Claude Code & Agent SDK by Michaël Bettan

Definition: Claude Code is Anthropic's agentic coding tool. It reads your codebase, edits files, runs commands, and verifies its own work through a tool-using loop. The same harness ships as a library (the **Claude Agent SDK**) and runs as a hosted service (**Claude Managed Agents**); both have their own guide (see the scope note below).

The agentic loop: every task cycles through three blended phases: **gather context** (search, read) → **take action** (edit, run) → **verify results** (tests, diffs), repeating until done. Each tool result informs the next step, and you can interrupt at any point. Hooks, skills, subagents, and MCP all plug into specific phases of this loop.

Workflow: 1. Gather context (`CLAUDE.md` , `.claude/rules/` , and auto-memory `MEMORY.md` load; Claude searches and reads files, or delegates noisy exploration to a subagent) → **2. Permission check** (`PreToolUse` hooks run first → `deny` rules → `ask` rules → protected-path & critical-path checks → `allow` rules → active permission mode → `PermissionRequest` hooks → interactive prompt or SDK `canUseTool`) → **3. Take action** (edits, shell commands, MCP tool calls, or skill invocations execute) → **4. Verify** (`PostToolUse` / `FileChanged` hooks format or audit; Claude runs tests and reads results before `Stop` hooks evaluate completion).

Surfaces & execution modes: Claude Code (developer layer: terminal CLI, the Code tab of the Claude desktop app, VS Code / JetBrains extensions, `claude.ai/code` on the web, mobile iOS/Android, Slack, and CI/CD pipelines; the UI changes, the agentic loop is identical) · **Cowork & Dispatch** (**Cowork** is the Claude Desktop app tab for multi-step knowledge work run inside an app-hosted VM isolated from the host OS; **Dispatch** hands off tasks remotely from mobile/web to your desktop session) · **Agent SDK** (the Claude Code loop, tools, permissions, hooks, and subagents as a TypeScript or Python library you run in your own process).

- **Local Worktree vs. Cloud/Web & Teleportation (`&` and `/teleport`):** Desktop "Local worktree" and CLI `--worktree` (`-w`) run locally inside an isolated Git worktree (`.claude/worktrees/<name>`), sharing `.git` history and repo auto-memory (`MEMORY.md`) while **omitting gitignored files** (`node_modules/` , `.env` — run `npm install` or use `Setup` / `SessionStart` hooks). **Cloud/Web mode** (`claude.ai/code` , Desktop cloud mode, Mobile app — which is **100% cloud-only**, and CLI prompts prefixed with `& <prompt>`) runs in an Anthropic-managed container that clones from GitHub (defaulting to `main` unless steered) without your local uncommitted files, `.env` secrets, or local `stdio` MCP servers. Pull any cloud session and its branch back to your local terminal with `/teleport` (or `claude --teleport <session-id>`).
- **Deep Agent & Agent Harness architecture (4 pillars + 4 context strategies):** Unlike a **shallow ReAct agent** (which appends every tool output into one growing message list until **context rot** degrades reasoning), Claude Code is an **agent harness** built on four Deep Agent pillars: **(1) Explicit planning tool (`TodoWrite` / `TodoRead`)** maintaining live task state (`pending` , `in_progress` , `completed`) as a context anchor to prevent blind retry loops; **(2) Hierarchical subagents (`Agent`)** isolating noisy exploration; **(3) Filesystem as an external context engine (`Read` , `Write` , `Edit` , `Glob` , `Grep`)** offloading state to disk and retrieving slices via `Glob` / `Grep` + paginated `Read(offset, limit)` (avoiding *under-retrieval*, *over-retrieval*, and *misaligned retrieval*); and **(4) System prompt heuristics** (*Understand* → *Act* → *Verify*, specialized file tools over `cat` / `sed` , parallel tool calls). These map to the **4 context engineering strategies**: **Write** (`CLAUDE.md` , `MEMORY.md` , `TodoWrite`), **Select** (lazy subdir `CLAUDE.md` , `paths:` rules, `ToolSearch`), **Compress** (`/compact` , `/clear`), and **Isolate** (subagents, `context: fork`).

> MENTAL MODEL
> The model *proposes*; the harness *disposes*. Anything written in `CLAUDE.md` or `.claude/rules/` is advisory guidance the model weighs. Anything in `settings.json` permission rules, hooks, or managed policy is enforced deterministically by the client regardless of what the model decides.

> SCOPE NOTE
> The Agent SDK API (`query()` , `ClaudeAgentOptions` , `setting_sources` , `AgentDefinition`), agent-loop and orchestration patterns, and Managed Agents now live in the Claude Agent SDK guide. This guide covers configuring and using Claude Code itself.

1. Permission modes and allow / ask / deny rules

A permission mode sets the session's baseline for what runs without prompting. Rules then carve specific tools, paths, or commands in or out. Set the starting mode with `--permission-mode` or `permissions.defaultMode` (interactive CLI defaults to `auto` on supported plans in v2.1.283+; project `.claude/settings.json` and `.claude/settings.local.json` **ignore** `defaultMode: "auto"` and `"bypassPermissions"`).

Mode	Runs without asking	Still prompts / blocked	Use when
`default` (`manual`)	File reads + built-in read-only shell commands (`ls` , `pwd` , `git status` / `diff` / `log` , `rg` , `find` without `-exec` / `-delete`) in working dirs	Every file edit, mutating shell command, and path outside working directories	Sensitive repos; reviewing every mutating action

Mode	Runs without asking	Still prompts / blocked	Use when
`acceptEdits`	Everything in `default` + CWD file edits and basic file-manipulation utilities (`sed` , `cp` , `mv` , `mkdir` , `rmdir` , `touch` , `rm`)	Other Bash commands, paths outside CWD, **protected repo paths**, and critical-path `rm`	Iterating on code you are reviewing
`plan` (`Shift+Tab` / `/plan`)	File reads, read-only shell commands, read-only MCP calls, and read-only subagents (`Explore` , `Plan`)	All file edits, package installs, commits, and mutating commands until plan approval (`PreToolUse` `"allow"` does **not** bypass `plan` mode's edit block)	Spec-driven design (1. Explore & clarify via `AskUserQuestion` → 2. Draft spec → 3. Review → 4. Exit plan mode to execute)
`auto`	Actions approved by a background safety classifier; **strips broad `allow` rules** (`Bash(*)` , `Bash(python*)` , `Bash(node*)` , `Agent`) on entry	Classifier-flagged actions (curl-pipe-sh, prod deploys, force push, mass delete), protected paths, deny/ask rules; falls back to `default` if prerequisites fail	Long interactive tasks where prompt fatigue is the problem (disable org-wide via `"disableAutoMode": "disable"`)
`dontAsk`	Reads in working directories and explicitly pre-approved (`allow`) tools	Anything that would prompt is silently **denied** (`PermissionRequest` hooks and SDK `canUseTool` are skipped)	Locked-down CI, scripts, and background jobs
`bypassPermissions`	All tools and commands except the hard guardrails in the next column	Explicit `deny` rules, explicit `ask` rules, `PreToolUse` hook denials, **protected repo paths**, and **critical-path `rm` / `rmdir`**	Isolated containers and disposable VMs only (disable org-wide via `"disableBypassPermissionsMode": "disable"`)

> **COMMON TRAP**
> " `acceptEdits` only touches file edits." It also pre-approves CWD shell file utilities (`sed` , `cp` , `mv` , `rm`). Pair it with `deny: ["Bash(rm *)"]` when deletions must stay gated.

Rule evaluation and path / tool specifiers:

- **Evaluation order: `PreToolUse` hooks → `deny` → `ask` → protected-path & critical-path checks → `allow` → active mode → `PermissionRequest` hooks → user prompt / `canUseTool`** . First matching rule in `deny → ask → allow` wins *regardless of specificity*: a broad `deny: ["Bash(aws *)"]` beats a narrow `allow: ["Bash(aws s3 ls)"]` , and a matching `ask` still prompts even if a more specific `allow` matches.
- **Protected paths & critical-path removals:** Writes to `.git` , `.claude` , `.vscode` , `.idea` , `.husky` , `.mcp.json` , and shell profiles are checked **before** `permissions.allow` — putting `Edit(.claude/**)` in `allow` never pre-approves writes there, and even `bypassPermissions` still prompts on them. Removals targeting `/` , `~` , or CWD (`rm` / `rmdir`) cannot be auto-approved by `allow` rules or `PreToolUse` hooks.
- **Bare vs scoped `deny` rules:** A bare tool name in `deny` (`"Bash"` , `"WebFetch"` , `"mcp__*"` , `"Agent(Explore)"`) **removes the tool from Claude's context entirely**; a scoped rule (`"Bash(rm *)"`) keeps `Bash` in context and blocks matching calls.
- **File path specifiers (`Read` and `Edit` only):** Only `Read(...)` and `Edit(...)` accept path globs — **`Edit(...)` governs `Edit` , `Write` , and `NotebookEdit`** , whereas `Write(path)` , `NotebookEdit(path)` , and `Glob(path)` rules **never match**!
- **Path prefix anchors:**
- `//path` → **absolute filesystem path** from root (e.g. `Read(//etc/**)` , `Edit(//Users/me/secrets/**)`).
- `~/path` → relative to `$HOME` .
- `/path` (single slash) → anchored to `<cwd>/path` in project/local settings **or `~/.claude/path` in `~/.claude/settings.json` (not** the filesystem root!).
- `path` or `./path` → relative to the current working directory.
- **Cross-tool coupling:** `Read(path)` in `deny` also blocks `Edit` / `Write` on that path **and** blocks `@file` prompt mentions; `Edit(path)` in `allow` also grants `Read` . Putting `WebFetch(domain:*)` in `allow` **also opens all network hosts to sandboxed Bash commands** (use explicit per-domain rules like `WebFetch(domain:docs.example.com)`).
- **MCP rule syntax:** `mcp__<server>__<tool>` (one tool), `mcp__<server>` or `mcp__<server>__*` (all tools on server), `mcp__github__get_*` (wildcard in tool suffix only). **Never use `(...)` on `mcp__` rules** in settings files (e.g. `mcp__github__*(repo)` is skipped with a startup warning), and unanchored `"*"` or `"mcp__*"` in `allow` is skipped.

```
// .claude/settings.json (team-shared; requires workspace trust
dialog for allow rules)
{
  "permissions": {
    "defaultMode": "acceptEdits",
    "allow": ["Bash(npm run *)", "Edit(/src/**)",
"mcp__github__get_*"],
    "ask":   ["Bash(git push *)"],
    "deny":  ["Read(./.env)", "Read(//etc/**)", "Bash(rm *)",
"mcp__github__delete_repo"]
  }
}
```

OS Sandboxing (`/sandbox`) vs Permission Rules:

- **Permission rules** gate *which tool calls Claude may invoke*; **OS sandboxing** (`sandbox.enabled: true` ; Seatbelt on macOS, `bubblewrap` on Linux/WSL2) restricts *filesystem writes and network hosts of child OS processes* spawned by `Bash` .
- With `sandbox.autoAllowBashIfSandboxed: true` , sandboxed Bash commands run without prompting (`sandbox.filesystem.allowWrite` / `denyWrite` / `denyRead` and `sandbox.network.allowedDomains` enforce OS boundaries).
- If a command fails inside the sandbox, Claude may retry with `dangerouslyDisableSandbox: true` (which falls back to the normal permission prompt) unless `"allowUnsandboxedCommands": false` is set for strict sandboxing.

2. Settings and memory precedence

Precedence (high → low)	Where	Who it affects	Typical content
1. Managed	Server-managed, MDM/OS policy, or `managed-settings.json` (+ `managed-settings.d/*.json`)	Everyone the org deploys to; nothing below overrides a managed key	Security policy, compliance, model/plugin/MCP locks
2. Command line	`--settings <file-or-json>` and CLI flags (`--permission-mode`, `--allowedTools`)	This session only	One-off overrides, CI flags
3. Project local	`.claude/settings.local.json` (**always at git repo root**, even from subdirs/worktrees)	You, this repository (gitignored)	Personal overrides, local testing
4. Shared project	`.claude/settings.json`	Everyone who clones the repo	Team permissions, hooks, plugins, `env`
5. User	`~/.claude/settings.json`	You, every project	Personal defaults and rules

MERGE RULE

Arrays merge across layers; scalars override. Permission rule arrays from every layer combine (`deny → ask → allow`), so a `deny` at any layer still blocks. A scalar `model` in managed settings is only a starting default; the hard locks are `availableModels` (allowlist) and `deniedModels` (denylist, which wins over `availableModels`).

Memory / instruction source	Location & loading behavior	Shared with
Managed policy `CLAUDE.md`	OS system path (`/Library/Application Support/ClaudeCode/CLAUDE.md` on macOS, `/etc/claude-code/CLAUDE.md` on Linux/WSL, `C:\Program Files\ClaudeCode\CLAUDE.md` on Windows) or `"claudeMd"` string in `managed-settings.json`; **cannot be excluded**	Whole organization
User `CLAUDE.md` & rules	`~/.claude/CLAUDE.md` and `~/.claude/rules/*.md` (loaded at session start before project files)	Only you, all projects
Project `CLAUDE.md` & `AGENTS.md`	`./CLAUDE.md` or `./.claude/CLAUDE.md` (plus `AGENTS.md` when active); discovered upward from CWD and loaded in **root-to-CWD order** at launch. Subdirectory `CLAUDE.md` files load **lazily on demand** when files in that subtree are read/edited	Team, via source control
Path-scoped `.claude/rules/*.md`	Discovered recursively in `.claude/rules/`. Rules **without** `paths:` load at launch; rules **with** `paths: ["src/api/**/*.ts"]` load on demand when matching files are read	Team, via source control
Local `CLAUDE.local.md`	`./CLAUDE.local.md` (gitignored; appended after `CLAUDE.md` in each directory)	Only you, this project
Auto memory (`MEMORY.md`)	`~/.claude/projects/<project>/memory/MEMORY.md` (plus topic files). Claude writes notes itself; **first 200 lines or 25 KB** of `MEMORY.md` load at startup (toggle via `/memory`, `"autoMemoryEnabled": false`, or `CLAUDE_CODE_DISABLE_AUTO_MEMORY=1`)	Only you, per git repo (shared across worktrees)

- **`AGENTS.md` + `CLAUDE.local.md` trap:** Claude Code natively reads `AGENTS.md` alongside `CLAUDE.md`, **except** that under the default `auto` project-instructions setting, creating a `CLAUDE.local.md` file switches detection to `claude-md` only and **silently stops loading `AGENTS.md`** unless `/config` → *Project instructions* is set to `claude-md-and-agents-md`.
- **Size, `@path` imports, subdirectory lazy-load trap & monorepo exclusions:** Keep each `CLAUDE.md` **under 200 lines**; `@path/to/file` imports in root `CLAUDE.md` expand eagerly at launch (max 4 hops) and **do not save context tokens** (and manual `@file` mentions only resolve paths within or below the launch CWD / `--add-dir`). Conversely, placing domain rules in a subdirectory `packages/frontend/CLAUDE.md` **does not load them at startup** until a file in `packages/frontend/` is read/edited — use `.claude/rules/frontend.md` with `paths: ["src/components/**"]` for glob-scoped lazy loading. In monorepos, skip irrelevant ancestor files with `"claudeMdExcludes": ["**/monorepo/CLAUDE.md"]`. `--add-dir` loads skills/commands/agents from extra directories, but **not** `CLAUDE.md` or `.claude/rules/` unless `CLAUDE_CODE_ADDITIONAL_DIRECTORIES_CLAUDE_MD=1`.
- **Bootstrapping (`/init`, `#`, `/memory`), Context Rot taxonomy & `/clear` vs `/compact`:** Run `/init` to scan a repo and generate a starter `./CLAUDE.md` (tech stack, commands, architecture); prefix a message with `# <rule>` to save a rule directly to a chosen `CLAUDE.md` tier, or use `/memory` to edit memory files. Unmanaged long sessions suffer from three **Context Rot** failure modes: **Context poisoning** (a hallucination or bad tool output enters history and biases future turns), **Context confusion** (irrelevant files/tools dilute attention), and **Context clash** (contradictory instructions across layers — diagnose with `/doctor prompt-audit` and inspect token breakdown with `/context`). Use `/compact` to summarize history when continuing the same task, or `/clear` to wipe the conversation transcript completely when switching tasks or recovering from **context poisoning** (since `/compact` can preserve poisoned claims inside its summary, whereas `/clear` resets history while retaining `CLAUDE.md` and `MEMORY.md`). Note that `.claudeignore` **does not exist** — use `permissions.deny: ["Read(./secrets/**)"]`.
- **Output Styles (`/output-style`) & Custom Status Line (`/statusline`):**

- `/output-style` (`.claude/output-styles/*.md` > `~/.claude/output-styles/*.md`, or `"outputStyle"` in `settings.json`): Replaces the default software-engineering prose instructions in Claude Code's **system prompt** while preserving all tools, hooks, and subagents. Built-in styles: `Default`, `Explanatory` (educational architectural insights), and `Learning` (collaborative coaching with `TODO(human)` markers). Custom Markdown styles use YAML frontmatter (`name`, `description`, `keep-coding-instructions: true|false`) to enforce response protocols (e.g., concise bulleted summaries or structured key-value notes). **Exam trap:** `/output-style` is prompt-level guidance for interactive sessions — never use it as a substitute for `--output-format json --json-schema` in headless CI pipelines.
- `/statusline` (`"statusLine": {"type": "command", "command": "..."}` in `settings.json`): Runs an executable script that receives session metadata as **JSON on** `stdin` (`model`, `output_style`, `cwd`, cost/context stats, and `transcript_path` pointing to `~/.claude/projects/<encoded-cwd>/<session-uuid>.jsonl`) and prints a persistent terminal footer (with ANSI colors) to `stdout` at **zero context token cost**.

3. CLAUDE.md hierarchy, `@` imports and path-scoped rules

Instruction is for...	Put it in	In git?	Trap
The whole team, every task	`./CLAUDE.md` or `./.claude/CLAUDE.md`	Yes	Keep it short (under ~200 lines); move topic detail into imports or rules
Only you, every project	`~/.claude/CLAUDE.md`	No	Teammates never receive it: team conventions placed here "work on my machine" only
Only you, this repo	`./CLAUDE.local.md`	No (gitignored)	Not shared across worktrees; import `@~/.claude/<file>.md` instead to reuse personal notes
One package or directory	`<subdir>/CLAUDE.md`	Yes	Loads **lazily**, only once Claude reads files in that subtree
One file type spread across many directories (tests, API handlers)	`.claude/rules/<topic>.md` with `paths:` globs	Yes	A directory `CLAUDE.md` cannot target "every `*.test.tsx` anywhere"
The whole org, not excludable	Managed policy `CLAUDE.md`	IT-deployed	Users cannot override or exclude it

`@` imports (modular CLAUDE.md):

```
# CLAUDE.md
See @README.md for the project overview and @package.json for npm scripts.

## Standards
- Testing conventions: @docs/testing.md
- API conventions: @docs/api-conventions.md
- My personal overrides: @~/.claude/my-project-instructions.md
```

- Relative paths resolve **from the file containing the import**, not from the working directory. Absolute and `~` paths are allowed.
- Imported files can import others, up to **four hops** deep.
- Imports load **at launch**: they organize instructions but **do not save context tokens**. For on-demand content use path-scoped rules or skills.
- The first time a project uses imports from outside the repo, Claude Code shows a one-time approval dialog.
- Team pattern: keep the root `CLAUDE.md` as a short index of `@imports`, so each topic file has one owner and only relevant standards are pulled in per package.

Path-scoped rules (`.claude/rules/*.md`):

```
---
paths:
  - "src/api/**/*.ts"
  - "**/*.test.{ts,tsx}"
---
# API & test conventions
- Validate every request body with the shared schema helpers.
- Tests use the factories in test/factories; never hit the real network.
```

- Rules **without** `paths` load at launch like `CLAUDE.md`. Rules **with** `paths` load only when Claude reads a matching file, so they cost no tokens on unrelated tasks.
- Globs support `**` and brace expansion (`*.{ts,tsx}`).
- **Diagnose with** `/memory`: it lists every `CLAUDE.md`, `CLAUDE.local.md` and rules file loaded in the current session. Run it first when "Claude ignores my instruction".

> **EXAM TRAPS**
> (1) A new teammate's Claude ignores the team's conventions → they live in someone's `~/.claude/CLAUDE.md` (user scope), not the project file. Move them to `./CLAUDE.md` or `.claude/rules/`. (2) Test files sit next to source files across 30 directories → one rule with `paths: ["**/*.test.*"]`, not 30 directory `CLAUDE.md` files. (3) Splitting a big `CLAUDE.md` into `@imports` does not reduce tokens; path-scoped rules do.

4. Which mechanism carries which rule

DECISION RULE
Ask two questions: **must it hold 100% of the time?** (then settings rules or a hook, never prose) and **does it matter for every task?** (then always-on; otherwise on-demand). Pick the lowest-context mechanism that satisfies both.

You need...	Use	Why not the alternatives	Context cost
Conventions for every task (test command, build steps)	**`CLAUDE.md`**	A hook can't express style; a skill may not trigger	Every session; dilutes past ~200 lines
Guidance only for `src/db/**`	**`.claude/rules/db.md`** with **`paths:`**	In `CLAUDE.md` it wastes tokens on unrelated tasks	Only when matching files are touched
Block a tool, path, `@file` mention, or command prefix, always	**`permissions.deny`** (managed if org-wide)	`CLAUDE.md` is advisory; `PreToolUse` skips `@file` mentions	Zero (bare tool deny even removes tool schema)
Inspect/rewrite args or block dynamically before a tool runs	**`PreToolUse`** hook (exit `2` or `permissionDecision: "deny"`)	Permission rules match globs/prefixes, not payload content	Near-zero (only injected reason/context)
Deterministic side effect after edits (formatter, linter, audit)	**`PostToolUse`** hook (or **`FileChanged`** if edited via `Bash`)	Instructions get skipped; `PreToolUse` fires before disk write	Near-zero
Wide codebase search or independent parallel check	**Subagent** (`.claude/agents/`)	Main context fills with throwaway file reads	Isolated window; returns summary only
Repeatable workflow or slash playbook (`/release`)	**Skill** (`SKILL.md`; add `disable-model-invocation: true` for side effects)	In `CLAUDE.md` you pay for the full playbook every session	Description only (or zero if manual-only) until invoked
Change how Claude formats & communicates every response (YAML, teaching, HTML)	**Output style** (`/output-style`, `.claude/output-styles/*.md`)	`CLAUDE.md` adds on top of SWE prose defaults; skills are task-scoped	Replaces default SWE system prompt section
Persistent terminal footer showing live session/transcript state	**Status line** (`/statusline`, `statusLine` in `settings.json`)	Hooks only run on events; output styles waste output tokens	**Zero context tokens** (reads JSON on `stdin` / `transcript_path`)
Track dynamic multi-step task progress during long-horizon execution	**`TodoWrite`** / **`TodoRead`** (built-in planning tool)	Implicit CoT drifts as context grows; `plan` mode only gates pre-execution	Compact task-state JSON in context
Ship skills, agents, hooks & MCP across many repos	**Plugin** via a marketplace	Copy-pasted `.claude/` folders drift across repos	Per bundled component
Org policy users cannot override	**Managed settings** (+ managed `CLAUDE.md` for guidance)	Project/user files can be edited locally	Zero

How mechanisms combine across scopes (`managed / user / project / local / plugin`):

- **`CLAUDE.md`** & **`.claude/rules/`** — **Additive:** Every matching file across all layers loads into context together.
- **`Permissions`** & **`Hooks`** — **Merge & most-restrictive wins:** Permission rules merge across all settings files (`deny > ask > allow`). Hooks from **every** active layer register and execute **in parallel** on matching events; `PreToolUse` decisions resolve by **`deny > defer > ask > allow`**.
- **`Subagents`**, **`Skills`**, & **`MCP servers`** — **Override by name:** Only one definition wins when names collide (plugins use `plugin:name` namespaces). **Precedence trap:** Subagents resolve **`managed > --agents > project > user > plugin`** (**project beats user**), whereas Skills resolve **`managed > user > project > nested > plugin > synced`** (**user beats project**), and MCP servers resolve **`local > project > user`**.
- **Spec-Driven Development (SDD) & Two-Phase Model Switching: (1) Plan (`plan` mode + Opus):** Toggle `plan` mode (`Shift+Tab`), select **Opus** (`/model`), research docs/code, and write a persistent spec (`SPEC.md`). **(2) Clean Context (`/clear`):** Wipe exploratory research noise. **(3) Implement & Verify (`acceptEdits` + Sonnet + `@SPEC.md`):** Switch to **Sonnet**, attach `@SPEC.md`, implement via `TodoWrite` steps, and verify visually via **Playwright MCP** (`mcp__playwright__*`) or an independent **Reviewer** session.

5. Hooks and exit-code semantics

What a hook is: a lifecycle handler configured in `settings.json` (user, project, local, or managed policy), a plugin's `hooks/hooks.json`, or a **skill/subagent YAML frontmatter** (`hooks:`). Hooks receive event JSON on `stdin` (`session_id`, `cwd`, `hook_event_name`, plus tool fields `tool_name`, `tool_input`). An empty `"matcher": ""` (or `"*"`) matches every

occurrence of that event; keep high-frequency synchronous hooks lightweight (or set `"async": true` for non-blocking logging/notifications) to avoid per-turn latency.

- **5 handler types:** `"command"` (shell/exec script; supports `"async": true` for background side effects), `"http"` (POSTs event JSON to a URL), `"mcp_tool"` (invokes a configured MCP server tool), `"prompt"` (single-turn fast LLM check returning `{"ok": true|false, "reason": "..."}`), and `"agent"` (multi-turn verifier subagent with `Read` / `Grep` / `Glob`, up to 50 turns).

Exit code	Meaning	Effect on the event
`0`	Success / allow	Event proceeds. `stdout` is parsed for JSON decision fields (`hookSpecificOutput`, `decision`, `continue`). Plain-text `stdout` is added to Claude's context **only** on `SessionStart`, `Setup`, `UserPromptSubmit`, and `PostModelSwitch`. Exit `0` without JSON does *not* bypass permission prompts.
`2`	**Blocking error**	**Blocks** the action (on blockable events). `stderr` is fed back to Claude as the block reason. Valid JSON on `stdout` is still read but **can't override the block** (even `"allow"`); a JSON blocking-decision reason, if present, replaces `stderr` as the message.
`1` or other non-zero (no valid JSON)	**Non-blocking error (fail-open default)**	**Exam trap:** `exit 1` without valid hook JSON logs `stderr` as a warning and **lets the tool call proceed** — **unless `"onFailure": "block"`** is set on the `command` / `http` hook (which turns non-zero exits, non-2xx HTTP responses, and **timeouts** into blocks). *(If `stdout` contains valid hook JSON, Claude Code applies the JSON decision and ignores the exit code; `WorktreeCreate` / `WorktreeRemove` fail on any non-zero exit.)*

Event	Fires when	Matcher	Exit `2` / JSON block effect
`PreToolUse`	Before a tool call executes in **every** mode (**skips `@file` prompt mentions**)	Tool name (`Bash`, `Edit|Write`, `mcp__srv__tool`) + optional `if` (`"Bash(rm *)"`)	**Blocks** tool call; `stderr` or `permissionDecisionReason` sent to Claude
`PermissionRequest`	Before a permission dialog shows (or before `-p` /background auto-deny outside `dontAsk`)	Tool name	Denies via `hookSpecificOutput.decision.behavior: "allow"|"deny"`
`PostToolUse`	Immediately after a tool succeeds (`Edit|Write` misses `Bash` file edits)	Tool name	Cannot undo execution; feeds `stderr` / `decision: "block"` reason to Claude or redacts output via `updatedToolOutput`
`PostToolUseFailure`	After a tool execution throws or exits non-zero (not permission/schema rejections)	Tool name	Feeds `stderr` / `additionalContext` to Claude alongside the tool error
`UserPromptSubmit`	User submits a prompt, before Claude processes it	*(none)*	**Blocks** prompt processing and erases the prompt from context
`Stop` / `SubagentStop`	Main agent (`Stop`) or subagent (`SubagentStop`) finishes its turn	*(none)* for `Stop`; agent `name` for `SubagentStop`	**Prevents stopping** and resumes work (check `.stop_hook_active` on `stdin`; hard cap of **8 consecutive continuations**)
`SubagentStart`	Subagent spawns or resumes	Agent `name` (`Explore`, custom)	Cannot block spawn; injects `additionalContext` into the subagent
`SessionStart` / `Setup`	Session starts (`SessionStart`) or `--init` / `--maintenance` runs (`Setup`)	`startup|resume|clear|compact` (`SessionStart`); `init|maintenance` (`Setup`)	Cannot block; `stdout` / `additionalContext` added to context; **`$CLAUDE_ENV_FILE`** available to persist `export` vars
`PreCompact`	Before context compaction (`PostCompact` fires after)	`manual|auto`	**Blocks compaction** (`exit 2` or `decision: "block"`)
`ConfigChange`	Settings, managed policy, or skill file changes mid-session	`user_settings|project_settings|local_settings|policy_settings|skills`	**Blocks** config change from taking effect (**except** `policy_settings`, which cannot be blocked)
`FileChanged` / `CwdChanged`	Watched file changes on disk (`Edit`, `Bash`, or external) / directory changes	Literal `|`-separated filenames (`.envrc|.env`) for `FileChanged`	Cannot block; catches `Bash` file edits missed by `PostToolUse(Edit|Write)`; **`$CLAUDE_ENV_FILE`** available
`WorktreeCreate` / `WorktreeRemove`	`--worktree` or `isolation: worktree` subagent creates/removes worktree	*(none)*	Replaces default `git worktree` (e.g. SVN/Mercurial); `WorktreeCreate` prints path on `stdout` (any non-zero exit fails)
`Notification` / `SessionEnd`	User alert (`permission_prompt`, `idle_prompt`) / session close (1.5s timeout)	Notification type / exit reason	Cannot block; alerting and cleanup only

```
// .claude/settings.json — fail-closed PreToolUse command hook with portable path
{
  "hooks": {
    "PreToolUse": [{
      "matcher": "Bash",
      "hooks": [{
        "type": "command",
        "if": "Bash(rm *)",
        "onFailure": "block",
        "command": "${CLAUDE_PROJECT_DIR}/.claude/hooks/no-rm.sh"
      }]
    }]
  }
}
```

- **Structured JSON on `stdout` (exit `0`) — 3 decision schemas:** 1. **`PreToolUse`** uses `hookSpecificOutput`:
  - `permissionDecision`: **`"deny"` > `"defer"` > `"ask"` > `"allow"`** (most restrictive wins across parallel hooks). `"deny"` blocks in **every** mode including `bypassPermissions`; `"defer"` pauses headless `-p`/SDK runs with `stop_reason: "tool_deferred"` for later `--resume`; `"allow"` bypasses normal prompts (unless matched by `permissions.deny`/`ask` or blocked by `plan` mode).
  - `updatedInput`: replaces the **entire** `tool_input` object before execution (omitted keys are dropped, not merged; `permissions` rules re-evaluate against the rewritten input; with `"allow"`, can auto-answer `AskUserQuestion`).
  - `additionalContext`: must be nested **inside `hookSpecificOutput`** alongside `hookEventName` (top-level `additionalContext` is silently ignored). 2. **`PermissionRequest`** uses `hookSpecificOutput.decision`: `{"behavior": "allow" | "deny", "updatedInput": {...}, "updatedPermissions": [...], "message": "..."}`. 3. **`PostToolUse`**, **`UserPromptSubmit`**, **`Stop`**, **`SubagentStop`**, **`PreCompact`**, **`ConfigChange`** use top-level `{"decision": "block", "reason": "..."}` (plus `hookSpecificOutput.updatedToolOutput` on `PostToolUse` to redact/replace tool output before Claude sees it). Top-level `{"continue": false, "stopReason": "..."}` halts the entire turn across events.
- **`@file` mention bypass trap:** `PreToolUse` on `Read` does **not** fire when a user types `@.env` in their prompt (Claude Code inlines `@file` directly without calling `Read`). Block both `Read` calls and `@file` mentions with `permissions.deny: ["Read(./.env)"]` (or `CLAUDE_CODE_DISABLE_FILE_MENTIONS=1`).
- **Portable paths & `$CLAUDE_ENV_FILE`:** Use `${CLAUDE_PROJECT_DIR}` in repo hooks and `${CLAUDE_PLUGIN_ROOT}` (code) / `${CLAUDE_PLUGIN_DATA}` (persistent state) in plugin hooks. Append `export KEY=VAL` to `$CLAUDE_ENV_FILE` inside `SessionStart`, `Setup`, `CwdChanged`, or `FileChanged` hooks to persist env vars across subsequent `Bash` calls.

6. Subagents

What they are: specialized assistants with their **own context window**, system prompt, tool set, and optional model/permission mode (created interactively via `/agents` or defined in `.md` files).

- **5-level scope precedence (highest → lowest on name collision): 1. Managed** (`.claude/agents/`) → **2. `--agents '{...}'` CLI flag** → **3. Project** (`.claude/agents/<name>.md`) → **4. User** (`~/.claude/agents/<name>.md`) → **5. Plugin** (`agents/<name>.md`, invoked as `plugin:agent`). *(Note: for subagents Project > User; for skills User > Project!)* Plugin subagents **ignore** `hooks`, `mcpServers`, and `permissionMode` in frontmatter for security.
- **Built-in vs Custom vs Fork (`/subtask`):**
- `Explore` and `Plan` (both run on the main conversation's model by default; a project/user subagent named `Explore` with `model: haiku` overrides the built-in for cheaper search) are **read-only**, **skip root `CLAUDE.md` and git status**, and are **one-shot** (return no `agentId`, **cannot be resumed**).
- `general-purpose` and all **custom subagents** load `CLAUDE.md` (unless `omitClaudeMd: true`), start in a **fresh context window** (receiving **only** their Markdown body as system prompt + `CLAUDE.md` + the single synthesized task prompt — **never** parent conversation history), and return an `agentId` that **can be resumed** via `SendMessage`.
- **Fork subagent (`/subtask`):** Unlike standard subagents, a fork **inherits the parent's full conversation history, system prompt, tools, and model**, and **shares the parent's prompt cache** (up to depth 2).
- **Dual role of `description`, untrusted-repo security trap & "missing raw artifact" trap:**
- A subagent's `description` is **injected directly into the MAIN agent's system prompt**: it controls **(1) when** the main agent delegates (use `"Use proactively after..."`) and **(2) how the main agent formats the task prompt** sent to the subagent. **Security trap:** Because `description` enters the *main* agent's system prompt, a malicious `.claude/agents/*.md` `description` in an untrusted repo can prompt-inject the **main agent** (which has full tools) even if the subagent's own `tools:` field is read-only!
- **Summary-only return trap:** When a subagent finishes, its internal conversation context is discarded and **only its final message** returns to the main agent (view internal steps via `Ctrl+R` / `Cmd+R`). If an intermediate tool call generated code, schemas, or architectural figures that were not written directly to disk or echoed in the return payload, the coordinator cannot inspect them — instruct subagents to write concrete files to disk or include raw output blocks in their final message.

```yaml
---
name: security-auditor
description: Audits diffs for OWASP issues. Use proactively after code changes.
tools: Read, Grep, Glob              # allowlist (omit to inherit all tools; no Agent = cannot spawn subagents)
disallowedTools: Write, Edit, Agent  # applied FIRST (Trap: Bash(git push *) strips ALL of Bash! Use permissions.deny)
model: sonnet                        # Per-call Agent(model) > frontmatter > CLAUDE_CODE_SUBAGENT_MODEL > parent model
permissionMode: plan                 # default | acceptEdits | auto | dontAsk | plan | bypassPermissions
maxTurns: 15                         # returns partial result on limit; resumable via SendMessage
skills: [owasp-checklist]            # preloads FULL skill body at startup (max 32; skips disable-model-invocation: true)
mcpServers: [github, {playwright: {type: stdio, command: npx, args: ["-y", "@playwright/mcp"]}}] # string reuses parent; inline is subagent-only
memory: project                      # user | project | local → .claude/agent-memory/<name>/MEMORY.md (200 lines/25KB) + auto-enables Read/Write/Edit
hooks:                               # subagent-scoped (Stop auto-converts to SubagentStop; requires folder trust even in -p)
  PreToolUse: [{matcher: "Bash", hooks: [{type: command, command: "${CLAUDE_PROJECT_DIR}/scripts/check.sh"}]}]
isolation: worktree                  # temporary git worktree; auto-cleaned if no changes
background: true                     # runs concurrently; cannot prompt interactively (AskUserQuestion fails)
omitClaudeMd: true                   # skips user/project/local CLAUDE.md (managed policy CLAUDE.md still loads)
---
```

- **`disallowedTools` specifier trap:** Putting `disallowedTools: Bash(git push *)` in subagent frontmatter **removes the entire `Bash` tool** from the subagent. Keep `Bash` in `tools` and block specific subcommands via `permissions.deny` or a `PreToolUse` hook.
- **`permissionMode` inheritance & privilege boundary:** If the parent session runs in `bypassPermissions`, `acceptEdits`, `auto`, or `plan`, the parent mode **overrides** the subagent's `permissionMode`. A subagent setting `permissionMode: bypassPermissions` is **ignored** unless the parent session is already in `bypassPermissions`.
- **Restricting which subagents can spawn (`Agent(type)`):** When running as the main thread via `claude --agent`, `tools: Agent(worker, researcher)` allowlists which subagents may be spawned (`Task(...)` is a legacy alias). Session-wide, block specific subagents via `permissions.deny: ["Agent(Explore)"]`.
- **Parallel subagents, Batch Concurrency & When Parallel Agents Fail:**
- **Batch concurrency control:** Claude runs multiple subagents sequentially unless explicitly told tasks are independent (*"in parallel"*). Because isolated subagents cannot see each other, dispatch concurrent tasks in bounded cohorts (e.g. 3 to 5 concurrent workers) to avoid hitting HTTP 429 rate limits or overflowing coordinator context, passing each worker an isolated workspace worktree and explicit target paths.
- **Builder–Reviewer context separation:** An agent reviewing code/UI it just built in the same session suffers from self-confirmation bias. Always run QA/review in a **separate instance or fresh subagent** that sees only the spec and diff/screenshot.
- **5 anti-patterns where parallel agents/worktrees lose: (1) Sequentially coupled interfaces** (e.g., building a backend API and its frontend consumer in parallel without a locked `SPEC.md` causes the frontend agent to hallucinate mock schemas), **(2) Shared mutable state/files** (migrations, `package.json`, global CSS), **(3) Small tasks (<5 min)** where worktree/merge overhead exceeds task time, **(4) High boundary-spec coordination cost**, and **(5) Exploratory root-cause debugging**.

7. Skills and progressive disclosure

3-level progressive disclosure: a skill is a directory containing `SKILL.md` (YAML frontmatter + markdown playbook) and optional `scripts/` or `references/` (`${CLAUDE_SKILL_DIR}` ; legacy `.claude/commands/<name>.md` files are unified under Skills). **(1) Session start:** only YAML frontmatter (`name` , `description` , `when_to_use` — **max 1,536 chars** per skill; total listing budget is **1% of context window** via `SLASH_COMMAND_TOOL_CHAR_BUDGET`) enters the system prompt. **(2) On activation:** the `SKILL.md` markdown body loads (re-attached after `/compact` up to **5,000 tokens per skill / 25,000 tokens total**). **(3) On execution:** bundled `scripts/` and reference files load **only when executed or read** (unused scripts consume **zero tokens**).

- **Frontmatter boundary (`---`) & script fallback traps:** Anything written below the closing `---` YAML frontmatter delimiter is **invisible during skill selection** — put all trigger keywords in frontmatter `description` / `when_to_use` . If a bundled script exits with a non-zero code at runtime, Claude may attempt to diagnose or fall back to standard tool reasoning; to ensure specific destructive commands or unvetted actions are never executed on script failure, enforce restrictions deterministically via `permissions.deny` or a `PreToolUse` hook.

6 scopes & precedence (highest → lowest): 1. Enterprise (`managed-settings`) > **2. Personal** (`~/.claude/skills/<name>/SKILL.md`) > **3. Project** (`.claude/skills/<name>/SKILL.md`) > **4. Nested** (`<subdir>/.claude/skills/` , auto-discovered on demand) > **5. Plugin** (`<plugin>/skills/<name>/SKILL.md` , namespaced `/plugin:skill`) > **6. Synced `claude.ai`** (`~/.claude/skills/synced/` , invoked as `/anthropic-skills:<name>`).

- **Personal variant of a team skill:** copy it to `~/.claude/skills/<different-name>/` so you can tweak it while teammates keep the project version. Reusing the **same name** silently shadows the project skill for you only (personal > project), which hides upstream fixes.

Frontmatter / Syntax	Effect & exam traps
`description` + `when_to_use`	Trigger text shown in context (combined cap: **1,536 chars**; put the primary trigger phrase first)
`disable-model-invocation: true`	**Manual-only (`/name`):** description is **omitted from context** (zero token cost until invoked), Claude cannot invoke it via the `Skill` tool, and subagents cannot preload it. Use for side-effecting playbooks (`/deploy` , `/release`)
`user-invocable: false`	Model-invocable background knowledge; hidden from the `/` menu (description stays in context)
`allowed-tools`	**Pre-approves** listed tools while the skill runs (does **not** restrict available tools; clears on next user turn). To actually **limit** tools, use `context: fork` with an `agent` whose `tools` list is narrow, or `permissions.deny`

Frontmatter / Syntax	Effect & exam traps
`context: fork` + `agent`	Runs the skill in an **isolated subagent with no conversation history** (`agent: Explore` also skips `CLAUDE.md` , so `SKILL.md` must be self-contained)
`paths` / `hooks` / `model` / `effort` / `shell`	`paths:` restricts auto-activation to matching file globs; `hooks:` registers skill-scoped lifecycle hooks

Frontmatter / Syntax	Effect & exam traps
`$ARGUMENTS` , `$ARGUMENTS[N]` , `$N`	Substitutes arguments passed to `/skill-name arg0 arg1` (appends `ARGUMENTS: ...` if omitted from body; also supports `${CLAUDE_SKILL_DIR}` and `${CLAUDE_SESSION_ID}`)
`!`command`` preprocessing	Shell command inside `!`...`` runs **before** the skill prompt reaches Claude and inlines its `stdout` . **Trap:** it **never shows a permission prompt** — outside `auto` mode, an unapproved command or any non-zero exit code **aborts the entire skill invocation**

Surface	How a custom skill reaches it	Frontmatter & sync rules
Claude Code CLI / Desktop / IDE	Filesystem scopes + `claude.ai` downward sync (`~/.claude/skills/synced/`)	Full Claude Code frontmatter (`allowed-tools` , `context: fork` , `hooks` , `argument-hint`). **One-way sync:** `claude.ai` skills sync *down* to Claude Code, but local `.claude/skills/` **never** sync *up* to `claude.ai` or the API
Agent SDK	`.claude/skills/` when `setting_sources` includes `"project"` / `"user"` , or preloaded per subagent via `skills`	Full body injected at startup when listed in subagent `skills: [...]`
Claude API (`/v1/skills`)	Uploaded via Skills API (`skill_id` in `container.skills` + `code_execution` tool; **workspace-wide**, pin version in prod)	Runs in Anthropic's networkless container. **Strict schema trap (verify: based on the Agent Skills spec; subject to change):** API `/v1/skills` accepts **only** `name` , `description` , `license` , `compatibility` , `metadata` , `allowed-tools` — uploading a `SKILL.md` with `argument-hint` , `disable-model-invocation` , `context` , or `hooks` fails validation
Managed Agents	Attached to the Agent definition (`skills: [...]`)	Runs in the session's cloud or self-hosted environment

8. Custom slash commands

Scope	Location	Invoked as	Shared?
Project	`.claude/commands/<name>.md` (or skill `.claude/skills/<name>/SKILL.md`)	`/name`	Yes: committed, every teammate gets it on clone/pull
Personal	`~/.claude/commands/<name>.md` (or `~/.claude/skills/<name>/SKILL.md`)	`/name`	No: only you, all projects
Plugin	`<plugin>/commands/` or `<plugin>/skills/`	`/plugin:name`	Via marketplace

Commands and skills are unified: a `.claude/commands/review.md` file and a `.claude/skills/review/SKILL.md` skill both create `/review` and accept the same frontmatter. Prefer a skill when you need supporting files (scripts, references).

Frontmatter	Effect
`description`	Shown in the `/` menu; also lets Claude invoke it automatically when relevant
`argument-hint`	Autocomplete hint such as `[focus-area]`
`allowed-tools`	Pre-approves listed tools while the command runs (it does not remove other tools)
`model`	Model override for this command
`context: fork` (+ `agent`)	Runs in a **forked subagent with its own context**; verbose output stays out of the main conversation, only the result returns
`disable-model-invocation: true`	Only you can trigger it (use for side effects such as deploys)
Body: `$ARGUMENTS` , `$1` , `@file` , `!`cmd``	Argument substitution, file inclusion, and shell preprocessing (see §7 for the `!`cmd`` trap)

```
---
description: Review the current branch against team standards
argument-hint: [focus-area]
allowed-tools: Read, Grep, Glob, Bash(git diff *)
context: fork
---
Review the changes on this branch (run `git diff main...HEAD`).
Focus: $ARGUMENTS
Apply the checklist in @docs/review-checklist.md and return
findings as a table
with file, line, severity and suggested fix.
```

DECISION RULE

Team workflow everyone should run the same way → **project** `.claude/commands/` (or skill), committed. Personal shortcut → `~/.claude/commands/` . Workflow that reads a lot of files or produces long output → add `context: fork` so the main session stays clean.

EXAM TRAP

A command saved in `~/.claude/commands/` "doesn't exist" for teammates: personal scope is not in version control.

9. Workflows: plan mode, the interview pattern and iterative refinement

Situation	Approach	Why
Diff fits in one sentence (typo, log line, rename)	**Direct execution**	Planning overhead exceeds its value
Clear bug with a stack trace in one file	**Direct execution**	Scope and fix are already known
Multi-file change, unfamiliar code, several valid approaches (framework migration, splitting a monolith, new integration)	**Plan mode** first	Explore safely and agree on an approach before edits; avoids costly rework
Large change whose approach is already decided	Plan mode to explore, then **direct execution** to implement	Combine: investigate read-only, then execute the agreed plan

- **Explore → plan → implement → commit.** Plan mode allows reads and read-only tools but blocks edits until you approve. Press `Ctrl+G` to open the plan in your editor and change it before Claude proceeds. `/model opusplan` uses Opus while planning and Sonnet while executing.
- **Keep discovery out of the main context:** delegate verbose exploration to the `Explore` subagent; only its summary returns.
- **Interview pattern** (unfamiliar domain, hidden requirements): *"Interview me in detail using the AskUserQuestion tool about this feature, covering edge cases, failure modes and trade-offs, then write a complete spec to SPEC.md."* Then start a **fresh session** that implements from `SPEC.md`, so the interview noise doesn't fill the implementation context.

Iterative refinement techniques:

Technique	Use when
2–3 concrete input → output examples	Prose descriptions get interpreted inconsistently (data transformations, output formats, naming)
Test-driven iteration: write tests first (including edge cases and performance limits), then iterate on failures	Behavior must be verifiable; failing tests tell Claude exactly what to fix
Give a pass/fail check (test command, build, linter, screenshot comparison)	Always: Claude can verify and self-correct instead of guessing
Specific feedback: actual vs expected output, exact error text	Replace "it doesn't work" with evidence
One message for interacting issues; sequential fixes for independent ones	Fixes that affect each other need to be considered together

> **EXAM TRAP**
> Plan mode is not a default for everything. Choosing plan mode for a one-line, well-understood fix (or direct execution for an architectural migration) is the wrong call.

10. Built-in tools: choosing the right one

Tool	What it does	Pick it when	Trap
Grep	Searches file **contents** (ripgrep regex); output modes `files_with_matches` (default), `content`, `count`; filter with `glob` or `type`; respects `.gitignore`	Find callers of a function, an error string, all imports of a module	It doesn't find files by name; escape regex metacharacters
Glob	Finds files by **name/path pattern** (`**/*.test.tsx`); results sorted by modification time, capped at 100	Locate files by extension, naming convention or directory	Doesn't search contents; **doesn't respect `.gitignore`** by default
Read	Reads a file (paged with offset/limit for large files)	Inspect a file before changing it	Required before `Edit`
Edit	Replaces an exact `old_string` with `new_string`	Targeted change in an existing file	Fails unless the file was read first, the match is exact and `old_string` is unique
Write	Creates or fully overwrites a file	New files, or a complete rewrite	Overwriting an existing file requires reading it first
Bash	Runs shell commands	Tests, builds, git, anything without a dedicated tool	Prefer Grep/Glob/Read over `grep` / `find` / `cat` for search and reading

Edit failure → fallback:

1. **Not read yet / file changed since read** → `Read` the file again, then retry the edit.
2. **`old_string` matches several places** → include more surrounding lines so it is unique, or set `replace_all: true` when every occurrence should change.

3. **Still no stable unique anchor** (repetitive generated code, many similar blocks) → `Read` the full file and `Write` the complete new content.

Incremental codebase exploration: start with `Grep` for entry points (function name, route, error message) → `Read` only the files that matter → follow imports with further `Grep` calls. To trace a function used through wrapper modules, first find every exported name that re-exports it, then `Grep` each name across the codebase. Don't read every file up front.

11. Plugins and marketplaces

Plugin anatomy:

- Manifest at `.claude-plugin/plugin.json` (`name`, `version`, `description`); component directories (`skills/`, `commands/`, `agents/`, `hooks/hooks.json`, `.mcp.json`, `.lsp.json`, `settings.json`) live at the **plugin root**, never inside `.claude-plugin/`. Plugin `settings.json` only supports the `agent` key. A marketplace catalog is defined by `.claude-plugin/marketplace.json` (each entry's `source` points to the plugin directory) and supports **granular installation** of individual components.
- **`${CLAUDE_PLUGIN_ROOT}` vs `${CLAUDE_PLUGIN_DATA}`:** Installed plugins are copied into a read-only versioned cache (`~/.claude/plugins/cache/...`), which `${CLAUDE_PLUGIN_ROOT}` resolves to — **anything written inside `${CLAUDE_PLUGIN_ROOT}` is wiped on plugin update.** Store persistent state, SQLite DBs, or `node_modules` in **`${CLAUDE_PLUGIN_DATA}`** (`~/.claude/plugins/data/<id>/`), which survives updates.
- **Version bump rule:** `version` in `plugin.json` overrides `marketplace.json` `version` and git commit SHAs. If you push new code without bumping `version` in `plugin.json`, `claude plugin update` sees the cached version and **skips fetching**.
- Declare install-time secrets in `plugin.json` `userConfig` with `"sensitive": true` (stored in OS keychain and exposed as `${CLAUDE_PLUGIN_OPTION_<KEY>}`).

Marketplace governance (`managed-settings.json`):

- `extraKnownMarketplaces`: pre-registers org catalogs (`github`, `git`, `url`, `npm`, `file`).
- `enabledPlugins`: `"plugin@marketplace": true` force-enables a plugin in managed settings (users cannot disable it); `false` blocks it.
- `strictKnownMarketplaces`: allowlist of marketplace sources (`[]` blocks **every** marketplace, including the official directory); `blockedMarketplaces` blocklists specific sources.
- `disableSideloadFlags: true`: blocks `--plugin-dir` CLI sideloading (which bypasses marketplace allowlists).
- `strictPluginOnlyCustomization: true` (or selective `strictCategories: ["hooks", "mcp"]`): blocks non-managed local/user/project customizations so only managed settings and approved plugins can define hooks, skills, agents, or MCP servers.

12. MCP in Claude Code

Scope (`--scope`)	Stored in	Precedence & sharing
`local` *(default for `claude mcp add`)*	`~/.claude.json` under `projects["<path>"].mcpServers`	**1st (highest)**; private to you in this project
`project`	`.mcp.json` at the repo root (committed to git)	**2nd**; shared with the team (prompts for approval before first use; approval + tool permissions persist in gitignored **`.claude/settings.local.json`**)
`user`	`~/.claude.json` top-level `mcpServers`	**3rd (lowest)**; private to you across all projects
Session (`--mcp-config`)	Passed via `claude --mcp-config <file>`	**Ephemeral**; **additive** with `local` / `project` / `user` unless paired with **`--strict-mcp-config`**
Plugin MCP	`<plugin>/.mcp.json` or inline in `plugin.json`	Lifecycle tied to plugin; tools named **`mcp__plugin_<plugin>_<server>__<tool>`**

- **MCP Architecture (M × N → M + N & 1:1 Client–Server rule) & CLI syntax:** An **MCP Host** (Claude Code) instantiates one dedicated **MCP Client** per connected **MCP Server** in a strict **1:1 relationship**. Add remote HTTP servers via `claude mcp add --transport http <name> <url> -s <scope>` and local stdio servers via `claude mcp add <name> -s <scope> -- npx -y @pkg/server` (`--` separates Claude flags from the subprocess command). Inside a session, use **`/mcp`** to inspect health or dynamically enable/disable servers to save context tokens.
- **Transports & `${VAR}` expansion:** Prefer **HTTP** (`--transport http`) for remote servers and **stdio** for local subprocesses (SSE is deprecated). `${VAR}` and `${VAR:-default}` expand in `command`, `args`, `env`, `url`, and `headers`. If `VAR` is unset with no default, Claude Code sends the literal `${VAR}` string (warning in `claude mcp list`). **Credential variable strip:** `ANTHROPIC_API_KEY` and `CLAUDE_CODE_*` auth variables resolve to **empty string** inside remote HTTP/SSE `url` / `headers` — always use a custom env var name like `${SEARCH_MCP_TOKEN}`.

- **Managed MCP: Provisioning vs Policy Filtering (two distinct mechanisms):** 1. **Exclusive file provisioning (`managed-mcp.json`):** Deploying `managed-mcp.json` at the OS system path gives IT **exclusive control** — users cannot add, modify, or use *any* MCP servers from `.mcp.json` , `~/.claude.json` , or plugins. (To provision remote `http` / `sse` org servers *without* wiping user servers, use `managedMcpServers` in server-managed or MDM settings). 2. **Policy filtering (`allowedMcpServers` / `deniedMcpServers` + `allowManagedMcpServersOnly`):** Configured in `managed-settings.json` . Each entry is an object matching by `{"serverName": "..."}` , `{"serverCommand": [...]}` (exact command+args array for stdio), or `{"serverUrl": "https://*.corp.com/*"}` (wildcard URL pattern for remote servers). `deniedMcpServers` always wins over `allowedMcpServers` . **Trap:** once *any* `serverUrl` entry appears in `allowedMcpServers` , remote HTTP/SSE servers must match a `serverUrl` pattern (`serverName` entries stop matching remote servers!). Setting `allowManagedMcpServersOnly: true` restricts the allowlist to the managed file while still merging `deniedMcpServers` from all layers.
- **MCP Tool Search (`ENABLE_TOOL_SEARCH`):** By default (`true` /unset), all MCP tool schemas are **deferred** behind `ToolSearch` so only tool names load at session start (mark high-frequency servers with `"alwaysLoad": true` in `.mcp.json` to preload their full schemas). Set `ENABLE_TOOL_SEARCH=auto` to defer only when MCP schemas exceed 10% of the context window (`auto:<N>` for custom `%`), or `false` to preload all MCP tools up front.

13. Headless, CI/CD (`claude -p`), Checkpointing and GitHub Actions

Flag	Effect & CI nuances
`-p` / `--print`	Non-interactive execution. **Default mode behavior:** plain `claude -p` (with no SDK `canUseTool` or `--permission-prompt-tool`) runs in `default` mode and **immediately auto-denies** any unapproved tool call (`PermissionRequest` hooks still get a chance to approve before denial unless mode is `dontAsk`).
`--bare`	**Clean-room CI mode:** skips auto-discovery of `CLAUDE.md` , auto-memory, hooks, plugins, skills, and `.mcp.json` ; ignores OAuth/keychain login (requires `ANTHROPIC_API_KEY` or cloud creds). Explicit `--settings <file>` , `--mcp-config <file>` , `--plugin-dir` , and `--add-dir` still work.
`--output-format text|json|stream-json`	`json` returns a single summary object (`result` , `session_id` , `total_cost_usd` , `subtype`); `stream-json` emits newline-delimited JSON events (`--include-partial-messages` streams token deltas; `--verbose` includes full turn events).
`--json-schema '<schema>'`	Enforces JSON Schema output after the agent finishes tool calls. **Trap:** the validated JSON object is returned in `.structured_output` (`jq '.structured_output'`), **not** `.result` ! Check `.subtype == "error_max_structured_output_retries"` on schema failure.
`--allowedTools` vs `--tools`	`--allowedTools "Edit(/src/**),Bash(npm test *)"` pre-approves matching calls (`permissions.deny` still wins); `--tools "Read,Grep"` restricts which built-in tools exist in context (`""` disables all built-ins).
`--max-turns <N>` / `--max-budget-usd <N>` / `--fallback-model <m>`	Caps agentic iterations (exits with `subtype: "error_max_turns"`), caps API spend, and auto-switches models when the primary model is overloaded (`529`).
`--system-prompt` vs `--append-system-prompt`	`--system-prompt` replaces the default Claude Code prompt **and drops the automatic git status/branch snapshot**; `--append-system-prompt` (or `--append-system-prompt-file`) preserves defaults + git snapshot.
`--continue` (`-c`), `--resume <id>` (`-r`), `--fork-session` , `--no-session-persistence` , `--worktree` (`-w`)	Continue/resume prior sessions, branch into a new session ID (`--fork-session`), skip writing transcripts to `~/.claude/projects/` (`--no-session-persistence`), or isolate the run in a git worktree (`-w`).

> **CRITICAL CI SECURITY TRAP (`-P` WITHOUT `--BARE`)**
> When running `claude -p` on an untrusted PR checkout *without* `--bare` , project `permissions.allow` rules and subagent frontmatter hooks are skipped, **but repository `.claude/settings.json` hooks and `.mcp.json` stdio servers execute automatically without a workspace trust dialog!** Always pass `--bare` (or `--settings '{"disableAllHooks": true}'` + `--strict-mcp-config`) when running `-p` against untrusted code.

- **GitHub Actions Integration (`/install-github-app` & `anthropics/claude-code-action`):**
- **Prerequisites & setup:** Run `/install-github-app` inside a local Git repo connected to a GitHub remote with **repo admin** rights and the **GitHub CLI (`gh`) installed and authenticated** (`gh auth login`). Generates `.github/workflows/claude.yml` (responds to `@claude` mentions in Issues/PR comments, updates a live to-do checklist comment, and **never commits directly to `main`** — creates isolated `claude/issue/<N>` branches) and `claude-code-review.yml` (automated PR review on open/push).
- **CI auth & context discovery trap:** Use `secrets.CLAUDE_CODE_OAUTH_TOKEN` (subscription) only for small/medium usage; use `secrets.ANTHROPIC_API_KEY` for high-frequency/multi-repo CI to avoid subscription rate limits. Because ephemeral `ubuntu-latest` CI runners lack local `CLAUDE.local.md` and `MEMORY.md` , any architecture/API contract needed by `@claude` in GitHub Actions **must be committed to `./CLAUDE.md` or `.claude/rules/`** .
- **Interactive Checkpointing & `/rewind` (`Esc+Esc`):** Automatically snapshots files modified by Claude Code's file-editing tools (`Edit` , `Write` , `NotebookEdit`) before each prompt, retained across sessions for **up to 30 days** with 3 restore modes: **(1) Restore code and conversation**, **(2) Restore conversation only**, and **(3) Restore code only** (reverts disk files while keeping failed attempt in chat context). **Critical trap:** `/rewind` and SDK `rewind_files()` **NEVER undo `Bash` side-effects** (`rm` , `mv` , `sed -i` , `npm install` , DB migrations) or manual IDE edits.

14. Code review and test generation in CI/CD

```
# Re-review after new commits: give prior findings so only new or unresolved issues are reported
claude -p "Review the diff in pr.diff using the criteria in review-criteria.md. \
Prior findings are in prior-findings.json: report only issues that are new or still unresolved." \
  --output-format json \
  --json-schema "$(cat review-schema.json)" \
  --allowedTools "Read,Grep,Glob" \
  | jq '.structured_output.findings'
```

Practice	Why
`-p` for every pipeline run	Without it Claude Code starts an interactive session and the job waits for input
`--output-format json` + `--json-schema` → read `.structured_output`	Machine-parseable findings (file, line, severity, issue, fix) your script can post as inline PR comments
Include prior findings on re-runs	Without them, each push re-posts the same comments; with them Claude reports only new or still-unaddressed issues
Independent review instance	The session that generated code keeps its own reasoning and is less likely to question it; review in a separate `claude -p` run that sees only the diff and criteria
Per-file passes + one integration pass for large PRs	Avoids attention dilution; see the Claude Agent SDK guide for decomposition patterns
Explicit criteria ("report bugs and security issues; skip style and naming") with severity definitions and examples	Vague "be conservative" prompts don't improve precision; false positives erode trust in every finding
Commit project context (`CLAUDE.md` with testing standards, fixtures, review criteria) and pass existing test files	CI runners only see committed files; showing existing tests prevents duplicate test scenarios
Blocking pre-merge check → synchronous `claude -p`. Overnight or weekly reports → Message Batches API	Batches cost less but can take up to 24 hours with no latency guarantee, so they suit only jobs nobody waits on

EXAM TRAPS

(1) Re-running review on every push without the earlier findings → duplicate PR comments. (2) Asking the generating session to "review your own work" instead of a fresh instance. (3) `--bare` skips `CLAUDE.md` auto-discovery, so pass review criteria explicitly (prompt or `--append-system-prompt-file`) when you need clean-room mode on untrusted code (§13).

15. Team rollout, distribution and spend controls

Managed delivery source	Precedence & OS path / mechanism	Key constraints
1. Server-managed / Gateway	`claude.ai` Admin Console (Teams/Enterprise) or self-hosted LLM Gateway	Fetched at startup + polled hourly (`forceRemoteSettingsRefreshInterval: true` fails closed if fetch fails). **Applies ONLY to `claude.ai` OAuth sessions** — bypassed if a user sets `ANTHROPIC_API_KEY`, runs `--bare`, or uses Bedrock/Vertex/Foundry (lock auth via `"forceLoginMethod": "claudeai"` + `"forceLoginOrgUUID"` in file/MDM policy). Delivers **one policy for the entire org**.
2. MDM / OS Registry	macOS plist (`com.anthropic.claudecode`) · Windows `HKLM\SOFTWARE\Policies\Anthropic\ClaudeCode`	Enforced by OS device management; supports per-group targeting via MDM device profiles.
3. File-based (`managed-settings.json` + `managed-settings.d/*.json`)	**macOS:** `/Library/Application Support/ClaudeCode/` **Linux / WSL:** `/etc/claude-code/` **Windows:** `C:\Program Files\ClaudeCode\` *(legacy `C:\Program Data\ClaudeCode\` is ignored)*	Drop-in files in `managed-settings.d/*.json` merge **alphabetically** on top of `managed-settings.json` (use `00-base.json`, `10-sec.json`). Verify active sources with `/status`.
4. Windows User Registry	`HKCU\SOFTWARE\Policies\Anthropic\ClaudeCode`	Lowest managed source.

- **Intra-managed source trap (`managedSourcesBehavior`):** By default, `managedSourcesBehavior` is **`"first-wins"`** — if Server-managed settings (Source 1) are non-empty, Claude Code **completely ignores** MDM (Source 2) and local `managed-settings.json` files (Source 3)! Set `"managedSourcesBehavior": "merge"` in the highest active managed source so Server, MDM, and file policies merge together.
- **The 3 `allowManaged...Only` lockdown keys:**
  - `allowManagedPermissionRulesOnly: true` — ignores `allow`, `ask`, and `deny` rules from user, project, local, `--settings`, and `--allowedTools`.
  - `allowManagedHooksOnly: true` — blocks user, project, and unforced plugin hooks (**SDK callback hooks and hooks from plugins force-enabled in managed `enabledPlugins` still run**).
  - `allowManagedMcpServersOnly: true` — restricts `allowedMcpServers` to managed settings (**`deniedMcpServers` still merges from all layers** so users can still block servers for themselves).

Authentication / Billing Setup	Visibility (`See spend`)	Hard Caps & Rate Protection (`Cap spend`)
Claude Subscription (Pro / Max 5x / Max 20x) *(Free tier excludes Claude Code)*	`/usage` (monitors rolling plan rate-limit windows; `/cost` is only for API key sessions)	Plan tier rate limits across rolling windows
Claude for Teams / Enterprise	Admin Console analytics + OpenTelemetry (`CLAUDE_CODE_ENABLE_TELEMETRY=1`)	Per-user/org spend limits in Admin Console
Claude Console (API keys)	`/cost` (per-session USD/token breakdown) + dedicated **"Claude Code"** workspace auto-created on first `claude login`	**Workspace spend limits & workspace rate limits** isolate developer usage from production API traffic
Bedrock / Vertex / Foundry	Cloud billing (`aws_account_id` / GCP project) + OTel metrics (`claude_code.cost.usage` , `claude_code.token.usage`)	Cloud budgets + **LLM Gateway (e.g. LiteLLM)** for per-user virtual keys, spend caps, and rate limits

16. Claude Code on Google Cloud (Vertex AI)

```
gcloud auth application-default login          # Application Default Credentials
export CLAUDE_CODE_USE_VERTEX=1
export CLOUD_ML_REGION=global                  # or a multi-region (us, eu) or region (us-east5)
export ANTHROPIC_VERTEX_PROJECT_ID=my-gcp-project
claude
```

Step / setting	Detail
GCP prerequisites	Enable the Vertex AI API, request access to the Claude models in **Model Garden**, and grant `roles/aiplatform.user` (needs `aiplatform.endpoints.predict`). A dedicated project simplifies cost tracking and access control
Setup wizard	Run `claude` → *3rd-party platform* → *Google Vertex AI*; it detects project/region, checks which models you can invoke and writes the `env` block of your user settings. Reopen with `/setup-vertex` (verify: requires a recent Claude Code version)
Regions	`CLOUD_ML_REGION=global` , a multi-region (`us` , `eu`) or a specific region. When using `global` , set `VERTEX_REGION_CLAUDE_*` to route models that have no global endpoint to a region
Optional	`ANTHROPIC_VERTEX_BASE_URL` (custom endpoint or gateway), `DISABLE_PROMPT_CACHING=1` , `ENABLE_PROMPT_CACHING_1H=1` (1-hour cache TTL, higher write cost), `CLAUDE_CODE_SKIP_VERTEX_AUTH=1` (gateway handles auth)
Behavior differences	`/logout` is unavailable (Google Cloud handles auth); MCP tool search is **off by default** (set `ENABLE_TOOL_SEARCH=true` on supported models); server-managed settings from `claude.ai` do not apply (use file or MDM managed settings)
Project ID override	`GCLOUD_PROJECT` , `GOOGLE_CLOUD_PROJECT` or the credentials file can override `ANTHROPIC_VERTEX_PROJECT_ID`
GitHub Actions	`anthropics/claude-code-action@v1` with `use_vertex: "true"` , authenticated through **Workload Identity Federation** (`google-github-actions/auth`) and a service account with the required Vertex AI role; no long-lived keys
Spend	Billed to the GCP project; track with Cloud Billing and Claude Code OpenTelemetry metrics, or an LLM gateway for per-user caps (§15)

EXAM TRAP

"Could not load the default credentials" means ADC isn't set up: run `gcloud auth application-default login` (or point to a service account) after setting the project and region variables. Changing `ANTHROPIC_API_KEY` doesn't help, because Vertex calls use Google Cloud credentials.

17. Failure modes and find the defect

Symptom / scenario	Root cause → correct fix	Tempting wrong answer
`Edit(.claude/**)` in `allow` or `bypassPermissions` still prompts on `.claude/settings.json`	Protected repo paths (`.git` , `.claude` , `.vscode` , `.idea` , `.husky` , `.mcp.json`) are checked **before** `allow` rules and even in `bypassPermissions`	Adding a broader `Edit(**)` allow rule
`Write(/etc/hosts)` slips past `deny: ["Write(/etc/**)"]`	File-write rules must use **`Edit(...)`** (`Write(...)` never matches), and a single `/` anchors to CWD/ `~/.claude/` → use **`Edit(//etc/**)`**	Changing the glob to `Write(/etc/*)`
`PreToolUse` hook on `Read` blocks `Read(.env)` , but typing `@.env` in prompt leaks `.env`	`@file` mentions are inlined before the prompt without calling `Read` → add **`permissions.deny: ["Read(./.env)"]`** (or `CLAUDE_CODE_DISABLE_FILE_MENTIONS=1`)	Switching the hook from `PreToolUse` to `UserPromptSubmit` regex
Guard hook exits `1` or times out and the destructive command still runs	Exit `1` (without JSON) and command/HTTP timeouts fail open → exit **`2`** with `stderr` reason and set **`"onFailure": "block"`** on the hook	Moving the guard to `PostToolUse`
Subagent with `disallowedTools: Bash(git push *)` loses all `Bash` access	Any specifier in subagent `disallowedTools` removes the **entire** underlying tool → keep `Bash` in `tools` and block `git push` via `permissions.deny` or `PreToolUse`	Wrapping the specifier in quotes

Symptom / scenario	Root cause → correct fix	Tempting wrong answer
Adding `CLAUDE.local.md` causes Claude Code to stop following `AGENTS.md`	Default `auto` project-instructions setting switches to `claude-md` only when `CLAUDE.local.md` exists → set `/config` Project instructions to `claude-md-and-agents-md`	Copying `AGENTS.md` into `.claudeignore`
Skill with `!`gh pr view`` aborts immediately in `default` mode, or `## When to Use` in `SKILL.md` body never triggers	`!`cmd`` never prompts (pre-approve `Bash(gh pr view *)` in `allow`), and text below `---` frontmatter is invisible before activation (put triggers in YAML `description`)	Adding `gh` to skill `allowed-tools` or adding more body bullets
`/rewind` (`Esc+Esc`) restores edited `.ts` files, but a file deleted via `Bash` (`rm`) stays gone	Checkpointing only tracks Claude Code's file-editing tools (`Edit`, `Write`, `Notebook Edit`), never `Bash` side effects → use Git or block `Bash(rm *)`	Choosing "Restore code and conversation" instead of "Restore code only"
`@claude` in GitHub Actions breaks an API contract documented in `CLAUDE.local.md`, or `claude --mcp-config f.json` still loads user MCPs	Ephemeral CI runners only see committed files (**commit `CLAUDE.md`**), and `--mcp-config` is additive unless paired with **`--strict-mcp-config`**	Enabling local auto-memory or deleting `.mcp.json`
Plugin SQLite DB inside `${CLAUDE_PLUGIN_ROOT}` is wiped on update, or update never fetches new commits	`${CLAUDE_PLUGIN_ROOT}` is wiped on update and `plugin.json` `version` wasn't bumped → store state in **`${CLAUDE_PLUGIN_DATA}`** and bump `version` in `plugin.json`	Putting `"version"` only in `marketplace.json`
Remote HTTP MCP server blocked despite `{"serverName": "sentry"}` in `allowedMcpServers`	Once *any* `serverUrl` entry exists in `allowedMcpServers`, remote servers must match a `serverUrl` pattern → add `{"serverUrl": "https://*.sentry.io/*"}`	Adding `mcp__sentry__*` to `permissions.allow`
`claude -p --json-schema` returns empty `.result` in CI, or untrusted PR executes repo hooks	Schema output is in **`.structured_output`** (not `.result`), and `-p` without `--bare` auto-runs repo `.claude/settings.json` hooks → read `.structured_output` and pass **`--bare`**	Running `claude -p` in `default` mode without `--bare`
Local `managed-settings.json` on macOS ignored after admin enables Server-managed settings	`managedSourcesBehavior` defaults to `"first-wins"` (Server > MDM > File) → set **`"managedSourcesBehavior": "merge"`** in the server-managed config	Moving the file to `/etc/claude-code/` on macOS

Find the defect: a team's "secure" setup. A team commits the following and believes `.env` is completely blocked, warehouse MCP tools are auto-approved, and their CI JSON schema script is safe. Find four defects.

```
// .claude/settings.json
{
  "permissions": { "allow": ["mcp__*", "Edit(.claude/**)"], "deny":
["Write(/.env*)"] },
  "hooks": { "PreToolUse": [{ "matcher": "Read",
    "hooks": [{ "type": "command", "command":
"/Users/alex/repo/.claude/hooks/block-env.sh" }] }] }
}
# .claude/hooks/block-env.sh
f=$(jq -r '.tool_input.file_path')
if [[ "$f" == *.env* ]]; then echo "blocked" >&2; exit 1; fi
// .mcp.json
{ "mcpServers": { "warehouse": { "type": "http", "url":
"https://wh.internal.example.com/mcp",
    "headers": { "Authorization": "Bearer ${ANTHROPIC_API_KEY}" } }
} }
# CI script
claude -p "Extract release notes" --json-schema
'{"type":"object","properties":{"tag":{"type":"string"}}}' --
output-format json | jq -r '.result.tag'
```

1. **Broken permission rules:** `"mcp__*"` in `allow` is unanchored and skipped (use `"mcp__warehouse__query"`); `"Edit(.claude/**)"` in `allow` never bypasses protected-path prompts on `.claude/`; and `"Write(/.env*)"` in `deny` **never matches** because file-write rules must use `Edit(...)` (and `Read(./.env*)` in `deny` is needed to block both `Read` / `Edit` / `Write` and `@.env` prompt mentions).
2. **Non-blocking, non-portable hook:** `block-env.sh` exits `1` (non-blocking fail-open) instead of `2`, lacks `"onFailure": "block"`, misses `@.env` prompt mentions, and hardcodes `/Users/alex/...` instead of `${CLAUDE_PROJECT_DIR}`.
3. **Stripped credential variable in `.mcp.json`:** `ANTHROPIC_API_KEY` resolves to an **empty string** in remote MCP `url` / `headers` (causing `401 Unauthorized`). Reference a custom env var (`"Bearer ${WAREHOUSE_MCP_TOKEN}"`).
4. **Wrong CI JSON field (`-p --json-schema`):** `--json-schema` places validated JSON in **`.structured_output.tag`**, not `.result.tag` (and untrusted PR CI runs should also pass `--bare` + `--permission-mode dontAsk`).

18. Glossary

Term	Definition
Permission evaluation order	`PreToolUse` hooks → `deny` → `ask` → protected/critical-path checks → `allow` → permission mode → `PermissionRequest` hooks → prompt / `canUseTool`
Protected paths	`.git`, `.claude`, `.vscode`, `.idea`, `.husky`, `.mcp.json`, shell profiles — checked before `allow` rules and still prompt in `bypassPermissions`
`Edit(//path)` vs `Read(/path)`	Only `Read` and `Edit` take path globs (`Edit` covers `Edit` / `Write` / `NotebookEdit`); `//` = filesystem root, single `/` = `<cwd>` or `~/.claude/`
Auto memory (`MEMORY.md`)	Claude-written notes at `~/.claude/projects/<project>/memory/MEMORY.md` (first 200 lines / 25 KB loaded at startup; disable via `CLAUDE_CODE_DISABLE_AUTO_MEMORY=1`)
`PreToolUse` decisions	`hookSpecificOutput.permissionDecision`: **`deny`** > **`defer`** > **`ask`** > **`allow`** across parallel hooks, plus `updatedInput` and `additionalContext`
`"onFailure": "block"`	Hook setting that converts non-zero exits (like `exit 1` without JSON), HTTP errors, and timeouts from fail-open warnings into hard blocks
Fork (`/subtask`) vs Subagent	Standard subagents get a fresh context window; a fork inherits parent conversation history, tools, model, and prompt cache (up to depth 2)
`${CLAUDE_PLUGIN_ROOT}` vs `DATA`	`ROOT` is the read-only versioned cache wiped on update; `DATA` (`~/.claude/plugins/data/<id>/`) persists state across plugin updates
`managed-mcp.json` vs `allowedMcpServers`	`managed-mcp.json` provisions an exclusive system-wide server set; `allowedMcpServers` / `deniedMcpServers` filter servers by `serverName`, `serverCommand`, or `serverUrl`
`--bare` (`claude -p`)	Clean-room headless flag that skips `CLAUDE.md`, auto-memory, hooks, plugins, skills, `.mcp.json`, and OAuth/keychain
`managed` Sources Behavior	Intra-managed rule: defaults to `"first-wins"` (`Server-Gateway > MDM/HKLM > File + managed-settings.d/*.json > HKCU`); set `"merge"` to combine
Deep Agent & `TodoWrite`	4-pillar agent harness (`TodoWrite` task anchor, isolated subagents, filesystem context engine, system prompt heuristics) preventing shallow ReAct context rot
`/rewind` (`Esc+Esc`)	Restores files (`Edit` / `Write` / `NotebookEdit`) and/or conversation (30-day retention), but **never undoes `Bash` side effects** (`rm`, installs, migrations)
`/output-style` & `/statusline`	`/output-style` swaps the SWE system prompt format (`Default` / `Explanatory` / `Learning` or `.claude/output-styles/*.md`); `statusLine` runs a 0-token footer script reading JSON & `transcript_path` on `stdin`
`&` prefix & `/teleport`	`<prompt>` dispatches a task to an Anthropic cloud container (`claude.ai/code`); `/teleport` (`--teleport`) pulls the cloud session and branch back locally