used throughout this note, skills live in `~/.claude/skills/` for personal use, or in `.claude/skills/` inside a repository for project-scoped use that travels with the codebase [5].

The frontmatter has two required fields with fixed constraints. The `name` may be at most 64 characters, may contain only lowercase letters, numbers, and hyphens, may not contain XML tags, and may not contain the reserved words `anthropic` or `claude`. The `description` must be non-empty, at most 1024 characters, and free of XML tags [2].

## 4 Staged loading

Skills use staged loading, described in the documentation as progressive disclosure [2]. The mechanism rests on three levels, shown in Figure 2.

The first level is the frontmatter. The `name` and `description` of every installed skill are made available to the model at the start of a session, at a cost of roughly one hundred tokens per skill [2]. This is what lets a project hold many skills without paying for all of their contents up front. At this stage the model knows only that the skill exists and what it claims to be for.

The second level is the body. When the model judges that a task matches a skill's description, it reads `SKILL.md` from disk and the body enters the context window. The body should stay under about five hundred lines, a few thousand tokens, so that once loaded it does not crowd out the conversation and other context [3].

The third level is the bundled files. Reference documents are read only when the body points to them and the task needs them. Scripts are normally run rather than read into context: executing a script keeps its source out of the context window and brings in only its output. A script can instead be read as reference where its internal logic matters, in which case it counts as a reference file. There is no token cost for bundled material until it is accessed, which means a skill can carry large reference files or datasets without penalty as long as they remain unread [2].

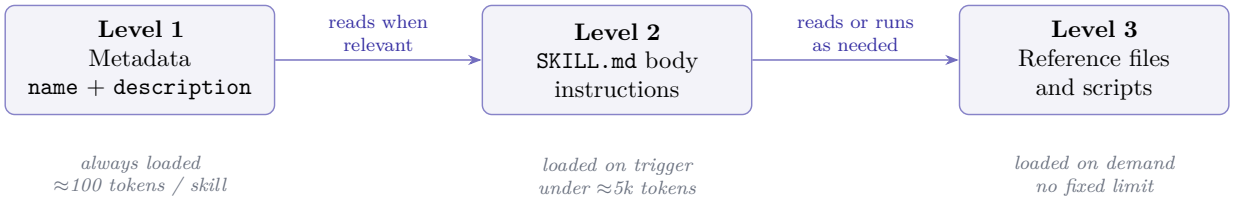


**Figure 2:** Staged loading. Only the metadata is resident at all times. The body enters context when the model selects the skill, and bundled files are read or executed only when the task requires them.

A skill has two invocation paths. Under automatic invocation the model decides whether to run a skill, by matching the description, and that decision is probabilistic. A skill can also be invoked directly by name, which runs it deterministically at the user's choice. The description carries a double load on the automatic path: it is both the documentation a reader sees and the trigger condition the model matches against.

## 5 Writing the description

Because the description is the trigger, it deserves more care than any other part of the skill. Write it in the third person, since it is read by the model as part of its own context, and a first or second person voice can degrade matching. State both what the skill does and when it should be used, and include the concrete terms a relevant task would contain.

A description such as "Handles releases" gives the model nothing to match against. A description such as "Drafts release notes from the pull requests merged between two version tags.

Use when cutting a release, updating a changelog, or summarising what changed in a version" names the operations and the triggers, so the model can select it against a real request [3].

**Explicit invocation.** If a skill should run only when called by name, Claude Code allows the frontmatter to disable automatic invocation, which turns the skill into an explicit command [5]. Use this for actions that should never fire on the model's own judgement.

## 6 Anatomy of a well-formed skill

Figure 3 shows the parts of a skill using a generic release-notes example. Inputs that the skill reads are drawn as chips on the left, the numbered steps of the body sit inside the container, and the artefacts the skill produces are chips on the right. The same convention serves any skill: the container is the unit of behaviour, the chips on either side are what it consumes and produces, and the inner boxes are the procedure the body encodes.

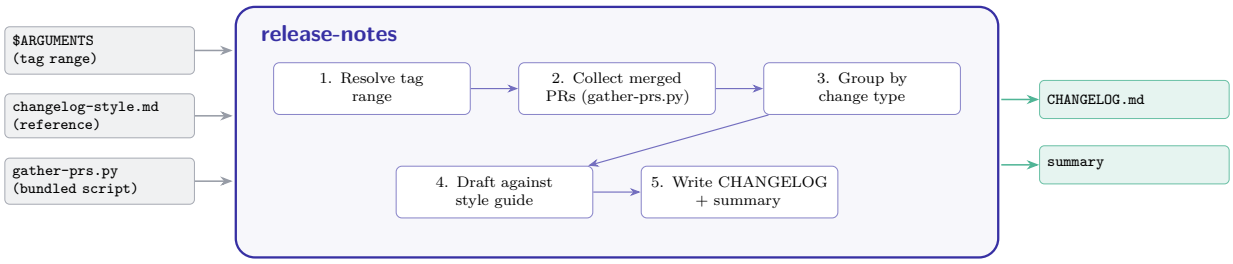


**Figure 3:** Anatomy of a skill (generic example). Grey chips are inputs the skill reads, including reference files and bundled scripts; the numbered boxes are the steps encoded in the body; green chips are the artefacts produced. When a bundled script is executed rather than read, its source does not enter context.

The body should be short and oriented to the task it serves. Assume the model already holds general knowledge, and add only what it does not have: project conventions, non-obvious procedures, specific file locations, rules that must hold. Every sentence competes for context once the body is loaded, so remove explanations of things the model already understands.

**References stay one level deep.** The body may link to reference files, but those files should not chain onward to further files, because the model may preview a deeply linked file with a partial read and then act on incomplete information. Link every reference file directly from `SKILL.md`. For any reference file longer than about one hundred lines, place a short table of contents at the top so the model can see the full scope even on a partial read [3].

**Conventions.** Use forward slashes in all paths so they work across operating systems. Use one term for each concept throughout the skill rather than alternating between synonyms, since consistent wording is easier for the model to follow. Avoid time-sensitive statements that will date; if older behaviour must be recorded, place it in a clearly marked section for legacy patterns rather than in the main flow [3].

**Degrees of freedom.** Match the level of constraint to the fragility of the task. Where several approaches are valid and the right one depends on context, give general direction and let the model choose. Where a sequence is fragile and must run exactly, give the precise command and state that it must not be altered [3].

## 7 Skills within the agent tool surface

Skills are one of several mechanisms that change how an agent behaves. They are easy to confuse, and choosing the wrong one is a common source of unreliable behaviour. Two questions separate them: who decides that the mechanism runs, and what guarantee it provides. Table 1 sets out the mechanisms against these two questions, as Claude Code implements them. Analogous mechanisms exist in other tools: the external tool connection follows the cross-vendor Model Context Protocol (MCP), and the project memory file has a cross-tool analogue in the `AGENTS.md` convention.

| Mechanism | Who decides it runs | What it provides | Reach for it when |
|---|---|---|---|
| Memory file (`CLAUDE.md`) | The runtime, always | Standing project context, present every turn | The instruction is short, general, and should stay in view at all times |
| Skill | The model, by matching the description; or the user, by name | Procedural knowledge and bundled resources, loaded in stages | The task needs domain procedure or reference material the model should find on its own |
| Slash command | The user, by typing `/name` | A saved prompt run on demand | You want to decide exactly when a prompt runs |
| Subagent | The model or the user, by delegating | A separate instance with its own context window and tools | Work should run in isolation to keep the main context focused |
| External tool (MCP) | The model, by calling the tool | A connection to an external service or data source | The model must act in the outside world, on a repository, a database, or a chat system |
| Hook | The runtime, on a lifecycle event | Deterministic execution that can block an action | A step must happen every time, or an action must be prevented |
| Plugin | Installed by the user | A bundle of the above for distribution | You are packaging capabilities to share across a team |

**Table 1:** The mechanisms that shape agent behaviour, separated by who decides that each runs and what it guarantees.

**Overlap between commands and skills.** In current Claude Code, slash commands and skills have converged: both can be invoked as `/name`, and files in `.claude/commands/` continue to work [5]. The practical reading is that a slash command and a skill sit on one continuum. The command end is a single prompt file triggered by name. The skill end adds automatic invocation by description, staged loading, and bundled files. Choose the command end to decide when it runs, and the skill end to let the model recognise the moment or to carry supporting files and scripts.

Figure 4 places the mechanisms on two axes: who decides the run, the model, the user, or the runtime, and how strong the effect is, from advisory to blocking. The two tend to track together, because only runtime-decided mechanisms can be made deterministic. A skill sits in the advisory, model-decided region, and a hook in the blocking, runtime-guaranteed region. A memory file is the exception, always loaded yet advisory. The positions are indicative rather than precise, but they show the gap that matters in practice.

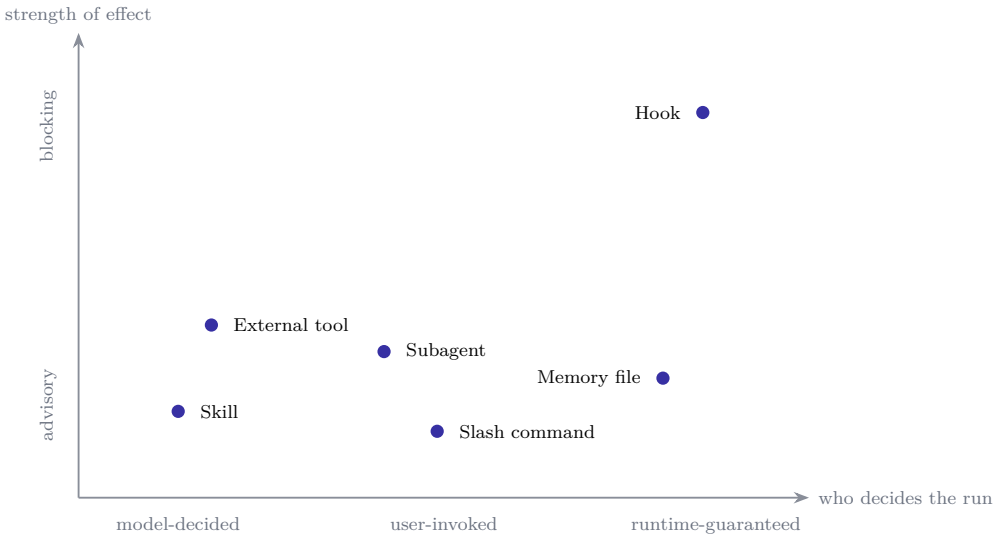


**Figure 4:** Schematic positioning of the mechanisms by who decides the run, the model, the user, or the runtime, and how strong the effect is, from advisory to blocking. The two tend to track together, since only runtime-decided mechanisms can be made deterministic: a skill sits low and left, a hook high and right. A memory file is the exception, always loaded by the runtime yet only advisory, so it sits low on the right. Among these mechanisms only a hook can block an action. Positions are indicative.

## 8 Skills and hooks

This pairing deserves separate treatment, because it is where guarantees are most often misplaced. On its automatic path a skill is model-invoked and probabilistic; a hook is runtime-invoked and deterministic. A hook is a shell command, or a prompt or agent handler among others, registered against a lifecycle event in `.claude/settings.json`. It fires every time its event and matcher conditions are met, whatever the model decides. These events occur at fixed points that the runtime controls: at session start, around each tool call, and when the model finishes responding, among others. A tool call is any action the agent takes through a tool, such as reading or writing a file, running a shell command, or querying an external service. The `PreToolUse` event fires immediately before such an action runs, while the agent waits, so a hook on it can inspect the action and block it. The `PostToolUse` event fires immediately after the action completes, so a hook on it can read the result. A `PreToolUse` hook blocks the pending action either by exiting with code 2 or by returning a deny decision in its output, which makes it the place to stop a dangerous command or protect a file [4].

Figure 5 shows where each acts during a session. A skill is read inside the model's reasoning, at a point the model chooses. A hook fires at fixed events that the runtime controls, around every tool call and at the session and turn boundaries.

The decision rule follows directly. If a step depends on judgement, varies with context, or encodes domain procedure, put it in a skill. If a step must happen every time the triggering event occurs, put it in a hook. A standing instruction in a skill body, however firmly worded, is something the model may read, defer, or skip. The same instruction expressed as a hook runs unconditionally. When a guarantee matters, for example loading a specification file at session start, running a validator before a commit, or formatting after every file write, write it as a hook and do not rely on skill prose to enforce it.

## 9 An evaluation-driven authoring process

Write evaluations before writing the skill. Run the model on representative tasks without the skill and record where it fails or lacks context. Turn those failures into a small set of test cases with

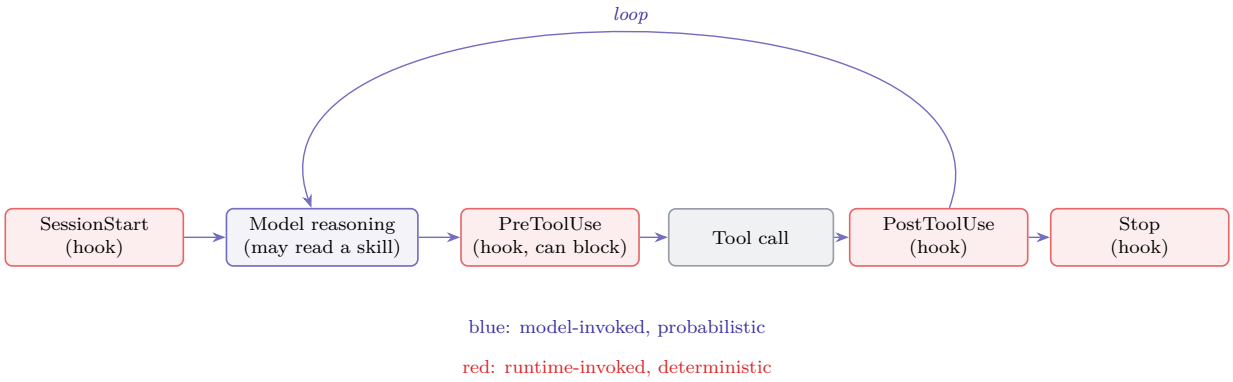


**Figure 5:** Where skills and hooks act during a session. The model reads a skill at a point of its own choosing inside its reasoning. Hooks fire at fixed lifecycle events controlled by the runtime: `SessionStart` at the beginning of a session, `PreToolUse` and `PostToolUse` immediately before and after each tool call, with `PreToolUse` able to block the call, and `Stop` when the model finishes responding.

expected behaviours. Measure the baseline without the skill, then write the minimum instructions needed to pass the tests, and iterate against them. This keeps the skill aimed at real gaps rather than imagined ones [3].

A reliable development loop uses two instances of the model [3]. Work with one instance to draft and refine the skill, drawing on the context that would otherwise be typed by hand. Test the result with a fresh instance that has only the skill loaded, and observe its behaviour on real tasks: whether it triggers when expected, follows the references, and applies the rules. Bring specific observations back to the drafting instance and revise. The frontmatter is the first thing to check when a skill fails to trigger, since the model selects on the name and description.

**Observe how the skill is read.** If the model reads files in an order that was not expected, the structure may be less clear than assumed. If it ignores a bundled file, that file may be unnecessary or poorly signalled. If it returns to the same file repeatedly, that content may belong in the body. Test with each model intended for use, since a body that suits a stronger model may need more detail for a faster one.

## 10 Patterns and faults

For multi-step tasks, give the body an explicit ordered workflow, and for fragile sequences include a checklist the model can copy and tick off as it proceeds. Where output quality depends on validation, build a loop: run a check, fix what it reports, run it again, and proceed only when it passes. For batch or destructive operations, have the model write a plan to a structured file and validate that file with a script before any change is applied, so that errors are caught before they take effect. Where output format matters, provide a template; where style matters, provide worked input and output examples rather than describing the style in the abstract.

Several patterns commonly cause trouble, drawn from the cited best practices and from common experience. Offering many alternatives leaves the model to choose without grounds, so give one default and name the exception. Windows-style backslash paths break on other systems. Deeply nested references lead to partial reads. Assuming a package is installed fails when it is not, so state dependencies explicitly. In bundled scripts, handle error conditions rather than failing and leaving the model to recover, and justify every constant rather than leaving unexplained values.

## 11 Trust and third-party skills

A skill should be treated as a software dependency. It can carry instructions, scripts, references, and links to external content, and the agent acts on those materials with whatever permissions it holds in the current environment. A skill that fetches external content is a particular concern, since that content can itself carry instructions and can change after the skill was first trusted. So a skill from a third party should be inspected before use: read the `SKILL.md`, the bundled scripts, the referenced resources, and any external URLs, and check each against the skill's stated purpose. This matters most when the skill can read local files, call tools, or send data outside the project [2].

## 12 Summary of guidance

A skill is a software artefact, and the guidance above is an application of ordinary engineering discipline to it. The comparison drawn in UML class style holds for structure, while invocation works differently: the model selects a skill probabilistically, so evaluations carry the weight a type system would in conventional code. A skill is selected by the model on the strength of its description and loaded in stages, so the description must state both function and trigger, and the body must stay short and assume general knowledge. Reference files are linked one level deep, and scripts are bundled for execution rather than for reading. The choice between a skill and the other mechanisms turns on two questions: who should decide that it runs, and how strong the guarantee must be. Judgement-dependent procedure belongs in a skill; a requirement that must hold every time belongs in a hook. Authoring proceeds by writing evaluations first, then the minimum instructions needed to pass them, and refining against observed behaviour. Because a skill may contain instructions and code that the agent can act on, a skill from a third party should be read in full before it is used.

## References


[1] Agent Skills. *Agent Skills Specification* (open standard, Apache-2.0). `https://agentskills.io`

[2] Anthropic. *Agent Skills*. Claude API documentation. `https://platform.claude.com/docs/en/agents-and-tools/agent-skills/overview`

[3] Anthropic. *Skill authoring best practices*. Claude API documentation. `https://platform.claude.com/docs/en/agents-and-tools/agent-skills/best-practices`

[4] Anthropic. *Hooks reference*. Claude Code documentation. `https://code.claude.com/docs/en/hooks`

[5] Anthropic. *Skills in Claude Code*. Claude Code documentation. `https://code.claude.com/docs/en/skills`