How to add skill support to agents

This article will guide you on how to add Agent Skills support to an AI agent or development tool.

This article will guide you on adding Agent Skills support to an AI agent or development tool. It covers the entire lifecycle: discovering the skill, notifying the model about that skill, loading the skill content into context, and maintaining its effectiveness over time.

The core integration process is the same regardless of the agent's architecture. The implementation details differ based on two factors:

1. **Where are skills stored?** A locally running agent can scan the user's file system for the skills directory. An agent hosted in the cloud or in a sandbox environment would require an alternative discovery mechanism—an API, remote registry, or packaged assets.
2. **How does the model access skill content?** If the model is capable of reading files, it can directly read the SKILL.md files. Otherwise, you will provide a dedicated tool or programmatically insert the skill content into the prompt.

The guide will highlight where these differences are important. You don't need to support every case – do what works best for your agent.

Prerequisite : Thorough understanding of the Agent Skills specification , which defines the SKILL.md file format, header fields, and directory conventions.

Core principle: Disclosure in stages.

All agents compatible with the skill follow the same 3-level information loading strategy:

Level	What has been loaded	Time	Token cost
1. Categories	Name + Description	Start of session	~50-100 tokens per skill
2. Instructions	The entire content of SKILL.md	When the skill is activated	
3. Resources	Script, reference document, asset	When the instructions mention them	Depending on the specific case

The model sees the catalog from the start, so it knows which skills are available. When it decides a skill is suitable, it loads the entire tutorial. If those tutorials reference supporting files, the model loads them

individually as needed.

This helps keep the underlying context compact while still providing the model with access to specialized knowledge on demand. An agent with 20 skills installed doesn't pay token costs for 20 complete instruction sets upfront—only paying for the instruction sets actually used in a given conversation.

Step 1: Discover your skills

When starting a session, find all available skills and load their metadata.

Areas that need scanning

Which folders you scan depends on the agent's environment. Most locally running agents scan at least two ranges:

1. **Project level** (relative to the working directory): Specific skills for a project or repository.
2. **User level** (relative to the home directory): The skills available across all projects for a given user.

Other scopes are also possible—for example, organization-wide skills deployed by administrators or skills bundled with the agent itself. The appropriate set of scopes depends on the agent's deployment model. Within each scope, consider scanning both the client-specific directory and the .agents/skills/ convention:

Scope	Path	Purpose
Project	`/.  /skills/`	Your client's original location
Project	`/.agents/skills/`	Interoperability between clients
User	`~/.  /skills/`	Your client's original location
User	`~/.agents/skills/`	Interoperability between clients

The .agents/skills/ directory has become a widely adopted convention for sharing skills between clients. While the Agent Skills specification doesn't define the location of the skill directory (it only defines its contents), scanning .agents/skills/ means that skills installed by other compliant clients will automatically be visible to your client, and vice versa.

Some implementations also scan the .claude/skills/ directory (both project-level and user-level) to ensure practical compatibility, as many existing skills are installed there. Other additional locations include parent directories down to the git root directory (useful for monorepos), XDG configuration directories, and user-configured paths.

What needs to be scanned?

In each skill folder, find subfolders containing a file with the exact name SKILL.md:

```
~/.agents/skills/ ??? pdf-processing/ ? ??? SKILL.md ? ???c phát hi?n ? ???
 scripts/ ? ??? extract.py ??? data-analysis/ ? ??? SKILL.md ? ???c phát hi
```

```
?n ??? README.md ? b? b? qua (không ph?i là th? m?c skill)
```

The actual scanning rules:

1. Ignore directories that don't contain skills, such as .git/ and node_modules/.
2. The .gitignore compliance option avoids scanning built artifacts.
3. Set reasonable limits (e.g., maximum depth of 4-6 levels, maximum of 2000 folders) to prevent over-scanning in large directory trees.

Name conflict resolution

When two skills have the same name, apply the rule of priority.

The common convention in existing implementations is that project-level skills override user-level skills.

Within the same scope (e.g., two skills named code-review found in both /.agents/skills/ and /./skills/), either the first or last skill found is acceptable—choose one and be consistent. Record a warning when a conflict occurs so users know that a skill has been hidden.

Consider the reliability

Project-level skills originate from the repository being used, which may be untrustworthy (e.g., a newly cloned open-source project). Consider performing trust checks to ensure project-level skills are loaded correctly – only load them if the user has marked the project directory as trusted. This prevents untrustworthy repositories from silently injecting directives into the agent's context.

The agent is hosted in the cloud and in a sandbox environment.

If your agent runs in a container or on a remote server, it will not have access to the user's local file system. The discovery process needs to behave differently depending on the skill scope:

1. Project-level skills are usually the easiest case. If the agent operates on a replicated repository (even within a sandbox environment), project-level skills will be included with the code and can be scanned from the repository's directory tree.
2. User-level and organization-level skills do not exist in a sandbox environment. You need to provide them from an external source—for example, by replicating the configuration repository, accepting skill or package URLs through the agent's settings, or allowing users to load skill directories via a web user interface.
3. Built-in skills can be packaged as static assets within the agent's deployment component, making them available in every session without needing to be loaded from external sources.

Once the skills are available to the agent, the rest of the lifecycle—parsing, revealing, triggering—works similarly.

Step 2: Parse the SKILL.md files.

For each SKILL.md file found, extract the metadata and body content.

Extract the beginning of the file.

A SKILL.md file has two parts: the YAML header located between the delimiters ---, and the markdown body after the closing delimiter. To parse it:

1. Look for the opening ellipsis (--) at the beginning of the file and the closing ellipsis (--) after it.
2. Parse the YAML block located between them. Extract the name and description (required), plus any optional fields.
3. Everything after the closing hyphen ---, when truncated, is the main content of the skill.

Handling YAML formatting errors

Skill files created for other clients may contain technically invalid YAML that their parsers accept. The most common problem is unquoted values ??containing colons:

```
# V? m?t k? thu?t, YAML không h?p l? – d?u hai ch?m gây l?
i khi phân tích cú pháp Mô t?: S? d?ng skill này khi: ng??i dùng h?i v? PDF
```

Consider a workaround such as enclosing those values ??in double quotes or converting them to YAML block scalar values ??before retrying. This improves client-to-client compatibility at minimal cost.

Flexible authentication

Warning about potential issues, but still load skills if possible:

1. Name doesn't match parent folder name ? warning, still loads.
2. Name exceeds 64 characters ? warning, still loading
3. Description is missing or blank ? skip skill (description is necessary to reveal), log error.
4. YAML cannot parse at all ? ignore skill, log error.

Record diagnostics so they can be displayed to the user (in debug commands, log files, or the user interface), but do not block skill loading due to formality issues.

The specification defines strict constraints on the name field (matching to parent directory, character set, maximum length). The flexible approach above intentionally loosens these constraints to improve compatibility with skills created for other clients.

What needs to be stored?

At a minimum, each skill record needs 3 fields:

School	Describe
`name`	From the top of the page
`description`	From the top of the page
`location`	Absolute path to the SKILL.md file

Store this information in a locked memory map `name`for quick retrieval during activation.

You can also store the body (the markdown content after the introduction) at discovery time, or read it from there `location`at trigger time. Storing makes the trigger process faster; reading it at trigger time uses less memory and updates changes to skill files between triggers.

The root directory of the skill (the parent directory of `location`) will be needed later to resolve relative paths and list packaged resources - retrieve it from there `location`when needed.

Step 3: Reveal the available skills for the model.

Tell the model which skills exist without loading their entire contents. This is level 1 of gradual revelation.

Build a skill catalog.

For each skill discovered, include the name, description, and optional location (path to the SKILL.md file) in any structured format that fits your stack—XML, JSON, or bulleted lists all work:

```
 pdf-processing Trích xu?t v?n b?n PDF, ?i?n bi?u m?u, h?p nh?t file. S? d?
ng khi x?
 lý các file PDF. /home/user/.agents/skills/pdf-processing/SKILL.md data-analysis
?p d? li?u, t?o bi?u ?? và l?p báo cáo tóm t?
t. /home/user/project/.agents/skills/data-analysis/SKILL.md
```

The field `location`serves two purposes: It allows the trigger to read the file (see Step 4), and it provides the model with a base path to resolve relative references within the skill body (such as scripts/evaluate.py). If your dedicated trigger provides the skill directory path in its output (see Wrapping Structure in Step 4), you can omit the location from the catalog. Otherwise, include it.

Each skill adds approximately 50-100 tokens to the catalog. Even with dozens of skills installed, the catalog remains compact.

Location of the catalog

There are two common approaches:

System prompt section : Add the category as a labeled section in the system prompt, along with brief instructions on how to use the skill. This is the simplest approach and works with any model that has access to the file reader tool.

Tool description : Embed the category into the description of a dedicated skill trigger tool (see Step 4). This keeps the system prompt clean and naturally blends discovery and triggering.

Both approaches are effective. Placing it in the system prompt is simpler and more widely compatible; embedding it in the tool description is neater when you have a dedicated trigger tool.

Behavioral guidance

Include a short tutorial block alongside the categories, guiding the model on how and when to use the skills. The wording will depend on the trigger mechanism you support (see Step 4):

If the model activates the skill by reading a file:

```
Các skill sau ?ây cung c?p h??ng d?n chuyên bi?t cho các nhi?m v? c? th?
. Khi m?t nhi?m v? phù h?p v?i mô t? c?a m?t skill, hãy s? d?ng công c? ??
c file c?a b?n ?? load file SKILL.md t?i v? trí ???c li?t kê tr??c khi ti?
p t?c. Khi m?t skill tham chi?u ??n các ???ng d?n t??ng ??i, hãy gi?i quy?
t chúng d?a trên th? m?c c?a skill (th? m?c cha c?a SKILL.md) và s? d?
ng các ???ng d?n tuy?t ??i trong nh?ng l?nh g?i công c?.
```

If the model activates skills through a specialized tool:

```
Các skill sau ?ây cung c?p h??ng d?n chuyên bi?t cho các nhi?m v? c? th?
. Khi m?t nhi?m v? phù h?p v?i mô t? c?a m?t skill, hãy g?i công c?
 activate_skill v?i tên c?a skill ?? load toàn b? h??ng d?n c?a nó.
```

Keep these instructions concise. The goal is to let the model know that the skills exist and how to load them – the skill content itself provides detailed instructions after loading.

Filter

Some skills should be excluded from the list. Common reasons:

1. The user has disabled the skill in the settings.
2. The permission system denies access to the skill.
3. The skill was refused activation due to a control model error (e.g., via the disable-model-invocation flag).

Completely hide filtered skills from the catalog instead of listing them and blocking them at activation. This prevents the model from wasting processing turns trying to load skills it cannot use.

When no skills are available

If no skills are found, completely skip the category and behavior guide. Do not display empty blocks or register skill tools without valid options – this will confuse the model.

Step 4: Activate the skill

When the model or user selects a skill, provide full instructions within the conversational context. This is level 2 of gradual revelation.

Model-based activation

Most implementations rely on the model's own judgment as the trigger mechanism, rather than implementing frame-side trigger matching or keyword discovery. The model reads the catalog (from Step 3), decides which skill is appropriate for the current task, and loads it.

Two deployment models:

Activation by reading a file : The model calls its standard file reader tool with the path SKILL.md from the catalog. No special infrastructure is needed - the agent's existing file reading capabilities are sufficient. The model receives the file contents as the tool's output. This is the simplest approach when the model has file access.

Activation using a dedicated tool : Register a tool (e.g., activate_skill) that receives the skill name and returns the content. This is necessary when the model cannot read the file directly and is optional (but useful) even if it can. Advantages over reading the raw file:

1. Control the content that is returned - for example, remove or retain the frontmatter of the YAML (see "What the model receives" section below).
2. Encapsulate content within structured tags for identification during context management.
3. List the packaged resources (e.g., references/*) along with the instructions.
4. Enforce user rights or request user consent.
5. Track triggers for analysis.

If you're using a dedicated trigger tool, limit the name parameter to a set of valid skill names (e.g., as an enum in the tool schema). This prevents the model from generating non-existent skill names. If no skill exists, don't register that tool.

Explicitly activated by the user.

Users can also activate skills directly, without waiting for the model to decide. The most common pattern is the slash command syntax or mention (/skill-name or $skill-name) that the crawler will intercept. The specific syntax is up to you – the main idea is that the crawler will handle the search and insertion, so the model receives the skill content without having to perform the activation action itself. An autocomplete utility (listing available skills as the user types) can also make it easier to find this information.

What the model received

When a skill is activated, the model receives instructions for that skill. There are two options regarding the specific content:

Entire file : The model sees the entire SKILL.md file, including the YAML frontmatter. This is a natural outcome of triggering by reading the file, where the model reads the raw file. This is also a valid option for specialized tools. The frontmatter may contain fields useful at the time of triggering—for example, compatibility notes, environment requirements that may provide information on how the model executes the skill instructions.

Content-only (frontmatter removed) : The tool will parse and remove the YAML frontmatter, returning only the markdown instructions. Among existing implementations with dedicated trigger tools, most use this method—removing the frontmatter after extracting the name and description during discovery.

Both methods work in practice.

Structured packaging

If you're using a dedicated trigger tool, consider packaging skill content within identifying tags. For example:

```
 # PDF Processing ## Khi nào nên s? d?ng skill này Hãy s? d?ng k? n?
ng này khi ng??i dùng c?n làm vi?c v?
```

```
i các file PDF. [rest of SKILL.md body] Skill directory: /home/user/.agents/skil
???ng d?n t??ng ??i trong skill này ???c tính t??ng ??i so v?i th? m?
c skill. scripts/extract.py scripts/merge.py references/pdf-spec-summary.md
```

This offers practical benefits:

1. The model can clearly distinguish skill instructions from other conversational content.
2. The system can identify skill content during context compression (Step 5).
3. Packaged resources are exposed to the model without preloading.

List the packaged resources.

When a dedicated trigger tool returns skill content, it may also list supporting files (scripts, references, assets) in the skill directory—but it shouldn't read them first. The model loads specific files on demand using its file-reading tools when the skill instructions reference them.

For large skill folders, consider limiting the list and note that it may not be exhaustive.

Allow access

If your agent has a file access control permission system, allow access to skill directories so the model can read packaged resources without triggering a user confirmation prompt. Without this, each reference to a script or packaged reference file will result in a permission request dialog, interrupting the workflow for skills that include resources outside of the SKILL.md file itself.

Step 5: Manage skill context over time.

Once the skill instructions have been incorporated into the conversational context, maintain their effectiveness throughout the session.

Protect skill content from contextual compression.

If your agent truncates or summarizes older messages when the context window is full, exclude skill content from truncation. Skill instructions are persistent behavioral guidelines—losing them mid-conversation will silently degrade agent performance without any visible errors. The model continues to function but without the specialized instructions that skills provide.

Common methods:

1. Mark the output of the skill tool as protected so that the clipping algorithm ignores it.
2. Use the structured tags from Step 4 to identify skill content and preserve it during compression.

Remove duplicate triggers

Consider tracking which skills have been activated in the current session. If the model (or user) attempts to load a skill already present in the context, you can skip reinserting it to avoid repeating the same instructions throughout the conversation.

Authorize sub-agents (optional)

This is an advanced model supported only by certain clients. Instead of inserting skill instructions into the main conversation, the skill is run in a separate sub-agent session. The sub-agent receives the skill instructions, performs the task, and returns its task summary to the main conversation.

This model is useful when the workflow for a skill is complex enough to require a dedicated, focused session.