Claude Code Handbook

Flavio Copes

Updated November 24, 2026

Contents

How Claude Code actually works . . . 2
Pick your interface . . . 9
Inside the Claude desktop app (Code tab) . . . 9
Install, auth, updates and migrating . . . 16
Models, effort, fast mode, advisor and output styles . . . 20
Interactive CLI mastery and session lifecycle . . . 24
Real workflows from start to finish . . . 28
Memory, rules and the .claude directory . . . 32
Permissions, Auto Mode and the Bash sandbox . . . 38
Five ways to run agents in parallel . . . 45
Goals, scheduled tasks, channels and deep links . . . 51
Skills: package repeated workflows . . . 53
Hooks: deterministic automation in the turn lifecycle . . . 55
MCP servers and connectors . . . 61
Plugins, code intelligence, evals, Code Mode and Mods . . . 62
Code review and security scanning . . . 66
Browsers, Computer Use, Cloud, Mobile and Artifacts . . . 68
Headless CLI, CI/CD, Agent SDK and teams . . . 73
Troubleshooting and when Claude gets stuck . . . 76
How Claude Code compares and how I use it . . . 79

Preface

This is the offline handbook edition of Claude Code Handbook, written by Flavio Copes.

You can read the online version with interactive diagrams and videos at https://flaviocopes.com/claude-code-guide/, and find more programming tutorials, courses, and handbooks at https://flaviocopes.com.

Claude Code is Anthropic's coding agent. Tell it what you want changed in a repository, and it reads the files, searches the codebase, edits code, runs your tests or build, reads any failure output, and iterates until the check passes.

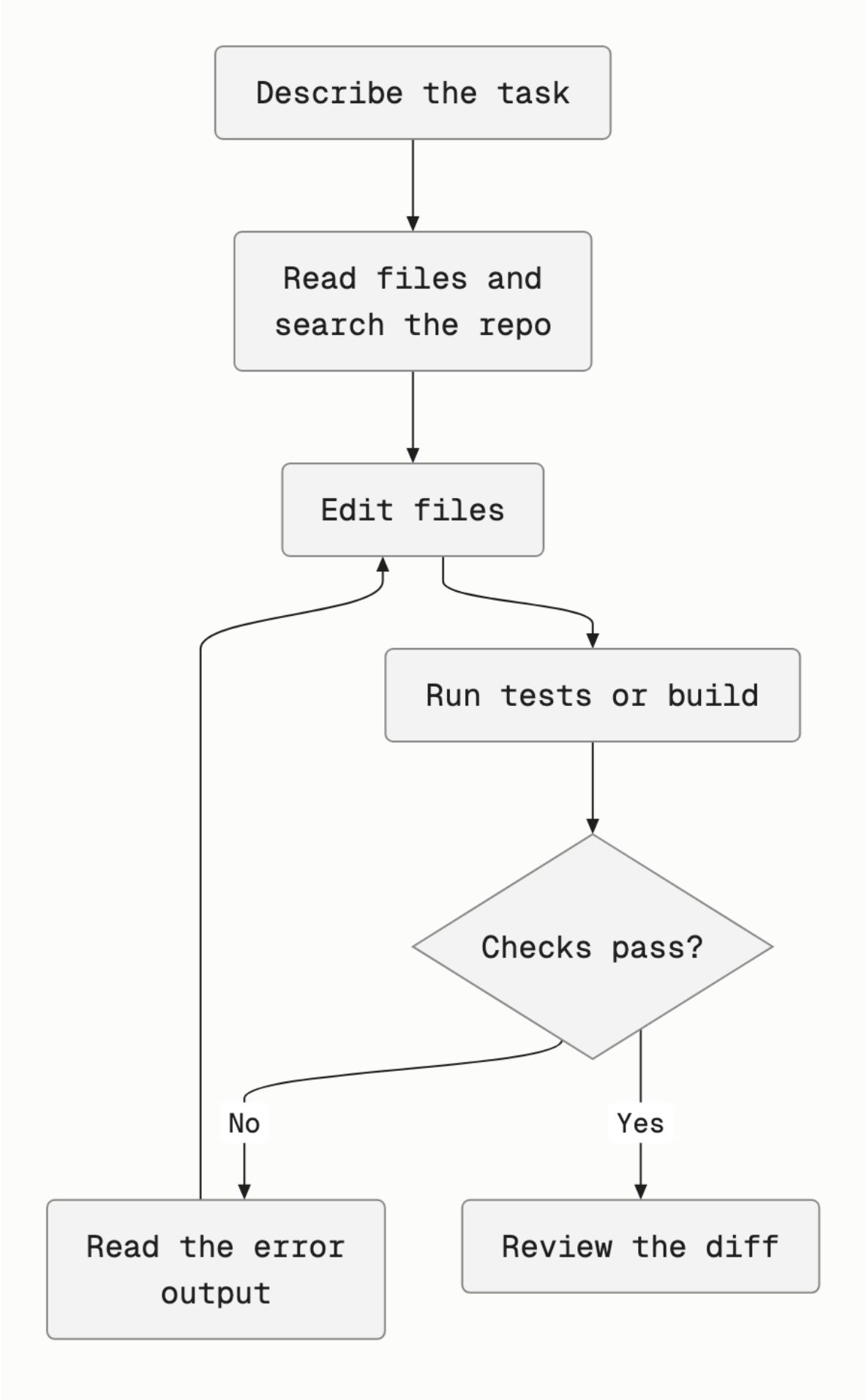
You can run it from a terminal, from the Claude desktop app, inside VS Code or JetBrains, in a cloud browser session, or from your phone. Under the hood, every surface runs the same agent loop against real files and real commands.

I use Claude Code every day on my Mac to maintain this website, build native macOS apps, write scripts, and run automated checks. Every CLI command, JSON measurement, hook trace, and terminal screenshot in this guide was captured on Claude Code v2.1.330.

How Claude Code actually works

In a regular chat window you paste snippets in and copy suggestions back out. You are the glue between the model and your filesystem.

Claude Code removes that copy-paste loop by giving Claude direct tools inside your working directory. When you submit a prompt, Claude Code enters an agentic loop: it inspects files, searches for definitions, applies targeted edits, executes commands such as your test suite or linter, reads the output, and keeps going until the task finishes or it needs a decision from you.


Describe the task
Read files and search the repo
Edit files
Run tests or build
Checks pass?
No
Yes
Read the error output
Review the diff

You still own the review. Passing tests won't catch a misunderstood requirement, an unnecessary rewrite of an adjacent module, or an outdated API pattern Claude pulled from memory.

The built-in toolset

Every Claude Code session comes with a set of built-in tools. You don't need to install anything for Claude to read code, edit files, or run commands. Knowing what tools exist helps you write sharper permission rules and understand what Claude is actually doing during a turn.

Tool	What it does	Prompts by default?
`Read`	Reads files, images, PDFs, and notebooks on disk	No
`Edit`	Applies targeted search-and-replace edits to an existing file	Yes
`Write`	Creates a new file or overwrites an existing one	Yes
`Bash`	Runs shell commands, build scripts, git, and tests	Yes
`Glob` / `Grep`	Searches file paths and file contents (via ripgrep/shell on Unix, built-in where configured)	No
`WebFetch` / `WebSearch`	Fetches a URL or runs a web search for current docs	Yes
`LSP`	Queries language servers for definitions, references, and live type errors	No
`Agent`	Spawns an isolated subagent (or teammate when Agent Teams is enabled)	No
`Workflow`	Executes a JavaScript dynamic workflow that orchestrates many subagents	Yes
`EnterWorktree` / `ExitWorktree`	Creates or switches into an isolated Git worktree mid-session	Some paths / removals
`Monitor`	Runs a background command or WebSocket stream and feeds incoming events back into the turn	Yes
`CronCreate` / `CronList` / `CronDelete`	Manages session-scoped recurring or one-shot prompts (`/loop`)	No
`RemoteTrigger`	Creates and manages cloud scheduled tasks on `claude.ai` (`/schedule`)	No

Tool	What it does	Prompts by default?
`ListAgents` / `SendMessage`	Lists and messages your other Claude Code sessions, teammates, or subagents	No
`Skill`	Loads and runs a skill's `SKILL.md` inside the session	Some skills
`ToolSearch`	Loads deferred MCP tool schemas on demand when a task needs them	No
`Artifact`	Publishes an interactive HTML or Markdown page to `claude.ai`	Yes
`AskUserQuestion`	Pauses to ask structured multiple-choice questions when requirements are ambiguous	No

One subtle detail that surprises people: Claude Code doesn't dump every tool description into every turn identically. On supported models, `ToolSearch` defers MCP server tool schemas until Claude actually searches for them, which keeps startup context small even when you connect dozens of external services.

Context window and prompt caching

Every turn in Claude Code sends the conversation prefix back to the API: the system prompt, built-in tool definitions, your `CLAUDE.md` files, your skill index, and every prior message and tool result in the session.

On Opus 5.5, Sonnet 5.5, Haiku 5.5, and Fable 5.2, the context window is **1M tokens**. Run `/context` inside a fresh session to see exactly what occupies space before you even ask your first question:



Look closely at those real numbers from my Mac on Opus 5.5:

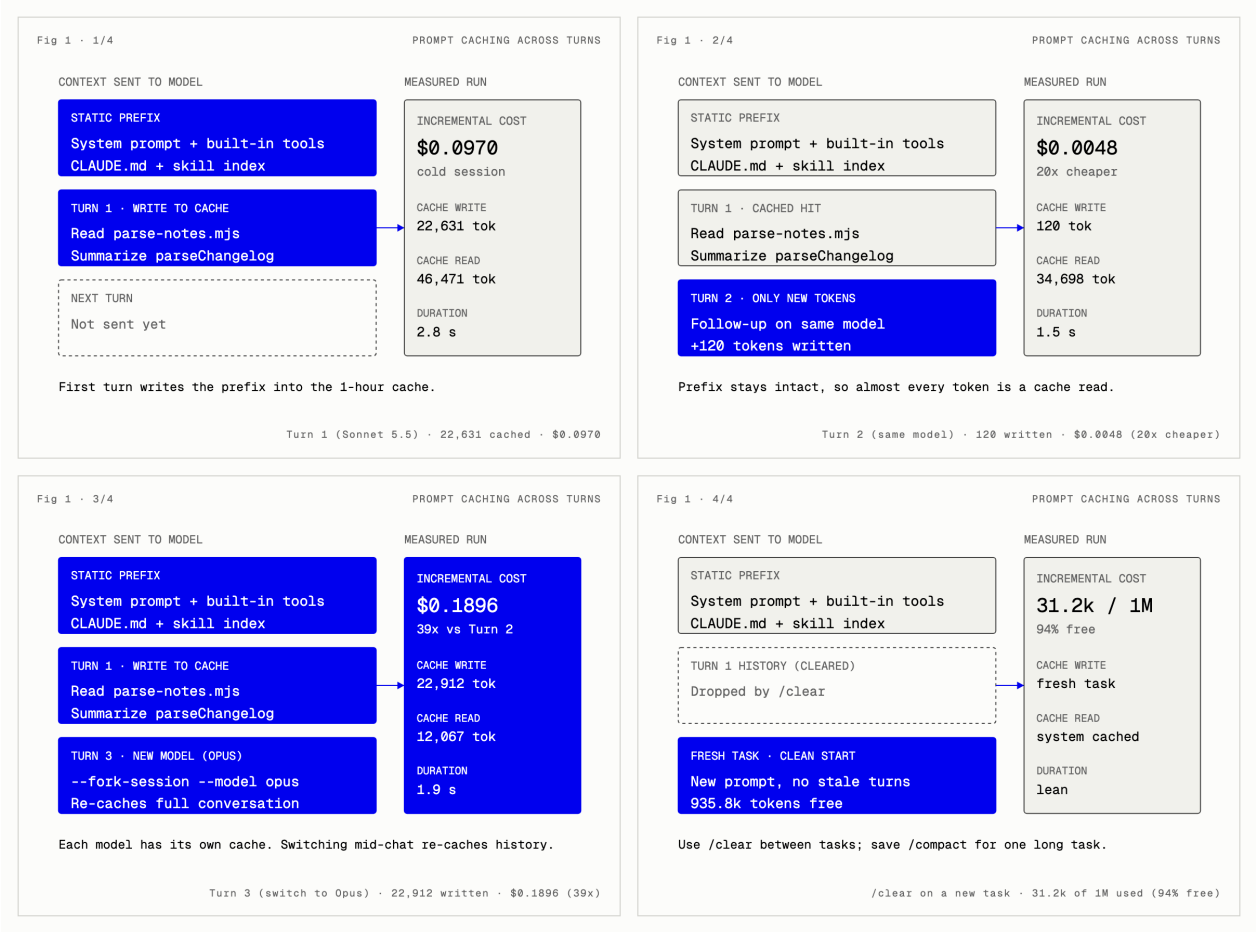
- **System prompt:** 4.4k tokens (0.4%)
- **System tools:** 14.6k tokens (1%)
- **MCP server instructions:** 61 tokens (<0.1%), even with **110 MCP tools** configured in that folder, because MCP schemas load on demand
- **Memory files (`CLAUDE.md`):** 2.1k tokens (0.2%)
- **Skills index:** 10k tokens (1%) across 109 available skills (user, synced, and plugin skills combined)
- **Autocompact buffer:** 33k tokens (3%) reserved so compaction can run cleanly
- **Free space:** 935.8k tokens (94%)

If Claude Code had to re-compute all of those tokens from scratch on every turn, long sessions would be painfully slow and expensive. Instead, Anthropic caches the static prefix on the server (with an ephemeral 1-hour cache in standard Claude Code runs). As long as the prefix stays unchanged, subsequent turns pay a fraction of the cost and run much faster.

To see the exact numbers in practice, I ran a three-turn benchmark in a small Node.js changelog-parser repo (`release-notes`) using `claude -p --output-format json`. The first run started fresh on Sonnet 5.5 and read `parse-notes.mjs`. Next, I resumed that same session (`-r <session-id>`) with a follow-up question on the same model. Finally, I forked the session (`--fork-session`) and switched mid-conversation to Opus 5.5:

Turn	Command pattern	Cache write	Cache read	Output tokens	Duration	Incremental cost (API list)
Turn 1 (cold start)	`claude -p --model sonnet --effort low`	22,631	46,471	171	2.8 s	$0.0970
Turn 2 (same model)	`claude -p -r <id> --model sonnet --effort low`	120	34,698	87	1.5 s	$0.0048
Turn 3 (switch to Opus)	`claude -p -r <id> --fork-session --model opus --effort low`	22,912	12,067	196	1.9 s	$0.1896

Look at the difference between Turn 2 and Turn 3. Continuing on the same model wrote just **120 tokens** to the cache and cost less than half a cent (**20x cheaper** than Turn 1). Switching from Sonnet to Opus mid-conversation invalidated the model cache, forced a **22,912-token** re-write, and cost **$0.1896** for a single one-sentence answer (**39x more expensive** than Turn 2).



Here is what breaks or preserves your prompt cache in Claude Code:

- **Switching models (`/model`) invalidates the cache.** Every model keeps its own cache. When your cache is still warm, `/model` warns you before switching. Using the `opusplan` model setting also switches models every time you move between Plan mode and execution.
- **Changing effort preserves the cache on current models.** On Opus 5.5, Sonnet 5.5, Haiku 5.5, and Fable 5.1+, changing `/effort` mid-session keeps the cached prefix intact on Anthropic subscriptions and API keys (older models and some third-party cloud providers still invalidate on effort change).
- **Toggling `/fast` uses a separate cache.** Fast mode and standard speed don't share caches, so enable `/fast` at the start of a session rather than halfway through a 150k-token chat.
- **Toggling `/advisor` preserves the cache.** Its server tool definition sits after the cache breakpoint.
- **Adding or removing MCP servers keeps the cache when tool search is on.** Because deferred MCP tools load on demand, connecting a new server mid-session doesn't disturb the cached prefix.
- **Compacting (`/compact`) resets the cached history.** The summary replaces older turns, so the next request writes a new shorter prefix once and then stays cheaper afterward.

Two CLI flags exist specifically for prompt-cache engineering in scripts and shared environments: `--system-prompt-snapshot on` (the default for interactive and cloud sessions) records the initial system prompt verbatim so resuming a session never drifts and breaks the cache, and `--exclude-dynamic-system-prompt-sections` moves per-machine fields (working directory, git status, memory paths) into the first user message so teams can share a warm system-prompt cache across machines. I cover prompt caching mechanics in more detail in how prompt caching works in

coding agents.

Pick your interface

Every Claude Code interface runs the same agent loop and reads your `CLAUDE.md`, settings, hooks, and MCP config (cloud sessions read what's committed to the repo plus cloud connectors). Pick the surface that matches what you're doing.

Interface	Where the code runs	Best for
Claude desktop app (Code tab)	Local folder, git worktree, cloud, or SSH host	Visual side-by-side diffs, inline comments, live previews, and PR tracking
Claude Code CLI (`claude`)	Your terminal or a remote server over SSH	Daily work, scripts, git worktrees, Agent View, Code Mode, and fast iteration
VS Code / Cursor extension	Your local project	Inline diffs, `@`-mentions from open tabs, and plan review beside your code
JetBrains plugin	Your local project	IntelliJ, PyCharm, WebStorm, and GoLand users who stay inside the IDE
Cloud sessions (`claude.ai/code` or `--cloud`)	Anthropic's remote environment or self-hosted runners	Tasks that run from GitHub without your laptop awake
Mobile app (iOS, Android) + Remote Control	Cloud session or your awake computer	Approving prompts, steering a task, or firing off a fix away from the desk
Slack / Claude Tag	Cloud sessions triggered from Slack channels	Delegating bug reports or PR fixes straight from team discussions

VS Code and JetBrains extensions

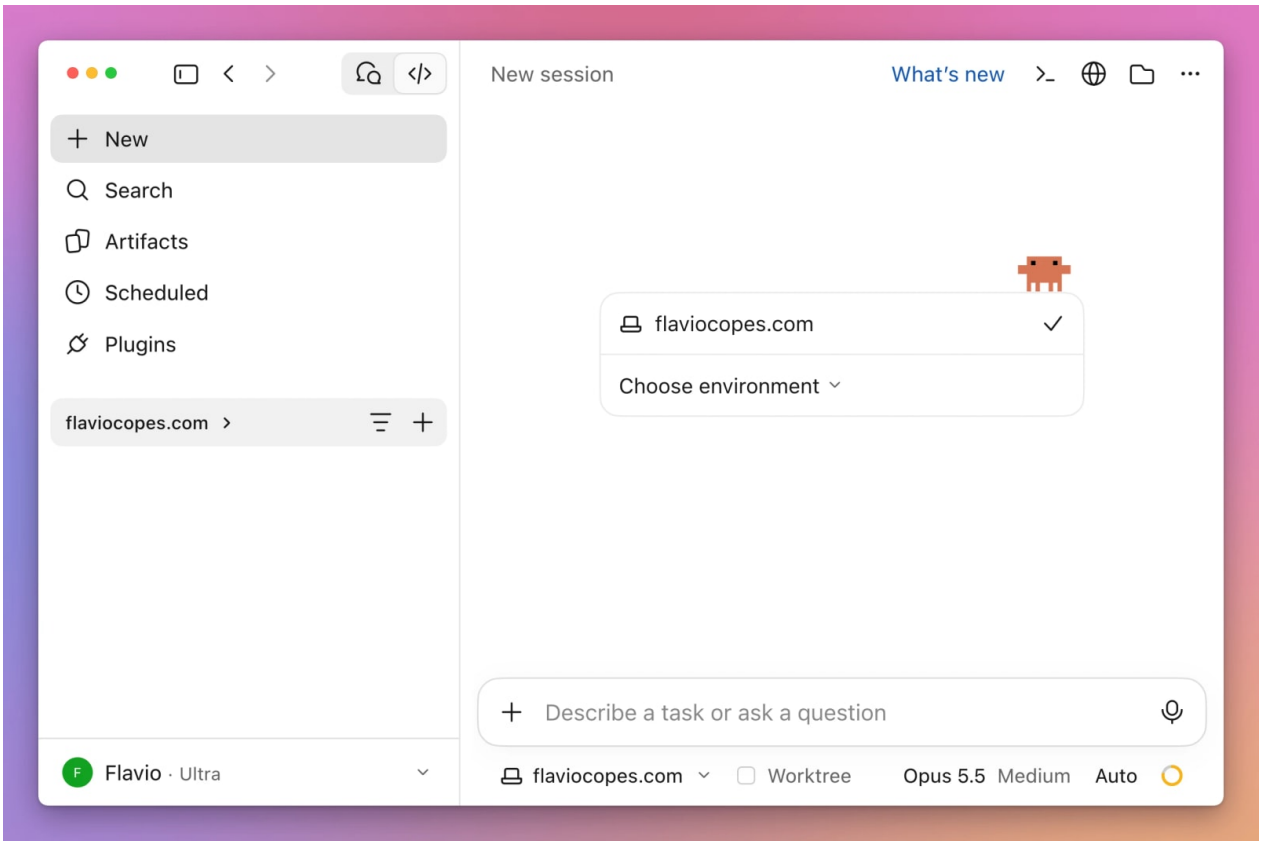
If you keep an editor open all day, the official VS Code extension (which also works in Cursor and other VS Code forks) and the JetBrains plugin put Claude Code in a panel beside your code. You get inline diffs in editor tabs, `@`-mentions that pull in your current file or selected lines, `/fork` and `/workflows` right in VS Code, and interactive plan review. Both extensions read the exact same `~/.claude/settings.json`, `.claude/settings.json`, skills, hooks, and MCP servers as the CLI.

You can also link a running CLI session to an open IDE automatically with `claude --ide` when one supported IDE window is open.

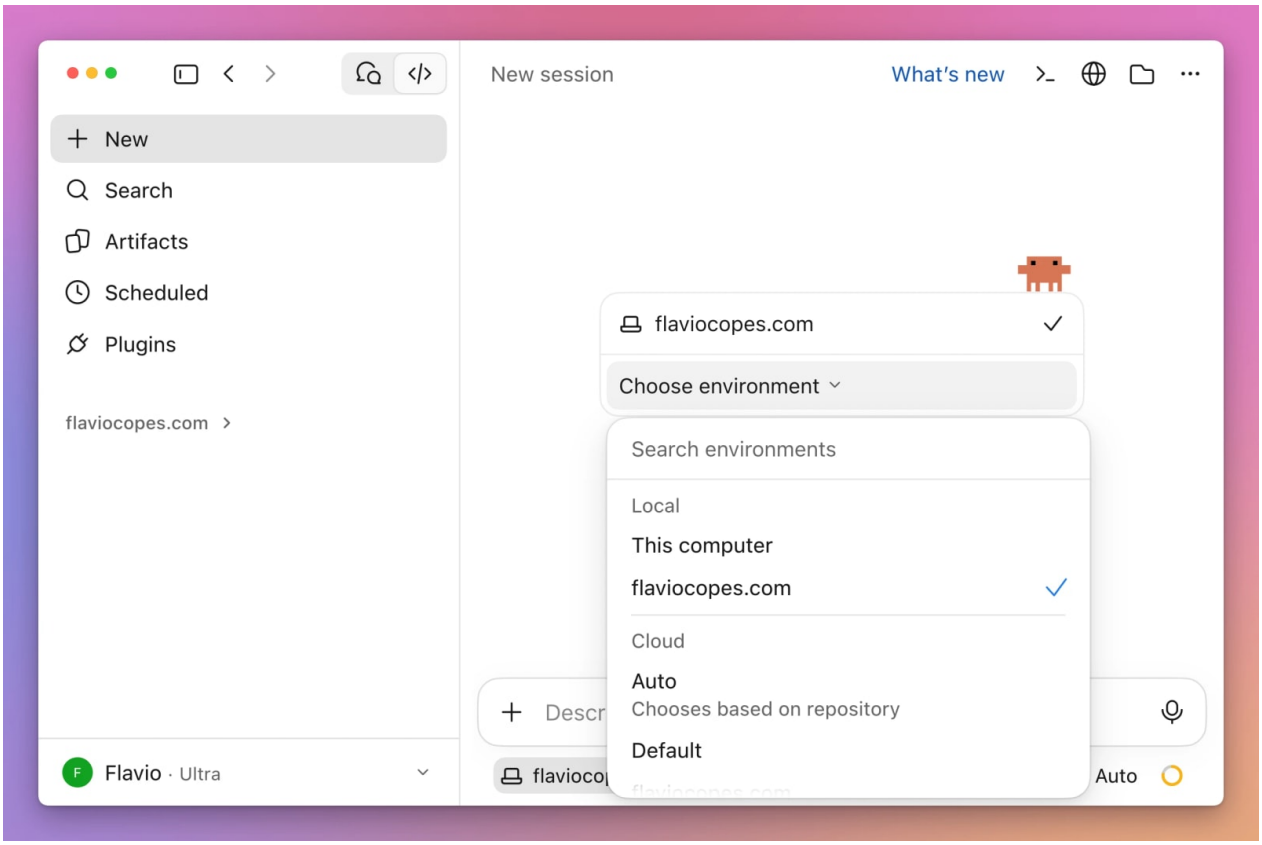
Inside the Claude desktop app (Code tab)

Many people assume Claude Code is a terminal-only tool. If you prefer visual diffs, live app previews, side-by-side panes, and clicking through pull requests without memorizing CLI flags, the **Code** tab (`</>`) inside the Claude desktop app (macOS, Windows with WSL 2, and Linux beta on Ubuntu/Debian) is a full workspace built around the same agent loop.

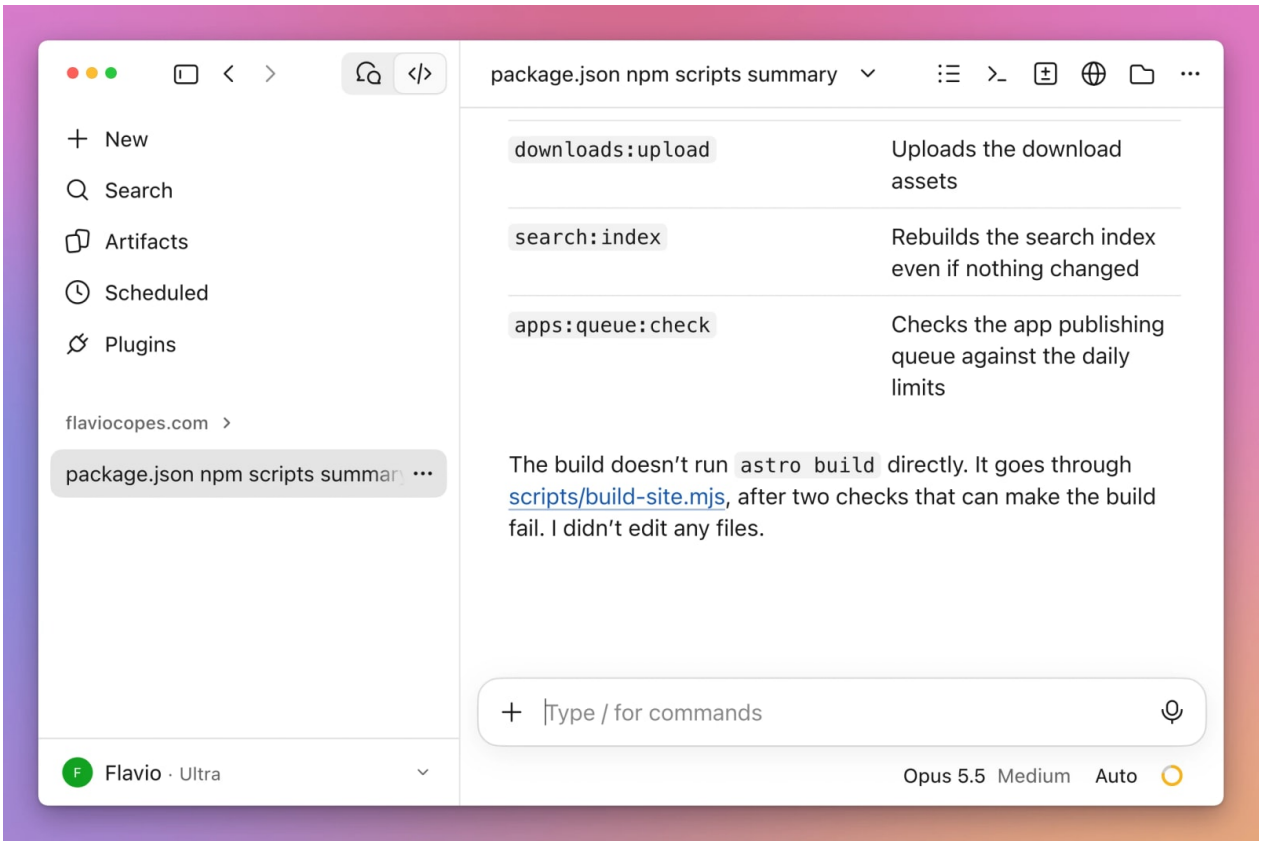
Open the app and switch to **Code**. The left sidebar gives you **New**, **Search**, **Artifacts**, **Scheduled** tasks, **Plugins**, and grouped session lists per repository. Along the bottom of the composer, you pick the project folder, toggle an isolated **Worktree**, choose your model and effort level, set the permission mode, click + to attach images, PDFs, skills, or connectors, or click the microphone for voice dictation:



Open **Choose environment** (or press `Cmd+Option+O` to create a saved environment) to run locally on your machine, connect over SSH to a remote Linux/macOS/Windows box, run inside WSL 2 on Windows, or hand the session to Anthropic's cloud:



Once a task is running, Claude links directly to file paths it reads or edits, shows live context and changes in the session details rail, and gives you top-right controls for background tasks, an integrated terminal, Changes diffs, and the Browser preview pane. Here I asked Desktop to summarize the npm scripts in this site's `package.json` without editing any files:



You can also steer a running turn without aborting it: type a correction while Claude is working, and it reads your message right after the current tool finishes.



Arrange panes, pop out windows and minimize

The Code tab organizes work into draggable panes: **Chat**, **Changes** (diff), **Browser**, **Terminal**, **Files** (built-in editor), **Plan**, **Background tasks**, **Subagent**, and on macOS the **iOS Simulator**.

- Drag any pane by its tab or handle to reorder or resize your layout, or press `Cmd+\` (`Ctrl+\` on Windows) to close the focused pane.
- **Pop any pane into its own window** (for example, keeping the Browser preview or Changes

diff on a second monitor) while Claude keeps working in the main window, and dock it back later.
- **Minimize** the Terminal, Browser, iOS Simulator, or a Side Chat so it sits tucked in the session rail (or floats neatly in the corner of the chat) until you need it again.

Live app previews and `.claude/launch.json`

In a local Desktop session, Claude doesn't just edit HTML and CSS blind. It starts your dev server in the built-in **Browser** pane (`Cmd+Shift+B`), takes screenshots, inspects the DOM, clicks buttons, reads console errors, and iterates until the page looks and behaves right. You can also click any HTML, PDF, or video path in the chat to preview it in the pane, press `Cmd+Shift+S` to pick a specific DOM element to reference in your prompt, and toggle **Keep cookies** in the pane's `...` menu so you stay logged in across restarts.

Claude writes preview server commands to `.claude/launch.json` at the root of your project folder. Here is the real `.claude/launch.json` sitting in this website's repository so Desktop can spin up either the Astro dev server or a built preview on dedicated ports:

```
{
  "version": "0.0.1",
  "configurations": [
    {
      "name": "astro",
      "runtimeExecutable": "npx",
      "runtimeArgs": ["astro", "dev", "--port", "4399"],
      "port": 4399
    },
    {
      "name": "dist-preview",
      "runtimeExecutable": "npx",
      "runtimeArgs": ["astro", "preview", "--port", "4401"],
      "port": 4401
    }
  ]
}
```

By default, **Auto-verify changes** is on, so Claude checks the running app automatically after edits. Add `"autoVerify": false` to `.claude/launch.json` (or turn it off in the Browser pane's `...` menu) if you prefer triggering visual verification manually.

Visual diff review, inline comments and `/code-review` cards

Open the **Changes** pane (`Cmd+Shift+D`, or click the `+12 -1` diff count in the session details panel) to inspect every edited file.

Instead of describing line numbers in chat, hover over any line in the diff, click the + button beside the line number, type your note, and press `Enter`. Add notes across as many lines and files as you want, then press **`Cmd+Enter`** (`Ctrl+Enter` on Windows) to submit all of them in one batch. Claude reads every inline comment, updates the code, and refreshes the diff.

Running `/code-review` inside a Desktop local, SSH, or WSL session also produces an interactive **Code review** card grouped by file. Click **Walk through in Changes** to jump line by line

through each finding with **Fix this one** or dismiss buttons, or click **Apply fixes** to resolve every open finding at once.

PR status monitoring, Auto-fix and Merge when ready

Once you open a pull request from Desktop (requires `gh` authenticated locally, or on the remote host for SSH sessions), the PR appears in the **Changes** card alongside live CI check status. Open the ... menu beside the PR title for two powerful toggles:

- **Auto-fix:** Claude watches the PR, fixes failing CI checks automatically, and in local sessions responds to new review comments left by collaborators, owners, or GitHub Apps.
- **Merge when ready:** Enables GitHub auto-merge (squash merge by default unless the repo uses a merge queue) once required checks pass. Pair it with **auto-archive** in Settings so finished PR sessions archive themselves cleanly.

Side chats, suggested tasks and working across Desktop sessions

Three small Desktop workflows save a surprising amount of mental overhead:

- **Side chats (`Cmd+;` or `/btw`):** Opens a side panel that can read everything in your main session up to that point so you can ask "why does this middleware skip OPTIONS?" without adding detours to the main conversation history.
- **Suggested task cards:** When Claude spots a separate issue outside your current task, it surfaces a card with **Start** (spawns a child session in its own thread/worktree), **Do it here**, **Options** (pick model/worktree/permissions first), or **Start all** when several follow-ups are listed.
- **Work across sessions:** Ask Claude in plain language about your other Code tab sessions: *"which session touched the auth refactor?"*, *"tell the API session the payload field changed"*, or *"archive the sessions whose PRs merged"*. Desktop routes messages between sessions with attributed cards, while still respecting each receiving session's permission rules.

Testing iOS apps in the iOS Simulator pane

If you build iOS apps on macOS (with Xcode 26 or Xcode 27 installed), Desktop includes a dedicated **iOS Simulator** pane (public beta). Whenever Claude builds, installs, or tests your app in a simulator, the pane opens automatically beside the chat and streams the device screen live. You can tap through the simulator yourself while Claude works, and on Xcode 27.1+ you can even test folding layouts on an iPhone Duo.

Because Desktop drives Apple's simulator directly, it doesn't require Computer Use and never hijacks your mouse cursor or hides your other windows.

SSH sessions, WSL 2, Desktop Computer Use and Scheduled Tasks

Desktop isn't limited to local macOS folders:

- **SSH sessions:** Add any Linux, macOS, or Windows server in **New environment > SSH** (reusing `~/.ssh/config` or an SSH key, with an optional remote worktree folder). Desktop automatically installs Claude Code on the remote host on first connect and saves the machine + folder as a reusable environment (`Cmd+Option+1-9`). Remember that Browser pane preview tools don't run on remote SSH hosts (forward ports instead), and you need `git` + `gh` installed on the remote box to open PRs there.

- **WSL 2 & Linux (beta):** On Windows, Desktop runs Code sessions inside your WSL 2 distribution natively; on Ubuntu and Debian, the Linux beta desktop app supports local Code sessions (without Computer Use yet).
- **Desktop Computer Use:** On macOS and Windows, enable Computer Use in Settings when Claude needs to drive GUI-only apps that have no CLI or API, with per-app approvals. (Users who previously enabled Cowork's **Dispatch** can also route phone tasks into local Code sessions from the sidebar, though Dispatch is closed to new sign-ups.)
- **Desktop Scheduled Tasks:** Click **Scheduled** in the sidebar to set up recurring tasks on your computer with full access to local files, local MCP servers, and per-task permission settings.

Essential Desktop keyboard shortcuts

Press `Cmd+/` on macOS (`Ctrl+/` on Windows and Linux) anytime to pull up the full cheat sheet. The shortcuts worth learning first:

Shortcut (macOS)	Windows / Linux	Action
`Cmd+N` / `Cmd+W`	`Ctrl+N` / `Ctrl+W`	New session / close session
`Cmd+Shift+D`	`Ctrl+Shift+D`	Toggle Changes diff pane
`Cmd+Shift+B`	`Ctrl+Shift+B`	Toggle Browser preview pane
`Cmd+Shift+S`	`Ctrl+Shift+S`	Select an element in the Browser pane
‘Ctrl+“	‘Ctrl+“	Toggle integrated Terminal pane
`Cmd+;`	`Ctrl+;`	Open a Side chat (`/btw`)
`Cmd+\`	`Ctrl+\`	Close focused pane
`Cmd+Enter`	`Ctrl+Enter`	Submit batched inline diff comments
`Cmd+Shift+M` / `I` / `E`	`Ctrl+Shift+M` / `I` / `E`	Open Permission mode, Model, or Effort menu
`Cmd+Option+1-9`	`Alt+1-9`	Select saved environment on a new session
`Ctrl+O`	`Ctrl+O`	Cycle view modes (including Verbose)

One small habit trip-up if you bounce between terminal and Desktop: **`Shift+Tab` does not cycle permission modes in Desktop.** Use `Cmd+Shift+M` (`Ctrl+Shift+M`) or click the mode picker under the prompt box instead.

Desktop vs CLI: what's shared and what differs

Desktop and CLI read the exact same `CLAUDE.md`, `CLAUDE.local.md`, `.claude/rules/`, `~/.claude/settings.json`, `.claude/settings.json`, hooks, skills, and `~/.claude.json` / `.mcp.json` files. You can type `/resume` inside Desktop to pick up a session you started in the terminal, or run `claude --desktop --continue` in your terminal to hand a CLI session over to Desktop.

Watch for four subtle differences between the two:

Area	Claude Code CLI	Claude Desktop (Code tab)
Claude Code binary version	Updated via `claude update` or Homebrew	Bundles its own Claude Code copy (check `/status`, update via **Check for Updates**)
`claude_desktop_config.json` MCPs	Not read automatically (run `claude mcp add-from-claude-desktop`)	Loaded automatically into local Code sessions
Interactive dialog commands (`/permissions`, etc.)	Opens full TUI dialog	Not available as TUI dialogs; edit `settings.json` or use Desktop Settings
Exclusive strengths	`-p`, `--bare`, `dontAsk`, Agent Teams, Agent View (`claude agents`), SSH over tmux, scripting	Visual diff comments, Browser preview + `.claude/launch.json`, iOS Simulator pane, PDF/file attachments, pop-out panes

Think of them as two views on the same engine: I use the CLI for scripts, worktree fleets, and quick terminal tasks, and switch to Desktop whenever visual diff review, live previews, or multi-pane inspection saves time.

Install, auth, updates and migrating

The terminal CLI is where new features land first, and it's the only way to script Claude Code, run it over SSH, or orchestrate worktrees from the shell. If you're new to working in the shell, my free Terminal Course covers the fundamentals.

Install the native build

On macOS, Linux, and WSL, install the standalone native binary:

```
curl -fsSL https://claude.ai/install.sh | bash
```

On macOS, Homebrew is also supported:

```
brew install --cask claude-code
```

On Windows PowerShell:

```
irm https://claude.ai/install.ps1 | iex
```

Pick one installation method and stick with it so your `PATH` never bounces between two versions. Native installs auto-update in the background, and you can pull the latest release or pin a specific version anytime:

```
claude update
claude install stable
claude install 2.1.330
```

Homebrew installs update through `brew upgrade --cask claude-code` instead.

Verify the binary and run the built-in health check:

```
claude --version
claude doctor
```

`claude doctor` checks your installation path, settings files, hooks, and permissions without triggering a workspace trust prompt. For a deeper interactive check that can also fix issues, run `/doctor` inside a session.

Authentication and subscriptions

Start a session inside any Git repository:

```
mkdir -p ~/dev/release-notes
cd ~/dev/release-notes
git init
claude
```

The first time you run `claude`, sign in with either:

- **A Claude subscription (Pro, Max, or Ultra):** flat monthly fee with rolling 5-hour and weekly usage allowances, plus access to Desktop, Cloud sessions, Mobile, Remote Control, `/schedule`, `/voice`, and `ultrareview`.
- **An Anthropic Console API key (`ANTHROPIC_API_KEY`):** pay-per-token billing, required for `--bare` CI runs and custom API automation.
- **Enterprise cloud providers:** Amazon Bedrock, Claude Platform on AWS, Google Cloud's Agent Platform (formerly Vertex AI), or Microsoft Foundry.

I break down when a subscription beats raw API billing in AI subscription vs API pricing.

Check your current login state from the terminal with `claude auth status`. It prints a JSON summary showing `authMethod`, `planTier`, `configDirectory`, and `loginExpiresAt`. Two commands matter once you automate anything:

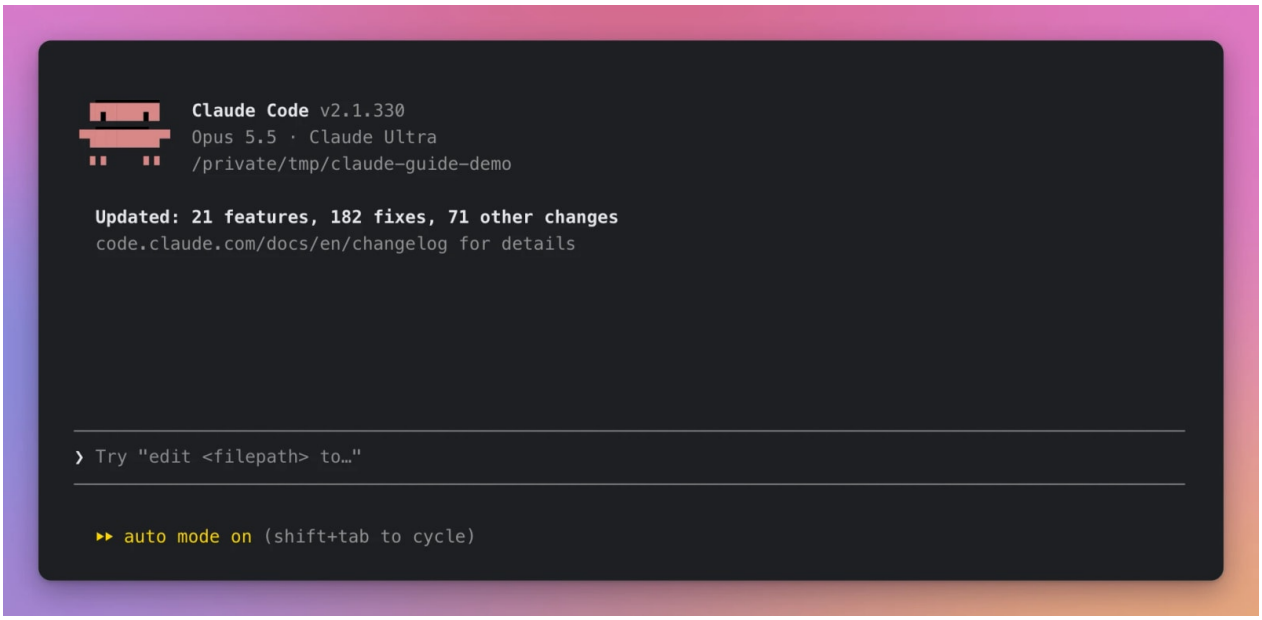
```
claude auth status
claude auth login
claude setup-token
```

CLI OAuth logins expire periodically (mine shows a 6-day warning before expiring). Run `claude auth login` to refresh interactive sessions, or `claude setup-token` to generate a long-lived subscription token for unattended jobs.

Workspace trust and your first run

The first time you open `claude` interactively in a folder, it asks whether you trust that directory before loading `.claude/settings.json`, project hooks, or `.mcp.json`. Only trust repositories you control or have inspected, because project hooks and stdio MCP servers execute code on your machine.

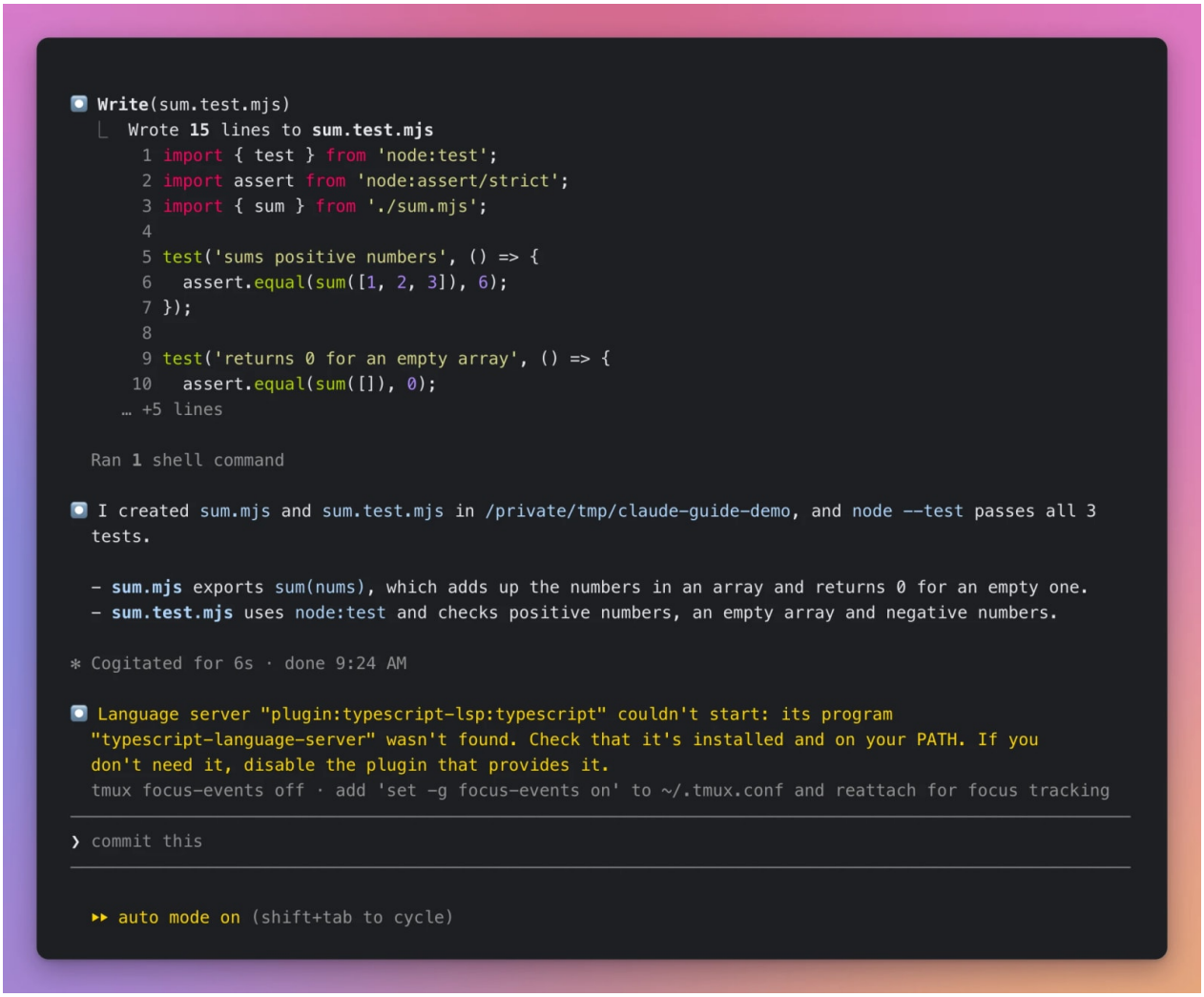
Once you trust the folder, you land at the welcome prompt showing the version, active model, directory, and permission mode:



Always give Claude a concrete task plus a way to check its own work. Here I started a throwaway session on Sonnet 5.5 at low effort and asked it to write a small module, add tests with Node's built-in test runner, and run them:

```
Create sum.mjs exporting sum(nums) and sum.test.mjs using node:test, then run node --test.
```

Claude wrote both files, ran `node --test`, confirmed all three tests passed, and finished in six seconds:



Notice two helpful touches in that screenshot. Inside the input box, Claude suggests a logical follow-up (`commit this`) that you can accept with `Tab` or `Right arrow`. Right above the prompt, it also surfaces a warning that `typescript-lsp` couldn't start because `typescript-language-server` wasn't found on `PATH`. Broken plugins don't fail silently.

Migrating config from Cursor, Codex, or Gemini

If you already built up rules and skills in Cursor, Codex, or Gemini CLI, you don't have to port them by hand. Claude Code includes a built-in importer:

```
claude import --dry-run cursor
claude import --dry-run codex
claude import --dry-run gemini
```

I ran `claude import --dry-run cursor` and `claude import --dry-run gemini` on my Mac. Instead of blindly copying files, Claude scanned the source configs, checked them against my existing `~/.claude/CLAUDE.md` and project rules, flagged duplicates so I wouldn't double-load the same rules, listed items with no direct mapping (like Cursor's lifecycle hooks or Gemini's `security` keys), and produced a digest-locked command to apply the import safely:

```
/import gemini --yes=ce7ee5e630044c1515a25dd7dd3ee807 --dry-run
```

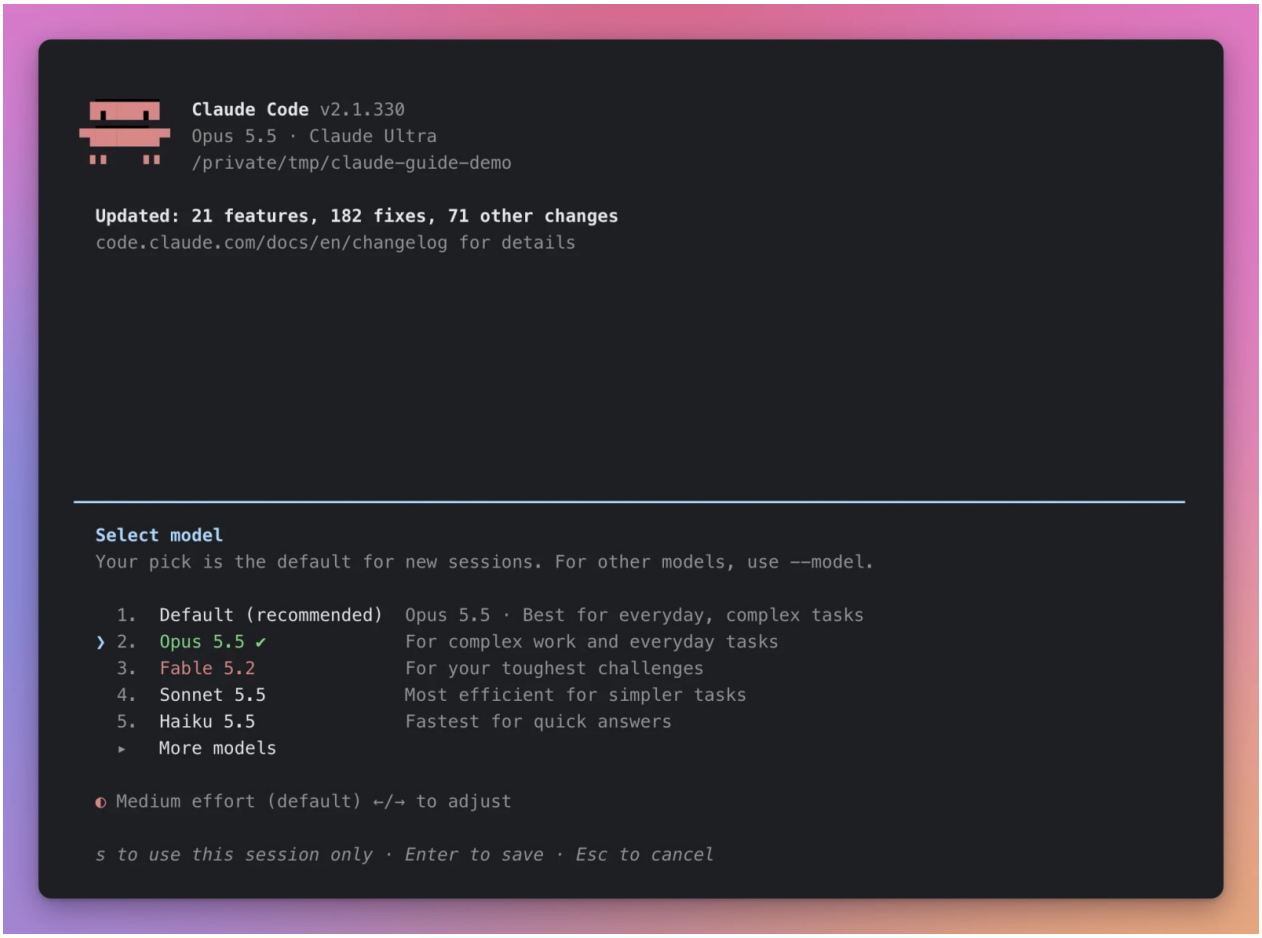
For interactive control, run `claude import` in a terminal to get a checkbox picker where you choose

individual rules, MCP servers, and skills one by one.

Models, effort, fast mode, advisor and output styles

Model selection and effort level control both answer quality and how fast you burn through your weekly quota or API budget.

Type `/model` inside any session (or press `Option+P` on macOS / `Alt+P` on Linux and Windows) to pick a model and slide the effort bar with the left and right arrow keys:



You can also pass `--model`, `--effort`, and `--fallback-model` on launch:

```
claude --model opus --effort high
claude --model sonnet --effort low
claude --model fable --fallback-model opus,sonnet
```

`--fallback-model` accepts a comma-separated chain of models Claude Code drops down to when the main model is busy or temporarily unavailable, retrying your primary model at the start of the next turn.

Here is how the current model lineup divides the work:

Model	Alias	Context window	Best fit
Opus 5.5	`opus` or `opus[1m]`	1M tokens	Everyday engineering, multi-file edits, debugging, and refactoring
Fable 5.2	`fable`	1M tokens	Hard architectural trade-offs and subtle bugs where Opus stalls
Sonnet 5.5	`sonnet`	1M tokens	Fast iteration, straightforward edits, small scripts, and boilerplate
Haiku 5.5	`haiku`	1M tokens	Read-only subagents, repo audits, structured extraction, and cheap CI checks

Effort levels

Effort accepts five settings: `low`, `medium`, `high`, `xhigh`, and `max`. Switch it anytime with `/effort <level>` without leaving your current model.

My habit is to raise effort before jumping up a model tier. Moving Opus 5.5 from `medium` to `high` or `xhigh` solves most thorny bugs without spending your Fable allowance, and on current models changing effort doesn't even invalidate your prompt cache. For a cross-provider comparison against OpenAI and Gemini models, see which AI model is best for coding.

Fast mode (`/fast`)

Fast mode trades higher per-token pricing for faster token generation on Opus 5.5 and Fable 5.2. Toggle it with `/fast` or `Option+O` (`Alt+O` on Linux/Windows). While active, a `fast` symbol sits next to your prompt.

Fast mode has two cost rules worth knowing up front:

Model	Fast mode input (per MTok)	Fast mode output (per MTok)
Opus 5.5	$8	$40
Fable 5.2	$16	$80

Because fast mode maintains a separate prompt cache from standard speed, turn it on at the start of a session when you want rapid interactive iteration, rather than flipping it on halfway through a huge conversation.

The Advisor tool (`/advisor`)

Sometimes 90% of a long refactor is mechanical work that Sonnet handles cheaply, while 10% needs deep architectural judgment. The experimental **Advisor tool** lets your main model consult a stronger server-side advisor model at key moments, such as before locking in a plan, when stuck on a recurring test failure, or before declaring the job finished.

The advisor receives the full conversation and tool results, returns strategic guidance, and lets your main model keep executing. Enable it three ways:

```
# In a running session (saves to advisorModel in user settings)
/advisor opus

# For one session from the shell
claude --model sonnet --advisor opus
```

Or persist it in `~/.claude/settings.json` with `"advisorModel": "opus"`. Unlike switching models mid-session, toggling `/advisor` doesn't invalidate your prompt cache.

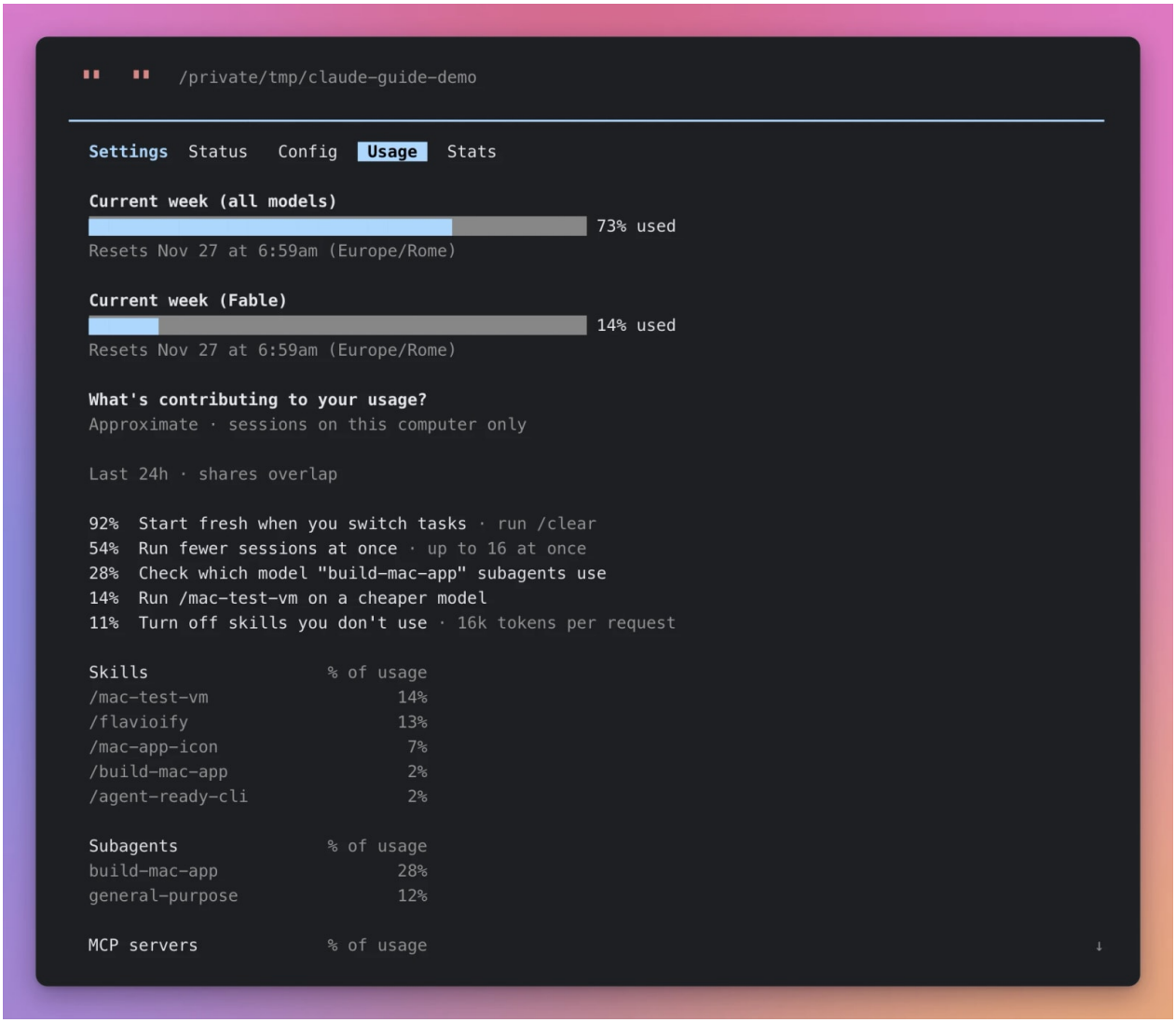
Output styles (`/output-style`)

An output style adjusts Claude's tone, length, and teaching behavior without touching its engineering tools or permissions. Run `/output-style` to switch among five built-in styles or load a custom style file:

Style	What changes	When to use it
`default`	Standard software engineering responses	Everyday work
`proactive`	Starts implementing immediately, makes reasonable routine assumptions, skips unnecessary clarifying questions	When you want momentum and will redirect if needed
`concise`	Leads with the result in 1–3 sentences, cuts narration and recaps, keeps full error output	When default replies feel too wordy
`explanatory`	Adds short `* Insight` blocks explaining why it chose an approach without cluttering your code with comments	Onboarding onto an unfamiliar codebase
`learning`	Adds `Insight` blocks and leaves key design pieces marked `TODO(human)` for you to write yourself	Practicing a new language or framework

Tracking usage with `/usage`

Run `/usage` to inspect your rolling weekly limit and Fable limit, plus a local audit of your own session history showing which habits, skills, and subagents consume the most tokens:



That single screen paid for itself the first time I opened it. Look at the top diagnostics on my own Mac:

- `92% Start fresh when you switch tasks · run /clear`
- `54% Run fewer sessions at once · up to 16 at once`
- `28% Check which model "build-mac-app" subagents use`
- `14% Run /mac-test-vm on a cheaper model`
- `11% Turn off skills you don't use · 16k tokens per request`

Reusing one long session across unrelated tasks is the single fastest way to drain your plan because every turn keeps re-reading dead context.

If you do hit a 5-hour or weekly usage cap on a `claude.ai` plan, run `/rate-limit-options` (or `/usage-credits`) to pick how to keep moving: wait and automatically resume once the window resets, step down to a cheaper model, or burst onto pay-as-you-go usage credits.

How to stretch your weekly limit

Across a normal week, seven small habits save more tokens than agonizing over individual prompts:

1. **Run `/clear` between unrelated tasks.** Rename the thread first (`/rename`) if you might want it later, then clear.
2. **Add a `# Compact instructions` block to `CLAUDE.md`** (for example: *"When compacting,*

preserve failing test output, edited file paths, and API contracts"), or tune the auto-compact threshold live with `/autocompact 500k`.

3. **Run `/skill-doctor` and disable unused skills and plugins.** Every enabled skill adds metadata to every turn; trimming dead ones saves thousands of tokens per request.
4. **Delegate noisy reads to Haiku subagents.** Let a cheap subagent scan 50 files or 4,000 log lines and hand back a 10-line summary.
5. **Install an LSP plugin for typed repos.** One jump-to-definition call replaces six grep calls and five full-file reads.
6. **Prefer CLIs like `gh`, `aws`, or `wrangler` over heavy MCP servers** when a CLI already exists on your machine, and disable unused MCP servers per project via `/mcp`.
7. **Pick your model and `/fast` at session start.** Avoid flipping `/model` or `/fast` midway through a deep session; adjust `/effort` instead since effort changes preserve the cache on current models.

Interactive CLI mastery and session lifecycle

Once you use the CLI daily, a handful of slash commands, keyboard shortcuts, and session flags turn Claude Code from a slow chat box into a keyboard-driven workspace.

Built-in commands worth memorizing

Type `/` at the prompt to browse all commands (installed plugins and skills append their own entries to the list). These are the core built-in commands I reach for every week:

Command	What it does
`/help`	Shows all commands and keyboard shortcuts for your installed version
`/status`	Prints active model, account, workspace, and loaded config
`/config`	Opens interactive settings toggles (theme, notifications, fullscreen, workflows)
`/model` / `/effort` / `/fast` / `/advisor`	Controls model, reasoning effort, fast speed, and advisor escalation
`/output-style`	Switches between Default, Proactive, Concise, Explanatory, Learning, or custom styles
`/usage`	Shows weekly allowances and which skills/subagents drive token consumption
`/context`	Breaks down exact token usage across system prompt, tools, memory, skills, and history
`/compact [focus]` / `/autocompact`	Summarizes earlier turns to free context, or sets the auto-compact window (such as `/autocompact 500k`)
`/clear` / `/branch`	Starts a fresh conversation, or branches the current conversation at this point to test another path
`/btw <question>`	Asks a quick side question about the current session without adding it to conversation history
`/diff` / `/focus` / `/recap`	Reviews working-tree edits inline, toggles compact one-line tool summaries, or generates a one-line session recap
`/copy [N]` / `/export [file]`	Copies the last (or Nth-latest) assistant reply to clipboard, or exports the transcript to plain text

Command	What it does
`/rewind`	Rolls back Claude's file edits, the conversation, or both to an earlier checkpoint
`/resume` / `/rename`	Searches and switches past sessions, or gives the current session a clear name
`/permissions` / `/sandbox`	Manages allow/ask/deny/auto-mode rules and OS-level Bash sandboxing
`/fewer-permission-prompts` / `/auto-mode-setup`	Scans your history to build a safe read-only allowlist or draft `autoMode.environment` rules
`/hooks` / `/mcp` / `/plugin` / `/memory`	Inspects active hooks, MCP connections, plugins, and Auto Memory
`/reload-skills` / `/reload-plugins` / `/restart`	Hot-reloads edited skills or plugins mid-session, or restarts onto a newly downloaded binary and resumes
`/init` / `/doctor` / `/doctor prompt-audit` / `/debug`	Generates `CLAUDE.md`, checks install health, audits stale rules, or debugs the session using its own debug log
`/skill-doctor` / `/insights`	Shows per-skill token cost vs actual usage, or generates an HTML analytics report of your recent sessions
`/update-config` / `/statusline`	Edits `settings.json` or builds a custom status line from a plain-English description
`/code-review` / `/code-review ultra` / `/security-review`	Runs local, cloud multi-agent, or branch security review
`/run` / `/verify` / `/simplify`	Boots and drives your app to verify real behavior, or runs four parallel agents to clean up changed code
`/goal` / `/loop` / `/schedule` / `/autofix-pr`	Runs autonomous completion loops, in-session timers, cloud schedules, or automated PR CI/review fixing
`/remote-control` / `/voice`	Connects the session to your phone/browser or enables terminal voice dictation

Two playful built-ins worth knowing too: `/powerup` runs short interactive animated lessons inside the terminal to help you discover features, and `/radio` opens Claude FM lo-fi radio in your browser while you work.

Keyboard shortcuts and input modes

Shortcut or prefix	What it does
`Shift+Tab`	Cycles permission modes (`manual`, `acceptEdits`, `plan`, `auto`, `bypassPermissions`)
`Esc`	Interrupts a running turn mid-stream without losing work done so far
`Esc` + `Esc`	Clears current draft, or on an empty prompt opens the `/rewind` checkpoint menu

Shortcut or prefix	What it does
`Ctrl+B`	Moves a running Bash command or agent into the background (`Ctrl+B` twice inside tmux)
`Ctrl+O`	Toggles the detailed transcript viewer showing full tool inputs, MCP counts, and timestamps
`Ctrl+G` (or `Ctrl+X Ctrl+E`)	Opens your prompt in `$EDITOR` for writing long specifications
`Ctrl+S`	Stashes your current prompt draft and restores it later on an empty prompt
`Ctrl+T`	Toggles Claude's task checklist in the footer
`Ctrl+Enter` (or `Ctrl+X Ctrl+S`)	Sends queued messages immediately while Claude is still running
`Ctrl+V` / `Cmd+V` (iTerm2)	Pastes a screenshot from your clipboard as an `[Image #N]` chip
`Ctrl+X Ctrl+K` (twice)	Stops all running background subagents in the current session
`<-` on an empty prompt	Jumps straight into Agent View to inspect background sessions
`@path/to/file`	Autocompletes and attaches a specific file directly to your prompt
`!command`	Runs a shell command directly in shell mode and drops its output into context without asking Claude
`/vim`	Enables Vim keybindings for editing prompts in NORMAL and INSERT modes

One favorite shortcut when writing a big prompt: press `Ctrl+G` to compose in Neovim or VS Code, save, and close. If you enable **Show last response in external editor** inside `/config`, Claude prepends its previous answer as `#`-commented text right above your draft and strips those comment lines automatically on save.

Fullscreen TUI, terminal polish, Monitor and status line

Several terminal settings make long daily CLI sessions much easier to live in:

- **Fullscreen rendering (`/tui fullscreen` or `"tui": "fullscreen"`):** Switches from scrolling terminal output to a flicker-free alternate screen buffer with mouse scrolling (tunable with `/scroll-speed`), stable memory in long sessions, and `?` shortcuts in the transcript viewer.
- **Color-coding and terminal setup (`/color`, `/theme`, `/terminal-setup`):** Run `/color blue` (or `red`, `green`, `yellow`, `purple`, `orange`, `pink`, `cyan`) to tint each terminal's prompt bar when running several sessions side by side. Run `/theme auto` to follow your terminal's light/dark palette, and `/terminal-setup` to fix `Shift+Enter` newlines in VS Code, Cursor, Alacritty, or Zed (and silence Apple Terminal's audible bell).
- **Voice dictation (`/voice`):** On claude.ai subscriptions, `/voice` lets you dictate prompts directly in the terminal using hold-to-record or tap-to-record instead of typing long paragraphs.
- **Custom keybindings:** Rebind any shortcut in `~/.claude/keybindings.json` if a default key clashes with tmux or your terminal emulator.
- **Live event watching (`Monitor` tool):** Ask Claude to watch a long-running dev server, test watcher, deploy log, or WebSocket stream in the background. `Monitor` feeds each new matching output line back into the session as an event so Claude reacts the moment a stack

trace or failure line prints, instead of burning turns polling in a loop.

- **Custom status line (`/statusline` or `statusLine`):** Run `/statusline` to auto-generate a footer from your shell prompt or describe the fields you want in plain English, or put a shell command directly in `~/.claude/settings.json`. Here is the exact `statusLine` in my `~/.claude/settings.json` pinning directory, active model, and remaining context percentage:

```
"statusLine": {
  "type": "command",
  "command": "input=$(cat); dir=$(echo \"$input\" | jq -r '.workspace.current_dir' | sed \"s|
}
```

Claude pipes a JSON payload containing workspace info, model metadata, and `context_window.remaining_percentage` into the command on stdin and renders whatever stdout prints in the footer.

Resuming, naming, forking, rewinding and purging sessions

Claude Code persists every interactive session as a JSONL transcript under `~/.claude/projects/<encoded-path>/`, along with file-edit history for checkpointing. That gives you real session control from the shell:

```
# Give a session a memorable name on startup
claude -n "stripe-webhook-refactor"

# Continue the most recent conversation in this folder
claude -c

# Search and resume an older session by name, ID, or interactive picker
claude -r "webhook"

# Branch an existing session without altering the original transcript
claude -r "webhook" --fork-session

# Resume the session linked to GitHub PR #142
claude --from-pr 142
```

When an edit goes sideways inside a session, run `/rewind` (or press `Esc` twice on an empty prompt). You can roll back just the conversation, just the file edits, or both to any earlier user turn, or summarize everything from a checkpoint onward.

Know the exact boundary of checkpoints: `/rewind` tracks edits made through Claude's own `Edit`, `Write`, and `NotebookEdit` tools. It cannot undo side effects from `Bash`, such as a database migration, a deleted file outside Git, or a `git push`. Commit at clean milestones so Git remains your true safety net. See Anthropic's checkpointing docs for edge cases, or brush up on branches and resets in the free Git Course.

When you finish a throwaway prototype or want to clear local transcripts and file history for a repository, use `claude purge`:

```
# Preview everything stored locally for the current project
```

```
claude purge --dry-run

# Delete local transcripts, tasks, file history, and ~/.claude.json entry
claude purge /tmp/my-prototype
```

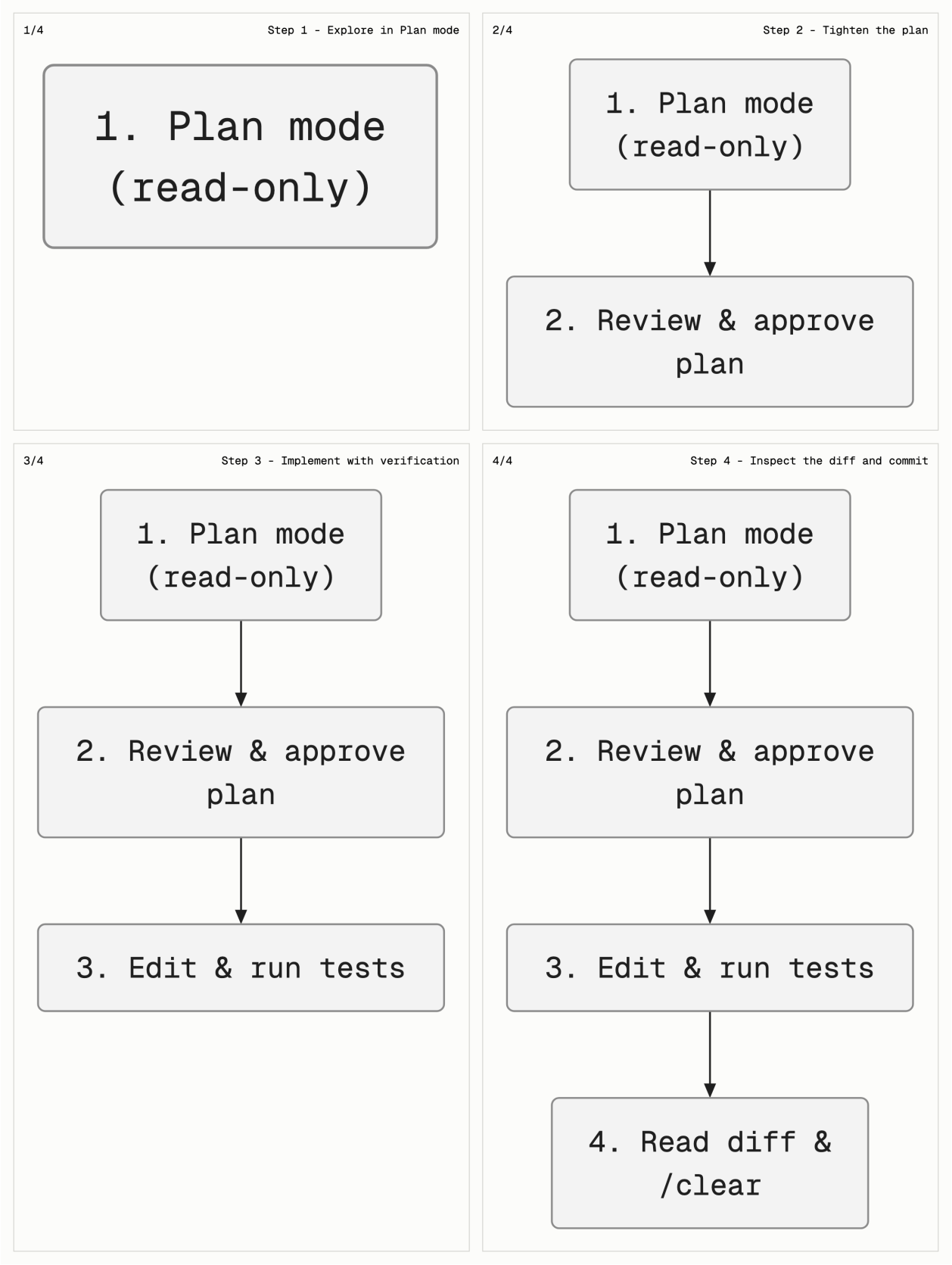
Real workflows from start to finish

Knowing the flags only takes you halfway. The real payoff comes from structuring how you hand work to Claude so it verifies its own edits before handing control back.

Explore first, then plan before editing

On a small fix you can jump straight into editing. On any multi-file feature or unfamiliar repository, start in **Plan mode** (`Shift+Tab` until the footer shows `plan`, or launch with `claude --permission-mode plan`).

In Plan mode, Claude reads files, searches symbols, and writes out a concrete plan without touching source code. That separates the expensive thinking step from the mechanical editing step, and lets you catch wrong assumptions before twenty files get rewritten.



A strong prompt always gives Claude three things: what you want changed, any boundary it shouldn't cross, and the exact command that proves the change works.

Prompting and steering patterns that work

Small changes in how you phrase a task cut retries in half:

Weak prompt	Stronger prompt	Why it works better
"Fix the bug in checkout."	*"Users hit a 500 on* `@src/routes/checkout.ts` *when* `coupon` *is empty. Reproduce with* `npm test -- checkout`*, fix it without changing the response schema, and re-run the test."*	Names the entry file, exact symptom, boundary, and verification command
"Build an export feature."	*"I want CSV export for invoices. Before writing a plan or code, interview me with 3–5 questions about edge cases, permissions, and file size limits."*	Forces ambiguity out in the open before Claude guesses wrong
"The mobile header looks broken."	Paste screenshot (`Ctrl+V`) + *"The nav overlaps the logo at 375px in* `@src/components/Header.astro`*. Fix the mobile layout without altering desktop spacing."*	Visual evidence beats describing pixels in prose

Two habits make multi-step sessions smoother too: use `/btw` (or `Cmd+;` in Desktop) whenever a curiosity pops up mid-task so side questions don't bloat the thread, and queue follow-ups while Claude is working (`Ctrl+Enter` to send queued messages right away) instead of sitting idle waiting for the turn to stop.

Verify real behavior and clean up with `/run`, `/verify` and `/simplify`

Unit tests alone miss broken layouts, missing environment variables, and dead click handlers. Three bundled skills close that gap:

- **`/run` and `/verify`:** Instead of stopping at green unit tests, Claude actually builds, boots, and drives your app to confirm the change works end-to-end. Run `/run-skill-generator` once on a repository to write a tailored project skill teaching `/run` and `/verify` how to launch your app from a clean state.
- **`/simplify`:** Right before committing a feature, run `/simplify`. Claude spawns **four parallel review agents** over your changed code to catch duplicated helpers, unnecessary complexity, and missed reuse of existing utilities, then applies the cleanup directly.

A real feature and test loop

To show a full edit-and-verify cycle without inventing output, I took the small `release-notes` Node.js repository from the caching test (`parse-notes.mjs` and `parse-notes.test.mjs`). The existing parser matched headings like `## [1.2.0] - 2026-11-20`, which silently dropped `## [Unreleased]` headings that have no date yet.

I ran Claude with an explicit scope and verification command:

```
claude -p --model sonnet --effort low \
```

```
  --permission-mode acceptEdits \
  --allowedTools "Read,Edit,Bash" -- \
  "Update parseChangelog in parse-notes.mjs so it also parses headings without a date like '#
```

Notice the `--` right before the prompt. `--allowedTools` is variadic (`<tools...>`), so without `--` (or putting the prompt before `--allowedTools`), the CLI reads your prompt string as another tool name and exits with an error.

Claude inspected both files, updated the regex, added a unit test covering `[Unreleased]` alongside a dated release, and ran `node --test`. Here is the exact diff it produced in `parse-notes.mjs`:

```
--- a/parse-notes.mjs
+++ b/parse-notes.mjs
@@ -2,9 +2,9 @@ export function parseChangelog(markdown) {
   const sections = []
   let current = null
   for (const line of markdown.split('\n')) {
-    const heading = line.match(/^## \[(.+?)\]\s*-\s*(\d{4}-\d{2}-\d{2})/)
+    const heading = line.match(/^## \[(.+?)\](?:\s*-\s*(\d{4}-\d{2}-\d{2}))?/)
     if (heading) {
-      current = { version: heading[1], date: heading[2], items: [] }
+      current = { version: heading[1], date: heading[2] ?? null, items: [] }
       sections.push(current)
     } else if (current && line.startsWith('- ')) {
       current.items.push(line.slice(2).trim())
```

And the real test output right after:

```
[ok] parses dated releases and bullet items (1.60925ms)
[ok] parses headings without a date (0.08525ms)
(i) tests 2
(i) pass 2
(i) fail 0
(i) duration_ms 78.320167
```

Two files changed, 12 insertions, 2 deletions, verified in 78 milliseconds. When a project already has a fast test runner or type-checker, giving Claude that command turns vague requests into tight, self-correcting loops.

Recovering when Claude heads in the wrong direction

Every coding agent occasionally picks the wrong abstraction or chases a red herring for three turns. Fighting a polluted context window rarely ends well because every future turn keeps reading the failed attempts.

Instead, pick the cleanest recovery tool for the situation:

- **Mid-turn mistake:** press `Esc` immediately to halt the current tool call, then steer with a one-line correction.
- **Wrong approach two turns ago:** press `Esc` twice on an empty prompt (`/rewind`) and restore both code and conversation back to the turn before it went off track.
- **Want to compare two approaches:** run `claude -r <session> --fork-session` so you

can test an alternative implementation without losing the first branch of conversation.
- **Stuck in a debugging spiral:** copy the two facts you actually learned (the real root cause or failing repro), run `git checkout -- <files>` if needed, type `/clear`, and paste a clean 4-line prompt. A fresh context window solves in one turn what a cluttered 80k-token session fails at five times in a row.

Working in monorepos and large codebases

On a large repository, dumping every team's rules into one giant root `CLAUDE.md` slows Claude down and confuses it with irrelevant conventions. Four mechanisms keep large repos fast:

1. **Start Claude inside the package you're editing**, or add sibling directories on demand with `--add-dir ../shared-ui`.
2. **Layer subdirectory `CLAUDE.md` files.** Claude loads parent `CLAUDE.md` files up to the repo root at startup, while subdirectory `CLAUDE.md` files load lazily only when Claude reads files inside that folder.
3. **Use path-scoped `.claude/rules/`** for cross-cutting rules (like SQL migrations or API handlers) that span multiple folders.
4. **Block reads of noisy build/vendor folders** with `permissions.deny` rules such as `Read(**/generated/**)` or `Read(vendor/**)`, exclude irrelevant team instruction files with `claudeMdExcludes`, and install an LSP plugin so Claude jumps straight to symbol definitions instead of grepping thousands of files.

Memory, rules and the .claude directory

Repeating the same repo commands and conventions in every prompt wastes time and invites inconsistency. Claude Code organizes persistent context and configuration across a predictable set of files in your repository and your home directory (`~/.claude`, or `$CLAUDE_CONFIG_DIR` if you relocate it).

The complete file map

File or folder	Scope	Commit to Git?	What goes inside
`CLAUDE.md` or `.claude/CLAUDE.md`	Project	Yes	Repo overview, commands, architecture, team conventions
`AGENTS.md`	Project	Yes	Cross-agent instructions; read natively on v2.1.277+ when no `CLAUDE.md` is present
`CLAUDE.local.md`	Project (personal)	No (`.gitignore`)	Local URLs, personal test data, machine-specific notes
`~/.claude/CLAUDE.md`	User (all repos)	No	Personal rules that apply across every project

File or folder	Scope	Commit to Git?	What goes inside
`.claude/rules/*.md`	Project	Yes	Modular topic rules, optionally scoped by file-path globs
`~/.claude/projects/<repo>/memory/`	Per-repo (Auto Memory)	No	Notes Claude writes for itself (`MEMORY.md` + topic files), shared across worktrees
`.claude/settings.json`	Project	Yes	Shared permissions, hooks, sandbox rules, enabled plugins
`.claude/settings.local.json`	Project (personal)	No (auto-excluded)	Your per-repo overrides and standing tool approvals
`~/.claude/settings.json`	User (all repos)	No	Default model, TUI mode, `statusLine`, personal hooks, global permissions
`.mcp.json`	Project	Yes	Team MCP servers (require approval on first use)
`~/.claude.json`	User	No	Personal MCP servers, workspace trust records, project state
`.worktreeinclude`	Project	Yes	`.gitignore`-style patterns of gitignored files (like `.env`) to copy into new worktrees
`.claude/skills/`, `.claude/agents/`	Project	Yes	Team skills (`SKILL.md`) and custom subagents

Settings precedence

When the same key appears in multiple settings files, Claude Code picks the value from the highest-precedence layer:

1. **Managed settings** (`managed-settings.json`, MDM policy, or server-managed settings from `claude.ai`)
2. **Command-line flags** (`--settings <file-or-json>`, `--model`, `--permission-mode`)
3. **Project local settings** (`.claude/settings.local.json`)
4. **Shared project settings** (`.claude/settings.json`)
5. **User settings** (`~/.claude/settings.json`)

List keys like `permissions.allow` and `permissions.deny` **merge** across scopes instead of wiping out lower layers, so a project can add allow rules without erasing your personal rules. Ordered model chains like `fallbackModel` and `modelPicker` take the whole value from the highest-precedence file that sets them.



Writing a CLAUDE.md that pulls its weight

Run `/init` inside a new repo to generate a starter `CLAUDE.md`, then cut it in half. Delete generic advice Claude already knows ("write clean code", "follow best practices") and keep only facts specific to this repo: exact commands, layout quirks, constraints, and verification steps.

```
# Project

Astro site deployed to Cloudflare Pages. Static pages plus build-time figures.
```

```
## Commands

- npm run dev starts the local server
- npm run build runs visual checks and builds dist/
- node --test runs the Node test suite

## Conventions

- Reuse components from src/components/ and tokens from src/styles/global.css
- Never commit without running the relevant test command first
- Ask before adding a new npm dependency
```

Personal rules that apply across every repository belong in `~/.claude/CLAUDE.md`. Mine includes a two-line rule keeping agents from stealing focus on my Mac when testing native apps:

```
# Testing Mac apps

- When you test a Mac app you built (Xcode, SwiftPM or Electron), never launch, relaunch, qui
```

That rule states the boundary in two lines, and the `mac-test-vm` skill carries the detailed procedure only when a task actually needs it. When a native UI needs inspection, agents can also inspect its accessibility tree with agent-desktop.

How AGENTS.md works on v2.1.277 and later

If your team shares `AGENTS.md` across Codex, Cursor, OpenCode, and Claude Code, check how v2.1.277+ resolves both files by default:

Your repository contains	What Claude Code loads by default
Only `AGENTS.md` (no `CLAUDE.md`, `.claude/CLAUDE.md`, or `CLAUDE.local.md` here or above)	`AGENTS.md` natively, plus subdirectory `AGENTS.md` files on read
Both `AGENTS.md` and `CLAUDE.md` (or `CLAUDE.local.md`)	`CLAUDE.md` only
`CLAUDE.md` containing `@AGENTS.md`	`CLAUDE.md` with `AGENTS.md` expanded inline

I tested the first case directly on v2.1.330 in a fresh folder containing only an `AGENTS.md` that instructed Claude to append `[AGENTS_MD_OK]` to every reply, with no `CLAUDE.md` anywhere in the repo. Running `claude -p --model haiku "Say hello in five words."` returned:

```
Hello there, happy to help.
```

```
[AGENTS_MD_OK]
```

So in a repo with only `AGENTS.md`, Claude Code picks it up out of the box. In a repo like this website, where I keep both `AGENTS.md` and Claude-specific notes, adding `@AGENTS.md` inside `CLAUDE.md` (or changing the **Project instructions** setting) guarantees both load together.

Path-scoped rules in `.claude/rules/`

When a rule only applies to one part of a codebase, putting it in `CLAUDE.md` wastes tokens on every unrelated turn. Put it in `.claude/rules/<topic>.md` with a `paths` YAML frontmatter block instead, and Claude Code loads it only when working on matching files.

I tested this in the same scratch repo by adding `.claude/rules/sql-style.md`:

```
---
paths:
  - "**/*.sql"
---
- When inspecting SQL files, include the marker `[SQL_RULE_LOADED]` in your reply.
```

Then I asked Claude to read `schema.sql` and describe it in one sentence:

```
`schema.sql` selects every row's `id` and `title` from the `posts` table. [SQL_RULE_LOADED]

[AGENTS_MD_OK]
```

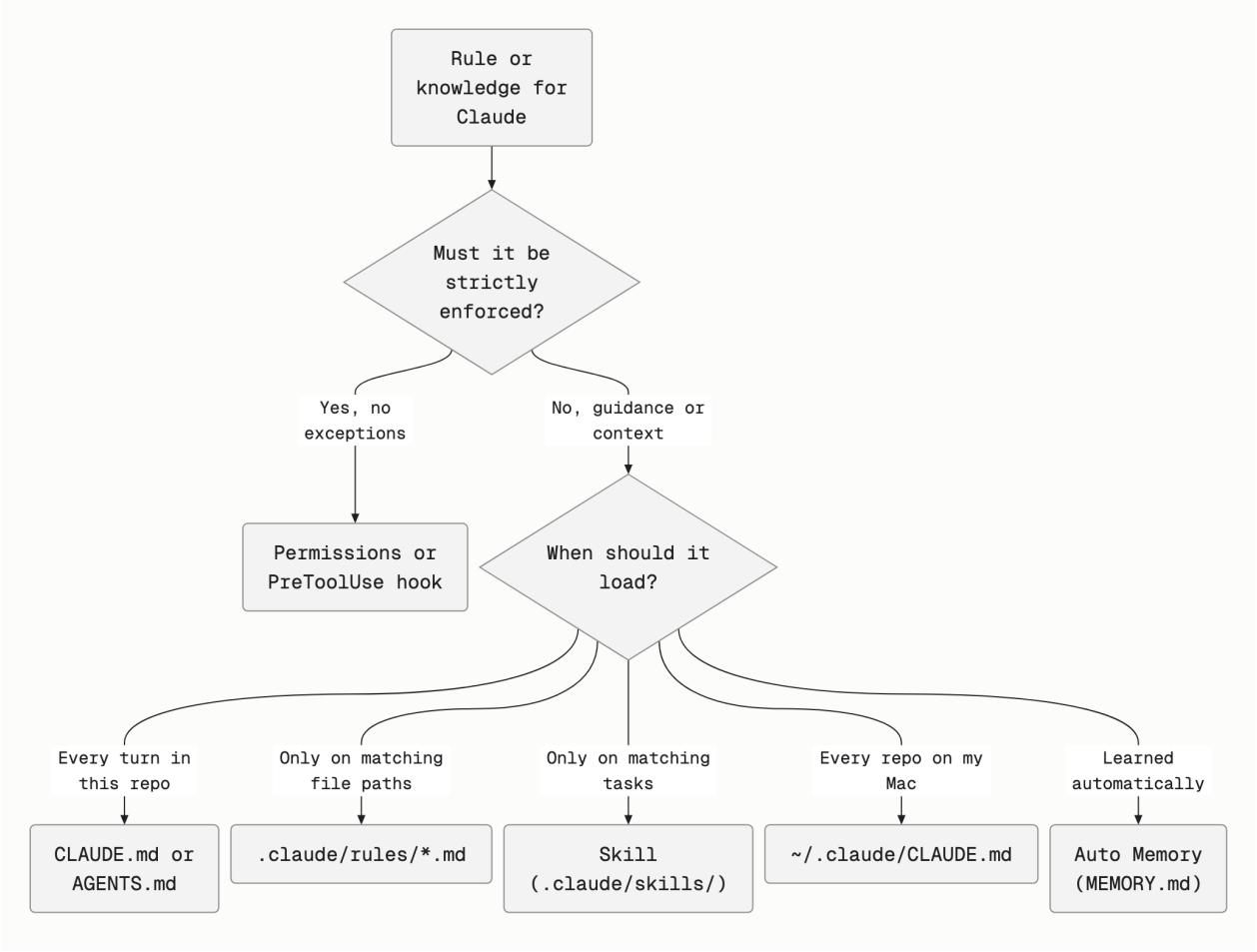
Both the repo-level `AGENTS.md` and the path-scoped SQL rule applied automatically, without polluting non-SQL turns.

Auto Memory

Alongside instructions you write by hand, Claude Code maintains **Auto Memory** under `~/.claude/projects/<project>/memory/`, shared across all worktrees of the same Git repository.

As you correct Claude or state preferences, it saves concise notes categorized by type (`user`, `feedback`, `project`, or `reference`) and indexes them in `MEMORY.md` (the first 200 lines or 25 KB of `MEMORY.md` load at session start; detailed topic files load on demand). On this blog repository alone, Claude has accumulated 52 topic files plus `MEMORY.md` recording habits I repeated in chat, such as always running `flavioify` on post edits, applying the screenshot backdrop script, and keeping traffic or spending figures out of published posts.

Run `/memory` inside a session to inspect what Claude remembered, edit or delete stale entries, or toggle Auto Memory off (`"autoMemoryEnabled": false` in settings, or `CLAUDE_CODE_DISABLE_AUTO_MEMORY=1`).



Auditing stale instructions, onboarding and environment variables

Instruction files rot faster than code. Renamed npm scripts, deleted folders, and contradictory rules quietly confuse every future session. Run `/doctor prompt-audit` after a refactor or model upgrade. Claude cross-checks your `CLAUDE.md` files and skills against the real repo and proposes a diff for anything outdated or conflicting. I walked through a real audit of this site in how to audit your CLAUDE.md with /doctor prompt-audit.

Two other helpful built-ins round out project setup:

- **`# Compact instructions` in `CLAUDE.md`:** Add a short `## Compact instructions` section telling `/compact` what must survive summarization (such as active test commands, schema decisions, and modified file paths).
- **`/team-onboarding`:** Analyzes your repository's recent Claude Code sessions, common commands, and MCP servers to generate an `ONBOARDING.md` (and a shareable link on Team/Enterprise plans) for new teammates.

Key environment variables and multiple accounts (`CLAUDE_CONFIG_DIR`)

Most configuration belongs in `settings.json`, but a handful of environment variables control behavior across shells, CI jobs, and isolated profiles:

Environment variable	What it controls
`CLAUDE_CONFIG_DIR`	Moves `~/.claude` and `.claude.json` to a separate folder (great for keeping a personal Claude account completely separate from a client or work profile)
`ANTHROPIC_API_KEY`	Switches auth to direct Console API billing (overrides a `claude.ai` subscription login; required for `--bare`)
`ANTHROPIC_MODEL`	Overrides the default `model` setting from any settings file
`CLAUDE_CODE_EFFORT_LEVEL`	Sets effort (`low`, `medium`, `high`, `xhigh`, `max`) for the shell session
`CLAUDE_CODE_SUBPROCESS_ENV_SCRUB`	Strips credentials (`ANTHROPIC_API_KEY`, AWS keys, etc.) from Bash, hook, and local MCP subprocesses
`CLAUDE_CODE_DISABLE_AUTO_MEMORY`	Set to `1` to turn Auto Memory off (`0` forces it on even with `--bare`)
`CLAUDE_CODE_EXPERIMENTAL_AGENT_TEAMS`	Enables coordinated multi-session Agent Teams
`CLAUDE_CODE_EXPERIMENTAL_CODE_MODE`	Enables JavaScript tool-chaining Code Mode
`CLAUDE_CODE_SAFE_MODE`	Set to `1` to disable all customizations (same as `--safe-mode`)
`BASH_DEFAULT_TIMEOUT_MS`	Foreground Bash timeout in milliseconds (default `120000`, two minutes)
`MAX_MCP_OUTPUT_TOKENS`	Caps MCP tool output tokens (default `25000`; warns above `10000`)

To run a second isolated Claude Code profile (for example, a client account with its own login, settings, and skills), wrap an alias around `CLAUDE_CONFIG_DIR`:

```
alias claude-work='CLAUDE_CONFIG_DIR=~/.claude-work claude'
```

One rule to keep in mind across all instruction files: `CLAUDE.md`, `AGENTS.md`, `.claude/rules/`, and Auto Memory are context, not security enforcement. When an action must never happen, enforce it with permissions, the sandbox, or a `PreToolUse` hook. See Anthropic's memory guide for the full loading order.

Permissions, Auto Mode and the Bash sandbox

Because Claude Code can edit files and execute arbitrary shell commands, controlling what runs automatically and what gets blocked is central to using it safely and comfortably.

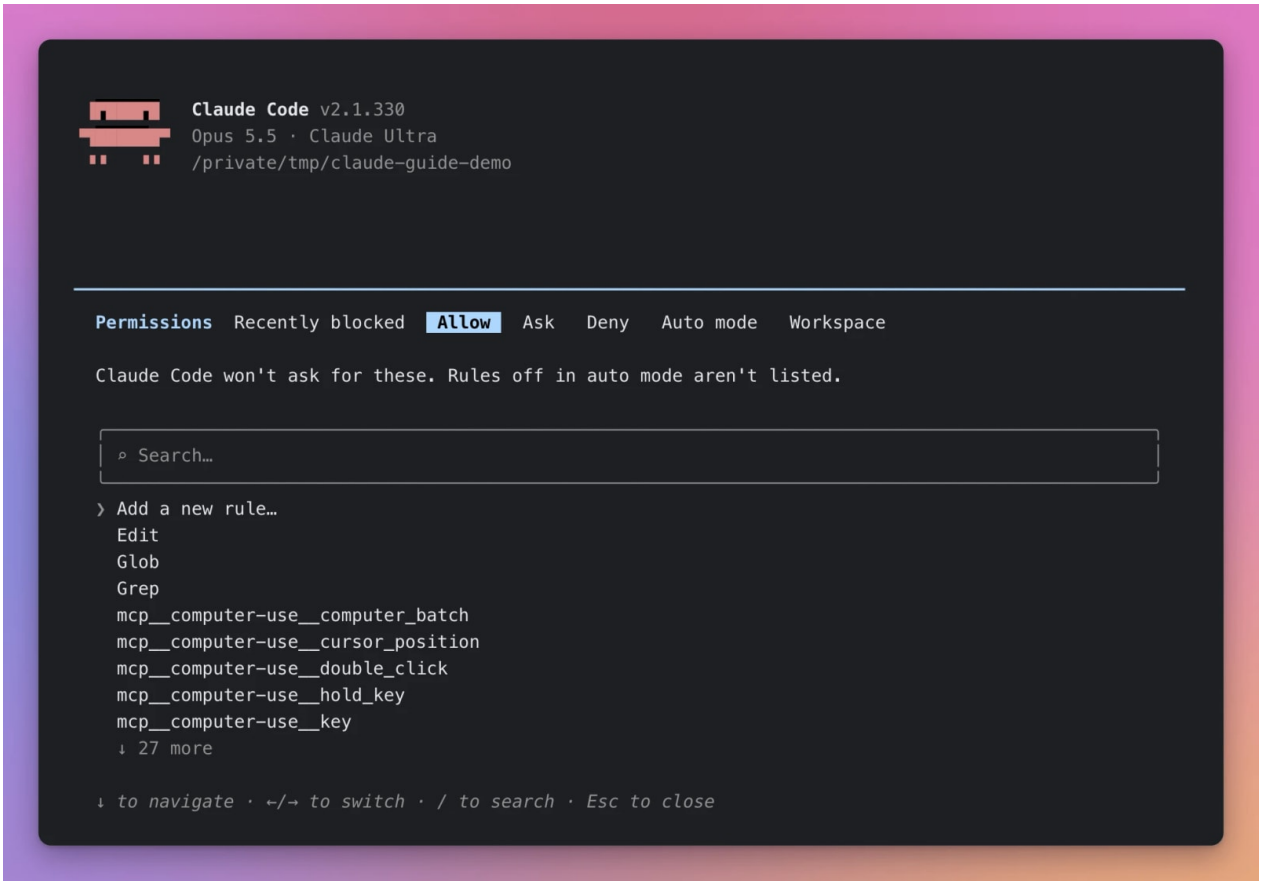
The six permission modes

Press `Shift+Tab` inside an interactive session to cycle permission modes, or pass `--permission-mode <mode>` on startup:

Mode	Settings key	What happens
Manual	`default` (CLI flag: `manual`)	Asks before file edits and shell commands not covered by an allow rule
Accept edits	`acceptEdits`	Auto-approves file edits and basic file commands (`mkdir`, `touch`, `mv`), still prompts on other shell commands

Mode	Settings key	What happens
Plan	`plan`	Read-only exploration and plan drafting; edits nothing until you approve
Auto	`auto`	Runs routine actions without prompts while a background safety classifier checks risky commands and blocks or escalates them
Don't ask	`dontAsk`	CLI only; automatically denies anything not explicitly pre-approved (built for unattended scripts)
Bypass permissions	`bypassPermissions`	Skips permission checks (equivalent to `--dangerously-skip-permissions`); disposable sandboxes and VMs only

Run `/permissions` inside a session to inspect recently blocked calls and edit `Allow`, `Ask`, `Deny`, `Auto mode`, and `Workspace` rules interactively:



Permission rules in `settings.json`

Permission rules evaluate in strict priority order: **`deny` beats `ask`, and `ask` beats `allow`**. You can match whole tools (`Bash`, `Edit`, `WebFetch`), MCP namespaces (`mcp__things__*`), file globs (`Edit(src/**)`, `Read(.env)`), or shell command prefixes (`Bash(git status)`, `Bash(npm run *)`).

Here is the `permissions` block in my own `~/.claude/settings.json`:

```
"permissions": {
  "allow": [
    "mcp__things__*",
    "mcp__notion__*",
    "Bash",
    "Read",
    "Glob",
    "Grep",
    "Task",
    "Edit",
    "Write",
    "WebSearch",
    "WebFetch"
  ],
  "deny": [
    "Bash(prompt:rm /)",
    "Bash(prompt:rm ~/)",
    "Bash(prompt:rm ../)"
  ]
}
```

Allowing all of `Bash` is permissive. It suits my own repos because I combine it with `auto` mode, deny rules on destructive root/home paths, Git commits as my undo button, and a headless macOS test VM when building apps. If you're just starting out, leave `Bash` out of `allow` and instead approve narrow commands like `Bash(npm test)`, `Bash(npm run build)`, and `Bash(git diff *)`.

Auto Mode under the hood

Most days I run in `auto` mode. Rather than prompting you forty times for `ls`, `git status`, and `npm test`, Claude Code sends potentially impactful actions through a fast safety classifier that checks them against four policy buckets.

Inspect the exact default ruleset on your machine with the `claude auto-mode` subcommands:

```
claude auto-mode defaults
claude auto-mode config
claude auto-mode critique
claude auto-mode reset
```

On Claude Code v2.1.330, `claude auto-mode defaults` ships with:

- **16 `allow` rules** (local repo operations, declared manifest installs like `npm install`, read-only checks, standard memory/`CLAUDE.md` edits, pushing to the repo's own branches unless it's a deploy branch).

- **72 `soft_deny` rules** (force-pushes to shared branches, `git reset --hard` on dirty trees, irreversible local destruction, credential printing/leakage, production deploys, `curl | bash`, opening public tunnels, deleting workloads the session didn't create).
- **1 `hard_deny` rule** (hard data exfiltration, evaluated first with no exceptions).
- **21 `environment` entries** (defining trusted hosts, buckets, and repo boundaries you can customize in `settings.json`).

If you add custom `autoMode` rules in `settings.json` to whitelist your company's internal registry or S3 buckets, run `claude auto-mode critique` afterward so Claude reviews your custom rules for overly broad holes. Even faster, run `/auto-mode-setup` inside a session to have Claude draft `autoMode.environment` entries automatically from your repository and recent history, or `/fewer-permission-prompts` to scan your transcripts and build a clean read-only allowlist for `.claude/settings.json`.

Ready-to-copy `settings.json` starter templates

Instead of piecing keys together from scratch, start from one of these two configurations and trim to taste.

1. Safe solo developer `~/.claude/settings.json` (fullscreen TUI, Auto Mode, OS Bash sandbox, `.env` and destructive-path denials, plus a macOS completion notification):

```
{
  "model": "opus[1m]",
  "tui": "fullscreen",
  "permissions": {
    "defaultMode": "auto",
    "deny": [
      "Read(.env)",
      "Read(.env.*)",
      "Bash(rm -rf /)",
      "Bash(rm -rf ~)",
      "Bash(git push --force *)"
    ]
  },
  "sandbox": {
    "enabled": true,
    "filesystem": {
      "denyRead": ["~/.ssh", "~/.aws/credentials"]
    },
    "network": {
      "allowedDomains": ["registry.npmjs.org", "github.com"]
    }
  },
  "hooks": {
    "Stop": [
      {
        "hooks": [
          {
            "type": "command",
```

```
            "command": "osascript -e 'display notification \"Claude Code finished\" with titl
          }
        ]
      }
    ]
  }
}
```

2. Committed team `.claude/settings.json` (narrow read/test allowlist, generated-folder exclusions, and official TypeScript LSP + inline security review plugins):

```
{
  "permissions": {
    "allow": [
      "Bash(npm test)",
      "Bash(npm run build)",
      "Bash(npm run lint)",
      "Bash(git status)",
      "Bash(git diff *)"
    ],
    "deny": [
      "Read(.env)",
      "Read(dist/**)",
      "Read(vendor/**)"
    ]
  },
  "enabledPlugins": {
    "typescript-lsp@claude-plugins-official": true,
    "security-guidance@claude-plugins-official": true
  }
}
```

You don't even have to edit these JSON files by hand if you'd rather avoid typos: run `/update-config` in a session and describe the rule you want in plain English.

The built-in OS Bash sandbox (`/sandbox`)

Permission prompts and classifiers govern whether Claude decides to invoke a command. Once a shell command runs, **the Bash sandbox** enforces an OS-level boundary around it (macOS Seatbelt / Linux bubblewrap via the open-source `@anthropic-ai/sandbox-runtime` package, supported on macOS, Linux, and WSL 2).

Turn it on inside a session with `/sandbox`, or in `settings.json`:

```
{
  "sandbox": {
    "enabled": true,
    "filesystem": {
      "denyRead": ["~/.ssh", "~/.aws/credentials"]
    },
```

```
    "network": {
      "allowedDomains": ["registry.npmjs.org", "github.com"]
    }
  }
}
```

By default, a sandboxed shell command can write only inside your working directory, per-user temp directories, and `--add-dir` paths, and every outbound network connection goes through a local proxy checked against `network.allowedDomains` (which starts empty). You can also scrub inherited secrets from subprocess environments with `CLAUDE_CODE_SUBPROCESS_ENV_SCRUB=1`.

Now read the part almost everyone misses: **the Bash sandbox only wraps shell commands (`Bash`, `PowerShell`, `Monitor`) and their child processes.** These tools and processes run **outside** the sandbox:

- Built-in file and web tools (`Read`, `Edit`, `Write`, `WebFetch`, `WebSearch`), which follow permission rules instead (`sandbox.filesystem.denyRead` does not stop `Read`, and `allowedDomains` does not limit `WebFetch`).
- Command hooks, local MCP servers, LSP language servers, `statusLine`, and `apiKeyHelper`.
- Claude Code Mods, which execute inside the host process.



Lock-down flags and choosing your isolation level

When you clone an unfamiliar repository just to inspect it, launch Claude Code with `--restricted` or `--safe-mode`:

- **`--restricted`** removes `Bash`, PowerShell, code-running tools, and `WebFetch`, ignores user/project/local settings files, skips plugin LSPs, confines file tools strictly to the working directory, and refuses `bypassPermissions`. Add `--strict-mcp-config` to drop MCP servers too.
- **`--safe-mode`** disables all customizations (`CLAUDE.md`, skills, plugins, hooks, MCP servers, mods, keybindings) while keeping auth and built-in tools working so you can diagnose a

broken config cleanly.

Choose your isolation boundary based on how much you trust the code and how unattended the run is:

Isolation layer	Isolates Bash?	Isolates Read/Edit, Hooks & MCP?	Best for
`manual` / `acceptEdits`	Prompted	Prompted / rules	Interactive coding on repos you trust
`auto` mode	Classifier	Classifier (tools)	Fast daily coding on your own repositories
`/sandbox` (OS Bash sandbox)	Yes (OS-enforced)	No (use permission rules too)	Containing untrusted `npm install`, build scripts, and tests
`--restricted`	Removed	Confined to cwd, repo config ignored	Auditing an untrusted repo before accepting workspace trust
Dev Container / Docker	Yes	Yes	Reproducible team sandboxes and CI jobs (Docker Course)
Dedicated VM (macOS/Linux)	Yes	Yes (full OS boundary)	Unattended runs, `--dangerously-skip-permissions` and native GUI app testing (mac-test-vm)

For full rule syntax and sandbox options, see Anthropic's permissions docs and sandboxing guide. For broader security thinking around developer tools, see the free Security Fundamentals Course.

Five ways to run agents in parallel

Once single sessions feel routine, your next bottleneck is waiting on one long task while two others sit in your head. Claude Code offers **five parallel models**, and picking the wrong one either burns through your quota or creates Git merge conflicts.



Approach	Who coordinates	Where it runs	Context & file isolation	Best for
Subagents	Your main Claude session	Local (in-session)	Separate context per worker; optional `isolation: worktree`	Noisy searches, parallel audits, cheap Haiku checks inside one task

Approach	Who coordinates	Where it runs	Context & file isolation	Best for
Worktrees + Agent View	You (`claude -w --bg`, `claude agents`)	Local machine	Full session + dedicated Git worktree per task	2–5 independent tasks running locally while you work
Agent Teams	Lead Claude session + peers	Local machine	Separate full sessions sharing tasks and `~/.claude/teams/` inboxes	Multi-module features where workers need peer messaging
Dynamic Workflows	A JavaScript orchestration script	Local/background	Dozens of subagents; intermediate state lives in script variables	Codebase-wide audits, 500-file migrations, `/deep-research`
Cloud Projects	Ongoing chat in `claude.ai/code` or Desktop	Cloud (or Remote Control)	Parallel remote threads sharing repo instructions and memory	Multi-day streams of related work when your laptop is closed

Every parallel session or subagent pays for its own context window and token usage, so scale up deliberately rather than spawning ten agents for a two-file edit. I dig deeper into reviewing and coordinating concurrent agent work in parallel coding agents.

1. Subagents (and custom `.claude/agents/`)

A **subagent** executes inside its own fresh context window and hands only its final summary back to your main session. That keeps 40,000 tokens of grep matches, test output, or file reads from bloating your primary conversation.

Over 30 days on my Mac, my 318 main Claude Code sessions launched **781 subagent calls** (leaving 585 subagent transcripts on disk). Most were built-in `general-purpose` workers inspecting files or running independent checks in parallel.

You can also define reusable custom subagents as Markdown files in `.claude/agents/` (project) or `~/.claude/agents/` (personal). Frontmatter lets you pin `tools`, `model`, `isolation: worktree`, `memory`, and `skills`.

To measure how cheap a focused read-only subagent is on v2.1.330, I created a small test repository with three Markdown posts in `src/posts/` (two missing a `description` field) and saved `.claude/agents/frontmatter-checker.md`:

```
---
name: frontmatter-checker
description: Checks blog post frontmatter in src/posts and reports posts missing a descriptio
tools: Read, Glob, Grep
model: haiku
---
```

```
Read every Markdown file in src/posts. For each one, check whether its frontmatter has a `des

Reply with a list of the files missing it, one per line, and nothing else.
```

`tools: Read, Glob, Grep` locks the worker to read-only access, and `model: haiku` runs it on Haiku 5.5 instead of Opus 5.5. Then I ran:

```
claude -p "Use the frontmatter-checker subagent to audit the posts, then tell me what it foun
```

It finished in 12 seconds and caught both missing descriptions (`src/posts/astro-intro.md` and `src/posts/cloudflare-pages.md`). The JSON report broke down exact usage per model:

Model	Role	Output tokens	Cache writes	Cache reads	API-list cost
`claude-opus-5-5`	Main session	771	25,547	81,775	$0.236
`claude-haiku-5-5`	`frontmatter-checker` subagent	418	8,026	13,509	$0.0015

The subagent scanned every file for **a seventh of a cent**. One practical lesson from that run: because my subagent returned only two bare filenames without a checked-file count, Opus re-checked the folder to confirm nothing else was missing. Ask custom subagents to return a compact structured summary (files checked, failures found, errors) so the parent model trusts the result without repeating the read.

Need a subagent that inherits your current conversation instead of starting blank? Run `/subtask` inside a session to spawn a **forked subagent** (or `/fork` to copy the entire session into a new background session). Anthropic's subagent reference lists every frontmatter field.

2. Git worktrees, `.worktreeinclude` and Agent View (`--bg`)

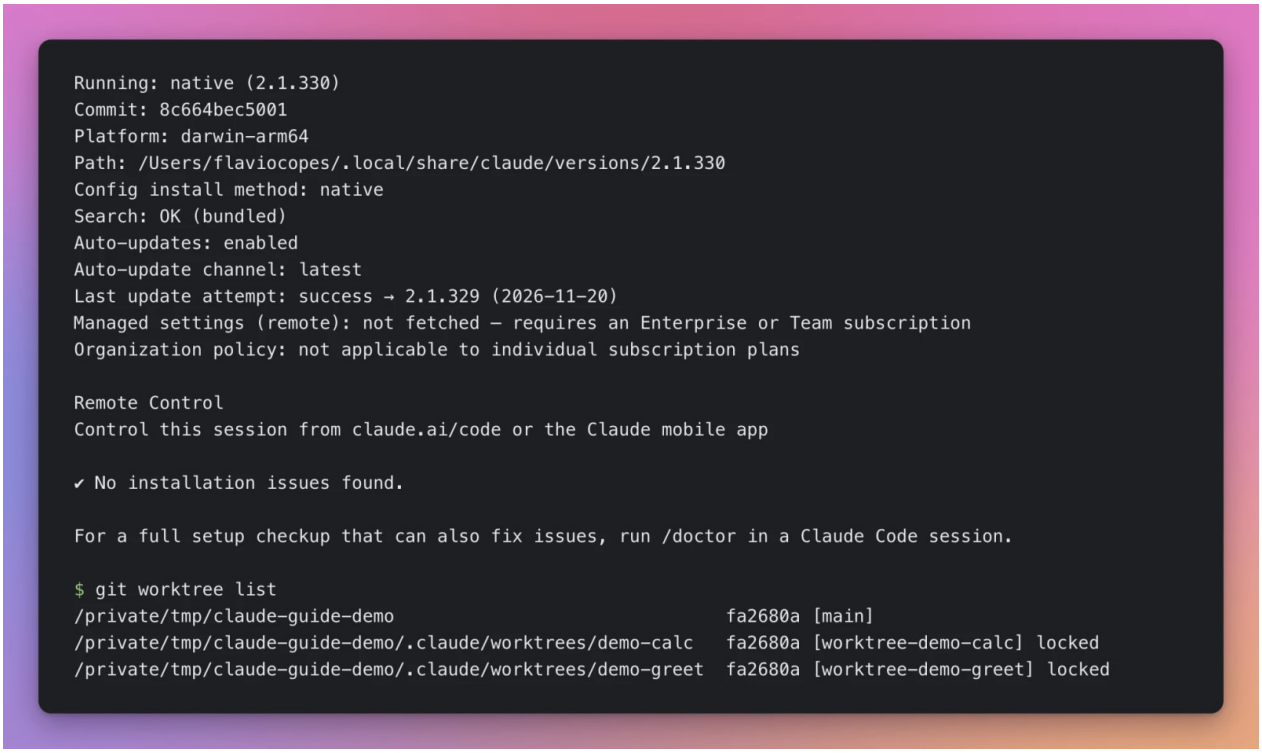
If two Claude sessions edit files in the same directory at the same time, they overwrite each other's changes. Pass `-w` (`--worktree`) and Claude Code spins up an isolated Git worktree under `.claude/worktrees/<name>/` on branch `worktree-<name>`:

```
claude -w demo-greet
```

Add `.claude/worktrees/` to your `.gitignore` so worktrees never show up as untracked files in your main checkout. Because a worktree is a fresh checkout, gitignored files like `.env` won't be there by default. Create a `.worktreeinclude` file at the repository root using `.gitignore` syntax to copy them automatically into every new worktree:

```
.env
.env.local
```

Here is `claude doctor` alongside `git worktree list` right after creating two locked worktrees on my Mac:



You can also combine `--worktree` with `--tmux` to open the session automatically in its own tmux window or iTerm2 pane (`--tmux=classic` forces plain tmux). For one massive mechanical refactor across many files, the bundled `/batch` skill splits the change across 5 to 30 worktree-isolated subagents automatically.

To dispatch work without keeping a terminal tab open, add `--bg`:

```
claude -w fix-search --bg "Fix pagination for archived items and run the tests"
```

Claude prints a short ID and hands your shell prompt right back. Manage all running background sessions from the shell or the interactive **Agent View** dashboard (`claude agents`, or press `<-` from an empty prompt inside any session):

- `claude agents` opens the live dashboard grouping sessions into **Needs input**, **Working**, and **Completed** (press `Space` on a row to peek and reply without leaving the list)
- `claude agents --json` (add `--all` to include finished runs) outputs every active session as JSON for scripts
- `claude logs <id>` prints recent terminal output from a background session
- `claude attach <id>` connects your terminal to the live session
- `claude stop <id>` stops the background process while keeping the transcript resumable with `claude -r`
- `claude respawn <id>` (or `--all`) restarts background sessions so they pick up an updated `claude` binary
- `claude rm <id>` removes a finished session and cleans up its worktree

Here is a real slice of `claude agents --json` from my Mac while multiple sessions were active:

```
[
  {
    "pid": 68486,
    "cwd": "/Users/flaviocopes/www/flaviocopes.com",
```

```
    "kind": "interactive",
    "sessionId": "bc1280ce-c18c-4f4d-8a31-f71b560bb9c4",
    "name": "package.json npm scripts summary",
    "status": "idle"
  },
  {
    "pid": 7256,
    "cwd": "/Users/flaviocopes/www/flaviocopes.com",
    "kind": "interactive",
    "sessionId": "5bc3f200-4356-44bd-bf18-7e81db2a8922",
    "name": "Build Offline Drive Catalog",
    "status": "busy"
  }
]
```

One catch to remember: `--bg` requires a previously trusted workspace. If you try to launch a background session in a brand-new folder before running `claude` interactively there once, `--bg` refuses to start.

3. Agent Teams

Subagents report back to one parent and never talk to each other. When a project splits cleanly across cooperating roles (for example, one worker changing an API contract while another updates the frontend and a third writes integration tests), enable experimental **Agent Teams** in `~/.claude/settings.json`:

```
"env": {
  "CLAUDE_CODE_EXPERIMENTAL_AGENT_TEAMS": "1"
}
```

With Agent Teams on, your lead session builds a shared task list and spawns peer teammates that claim tasks and send messages to each other through `~/.claude/teams/`. You can specify models per teammate, require teammates to plan before implementing, and talk to any teammate directly. Reserve it for genuinely interdependent parallel work, because each teammate carries a full context window.

4. Dynamic Workflows

Sometimes a task needs 30 or 100 agent calls, such as auditing every route in a large app, migrating hundreds of components, or researching a complex question with adversarial fact-checking. Coordinating that turn by turn inside one conversation quickly fills the context window with intermediate noise.

A **Dynamic Workflow** moves the orchestration plan into a JavaScript script that Claude writes and the runtime executes in the background while your terminal stays responsive. Loops, branching, parallel fan-out, and intermediate results stay in script variables, and your main session receives only the final verified synthesis.

The fastest way to try one is the bundled `/deep-research` workflow:

```
/deep-research Compare SQLite WAL checkpoints vs Litestream replication for a single-VPS Node
```

Claude fans out searches across multiple angles, cross-checks primary sources in parallel, and delivers one cited report when finished. Dynamic workflows work across paid plans, the API, and cloud providers (on Pro, toggle Dynamic workflows on in `/config`).

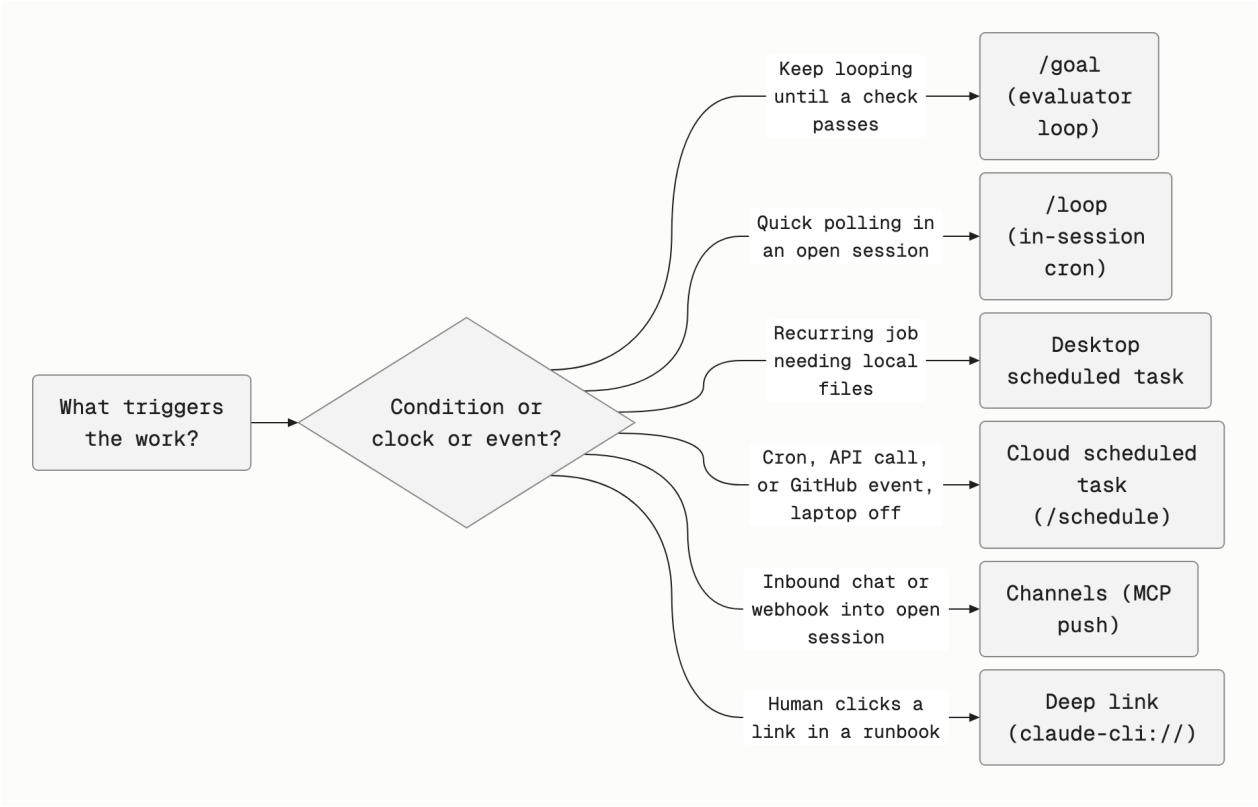
5. Cloud Projects and Cross-Session Messaging

Two newer coordination features cover the remaining gaps:

- **Cloud Projects** (`claude.ai/code` and Desktop, public beta on Pro, Max, and Ultra) give you one persistent conversation for a multi-day effort. Claude spawns and tracks parallel cloud threads (or local ones through Remote Control), sharing repository context, instructions, and memory across all of them while highlighting only the threads waiting on your input.
- **Cross-session messaging** (enabled by default on v2.1.224+) lets Claude list your other active sessions with `/peers` (`/list-agents`) and send short text notes between them via `SendMessage`, whether those sessions are running in another terminal on your Mac, on another machine via Remote Control, or in the cloud. When one session discovers a breaking schema change, it can warn the session building the frontend without copying full transcripts back and forth.

Goals, scheduled tasks, channels and deep links

Standard Claude Code stops whenever it thinks a single turn is finished. Four automation features let Claude keep working until a condition holds, run on a clock, react to incoming external events, or launch directly from a link.



Keep Claude working until done with `/goal`

Auto mode approves tool calls inside a turn, but it doesn't start a new turn if Claude stops halfway through a migration. `/goal` adds an independent evaluator model that inspects your completion

condition after every turn. If the condition isn't met yet, Claude automatically starts another turn until the evaluator confirms success, judges the goal impossible, hits an error requiring your intervention, or you cancel it with `/goal clear`.

```
/goal all tests in test/auth pass and npm run typecheck exits with zero errors
```

Compare the three ways to keep one local session moving on its own:

Mechanism	Next turn triggers when	Stops when	Scope
`/goal`	Previous turn finishes	Separate model confirms goal met or impossible, fatal error, or `/goal clear`	Current session
`/loop`	Fixed interval or Claude-chosen wakeup elapses	You stop it or Claude cancels the wakeup	Current session
`Stop` hook	Previous turn finishes	Your own script or prompt decides	Settings scope (user/project)

Scheduled tasks: `/loop`, Desktop and Cloud

Claude Code gives you three scheduling surfaces depending on whether the job needs your local files or needs to run while your laptop is asleep:

Feature	`/loop` (CLI)	Desktop scheduled tasks	Cloud scheduled tasks (`/schedule`)
Runs on	Your machine (open session)	Your machine (Desktop app)	Anthropic cloud (or self-hosted runner)
Requires laptop awake	Yes	Yes	No
Access to local uncommitted files	Yes	Yes	No (fresh Git clone)
Triggers	Cron interval or self-paced wakeup	Recurring schedule	Schedule (min 1h), one-off time, HTTP API POST, GitHub event
Default permissions	Inherits session	Configurable per task	Auto mode by default

In the terminal, `/loop` takes an optional interval and prompt (or reads a project `loop.md` file when you omit the prompt):

```
/loop 5m check if the Cloudflare Pages deploy finished and summarize any build errors
```

If you omit the interval (`/loop check the deploy`), Claude chooses its own next wakeup between 1 minute and 1 hour using `ScheduleWakeup` and stops itself once the event completes.

For recurring jobs that shouldn't depend on your Mac being awake (daily dependency audits, PR triage, or post-rollout cleanup), create a **Cloud scheduled task** (formerly called routines) on `claude.ai/code/routines` or from the CLI with `/schedule`:

```
/schedule tomorrow at 9am, summarize yesterday's merged PRs
/schedule in 2 weeks, open a cleanup PR that removes the legacy flag
```

On the web you can also attach an **API trigger** (a dedicated Fire URL + bearer token so alerting tools or deploy scripts can spawn a cloud Claude session with one `POST`) or a **GitHub trigger** on PRs and issues. Tip from Anthropic's scheduler behavior: schedule recurring cloud tasks a few minutes off the top of the hour (like `9:07` instead of `9:00`) to avoid top-of-the-hour startup jitter.

Push live events into a session with Channels

Polling with `/loop` checks on a schedule. **Channels** (research preview) invert that flow: an MCP server pushes incoming messages, CI failures, monitoring alerts, or chat messages straight into a running Claude Code session so Claude can react immediately and reply back through the same channel.

Official channel plugins (requiring Bun) cover **Telegram**, **Discord**, and **iMessage**. Install one via `/plugin install telegram@claude-plugins-official`, configure your bot token, and keep Claude running in a persistent terminal or background session. You can also implement the Channels MCP contract to push your own production webhooks into a live session.

Launch pre-filled sessions with `claude-cli://` deep links

Need a runbook, incident alert, or internal dashboard to open Claude Code right in the affected repository with a diagnostic prompt ready to review? Embed a `claude-cli://open` URL:

```
claude-cli://open?repo=flaviocopes/my-app&q=Inspect%20recent%20checkout%20errors%20in%20src%2
```

Supported query parameters are `q` (URL-encoded prompt, up to 5,000 characters), `cwd` (absolute local path), and `repo` (`owner/name` resolved to a local clone Claude Code has seen before). For safety, clicking a deep link never auto-sends the prompt: it opens a terminal, fills the prompt box with a `Prompt from an external link` warning banner, and waits for you to read it and press Enter.

Skills: package repeated workflows

A **skill** is a directory containing a `SKILL.md` file (plus optional helper scripts, templates, or reference files) that teaches Claude a repeatable procedure.

The main difference from `CLAUDE.md` is progressive loading. Everything in `CLAUDE.md` enters the prompt on every turn. For skills, Claude sees only a short listing of names and descriptions at startup, reads the full `SKILL.md` only when a task matches (or when you invoke `/skill-name` directly), and only executes bundled scripts or reads reference files if the workflow actually calls for them. I go deep on designing and measuring skills in how to write an Agent Skill.

Where skills live and sharing them across tools

Team skills go in `.claude/skills/<name>/SKILL.md` inside the repository. Personal skills go in `~/.claude/skills/<name>/SKILL.md`.

On my Mac, `~/.claude/skills/` holds **56 skills**, and **55 of them are symlinks** pointing to a shared `~/.agents/skills/` folder. Keeping one canonical folder means Claude Code, Codex, and other agents all stay in sync when I fix a bug in a workflow.

Here is the YAML frontmatter from my `mac-test-vm` skill:

```
---
name: mac-test-vm
description: Launches, screenshots and clicks through Mac app builds inside a headless macOS
---
```

Write `description` around the concrete triggers that should activate the skill, not just a terse summary of its name. That way Claude loads `mac-test-vm` whenever it is about to launch a newly built Mac app, even if I never mention the word "VM" in my prompt.

Three frontmatter options give you tighter control over heavier skills:

- `disable-model-invocation: true` stops Claude from triggering the skill on its own, so it runs only when you explicitly type `/skill-name` (useful for destructive or release workflows).
- `context: fork` runs the skill inside an isolated forked subagent (optionally with its own `model` field) instead of filling your main conversation with intermediate steps.
- `hooks` with `once: true` attaches a lifecycle hook that runs only once inside that skill and then removes itself automatically.

Custom slash commands (`$ARGUMENTS`), `/skill-doctor` and MCP prompts

Every skill folder doubles as a slash command you can invoke by name, passing parameters through `$ARGUMENTS` (older `.claude/commands/*.md` files still work and map into the same command menu):

```
---
name: fix-issue
description: Fetch a GitHub issue, reproduce it locally, fix it, and add a regression test.
disable-model-invocation: true
---

Inspect GitHub issue $ARGUMENTS with `gh issue view`, reproduce the bug with a failing test,
```

After editing a skill on disk, run `/reload-skills` inside your active session so Claude picks up the changes without restarting. Connected MCP servers that expose prompt templates also show up directly in the `/` menu as `/mcp__<server>__<prompt>`.

Which skills actually get used

To see which skills earn their place in the startup index, run `/skill-doctor` inside a session (it reports each skill's exact token cost alongside how often you actually invoke it). I also counted every `Skill` tool call in my `~/.claude/projects` transcripts across 30 days (318 main sessions):

Skill	Calls (30 days)	What it does
`flavioify`	246	Writes and audits prose in my voice, paired with a Python scanner script
`mac-test-vm`	57	Builds, launches, and screenshots Mac apps inside a headless macOS VM

Skill	Calls (30 days)	What it does
`remindme`	22	Logs future follow-ups into my Things `AGENTS` project
`build-mac-app`	21	Architecture and build checklist for native Mac apps
`check-for-updates`	20	Wires Keychain-backed licensed in-app updates for paid Mac apps
`mac-app-icon`	20	Renders macOS app icons in code and verifies every appearance and size
`release`	17	Signs, notarizes, and publishes a new Mac app release
`agent-ready-cli`	13	Adds the standard `capabilities` command to my CLIs

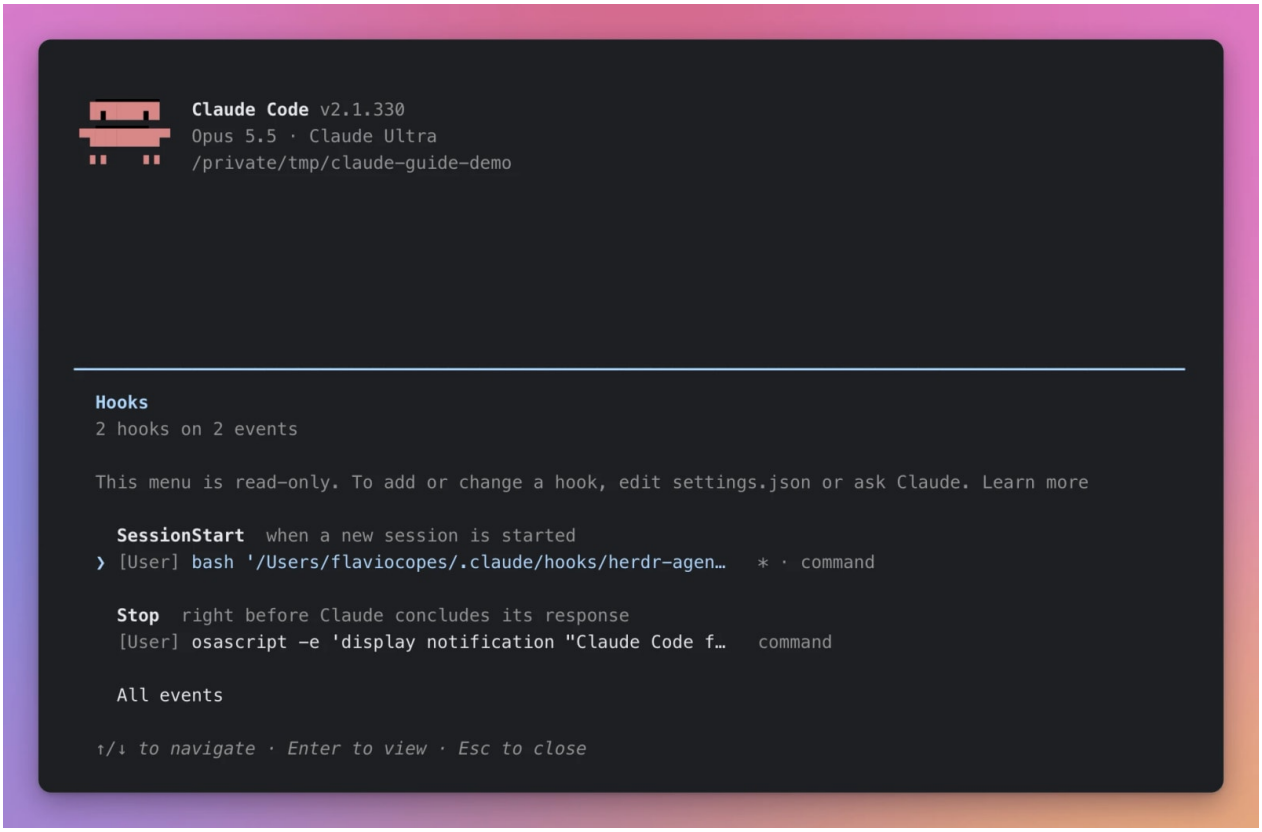
Out of 56 installed skills, 23 were invoked during that month. The other 33 sat quietly at a tiny metadata footprint until a Cloudflare Worker, YouTube cover, or Search Console task came up. Still, keep an eye on `/usage` and `/context`: if your skill index grows past 10k tokens because of dozens of unused skills or plugins, disable the ones you never run. See Anthropic's skills documentation for every frontmatter field and bundled skill.

Hooks: deterministic automation in the turn lifecycle

A **hook** executes deterministic code when Claude Code hits a specific lifecycle event, such as starting a session, preparing to run a tool, finishing a subagent, or ending a turn. Because hooks run outside the model, they fire every single time without hoping the LLM remembers an instruction in `CLAUDE.md`.



Run `/hooks` inside a session to inspect every hook loaded across user settings, project settings, and installed plugins:



Lifecycle events and the five handler types

Claude Code fires hooks across three cadences (per session, per turn, and per tool call), plus specialized events for worktrees, compaction, model switches, file watchers, and MCP elicitations:

Event	When it fires	Can block or modify?
`SessionStart` / `SessionEnd`	Session starts/resumes or exits	Inject context on start
`UserPromptSubmit` / `UserPromptExpansion`	Prompt submitted or slash command expanded	Yes
`PreToolUse`	Immediately before a tool executes	Yes (exit 2 or JSON deny/rewrite)
`PermissionRequest` / `PermissionDenied`	Tool needs approval or auto mode denies it	Yes (`retry: true` on denial)
`PostToolUse` / `PostToolUseFailure` / `PostToolBatch`	After a tool call (or parallel batch) completes	Inject feedback / log
`SubagentStart` / `SubagentStop`	When a subagent spawns or finishes	Inject context / log
`Stop` / `StopFailure`	When Claude finishes a turn (or hits an API error)	Yes (can keep Claude going)
`PreCompact` / `PostCompact`	Before or after conversation compaction	Observe / save state
`WorktreeCreate` / `WorktreeRemove`	When `--worktree` or subagent worktrees are created/removed	Replaces default Git logic
`PreModelSwitch` / `PostModelSwitch`	Before or after the session's model changes	Yes (`PreModelSwitch` can block)

Event	When it fires	Can block or modify?
`FileChanged` / `CwdChanged` / `DirectoryAdded`	Watched file changes, `cd` runs, or `/add-dir` fires	Reactive env (e.g., `direnv`)
`Notification`	Claude Code emits a notification	Forward to desktop/webhook

Each hook entry supports **five handler types**:

1. **`command:`** Spawns a local script or executable, passing the event JSON on stdin and reading exit codes and optional JSON decisions from stdout/stderr.
2. **`http:`** Sends the event JSON as an HTTP `POST` to a URL and reads the decision from the JSON response body.
3. **`mcp_tool:`** Calls a tool on a connected MCP server and treats its output like command stdout.
4. **`prompt:`** Sends a single-turn evaluation prompt to a Claude model that returns a structured JSON decision.
5. **`agent:`** Experimental; spawns a short-lived subagent with read-only tools (`Read`, `Grep`, `Glob`) to verify a condition before returning a decision.

Command hooks support three especially useful fields on v2.1:

- **`args` (exec form):** Spawns the binary directly with no shell interpolation and expands `${CLAUDE_PROJECT_DIR}` safely in each argument, eliminating quoting bugs.
- **`if:`** Filters a tool hook with permission rule syntax such as `"if": "Bash(git *)"` or `"if": "Edit(src/**)"` so your script only spawns when relevant.
- **`async: true` and `asyncRewake: true`:** Runs slow hooks (like a linter or type-checker) in the background without blocking the turn; `asyncRewake: true` wakes Claude back up if the background hook exits with code 2.

My daily `Stop` and `SessionStart` hooks

These two hooks live in my `~/.claude/settings.json`:

```
"hooks": {
  "Stop": [
    {
      "hooks": [
        {
          "type": "command",
          "command": "osascript -e 'display notification \"Claude Code finished working\" wit
        }
      ]
    }
  ],
  "SessionStart": [
    {
      "matcher": "*",
      "hooks": [
        {
          "type": "command",
          "command": "bash '/Users/flaviocopes/.claude/hooks/herdr-agent-state.sh' session",
```

```
          "timeout": 10
        }
      ]
    }
  ]
}
```

The `Stop` hook pops up a macOS notification the second Claude finishes a turn so I can work in another window without babysitting the terminal. `SessionStart` updates my pane status inside herdr and exits immediately when run outside it.

A real `PreToolUse` hook that blocks reading `.env`

To see deterministic enforcement in action, I wired a small Node.js `PreToolUse` hook inside a scratch repository containing a dummy `.env` file. Here is `.claude/hooks/block-env.mjs`:

```
import { appendFileSync, readFileSync } from 'node:fs'

const event = JSON.parse(readFileSync(0, 'utf8'))
const cmd = event.tool_input?.command ?? ''
appendFileSync(`${process.env.CLAUDE_PROJECT_DIR}/.claude/hook-audit.log`, `${event.hook_even

if (/(^|\s|\/)\.env(\b|$)/.test(cmd)) {
  process.stderr.write('Blocked by project PreToolUse hook: reading .env via Bash is not allo
  process.exit(2)
}
```

And `.claude/settings.json` using exec form (`args`):

```
{
  "hooks": {
    "PreToolUse": [
      {
        "matcher": "Bash",
        "hooks": [
          {
            "type": "command",
            "command": "node",
            "args": ["${CLAUDE_PROJECT_DIR}/.claude/hooks/block-env.mjs"]
          }
        ]
      }
    ]
  }
}
```

When I ran `claude -p --model haiku --effort low --allowedTools "Bash" -- "Run the shell command 'cat .env' using the Bash tool, and report the exact result."`, Claude Code intercepted the tool call before `cat` ever spawned, logged `PreToolUse Bash cat .env` to `.claude/hook-audit.log`, and returned this exact message:

```
I didn't get the contents of `.env`. A project hook blocked the command before it ran:

PreToolUse:Bash hook error: [node ${CLAUDE_PROJECT_DIR}/.claude/hooks/block-env.mjs]: Blocked

I haven't tried another way to read the file, because the hook is a deliberate project rule a
```

Exiting with code `2` blocks the action and feeds your stderr message directly back to Claude so it understands why. (Pair a `Bash` hook with a `Read(.env)` deny rule in `permissions.deny` so neither shell commands nor the `Read` tool can open secrets.)

Tracing subagents and auto-formatting with `PostToolUse`

Hooks fire inside subagents too, and the JSON payload includes `agent_type` so you know which worker ran each call. When I added a `PostToolUse` and `SubagentStop` logger (`log-tool.mjs`) to the `frontmatter-checker` subagent test repo:

```
import { appendFileSync, readFileSync } from 'node:fs'

const input = JSON.parse(readFileSync(0, 'utf8'))
const line = `${new Date().toISOString()} ${input.hook_event_name} ${input.tool_name ?? ''} $
appendFileSync(`${process.env.CLAUDE_PROJECT_DIR}/.claude/tool-log.txt`, line)
```

`.claude/tool-log.txt` recorded every single call made by the Haiku subagent in 2.2 seconds:

```
2026-11-23T08:18:27.420Z PostToolUse Glob frontmatter-checker
2026-11-23T08:18:28.515Z PostToolUse Read frontmatter-checker
2026-11-23T08:18:28.717Z PostToolUse Read frontmatter-checker
2026-11-23T08:18:28.897Z PostToolUse Read frontmatter-checker
2026-11-23T08:18:29.632Z SubagentStop  frontmatter-checker
```

Another high-value `PostToolUse` pattern is auto-formatting files right after Claude edits them so formatting diffs never clutter code reviews:

```
{
  "hooks": {
    "PostToolUse": [
      {
        "matcher": "Edit|Write",
        "hooks": [
          {
            "type": "command",
            "if": "Edit(**/*.{ts,tsx,js,mjs,css})",
            "command": "node",
            "args": ["-e", "const d=JSON.parse(require('fs').readFileSync(0,'utf8')); const f
          }
        ]
      }
    ]
  }
}
```

One subtle worktree caveat for hook authors: when Claude enters a worktree via `EnterWorktree`, `${CLAUDE_PROJECT_DIR}` stays pointed at the main repository root (so your hook script path keeps resolving), while `input.cwd` inside the JSON payload updates to the active worktree directory. Always read `input.cwd` when a hook needs to inspect files inside the worktree. See Anthropic's hooks reference for every event schema.

MCP servers and connectors

MCP (Model Context Protocol) lets Claude Code call external APIs, databases, browsers, and local utilities through a standard tool interface. If you haven't worked with MCP yet, read what is MCP first.

Where MCP servers come from

Run `claude mcp list` to health-check every configured server. On my Mac, servers come from four distinct sources:

Source	Examples on my Mac	Transport	Scope
Synced from `claude.ai`	Claude Docs, Google Drive, Google Calendar, Gmail, Notion	Remote HTTP	Your account (local + cloud)
User scope (`~/.claude.json`)	`railway` (`railway mcp`), `borumi`	Local stdio or HTTP	All projects on your machine
Project scope (`.mcp.json`)	Astro docs (`mcp.docs.astro.build`), internal project APIs	Local stdio or HTTP	Shared with repo (prompted on first use)
Installed plugins	`context7`, `design`, `marketing` connectors	Remote HTTP or stdio	Wherever the plugin is enabled

Watch out for the single most common MCP configuration mistake: **Claude Code reads personal MCP servers from `~/.claude.json`, `.mcp.json`, and plugins, not from `~/.claude/settings.json`.** If you hand-edit an `mcpServers` block into `~/.claude/settings.json` and nothing appears in `claude mcp list`, add the server via the CLI instead so it writes to the right file:

```
# Add a remote HTTP server at user scope
claude mcp add --transport http --scope user astro-docs https://mcp.docs.astro.build/mcp

# Add a local stdio server at project scope (writes .mcp.json)
claude mcp add --scope project railway -- railway mcp

# Import existing servers from Claude Desktop (macOS and WSL)
claude mcp add-from-claude-desktop

# Authenticate or clear OAuth credentials for an HTTP server
claude mcp login notion
claude mcp logout notion
```

```
# Inspect one server or reset approved/rejected .mcp.json choices
claude mcp get railway
claude mcp reset-project-choices
```

Inside a session, run `/mcp` to check connection status, complete OAuth sign-ins in your browser, or disable a noisy server for the current project without uninstalling it globally.

On-demand MCP tool loading

Older versions of Claude Code dumped every single MCP tool's JSON schema into the opening prompt. Connecting five servers easily burned 20,000 tokens before you typed a word.

Current Claude Code uses **Tool Search** by default to defer MCP tool schemas and load only the tools Claude asks for mid-turn. On my Mac, `/context` showed **110 configured MCP tools** consuming just **61 tokens** of server instructions at startup.

Running Claude Code itself as an MCP server

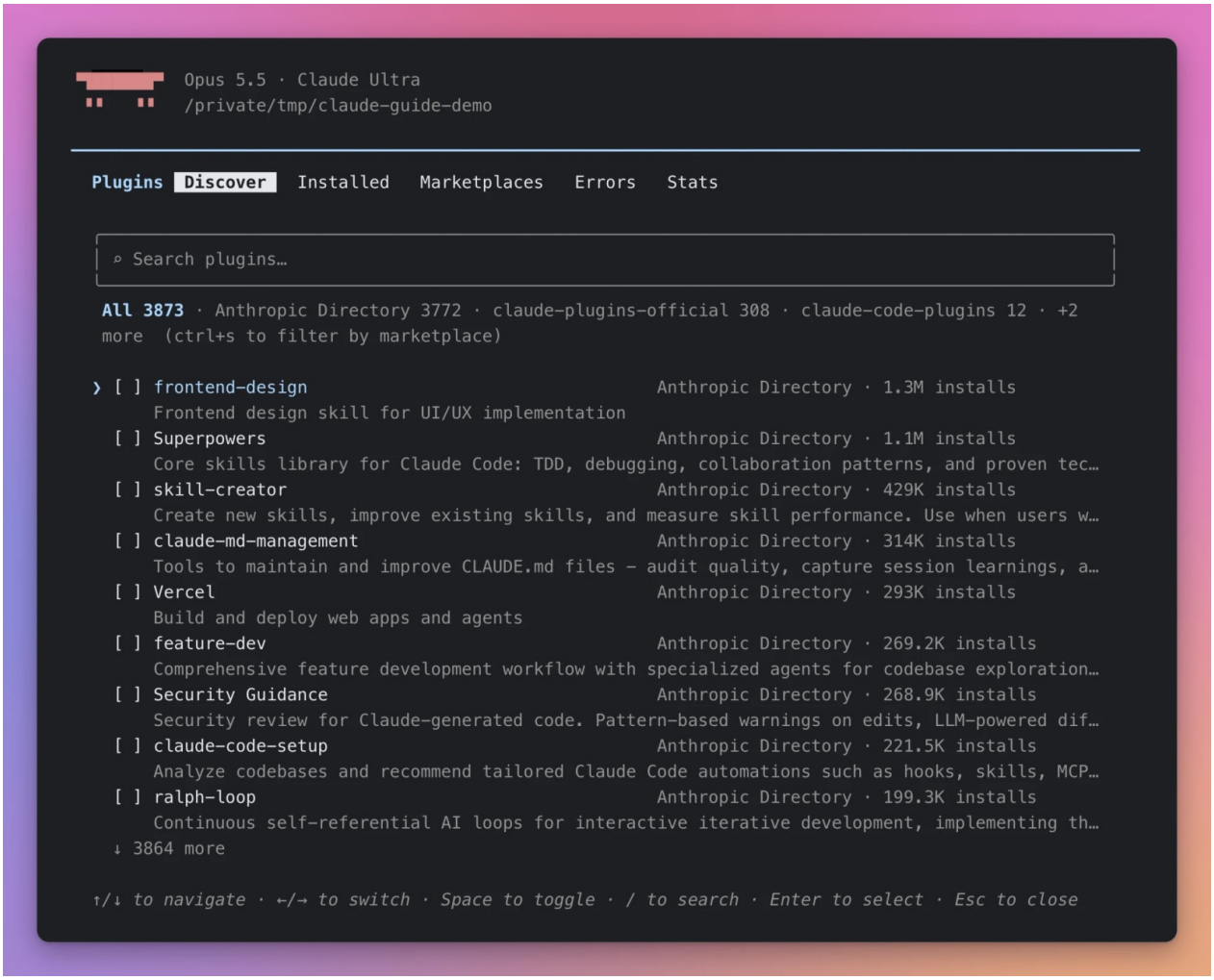
Less known, and occasionally super handy: `claude mcp serve` exposes Claude Code's own file, search, and shell capabilities as an MCP server that another MCP client can call.

If you want to build your own server, I wrote a complete walkthrough in how to build an MCP server, and if your tools run long jobs or need interactive confirmation forms mid-call, see how to use MCP Tasks and Elicitations in Claude Code. Anthropic's MCP guide covers every transport and OAuth detail.

Plugins, code intelligence, evals, Code Mode and Mods

Skills, custom agents, hooks, LSP servers, and MCP servers each solve one slice of customization. A **plugin** packages any combination of them into a single versioned unit you install, enable, disable, update, and share through a marketplace.

Run `/plugin` inside a session to browse directories, inspect installed plugins, manage marketplaces, and check plugin errors:



Inspecting installed plugins and their real token cost

From the shell, `claude plugin list` separates locally installed plugins from plugins synced automatically from your `claude.ai` account. On my Mac that is 8 local plugins (6 enabled, 2 disabled: `github` and `playwright`) plus 4 synced from `claude.ai` (`frontend-design`, `design`, `cowork-plugin-management`, and `marketing`).

Even better, `claude plugin details <name>` prints a plugin's exact component inventory and projected token overhead (both always-on startup tokens and per-invocation tokens). Here are the real numbers from running `claude plugin details` across every enabled local plugin on my Mac:

Plugin	Components	Always-on cost	On-invoke cost
`code-review@claude-plugins-official`	1 skill (`code-review`)	~22 tok	~2.4k tok
`code-simplifier@claude-plugins-official`	1 agent (`code-simplifier`)	~66 tok	~910 tok
`commit-commands@claude-plugins-official`	3 skills (`clean_gone`, `commit`, `commit-push-pr`)	~105 tok	~180–600 tok
`context7@claude-plugins-official`	1 MCP server (`context7`)	~0 tok	Resolved at runtime
`typescript-lsp@claude-plugins-official`	1 LSP server (`typescript`)	~0 tok	No model context cost

Plugin	Components	Always-on cost	On-invoke cost
`hey@37signals`	1 skill (`hey`)	~88 tok	~15k tok

That table reveals two smart patterns. First, **MCP and LSP plugins cost ~0 tokens at startup** because MCP schemas defer via Tool Search and language servers run out of band. Second, a plugin with a huge skill (like `hey` at ~15k tokens on invoke) stays cheap until you actually invoke it, and disabling an unused plugin with `claude plugin disable <name>` removes even its always-on footprint without uninstalling it.

Custom marketplaces live under `extraKnownMarketplaces` in `settings.json` (for example, I pull `hey@37signals` from `basecamp/claude-plugins`), or you can load a local plugin folder or `.zip` for a single session with `claude --plugin-dir ./my-plugin`.

Code intelligence plugins (LSP)

Without a language server, Claude finds symbol definitions by running grep across your repo and catches TypeScript or Go errors only when it remembers to run the compiler. Installing a **code intelligence plugin** (`typescript-lsp`, `pyright-lsp`, `rust-analyzer-lsp`, `gopls-lsp`) gives Claude the `LSP` tool: instant jump-to-definition, find-references, and live diagnostics after every edit, at zero startup token cost.

One catch you already saw in my earlier `cli-live-task.webp` screenshot: LSP plugins expect the underlying language server binary on your `PATH`. Seeing `Language server "plugin:typescript-lsp:typescript" couldn't start: its program "typescript-language-server" wasn't found` means the plugin is enabled, but the binary isn't installed globally yet. Install it once (or disable the plugin if you don't write TypeScript):

```
npm install -g typescript-language-server typescript
```

Authoring, validating and grading plugins with `claude plugin eval`

If you build internal plugins for your team, Claude Code now ships the whole plugin development lifecycle in the CLI:

```
# Scaffold a new plugin at ~/.claude/skills/<name>/ (auto-loads next session)
claude plugin init my-team-workflow

# Validate plugin.json, skills, agents, hooks, and commands
claude plugin validate ./my-team-workflow

# Run an eval suite against the plugin plus a no-plugin baseline arm
claude plugin eval ./my-team-workflow
```

`claude plugin eval` runs test cases (`evals/**/case.yaml` or `prompt.md` + `graders/*.md`) both with your plugin enabled and against a no-plugin baseline so you can prove your skill or agent actually improves scores before shipping it to teammates.

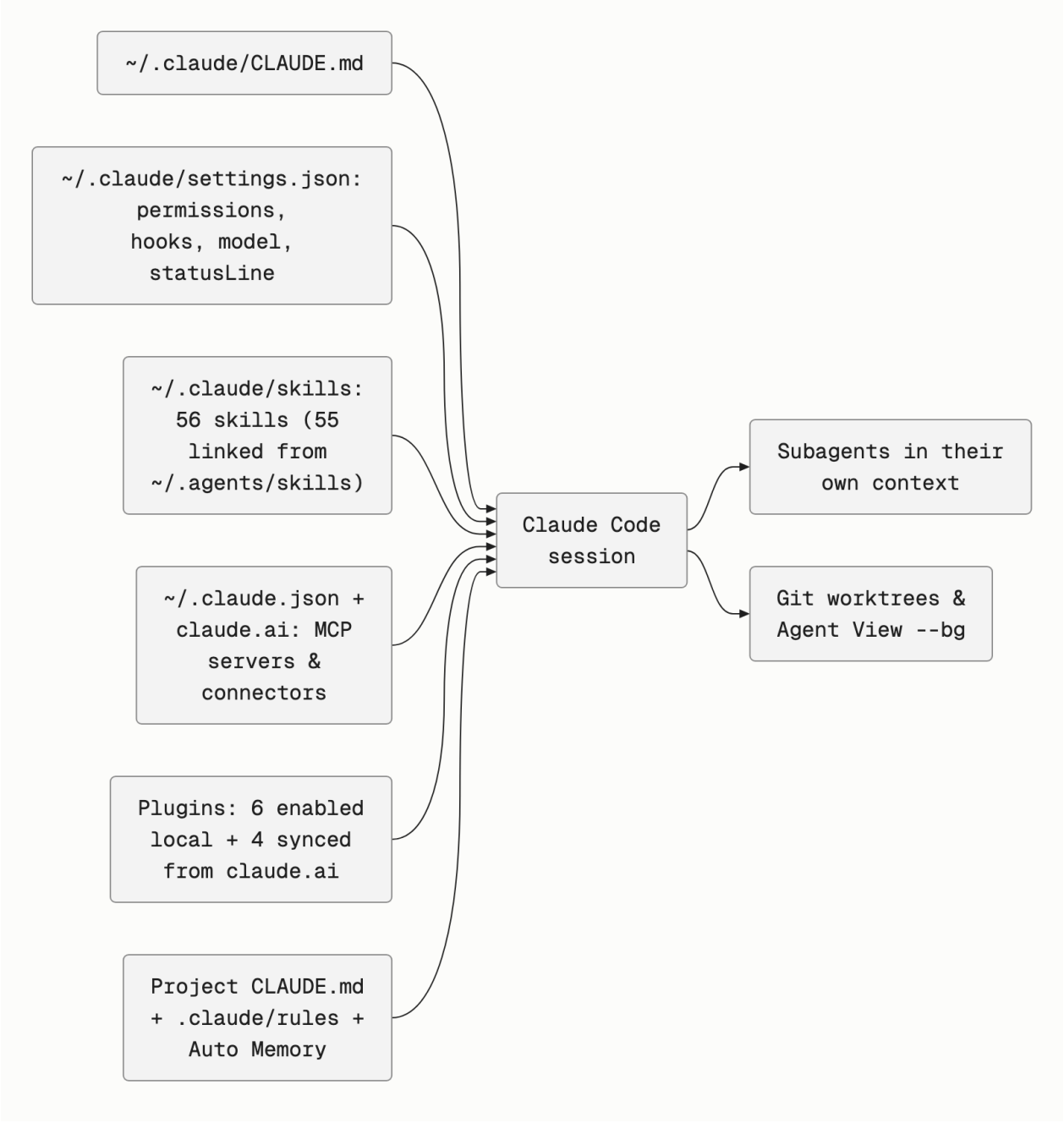
Treat third-party plugins with the same caution as npm packages: inspect a plugin's hooks and scripts before enabling it on sensitive repos, because plugin hooks execute automatically on lifecycle events with your user privileges. See Anthropic's plugins docs and plugin evals guide.

Code Mode, Mods and `easel`

Three features push Claude Code past standard text-and-tool turns:

- **Code Mode (`CLAUDE_CODE_EXPERIMENTAL_CODE_MODE=1`):** Instead of making one model round-trip per tool call (`Read`, wait, `Grep`, wait, `Edit`), Claude writes a short JavaScript snippet that chains multiple tool calls in a single turn and moves long loops to the background in auto mode so you can keep chatting. I tested it on a real project in how to use Code Mode in Claude Code.
- **Claude Code Mods:** Mods are executable extensions packaged inside plugins that can draw custom UI panes, buttons, and bands above the prompt, intercept or rewrite tool calls and events, register commands, and run unit tests offline with `claude plugin test`. After editing a plugin or mod on disk, run `/reload-plugins` to pick up the change mid-session without restarting. Because a mod runs inside Claude Code's host process outside the Bash sandbox, organizations can restrict user-installed mods through managed settings. I built a small command-guard mod in how Claude Code mods change the agent loop.
- **`easel` (built-in charting mod):** Added in v2.1.330 on macOS and Linux, `easel` is a built-in mod that draws visual charts right in the terminal whenever you ask for one, and renders charts automatically underneath MCP metrics responses (toggle automatic charts in `/config`).

Putting all of these configuration and extension layers together, here is how a Claude Code session on my Mac is assembled from disk:



Code review and security scanning

Asking an agent to ”review my code” inside the same session that just wrote it has an obvious flaw: the model is biased toward the assumptions it already made. Claude Code solves this with three dedicated review layers plus two security-focused plugins.

Local `/code-review`, cloud `ultrareview`, and GitHub PR Code Review

Review method	Where it runs	How you trigger it	Best for
`/code-review`	Local session	`/code-review`, `/code-review 1234`, `/code-review src/auth`	Quick pre-commit sanity checks on your current branch or a specific path

Review method	Where it runs	How you trigger it	Best for
`ultrareview` (`/code-review ultra` or `claude ultrareview`)	Cloud multi-agent sandbox	`/code-review ultra`, `claude ultrareview 142 --post`	Deep pre-merge bug hunts where findings are reproduced and verified
GitHub PR Code Review	Automatic on GitHub PRs	Claude GitHub App + `@claude review` (Team / Enterprise)	Every pull request automatically, tuned with `REVIEW.md`

Inside a session, `/code-review` accepts a branch diff, a revision range (`HEAD~15..HEAD`), a single commit (`abc1234^..abc1234`), a pull request number, or a directory path. Pass `--fix` to let Claude apply fixes directly or `--comment` to post findings.

For bigger or riskier branches, `ultrareview` launches a fleet of parallel reviewer agents in an Anthropic cloud sandbox that independently verify each bug before reporting it, cutting down style nits and false positives:

```
# Review the current branch against the default branch and print findings locally
claude ultrareview

# Review PR #142 and post findings as a comment on the pull request
claude ultrareview 142 --post

# Machine-readable output for scripts or CI
claude ultrareview 142 --json --trust-workspace
```

When you run `ultrareview` on a local branch (rather than a GitHub PR number), Claude uploads a bundle of the repo state while filtering out credential-shaped files such as `.env` and `*.tfvars`, and shows you the file count and estimated cost before launching. On PR targets, nothing leaves your machine.

Customizing PR reviews with `REVIEW.md`

If your team uses GitHub PR Code Review, put review-only rules in a `REVIEW.md` file at the repository root instead of stuffing them into `CLAUDE.md`. `REVIEW.md` lets you define what counts as a blocking bug, cap low-priority nits, ignore generated paths, and list invariants every PR must satisfy:

```
# Review instructions

## Always check
- Every API route handler validates request input before touching the database
- New SQL queries use bound parameters, never string interpolation
- Migrations are backward-compatible with the currently deployed app version

## Do not report
- Formatting or import order (Prettier and ESLint enforce those in CI)
- Generated files under dist/ or src/generated/
```

Security scanning and automated PR fixes (`/security-review`, `/autofix-pr`)

You don't have to install anything to run a fast security pass on your branch:

- `/security-review` is built straight into Claude Code and audits the diff between your current branch and origin's default branch for security vulnerabilities.
- `/autofix-pr` spawns a cloud session watching your checked-out branch's open GitHub PR (detected via `gh pr view`) and automatically pushes fixes whenever CI checks fail or reviewers leave comments. Pass a custom scope if you want it narrower, such as `/autofix-pr only fix lint and type errors`.

For deeper continuous coverage, two official security plugins catch vulnerabilities at different points in the workflow:

- `security-guidance` (268k+ installs in `/plugin`) runs checks inline as Claude edits files, finishes turns, and prepares commits or pushes, flagging patterns such as command injection, unsanitized HTML, or unsafe deserialization so Claude fixes them in the same session.
- `claude-security` scans an existing codebase (or just your branch diff) for vulnerabilities and turns confirmed findings into concrete patches you review and apply.

Browsers, Computer Use, Cloud, Mobile and Artifacts

Modern coding work rarely stays inside plain text files. You debug web apps in Chrome, verify native apps on macOS, kick off long jobs while closing your laptop, approve prompts from your phone, and share visual summaries with teammates.



Browser debugging with Claude in Chrome (`--chrome`)

Start Claude Code with `claude --chrome` to connect it to your Chrome browser via the Claude in Chrome extension. Claude can inspect rendered pages, read browser console logs and network failures, fill forms, test interactive flows, and extract data from live web apps without writing throwaway Playwright scripts.

If you prefer keeping browser access off by default (as I do in `~/.claude/settings.json` via `"disallowedTools": ["mcp__claude__claude-in-chrome"]` because I run browser checks inside my test VM), pass `--no-chrome` on launch or disable it in settings.

CLI Computer Use on macOS

On macOS, Claude Code can also enable **Computer Use** directly from the CLI so Claude sees the screen, opens apps, clicks buttons, and types text inside GUI-only tools and native apps.

A few built-in guardrails shape how CLI Computer Use behaves:

- You approve which specific applications Claude is allowed to interact with per session.
- Only one session can use Computer Use at a time, and screenshots are downscaled automatically.
- You can halt it at any instant if a click goes somewhere unexpected.

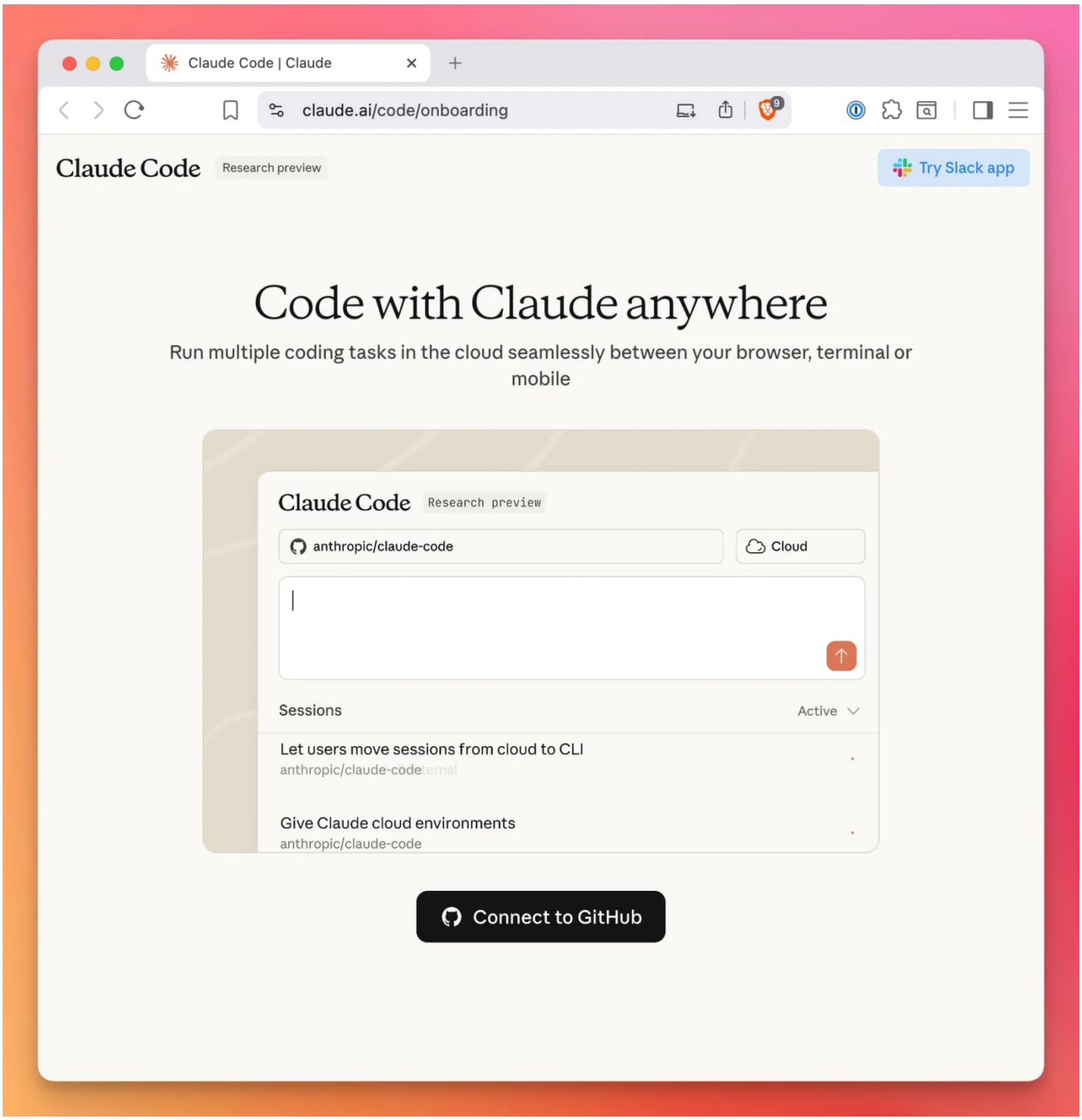
If you build native Mac apps every day, driving your live desktop still interrupts whatever else you're doing. That's why my daily setup runs native app tests inside a headless macOS VM (mac-test-vm) and inspects accessibility trees with agent-desktop instead of letting agents borrow my live cursor.

Cloud sessions, self-hosted runners and `--teleport`

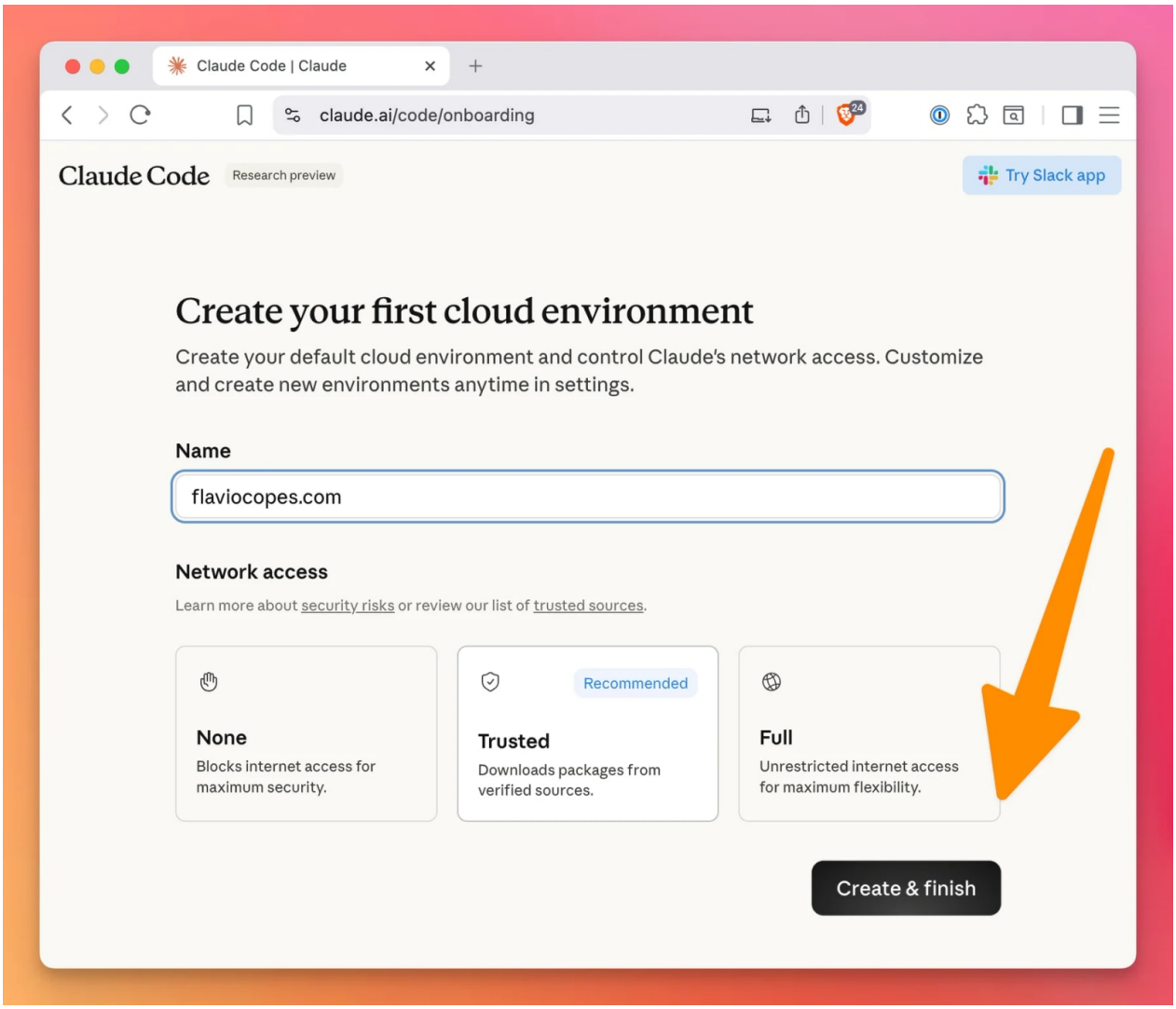
When you want a task to run without your laptop awake, open claude.ai/code or launch a cloud session straight from your terminal:

```
claude --cloud "Add dark mode support to the pricing table and open a PR"
```

Push any commits the task needs first, because cloud environments clone from GitHub and can't see local uncommitted files or your personal `~/.claude/skills/`.



Cloud environments let you attach environment variables, setup scripts, dependency caching, and a network access policy. Start with **None** or **Trusted** network access and open **Full** access only when a task genuinely needs arbitrary external hosts:



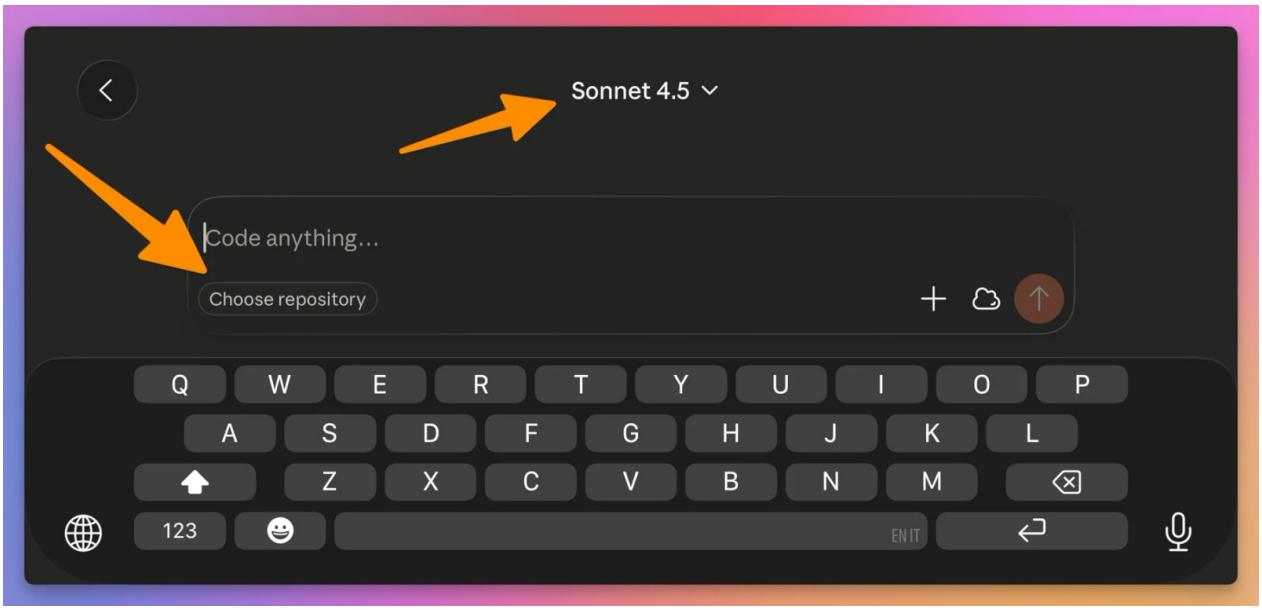
Two CLI flags move work between cloud sessions and your own infrastructure:

- `claude --teleport <session-id>` pulls an active or finished cloud session back down into your local terminal so you can keep iterating locally.
- `claude --environment ccpool_...` routes a cloud session onto a **self-hosted environment** runner running on your own infrastructure (useful when your code or internal registries can't leave your VPC).

Desktop handoff, Remote Control and Mobile

You don't have to pick one interface for the entire lifetime of a task:

- **CLI to Desktop:** Run `claude --desktop --continue` (or `--resume <id>`) in a terminal to open that exact live conversation inside the Claude desktop app when you want visual side-by-side diffs or a live browser preview.
- **Local to Phone/Web with Remote Control:** Type `/remote-control` inside a running local session (or start it with `claude --remote-control "billing-fix"`). Your local machine keeps executing the tools and keeps access to your local files and MCP servers, while you monitor progress, answer blocking questions, or steer the session from `claude.ai/code` or the Claude mobile app (iOS and Android).
- **Mobile cloud tasks:** Open the **Code** tab in the mobile app to start a fresh cloud session on a GitHub repository directly from your phone:



Checking progress or unblocking a question on my phone works well when I step away from my desk, though I always inspect real diffs on a full monitor before merging. I compare mobile options across coding agents in using coding agents from your phone.

Visual output: Artifacts, `/design`, `/design-sync`, `/slides` and `/dataviz`

Terminal output is terrible for comparing three UI options, reviewing a funnel breakdown, or walking a teammate through a complex PR. On Pro, Max, Ultra, Team, and Enterprise plans signed in via `/login`, Claude Code can publish a self-contained HTML or Markdown page straight from your session to a private **Artifact** URL on `claude.ai` that updates in place as you iterate (and can even pull live data through your connected MCP servers or offer file downloads):

```
Make an artifact that walks through this branch's diff inline and highlights every schema cha
```

Several bundled visual skills build on Artifacts directly from your prompt:

- **`/design` and `/design-sync`:** `/design` drafts multi-artboard UI mockups, landing pages, or screen flows as a Claude Design artifact. `/design-sync` goes one step further by converting your repository's React design system and uploading it to Claude Design so mockups use your **real production components** (authenticate once with `/design-login`).
- **`/slides`:** Generates a slide deck artifact from a brief, code audit, or quarterly summary.
- **`/dataviz`:** Picks chart types and theme-safe palettes for data dashboards and inline SVGs.
- **`/artifacts`:** Lists artifacts you own or share, letting you attach an existing one back into a session or copy its link.

Artifacts stay private until you click **Share** to generate an org-wide or public link. They are single-page captures with no custom backend, so use them for walkthroughs, reports, slides, and design comparisons rather than hosted internal apps.

Headless CLI, CI/CD, Agent SDK and teams

Interactive sessions shine when you're in the loop. Once a check or workflow proves useful, you can script it in Bash, run it in CI, or embed Claude Code directly inside your own TypeScript or Python program.

Headless scripting with `-p`, `--json-schema` and `--bare`

Pass `-p` (`--print`) to run a single non-interactive prompt, print the response to stdout, and exit. Combine it with `--output-format json` (or `stream-json` for live NDJSON events), `--json-schema` to validate the response shape, `--max-budget-usd` to hard-cap spend, and `--no-session-persistence` so scripted runs don't clutter your `/resume` history.

Here is a real structured audit I ran against the `release-notes` test repo on Haiku 5.5:

```
claude -p --model haiku --effort low \
  --allowedTools "Read,Glob,Grep" \
  --no-session-persistence \
  --max-budget-usd 0.25 \
  --output-format json \
  --json-schema '{"type":"object","properties":{"packageName":{"type":"string"},"releaseCount
  "Inspect package.json and CHANGELOG.md and return the requested fields."
```

It completed in **2.54 seconds**, cost **$0.0082**, and returned this validated `structured_output` object:

```
{
  "packageName": "release-notes",
  "releaseCount": 2,
  "latestVersion": "1.2.0",
  "scripts": ["test"]
}
```

Four flags matter most when putting `-p` inside shell scripts or cron jobs (if you're building shell wrappers around it, see the free Shell Scripting Course):

- **`--bare`:** Strips out user and project hooks, LSPs, plugin sync, Auto Memory, keychain reads, and automatic `CLAUDE.md` discovery so CI runs are 100% deterministic. You pass exact context explicitly via `--system-prompt`, `--settings`, `--mcp-config`, or `--add-dir`. Importantly, `--bare` authenticates **strictly via `ANTHROPIC_API_KEY` or `apiKeyHelper`** and refuses OAuth logins. When I ran `claude -p --bare --model haiku -- "Reply with ok"` on my Mac without an API key exported, it exited immediately with `Not logged in · Please run /login`.
- **`--permission-prompts none`:** Automatically denies any tool call that would otherwise prompt for approval during a `-p` run, while still respecting your permission mode and `--allowedTools`.
- **`--output-format stream-json`:** Streams structured NDJSON events as Claude thinks and runs tools. Add `--include-partial-messages`, `--include-hook-events`, or `--forward-subagent-text` when building live dashboards or log processors on top of the CLI.
- **Remember `--` before the prompt** whenever you use variadic flags like `--allowedTools`, `--disallowedTools`, `--add-dir`, or `--mcp-config`.

Keep in mind that `-p` skips the interactive workspace trust dialog and is **not** read-only by default. Always pass `--allowedTools "Read,Glob,Grep"` or `--permission-mode dontAsk` for read-only audits.

GitHub Actions and GitLab CI/CD

To let teammates mention `@claude` in GitHub issues and pull requests, or trigger Claude automatically on CI events, use Anthropic's official `claude-code-action`.

From any GitHub repository on your machine (with `gh` authenticated), run `/install-github-app` inside Claude Code. The wizard installs the Claude GitHub App, configures your authentication secret, and opens a PR adding workflow files under `.github/workflows/`. You can also wire the action manually in a custom workflow like `.github/workflows/claude.yml`:

```
name: Claude Code
on:
  issue_comment:
    types: [created]
  pull_request_review_comment:
    types: [created]
jobs:
  claude:
    if: contains(github.event.comment.body, '@claude')
    runs-on: ubuntu-latest
    permissions:
      contents: write
      pull-requests: write
      issues: write
      id-token: write
      actions: read
    steps:
      - uses: actions/checkout@v6
        with:
          fetch-depth: 1
      - uses: anthropics/claude-code-action@v1
        with:
          anthropic_api_key: ${{ secrets.ANTHROPIC_API_KEY }}
```

`id-token: write` handles GitHub App authentication, and `actions: read` lets Claude inspect failing CI runs on pull requests. My free GitHub Actions CI/CD Course covers workflow triggers, secrets, and job permissions if you want a refresher.

On GitLab, GitLab maintains an official beta Claude Code CI/CD integration built on top of the CLI and Agent SDK for turning issues into merge requests and responding to `@claude` mentions inside pipelines.

Building custom agents with the Claude Agent SDK

Shelling out to `claude -p` handles 80% of automation cleanly. Graduate to the **Claude Agent SDK** (`@anthropic-ai/claude-agent-sdk` in TypeScript or `claude-agent-sdk` in Python) when you need multi-turn programmatic control, typed streaming events, in-process custom MCP tools, custom approval callbacks, or external session storage in a database or object store.

Here is a minimal TypeScript agent that streams reasoning and tool calls while reviewing and fixing a module:

```
import { query } from '@anthropic-ai/claude-agent-sdk'

for await (const message of query({
  prompt: 'Inspect parse-notes.mjs for edge cases and fix any bugs you find.',
  options: {
    allowedTools: ['Read', 'Edit', 'Glob'],
    permissionMode: 'acceptEdits',
  },
})) {
  if (message.type === 'assistant' && message.message?.content) {
    for (const block of message.message.content) {
      if ('text' in block) {
        console.log(block.text)
      } else if ('name' in block) {
        console.log(`Tool: ${block.name}`)
      }
    }
  } else if (message.type === 'result') {
    console.log(`Done: ${message.subtype}`)
  }
}
```

The SDK reuses Claude Code's exact agent loop, `CLAUDE.md` loading, skills, hooks, subagents, file checkpointing, prompt caching, and OpenTelemetry tracing. See the TypeScript Agent SDK reference and Python Agent SDK reference for full options.

Teams, cloud providers and gateways

For companies deploying Claude Code across an engineering org rather than individual subscriptions, four pieces do the heavy lifting:

- **Cloud provider backends:** Claude Code runs directly against **Amazon Bedrock**, **Claude Platform on AWS**, **Google Cloud's Agent Platform**, and **Microsoft Foundry** using your existing IAM and cloud billing. (Note that consumer claude.ai features like Cloud sessions, Desktop, Mobile, `/schedule`, `/voice`, Remote Control, and `ultrareview` require a claude.ai account.)
- **Managed settings:** IT and security teams can push `managed-settings.json` via MDM or deliver **server-managed settings** from the `claude.ai` console to enforce `availableModels`, `deniedModels`, allowed MCP servers, plugin marketplaces, and mod policies.
- **Claude apps gateway (`claude gateway`):** Anthropic ships a self-hosted gateway that sits in front of Bedrock, AWS, Google Cloud, or Foundry to provide OIDC SSO login, per-group model routing, daily/weekly/monthly spend caps, and centralized OTLP telemetry.
- **OpenTelemetry observability:** Claude Code exports traces, token metrics, and cost events to any OpenTelemetry backend (Grafana, Datadog, Honeycomb) so teams can track adoption and spend without scraping developer laptops.

Troubleshooting and when Claude gets stuck

Even well-configured setups hit snags: a hook silently fails, a plugin breaks, context fills up, or Claude starts looping on the same failing test. Start with Claude Code's built-in diagnostic com-

mands before guessing.

The diagnostic toolkit

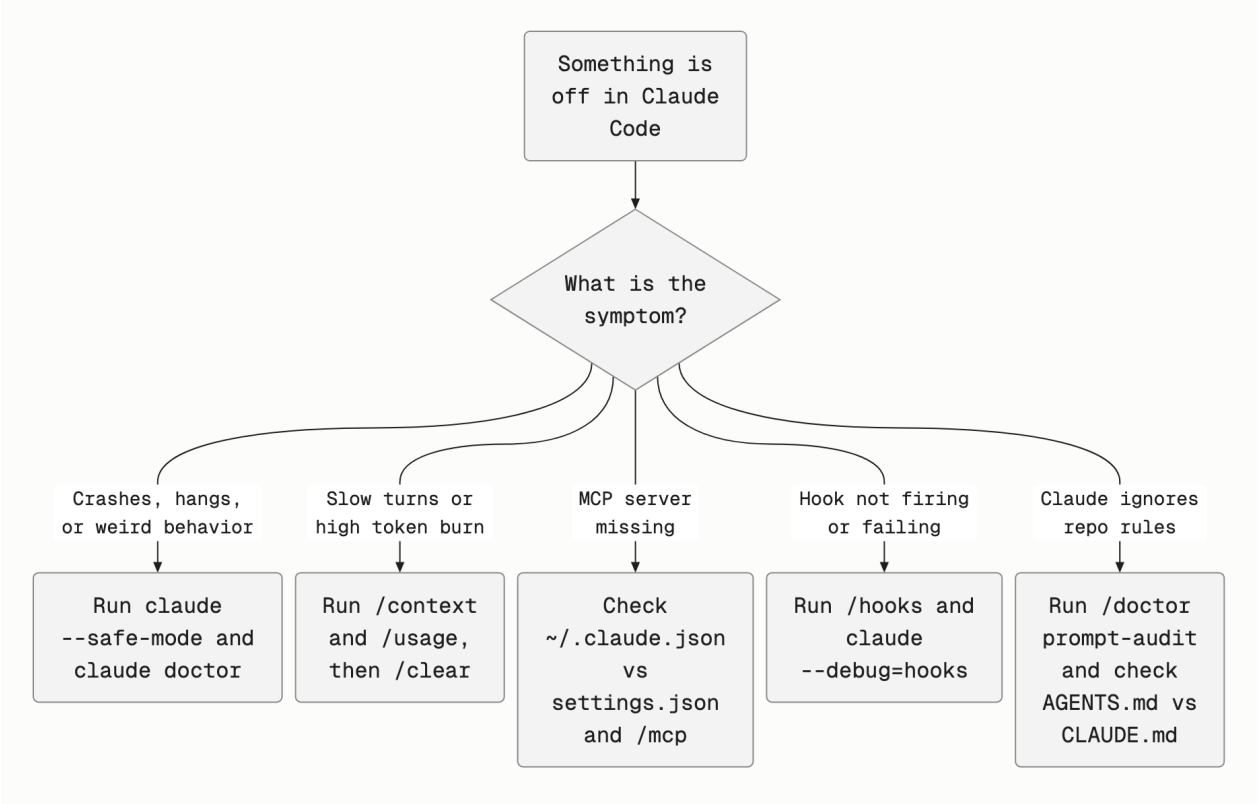
```
# Non-interactive install and config health check
claude doctor

# Start with ALL customizations disabled (CLAUDE.md, skills, plugins, hooks, MCP, mods)
claude --safe-mode

# Filter live debug logs to specific subsystems (or exclude noisy ones)
claude --debug=hooks,mcp
claude --debug='!hooks,!mcp' --debug-file /tmp/claude-debug.log
```

Inside a running session, `/context`, `/hooks`, `/mcp`, `/memory`, `/permissions`, `/status`, `/skill-doctor`, and `/doctor prompt-audit` show you the exact resolved state Claude is actually seeing rather than what you *think* you configured. Two other built-in diagnostics save hours when something behaves strangely:

- **`/debug`** enables debug logging for the active session and has Claude read its own session debug log to pinpoint why a hook, MCP server, or rule didn't fire.
- **`/insights`** generates a local HTML report analyzing your recent sessions across projects, highlighting recurring friction patterns and where runs go off track.



Twelve real problems and their exact fixes

Symptom	Likely cause	Fix
Claude repeats old mistakes or gets sluggish mid-session	Context rot from reusing one chat across multiple tasks	Summarize the 2–3 facts you need, run `/clear`, and start clean
One turn suddenly takes 3x longer and burns huge tokens	Prompt cache miss from switching `/model` or toggling `/fast` mid-chat	Pick your model and `/fast` at session start; change `/effort` instead of `/model` when possible
Auto-compact keeps firing back to back	Huge tool outputs (logs, minified bundles) filling the window	Offload noisy reads to a subagent, block `dist/` or `vendor/` in `permissions.deny`, or tune `/autocompact`
MCP server added in `~/.claude/settings.json` never appears	Personal MCP servers belong in `~/.claude.json`, not `settings.json` (and Desktop's `claude_desktop_config.json` isn't read by the CLI)	Add it with `claude mcp add` (or `claude mcp add-from-claude-desktop`) and verify with `claude mcp list`
Warning: `typescript-lsp couldn't start`	Plugin enabled, but `typescript-language-server` isn't on `PATH`	Run `npm install -g typescript-language-server typescript` or `claude plugin disable typescript-lsp`
`/sandbox` breaks `npm install` or local build scripts, or doesn't stop `Read`	Network domain isn't in `sandbox.network.allowedDomains`, or tool runs outside Bash sandbox	Add needed domains in `sandbox.network.allowedDomains`; add `Read`/`WebFetch` rules in `permissions.deny`
`claude --bg` refuses to launch in a new folder	Background sessions require an already trusted workspace	Run `claude` interactively in that folder once and accept the trust dialog
`claude -p --allowedTools "Bash" "prompt"` fails with missing prompt	`--allowedTools` is variadic and eats trailing positional arguments	Put the prompt before `--allowedTools` or add `--` before the prompt
`claude -p --bare` prints `Not logged in` even though `claude` works	`--bare` intentionally skips OAuth and macOS Keychain	Export `ANTHROPIC_API_KEY` or pass an `apiKeyHelper` via `--settings`
Unattended scheduled script fails after a few days	Interactive CLI OAuth session expired	Run `claude setup-token` for a long-lived subscription token, or use an API key
Desktop Code tab runs a different Claude Code version than terminal `claude`	Desktop bundles and updates its own Claude Code binary independently	Check `/status` inside Desktop and run **Claude > Check for Updates...**
`/permissions` says "isn't available in this environment" in Desktop	Terminal-only TUI dialogs don't open in Desktop's chat composer	Edit `~/.claude/settings.json` directly, run `/update-config`, or manage settings from the CLI

If you run Claude Code on a remote Linux VPS over SSH (see my free SSH Course) and hit

connection drops, keep interactive sessions inside `tmux` or use `claude --bg` so your task survives a closed laptop lid.

How Claude Code compares and how I use it

Claude Code isn't the only coding agent on my machine, and no single tool wins every scenario. Here is how it stacks up against the other tools I write about and run regularly:

Tool	Interface focus	Strengths	Where I pick something else
Claude Code	CLI, Desktop, IDE, Cloud, Mobile	Deepest customization ecosystem (hooks, skills, plugins, Mods, worktrees, Agent View, Dynamic Workflows, Agent SDK) and 1M context on Opus 5.5 / Fable 5.2	When I want to mix OpenAI/Gemini models or manage multiple harnesses in one UI
Codex	Desktop app, CLI, IDE, Cloud	Strong OpenAI coding models, clean desktop app with built-in browser preview and Goal mode	When I want Claude Code's hook/plugin/MCP depth or Opus/Fable
Cursor	Full AI-native IDE	Best inline tab-autocomplete and editor-first workflow when editing code directly by hand	When I'm orchestrating parallel CLI sessions, worktrees, or headless scripts
OpenCode	Open-source terminal & desktop	Provider-agnostic, open source, easy to pair with local models or OpenRouter	When I want Anthropic's first-party Auto Mode, Cloud sessions, and `ultrareview`
T3 Code / **Herdr**	Multi-agent control planes	Run Claude Code, Codex, OpenCode, and Pi side by side across repos, worktrees, and PRs	They wrap Claude Code rather than replacing it

For a side-by-side workflow comparison across CLI agents, see OpenCode vs Claude Code vs Codex.

How I actually use Claude Code

Most days I run Claude Code in fullscreen terminal mode (`"tui": "fullscreen"`) alongside the Claude desktop app and T3 Code, defaulting to `opus[1m]` with `auto` mode turned on.

My `~/.claude/CLAUDE.md` stays short and holds rules that apply everywhere (like keeping native Mac app testing inside my headless test VM), while multi-step procedures live in shared skills under `~/.agents/skills/` symlinked into `~/.claude/skills/` so every coding agent on my Mac reads

the same single copy. Each repository gets its own `CLAUDE.md`, and I run `/doctor prompt-audit` whenever a repo's scripts or layout change so stale paths don't quietly mislead future runs.

When I have two or three independent tasks, I isolate them in Git worktrees (`claude -w`) or dispatch background sessions (`--bg`) and let my `Stop` hook send a macOS notification when each turn completes. The biggest habit change from looking at my own `/usage` diagnostics was discipline around `/clear`: as soon as a task is committed, I clear the session instead of dragging 80k tokens of old history into the next feature.

Where I skip Claude Code: fixing a two-character typo in a file I already have open is faster by hand, and on fast-moving frameworks or newly shipped APIs I explicitly tell Claude to check current docs (via `context7` or web search) and always read the diff myself before merging. Give it clear verification commands, enforce hard boundaries with permissions and hooks rather than polite prose, keep sessions short, and read what comes back.