Extending Claude Code ◆

Skills, Subagents, Hooks and Plugins explained properly: what they are, when to use which, and how they combine.

Updated: July 2026 · Source: official Anthropic documentation (code.claude.com/docs) · Companion to the slash commands cheatsheet.

0. The extension map

which tool for which problem

Claude Code combines a model that reasons about code with built-in tools (files, search, execution, web). **Extensions** are the layer above: they customize what Claude knows, connect it to external services and automate workflows. Each one hooks into a different part of the agentic loop:

Feature	What it does	When to use it	Example
`CLAUDE.md`	Persistent context loaded **in every session**, automatically.	Project conventions, "always do X" rules.	"Use pnpm, not npm. Run tests before committing."
`Skill`	Reusable instructions, knowledge and workflows, loaded **on demand**.	Reusable content, reference documentation, repeatable tasks.	`/deploy` runs your deploy checklist; a skill with your API patterns.
`Subagent`	**Isolated** worker with its own context: does the work and returns only the summary.	Context isolation, parallel tasks, specialized workers.	Research that reads dozens of files but reports only the key findings.
`Hook`	Script/HTTP/prompt executed **deterministically** at lifecycle events.	Automations that must always run, without asking.	ESLint after every file edit.
`MCP`	Protocol connecting Claude to **external services** (tools and data).	Data or actions outside the filesystem.	Database queries, Slack posts, browser control.
`Agent team`	Multiple **independent Claude Code sessions that communicate** with each other (experimental).	Complex work requiring discussion and coordination.	Parallel reviewers on security, performance and tests.
`Plugin`	**Packages** skills + agents + hooks + MCP into an installable unit.	Reusing the same setup across repos, or distributing it.	Code-intelligence plugin for your language, from a marketplace.

▶ Build your setup over time: the right trigger for each addition

When this happens...	...add this
Claude gets a convention or command wrong **twice**	Line in `CLAUDE.md`
You keep typing the same prompt to kick off a task	Invocable skill `/name`
You paste the same playbook into chat for the third time	Skill
You copy data from a browser tab Claude can't see	`MCP` server
A side task floods the conversation with output you'll never reread	Subagent
You want something to happen **every time**, without asking	Hook
A second repository needs the same setup	Plugin

1. Skills

the most flexible extension

A skill is a folder with a `SKILL.md` file: YAML frontmatter that tells Claude **when** to use it, and markdown instructions that say **what** to do. You invoke it with `/skill-name`, or Claude loads it on its own when the description matches the task. Unlike CLAUDE.md, a skill's body **loads only when needed**: even lengthy reference material costs almost nothing until it's used.

Historical note: custom commands were merged into skills. A `.claude/commands/deploy.md` file and a `.claude/skills/deploy/SKILL.md` skill both create `/deploy` and work the same way; skills add supporting files, invocation control and automatic loading. They follow the open **Agent Skills** standard, shared across multiple AI tools.

◆ Anatomy and your first skill

```
# Structure of a skill
my-skill/
├── SKILL.md       # instructions (required)
├── template.md    # template to fill in
├── examples/      # sample outputs
└── scripts/       # executable scripts

# ~/.claude/skills/summarize-changes/SKILL.md
---
description: Summarizes uncommitted changes
  and flags risks. Use it when the user asks
  what changed or wants a commit message.
---

## Current changes
!`git diff HEAD`

## Instructions
Summarize the changes in 2-3 points, then list the
risks (missing error handling, hardcoded values,
tests to update). If the diff is empty, say so.
```

The three superpowers in the file alongside

- **Well-written description** → Claude invokes it on its own when you ask "what did I change?", without you typing /summarize-changes.
- **Dynamic injection** `!`command`` → Claude Code runs the command *before* Claude reads the skill and injects its output: the instructions arrive with the real diff already inside, not the one Claude imagines.
- **Supporting files** → templates, examples, scripts and reference docs cited by the SKILL.md, loaded only when needed.

Golden rule: keep the body concise

Once loaded, the skill **stays in context** for subsequent turns: every line is a recurring token cost. Write *what to do*, not narratives about why. Same conciseness test as CLAUDE.md (which itself should stay under ~200 lines: if it grows, move material into skills or `.claude/rules/`).

◆ Where they live and who wins

Level	Path	Applies to
Enterprise	`organization managed settings`	All users in the org (highest priority)
Personal	`~/.claude/skills/<name>/SKILL.md`	All your projects
Project	`.claude/skills/<name>/SKILL.md`	That project only (commit it to the repo!)
Plugin	`<plugin>/skills/<name>/SKILL.md`	Wherever the plugin is enabled, namespaced `plugin:skill`

On a name clash: enterprise > personal > project, and every level **overrides bundled skills** of the same name (a project-level `code-review` replaces the stock one). Skills also load from nested directories: in a monorepo, `packages/frontend/.claude/skills/` activates when Claude works on those files (qualified name: `/packages/frontend:deploy`). Changes on disk are picked up **live**, no restart needed.

◆ Frontmatter: the fields that matter

Field	What it's for
`description`	What the skill does and when to use it. It's the criterion Claude uses to decide to load it on its own: put the main use case first (truncated to 1,536 characters in the listing together with `when_to_use`).
`disable-model-invocation`	`true` = only you can invoke it with /name; Claude never loads it on its own. For workflows with side effects (deploy, commit) you don't want starting autonomously.
`user-invocable`	`false` = hidden from the / menu: background knowledge that makes no sense to invoke by hand.
`allowed-tools / disallowed-tools`	Tools usable without permission prompts while the skill is active / tools removed from the pool (e.g. no `AskUserQuestion` in an autonomous loop).
`model / effort`	Model and effort to use while the skill is active (override for the current turn, then back to session settings).
`context: fork + agent`	Runs the skill in a **forked subagent** instead of inline; `agent` picks which subagent type. The bridge between skills and subagents.
`argument-hint / arguments`	Hint shown in autocomplete (e.g. `[issue-number]`) and positional names for `$name` substitutions.
`paths`	Globs that restrict automatic activation to matching files only (e.g. only when you touch `*.sql`).
`hooks`	Hooks scoped to the skill's lifecycle.

◆ Substitutions available in the body

Variable	Expansion
`$ARGUMENTS`	All arguments passed at invocation (if absent, they're appended as `ARGUMENTS: ...`).
`$0, $1... / $ARGUMENTS[N]`	Single argument by position (0-based).
`$name`	Named argument declared in `arguments:` (e.g. `arguments: [issue, branch]` → `$issue`, `$branch`).
`${CLAUDE_SKILL_DIR}`	The skill's directory: for referencing bundled scripts and files regardless of the working directory.
`${CLAUDE_PROJECT_DIR} / ${CLAUDE_SESSION_ID} / ${CLAUDE_EFFORT}`	Project root, session ID, current effort level.

2. Subagents

isolated workers, clean context

A subagent is a specialized assistant with **its own context window**, a custom system prompt, and independent tools and permissions. The key use case: a side task that would flood the main conversation with searches, logs and file contents you'll never reread. The subagent does that work in its own context and **returns only the summary**. Define a custom subagent when you notice you keep launching the same kind of worker with the same instructions.

Why use them (in 5 points)

- **They preserve context**: exploration and research stay out of the main conversation.
- **They constrain capabilities**: you limit which tools they can use (e.g. read-only).
- **They're reusable**: those in ~/.claude/agents/ apply to all projects.
- **They specialize behavior**: a system prompt focused on one domain.
- **They control costs**: route simple tasks to fast, cheap models like Haiku.

◆ The built-in subagents

Agent	Tools	When Claude uses it
`Explore`	Read-only (Write/Edit denied)	Searching and understanding the codebase without modifying it. Claude specifies the depth: **quick**, **medium** or **very thorough**. Inherits the conversation's model (capped at Opus on the Anthropic API). Skips CLAUDE.md to stay fast and cheap.
`Plan`	Read-only	Codebase research during **plan mode**: exploration stays in a separate window while the main conversation remains read-only.
`general-purpose`	All tools	Complex multi-step tasks requiring both exploration and modification.

◆ Creating one: markdown file + frontmatter

A subagent is a markdown file in `.claude/agents/` (project) or `~/.claude/agents/` (personal). The quickest way: **ask Claude to write it** ("Create a code-improver subagent in ~/.claude/agents/ that... read-only, on Sonnet"). File changes are picked up within seconds, no restart.

```
# ~/.claude/agents/code-reviewer.md
---
name: code-reviewer
description: Expert reviewer. Use it
  proactively after code changes.
tools: Read, Glob, Grep
model: sonnet
---

You are a senior code reviewer. Analyze the
code and give specific, actionable feedback
on quality, security and best practices.
```

The body becomes the subagent's **system prompt**: it receives only that plus basic environment info, not Claude Code's full system prompt. The `description` is what Claude uses to decide when to delegate: write it clearly.

Priority on a name clash

Managed settings > CLI flag `--agents` (JSON, session only) > project `.claude/agents/` > `~/.claude/agents/` > plugins. Directories are scanned recursively: you can organize into subfolders (agents/review/, agents/research/); identity comes only from the `name` field.

Skill vs Subagent, in one sentence

Skill = reusable content you load into a context (adds to your window). **Subagent** = isolated worker that works in a separate window and returns only the summary. And they combine: a subagent can preload skills (`skills:` field), a skill can run in a subagent (`context: fork`).

◆ Frontmatter: the fields that matter

Field	What it's for
`name / description`	The only required fields: identifier (lowercase and hyphens) and delegation criterion.
`tools / disallowedTools`	Tool allowlist / denylist. If omitted, inherits everything. They support MCP patterns: `mcp__github` removes all of that server's tools, `mcp__*` all MCP tools.
`model`	`sonnet`, `opus`, `haiku`, `fable`, a full model ID, or `inherit` (default). The cost trick: research on Haiku, implementation on the big model.
`permissionMode`	`default`, `acceptEdits`, `auto`, `dontAsk`, `bypassPermissions`, `plan`: permissions independent from the parent session.
`skills`	Skills **preloaded** into the subagent's context at startup (full content, not just the description).
`isolation: worktree`	The subagent works in a **temporary git worktree**: an isolated copy of the repo, cleaned up automatically if it makes no changes. Zero conflicts with your working tree.
`background`	`true` = always runs in the background (in recent versions it's the default anyway when possible: you keep working, the result arrives).
`memory`	Persistent memory scope (`user`/`project`/`local`): the subagent **learns across sessions**.
`maxTurns / effort / color / hooks`	Turn limit, reasoning level, transcript color, hooks scoped to its lifecycle.

> **Subagent vs Agent team (experimental)**
> **Subagents** run inside your session and report only to the caller: contained costs, ideal for focused tasks. **Agent teams** are independent Claude Code sessions that message each other, with a shared task list and self-coordination: they're needed when teammates must discuss (competing hypotheses, parallel review, features with separate ownership), but each teammate is a full Claude instance, so it costs. Transition point: if your parallel subagents hit context limits or would need to talk to each other, the natural step is a team.

3. Hooks

determinism: it always happens, not "if Claude remembers"

Hooks are shell commands (or HTTP requests, or prompts) that fire at precise points in the Claude Code lifecycle. The philosophical difference from everything else: **they don't depend on the model's judgment**. A hook guarantees something always happens — formatting after every edit, blocking touches to certain files, notifying when input is needed — instead of hoping the LLM chooses to do it. For decisions that require judgment there are also **prompt-based** and **agent-based** hooks that use a model to evaluate the condition.

They're configured in the `"hooks"` block of a settings file (`~/.claude/settings.json` personal, `.claude/settings.json` project). The `/hooks` menu is read-only: to change them you edit the JSON or ask Claude.

◆ The main events

Event	When it fires
`SessionStart`	On session start or resume (matcher `compact` = after every compaction). The stdout is **added to Claude's context**.
`UserPromptSubmit`	When you submit a prompt, before Claude processes it.
`PreToolUse`	Before a tool executes. **Can block it** (exit code 2).
`PermissionRequest`	When a permission dialog is about to appear: it can answer on your behalf with JSON `{"behavior": "allow"}`.
`PostToolUse / PostToolUseFailure`	After a tool succeeded / failed.
`Notification`	When Claude Code sends a notification (matchers: `permission_prompt`, `idle_prompt`...).
`Stop / SubagentStart / SubagentStop`	When Claude finishes responding / a subagent starts or ends.
`ConfigChange / FileChanged / CwdChanged`	When configuration files, watched files on disk, or the working directory change (e.g. reloading direnv).

All hooks matching an event run **in parallel**; identical commands are deduplicated. Input arrives as JSON on stdin (convenient to read with `jq`).

◆ The three recipes everyone uses

```jsonc
// 1. Auto-format after every edit
// .claude/settings.json (in the project)
{
  "hooks": {
    "PostToolUse": [{
      "matcher": "Edit|Write",
      "hooks": [{
        "type": "command",
        "command": "jq -r '.tool_input.file_path'
          | xargs npx prettier --write"
      }]
    }]
  }
}
```

```jsonc
// 2. Desktop notification when input is needed
// ~/.claude/settings.json (Linux)
{
  "hooks": {
    "Notification": [{
      "matcher": "",
      "hooks": [{
        "type": "command",
        "command": "notify-send 'Claude Code'
          'Your attention is needed'"
      }]
    }]
  }
}
```

```bash
# 3. Protecting sensitive files
# .claude/hooks/protect-files.sh
# (registered on PreToolUse, matcher Edit|Write)
#!/bin/bash
INPUT=$(cat)
FILE_PATH=$(echo "$INPUT" | jq -r \
  '.tool_input.file_path // empty')

PROTECTED=(".env" "package-lock.json" ".git/")

for pattern in "${PROTECTED[@]}"; do
  if [[ "$FILE_PATH" == *"$pattern"* ]]; then
    echo "Blocked: $FILE_PATH" >&2
    exit 2   # exit 2 = blocks the action;
             # Claude receives the reason and
             # adapts its approach
  fi
done
exit 0
```

The contract in brief

- `exit 0` = all good, proceed.
- `exit 2` = **blocks** the action; stderr goes back to Claude as the explanation.
- On **SessionStart** the stdout is injected into the context: perfect for re-injecting conventions after a /compact (matcher `compact`) or the output of `git log --oneline -5`.
- On **PermissionRequest**, keep matchers **tight**: an empty matcher or `.*` would auto-approve everything, including writes and shell.

4. Plugins

the packaging layer

A plugin packages **skills + subagents + hooks + MCP servers** into a single installable unit, distributed as a Git repo (or zip/URL). Plugin skills are namespaced (`/my-plugin:review`), so multiple plugins coexist without conflicts. It's the answer to the trigger "a second repository needs the same setup": instead of copying .claude/ files around, you install.

How to use them

- `/plugin` opens the menu; direct subcommands: `list`, `install`, `enable`, `disable`.
- **Marketplaces** distribute plugin collections: Anthropic hosts an official one, and teams/communities publish others (public Git repos).
- `/reload-plugins` applies changes without restarting.
- For security reasons, plugin subagents ignore `hooks`, `mcpServers` and `permissionMode`: if you need them, copy the file into .claude/agents/.
- Neat trick: add a `.claude-plugin/plugin.json` to a skill folder and that skill becomes a plugin capable of including agents, hooks and MCP.

Vet before you trust

A plugin is **third-party code running in your environment**: hooks that execute shell commands, MCP servers that talk to the outside. Before installing from community marketplaces, read what it contains (skills, hooks and configurations are text files in the plugin's repo: they can be inspected). For teams, the "in-house" plugin, committed and reviewed like code, is the most solid pattern.

5. Combinations worth gold

when the pieces talk to each other

Recipe	How and why
Skill running in a subagent	`context: fork` (+ `agent:` to pick the type) in the skill's frontmatter: the workflow starts with /name but works in an isolated context and returns only the result. Ideal for long, noisy procedures.
Subagent with preloaded skills	`skills:` field in the agent's frontmatter: the worker is born with your API conventions or review checklist already on board, full content, without having to discover them.
Cheap subagent for exploration	A custom `Explore` agent with `model: haiku` overrides the built-in and keeps research on the cheapest model, while the main conversation stays on the powerful one.
Hook + security skill	A `PreToolUse` hook that blocks sensitive files (deterministic) + an invocable review skill (judgment): belt and suspenders.
Enthusiasm-proof deploy	`/deploy` skill with `disable-model-invocation: true` (runs only if you launch it) + restricted `allowed-tools` + `!`git status`` injection to decide based on reality.
MCP + a skill that uses it well	The MCP gives Claude the tools (e.g. the database); the skill teaches it your team's schema and query patterns. Tools + usage knowledge.
All together → plugin	When the combination works, package it: skills + agents + hooks + MCP in a plugin installable on every team repo.

Sources: code.claude.com/docs — features-overview, skills, sub-agents, hooks-guide, plugins · Features evolve fast: /release-notes is the truth for your version.