Claude Code

Complete Documentation

By Anthropic

Production date：2026-03-17

An agentic coding tool that lives in your terminal,
understands your codebase, and helps you code faster.

Compiled from code.claude.com/docs
For the latest version, visit the website.

Generated by: github.com/bepsvpt/cc-docs

Table of Contents

Part 1: Getting Started

Claude Code overview

> *Claude Code is an agentic coding tool that reads your codebase, edits files, runs commands, and integrates with your development tools. Available in your terminal, IDE, desktop app, and browser.*

Claude Code is an AI-powered coding assistant that helps you build features, fix bugs, and automate development tasks. It understands your entire codebase and can work across multiple files and tools to get things done.

Get started

Choose your environment to get started. Most surfaces require a Claude subscription or Anthropic Console account. The Terminal CLI and VS Code also support third-party providers.

Terminal

The full-featured CLI for working with Claude Code directly in your terminal. Edit files, run commands, and manage your entire project from the command line.

To install Claude Code, use one of the following methods:

Native Install (Recommended)

macOS, Linux, WSL:

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

Windows PowerShell:

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

Windows CMD:

```
curl -fsSL https://claude.ai/install.cmd -o install.cmd && install.cmd && del
install.cmd
```

Windows requires Git for Windows. Install it first if you don't have it.

> **Info:**
>
> Native installations automatically update in the background to keep you on the latest version.

Homebrew

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

> **Info:**
>
> Homebrew installations do not auto-update. Run `brew upgrade claude-code` periodically to get the latest features and security fixes.

WinGet

```
winget install Anthropic.ClaudeCode
```

> **Info:**
>
> WinGet installations do not auto-update. Run `winget upgrade Anthropic.ClaudeCode` periodically to get the latest features and security fixes.

Then start Claude Code in any project:

```
cd your-project
claude
```

You'll be prompted to log in on first use. That's it! Continue with the Quickstart →

> **Tip:**

> See advanced setup for installation options, manual updates, or uninstallation instructions. Visit troubleshooting if you hit issues.

VS Code

The VS Code extension provides inline diffs, @-mentions, plan review, and conversation history directly in your editor.

- Install for VS Code
- Install for Cursor

Or search for "Claude Code" in the Extensions view (`Cmd+Shift+X` on Mac, `Ctrl+Shift+X` on Windows/Linux). After installing, open the Command Palette (`Cmd+Shift+P` / `Ctrl+Shift+P`), type "Claude Code", and select **Open in New Tab**.

Get started with VS Code →

Desktop app

A standalone app for running Claude Code outside your IDE or terminal. Review diffs visually, run multiple sessions side by side, schedule recurring tasks, and kick off cloud sessions.

Download and install:

- macOS (Intel and Apple Silicon)
- Windows (x64)
- Windows ARM64 (remote sessions only)

After installing, launch Claude, sign in, and click the **Code** tab to start coding. A paid subscription is required.

Learn more about the desktop app →

Web

Run Claude Code in your browser with no local setup. Kick off long-running tasks and check back when they're done, work on repos you don't have locally, or run multiple tasks in parallel. Available on desktop browsers and the Claude iOS app.

Start coding at claude.ai/code.

Get started on the web →

JetBrains

A plugin for IntelliJ IDEA, PyCharm, WebStorm, and other JetBrains IDEs with interactive diff viewing and selection context sharing.

Install the Claude Code plugin from the JetBrains Marketplace and restart your IDE.

Get started with JetBrains →

What you can do

Here are some of the ways you can use Claude Code:

Claude Code handles the tedious tasks that eat up your day: writing tests for untested code, fixing lint errors across a project, resolving merge conflicts, updating dependencies, and writing release notes.

```
claude "write tests for the auth module, run them, and fix any failures"
```

Describe what you want in plain language. Claude Code plans the approach, writes the code across multiple files, and verifies it works.

For bugs, paste an error message or describe the symptom. Claude Code traces the issue through your codebase, identifies the root cause, and implements a fix. See common workflows for more examples.

Claude Code works directly with git. It stages changes, writes commit messages, creates branches, and opens pull requests.

```
claude "commit my changes with a descriptive message"
```

In CI, you can automate code review and issue triage with GitHub Actions or GitLab CI/CD.

The Model Context Protocol (MCP) is an open standard for connecting AI tools to external data sources. With MCP, Claude Code can read your design docs in Google Drive, update tickets in Jira, pull data from Slack, or use your own custom tooling.

`CLAUDE.md` is a markdown file you add to your project root that Claude Code reads at the start of every session. Use it to set coding standards, architecture decisions, preferred libraries, and review checklists. Claude also builds auto memory as it works, saving learnings like build commands and debugging insights across sessions without you writing anything.

Create custom commands to package repeatable workflows your team can share, like `/review-pr` or `/deploy-staging`.

Hooks let you run shell commands before or after Claude Code actions, like auto-formatting after every file edit or running lint before a commit.

Spawn multiple Claude Code agents that work on different parts of a task simultaneously. A lead agent coordinates the work, assigns subtasks, and merges results.

For fully custom workflows, the Agent SDK lets you build your own agents powered by Claude Code's tools and capabilities, with full control over orchestration, tool access, and permissions.

Claude Code is composable and follows the Unix philosophy. Pipe logs into it, run it in CI, or chain it with other tools:

```
## Monitor logs and get alerted
tail -f app.log | claude -p "Slack me if you see any anomalies"

## Automate translations in CI
claude -p "translate new strings into French and raise a PR for review"

## Bulk operations across files
git diff main --name-only | claude -p "review these changed files for security
issues"
```

See the CLI reference for the full set of commands and flags.

Sessions aren't tied to a single surface. Move work between environments as your context changes:

- Step away from your desk and keep working from your phone or any browser with Remote Control
- Kick off a long-running task on the web or iOS app, then pull it into your terminal with `/teleport`
- Hand off a terminal session to the Desktop app with `/desktop` for visual diff review
- Route tasks from team chat: mention `@Claude` in Slack with a bug report and get a pull request back

Use Claude Code everywhere

Each surface connects to the same underlying Claude Code engine, so your CLAUDE.md files, settings, and MCP servers work across all of them.

Beyond the Terminal, VS Code, JetBrains, Desktop, and Web environments above, Claude Code integrates with CI/CD, chat, and browser workflows:

I want to...	Best option
Continue a local session from my phone or another device	Remote Control
Start a task locally, continue on mobile	Web or Claude iOS app
Automate PR reviews and issue triage	GitHub Actions or GitLab CI/CD
Get automatic code review on every PR	GitHub Code Review
Route bug reports from Slack to pull requests	Slack
Debug live web applications	Chrome
Build custom agents for your own workflows	Agent SDK

Next steps

Once you've installed Claude Code, these guides help you go deeper.

- Quickstart: walk through your first real task, from exploring a codebase to committing a fix
- Store instructions and memories: give Claude persistent instructions with CLAUDE.md files and auto memory
- Common workflows and best practices: patterns for getting the most out of Claude Code
- Settings: customize Claude Code for your workflow
- Troubleshooting: solutions for common issues
- code.claude.com: demos, pricing, and product details

Quickstart

> *Welcome to Claude Code!*

This quickstart guide will have you using AI-powered coding assistance in a few minutes. By the end, you'll understand how to use Claude Code for common development tasks.

Before you begin

Make sure you have:

- A terminal or command prompt open
  - If you've never used the terminal before, check out the terminal guide
- A code project to work with
- A Claude subscription (Pro, Max, Teams, or Enterprise), Claude Console account, or access through a supported cloud provider

> **Note:**
>
> This guide covers the terminal CLI. Claude Code is also available on the web, as a desktop app, in VS Code and JetBrains IDEs, in Slack, and in CI/CD with GitHub Actions and GitLab. See all interfaces.

Step 1: Install Claude Code

To install Claude Code, use one of the following methods:

Native Install (Recommended)

macOS, Linux, WSL:

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

Windows PowerShell:

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

Windows CMD:

```
curl -fsSL https://claude.ai/install.cmd -o install.cmd && install.cmd && del
install.cmd
```

Windows requires Git for Windows. Install it first if you don't have it.

> **Info:**
>
> Native installations automatically update in the background to keep you on the latest version.

Homebrew

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

> **Info:**
>
> Homebrew installations do not auto-update. Run `brew upgrade claude-code` periodically to get the latest features and security fixes.

WinGet

```
winget install Anthropic.ClaudeCode
```

> **Info:**
>
> WinGet installations do not auto-update. Run `winget upgrade Anthropic.ClaudeCode` periodically to get the latest features and security fixes.

Step 2: Log in to your account

Claude Code requires an account to use. When you start an interactive session with the `claude` command, you'll need to log in:

```
claude
## You'll be prompted to log in on first use
```

```
/login
## Follow the prompts to log in with your account
```

You can log in using any of these account types:

- Claude Pro, Max, Teams, or Enterprise (recommended)
- Claude Console (API access with pre-paid credits). On first login, a "Claude Code" workspace is automatically created in the Console for centralized cost tracking.
- Amazon Bedrock, Google Vertex AI, or Microsoft Foundry (enterprise cloud providers)

Once logged in, your credentials are stored and you won't need to log in again. To switch accounts later, use the `/login` command.

Step 3: Start your first session

Open your terminal in any project directory and start Claude Code:

```
cd /path/to/your/project
claude
```

You'll see the Claude Code welcome screen with your session information, recent conversations, and latest updates. Type `/help` for available commands or `/resume` to continue a previous conversation.

> **Tip:**
>
> After logging in (Step 2), your credentials are stored on your system. Learn more in Credential Management.

Step 4: Ask your first question

Let's start with understanding your codebase. Try one of these commands:

```
what does this project do?
```

Claude will analyze your files and provide a summary. You can also ask more specific questions:

```
what technologies does this project use?
```

```
where is the main entry point?
```

```
explain the folder structure
```

You can also ask Claude about its own capabilities:

```
what can Claude Code do?
```

```
how do I create custom skills in Claude Code?
```

```
can Claude Code work with Docker?
```

> **Note:**
>
> Claude Code reads your project files as needed. You don't have to manually add context.

Step 5: Make your first code change

Now let's make Claude Code do some actual coding. Try a simple task:

```
add a hello world function to the main file
```

Claude Code will:

1. Find the appropriate file
2. Show you the proposed changes
3. Ask for your approval
4. Make the edit

> **Note:**

> Claude Code always asks for permission before modifying files. You can approve individual changes or enable "Accept all" mode for a session.

Step 6: Use Git with Claude Code

Claude Code makes Git operations conversational:

```
what files have I changed?
```

```
commit my changes with a descriptive message
```

You can also prompt for more complex Git operations:

```
create a new branch called feature/quickstart
```

```
show me the last 5 commits
```

```
help me resolve merge conflicts
```

Step 7: Fix a bug or add a feature

Claude is proficient at debugging and feature implementation.

Describe what you want in natural language:

```
add input validation to the user registration form
```

Or fix existing issues:

```
there's a bug where users can submit empty forms - fix it
```

Claude Code will:

- Locate the relevant code
- Understand the context

- Implement a solution
- Run tests if available

Step 8: Test out other common workflows

There are a number of ways to work with Claude:

Refactor code

```
refactor the authentication module to use async/await instead of callbacks
```

Write tests

```
write unit tests for the calculator functions
```

Update documentation

```
update the README with installation instructions
```

Code review

```
review my changes and suggest improvements
```

> **Tip:**
>
> Talk to Claude like you would a helpful colleague. Describe what you want to achieve, and it will help you get there.

Essential commands

Here are the most important commands for daily use:

Command	What it does	Example
`claude`	Start interactive mode	`claude`
`claude "task"`	Run a one-time task	`claude "fix the build error"`

Command	What it does	Example
`claude -p "query"`	Run one-off query, then exit	`claude -p "explain this function"`
`claude -c`	Continue most recent conversation in current directory	`claude -c`
`claude -r`	Resume a previous conversation	`claude -r`
`claude commit`	Create a Git commit	`claude commit`
`/clear`	Clear conversation history	`/clear`
`/help`	Show available commands	`/help`
`exit` or Ctrl+C	Exit Claude Code	`exit`

See the CLI reference for a complete list of commands.

Pro tips for beginners

For more, see best practices and common workflows.

Instead of: “fix the bug”

Try: “fix the login bug where users see a blank screen after entering wrong credentials”

Break complex tasks into steps:

```
1. create a new database table for user profiles
2. create an API endpoint to get and update user profiles
3. build a webpage that allows users to see and edit their information
```

Before making changes, let Claude understand your code:

```
analyze the database schema
```

```
build a dashboard showing products that are most frequently returned by our UK
customers
```

- Press `?` to see all available keyboard shortcuts

- Use Tab for command completion
- Press ↑ for command history
- Type `/` to see all commands and skills

What's next?

Now that you've learned the basics, explore more advanced features:

- How Claude Code works Understand the agentic loop, built-in tools, and how Claude Code interacts with your project
- Best practices Get better results with effective prompting and project setup
- Common workflows Step-by-step guides for common tasks
- Extend Claude Code Customize with CLAUDE.md, skills, hooks, MCP, and more

Getting help

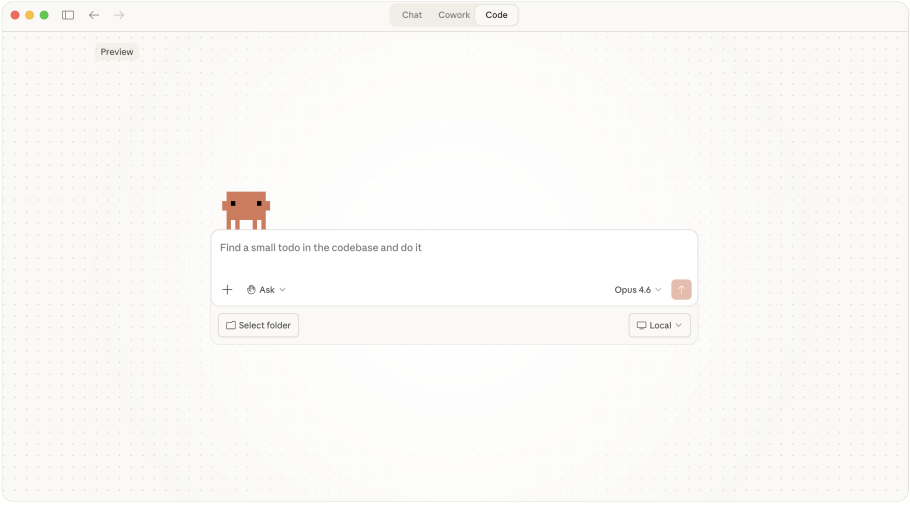
- **In Claude Code**: Type `/help` or ask "how do I..."
- **Documentation**: You're here! Browse other guides
- **Community**: Join our Discord for tips and support

Get started with the desktop app

> *Install Claude Code on desktop and start your first coding session*

The desktop app gives you Claude Code with a graphical interface: visual diff review, live app preview, GitHub PR monitoring with auto-merge, parallel sessions with Git worktree isolation, scheduled tasks, and the ability to run tasks remotely. No terminal required.

This page walks through installing the app and starting your first session. If you're already set up, see Use Claude Code Desktop for the full reference.



The Claude Code Desktop interface showing the Code tab selected, with a prompt box, permission mode selector set to Ask permissions, model picker, folder selector, and Local environment option

The desktop app has three tabs:

- **Chat**: General conversation with no file access, similar to claude.ai.
- **Cowork**: An autonomous background agent that works on tasks in a cloud VM with its own environment. It can run independently while you do other work.
- **Code**: An interactive coding assistant with direct access to your local files. You review and approve each change in real time.

Chat and Cowork are covered in the Claude Desktop support articles. This page focuses on the **Code** tab.

> **Note:**
>
> Claude Code requires a Pro, Max, Teams, or Enterprise subscription.

Install

Step 1: Download the app

Download Claude for your platform.

- macOS Universal build for Intel and Apple Silicon
- Windows For x64 processors

For Windows ARM64, download here.

Linux is not currently supported.

Step 2: Sign in

Launch Claude from your Applications folder (macOS) or Start menu (Windows). Sign in with your Anthropic account.

Step 3: Open the Code tab

Click the **Code** tab at the top center. If clicking Code prompts you to upgrade, you need to subscribe to a paid plan first. If it prompts you to sign in online, complete the sign-in and restart the app. If you see a 403 error, see authentication troubleshooting.

The desktop app includes Claude Code. You don't need to install Node.js or the CLI separately. To use `claude` from the terminal, install the CLI separately. See Get started with the CLI.

Start your first session

With the Code tab open, choose a project and give Claude something to do.

Step 1: Choose an environment and folder

Select **Local** to run Claude on your machine using your files directly. Click **Select folder** and choose your project directory.

> **Tip:**
>
> Start with a small project you know well. It's the fastest way to see what Claude Code can do. On Windows, Git must be installed for local sessions to work. Most Macs include Git by default.

You can also select:

- **Remote**: Run sessions on Anthropic's cloud infrastructure that continue even if you close the app. Remote sessions use the same infrastructure as Claude Code on the web.
- **SSH**: Connect to a remote machine over SSH (your own servers, cloud VMs, or dev containers). Claude Code must be installed on the remote machine.

Step 2: Choose a model

Select a model from the dropdown next to the send button. See models for a comparison of Opus, Sonnet, and Haiku. You cannot change the model after the session starts.

Step 3: Tell Claude what to do

Type what you want Claude to do:

- `Find a TODO comment and fix it`
- `Add tests for the main function`
- `Create a CLAUDE.md with instructions for this codebase`

A session is a conversation with Claude about your code. Each session tracks its own context and changes, so you can work on multiple tasks without them interfering with each other.

Step 4: Review and accept changes

By default, the Code tab starts in Ask permissions mode, where Claude proposes changes and waits for your approval before applying them. You'll see:

1. A diff view showing exactly what will change in each file
2. Accept/Reject buttons to approve or decline each change
3. Real-time updates as Claude works through your request

If you reject a change, Claude will ask how you'd like to proceed differently. Your files aren't modified until you accept.

Now what?

You've made your first edit. For the full reference on everything Desktop can do, see Use Claude Code Desktop. Here are some things to try next.

Interrupt and steer. You can interrupt Claude at any point. If it's going down the wrong path, click the stop button or type your correction and press **Enter**. Claude stops what it's doing and adjusts based on your input. You don't have to wait for it to finish or start over.

Give Claude more context. Type `@filename` in the prompt box to pull a specific file into the conversation, attach images and PDFs using the attachment button, or drag and drop files directly into the prompt. The more context Claude has, the better the results. See Add files and context.

Use skills for repeatable tasks. Type `/` or click + → **Slash commands** to browse built-in commands, custom skills, and plugin skills. Skills are reusable prompts you can invoke whenever you need them, like code review checklists or deployment steps.

Review changes before committing. After Claude edits files, a `+12 -1` indicator appears. Click it to open the diff view, review modifications file by file, and comment on specific lines. Claude reads your comments and revises. Click **Review code** to have Claude evaluate the diffs itself and leave inline suggestions.

Adjust how much control you have. Your permission mode controls the balance. Ask permissions (default) requires approval before every edit. Auto accept edits auto-accepts file edits for faster iteration. Plan mode lets Claude map out an approach without touching any files, which is useful before a large refactor.

Add plugins for more capabilities. Click the + button next to the prompt box and select **Plugins** to browse and install plugins that add skills, agents, MCP servers, and more.

Preview your app. Click the **Preview** dropdown to run your dev server directly in the desktop. Claude can view the running app, test endpoints, inspect logs, and iterate on what it sees. See Preview your app.

Track your pull request. After opening a PR, Claude Code monitors CI check results and can automatically fix failures or merge the PR once all checks pass. See Monitor pull request status.

Put Claude on a schedule. Set up scheduled tasks to run Claude automatically on a recurring basis: a daily code review every morning, a weekly dependency audit, or a briefing that pulls from your connected tools.

Scale up when you're ready. Open parallel sessions from the sidebar to work on multiple tasks at once, each in its own Git worktree. Send long-running work to the cloud so it continues even if you close the app, or continue a session on the web or in your IDE if a task takes longer than expected. Connect external tools like GitHub, Slack, and Linear to bring your workflow together.

Coming from the CLI?

Desktop runs the same engine as the CLI with a graphical interface. You can run both simultaneously on the same project, and they share configuration (CLAUDE.md files, MCP servers, hooks, skills, and settings). For a full comparison of features, flag equivalents, and what's not available in Desktop, see CLI comparison.

What's next

- Use Claude Code Desktop: permission modes, parallel sessions, diff view, connectors, and enterprise configuration
- Troubleshooting: solutions to common errors and setup issues
- Best practices: tips for writing effective prompts and getting the most out of Claude Code
- Common workflows: tutorials for debugging, refactoring, testing, and more

Advanced setup

> *System requirements, platform-specific installation, version management, and uninstallation for Claude Code.*

This page covers system requirements, platform-specific installation details, updates, and uninstallation. For a guided walkthrough of your first session, see the quickstart. If you've never used a terminal before, see the terminal guide.

System requirements

Claude Code runs on the following platforms and configurations:

- **Operating system**:
  - macOS 13.0+
  - Windows 10 1809+ or Windows Server 2019+
  - Ubuntu 20.04+
  - Debian 10+
  - Alpine Linux 3.19+
- **Hardware**: 4 GB+ RAM
- **Network**: internet connection required. See network configuration.
- **Shell**: Bash, Zsh, PowerShell, or CMD. On Windows, Git for Windows is required.
- **Location**: Anthropic supported countries

Additional dependencies

- **ripgrep**: usually included with Claude Code. If search fails, see search troubleshooting.

Install Claude Code

> **Tip:**
>
> Prefer a graphical interface? The Desktop app lets you use Claude Code without the terminal. Download it for macOS or Windows.
>
> New to the terminal? See the terminal guide for step-by-step instructions.

To install Claude Code, use one of the following methods:

Native Install (Recommended)

macOS, Linux, WSL:

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

Windows PowerShell:

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

Windows CMD:

```
curl -fsSL https://claude.ai/install.cmd -o install.cmd && install.cmd && del
install.cmd
```

Windows requires Git for Windows. Install it first if you don't have it.

> **Info:**
>
> Native installations automatically update in the background to keep you on the latest version.

Homebrew

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

> **Info:**
>
> Homebrew installations do not auto-update. Run `brew upgrade claude-code` periodically to get the latest features and security fixes.

WinGet

```
winget install Anthropic.ClaudeCode
```

> **Info:**

> WinGet installations do not auto-update. Run `winget upgrade Anthropic.ClaudeCode` periodically to get the latest features and security fixes.

After installation completes, open a terminal in the project you want to work in and start Claude Code:

```
claude
```

If you encounter any issues during installation, see the troubleshooting guide.

Set up on Windows

Claude Code on Windows requires Git for Windows or WSL. You can launch `claude` from PowerShell, CMD, or Git Bash. Claude Code uses Git Bash internally to run commands. You do not need to run PowerShell as Administrator.

Option 1: Native Windows with Git Bash

Install Git for Windows, then run the install command from PowerShell or CMD.

If Claude Code can't find your Git Bash installation, set the path in your settings.json file:

```
{
  "env": {
    "CLAUDE_CODE_GIT_BASH_PATH": "C:\\Program Files\\Git\\bin\\bash.exe"
  }
}
```

Option 2: WSL

Both WSL 1 and WSL 2 are supported. WSL 2 supports sandboxing for enhanced security. WSL 1 does not support sandboxing.

Alpine Linux and musl-based distributions

The native installer on Alpine and other musl/uClibc-based distributions requires `libgcc`, `libstdc++`, and `ripgrep`. Install these using your distribution's package manager, then set `USE_BUILTIN_RIPGREP=0`.

This example installs the required packages on Alpine:

```
apk add libgcc libstdc++ ripgrep
```

Then set `USE_BUILTIN_RIPGREP` to `0` in your `settings.json` file:

```
{
  "env": {
    "USE_BUILTIN_RIPGREP": "0"
  }
}
```

Verify your installation

After installing, confirm Claude Code is working:

```
claude --version
```

For a more detailed check of your installation and configuration, run `claude doctor`:

```
claude doctor
```

Authenticate

Claude Code requires a Pro, Max, Teams, Enterprise, or Console account. The free Claude.ai plan does not include Claude Code access. You can also use Claude Code with a third-party API provider like Amazon Bedrock, Google Vertex AI, or Microsoft Foundry.

After installing, log in by running `claude` and following the browser prompts. See Authentication for all account types and team setup options.

Update Claude Code

Native installations automatically update in the background. You can configure the release channel to control whether you receive updates immediately or on a delayed stable schedule, or disable auto-updates entirely. Homebrew and WinGet installations require manual updates.

Auto-updates

Claude Code checks for updates on startup and periodically while running. Updates download and install in the background, then take effect the next time you start Claude Code.

> **Note:**
>
> Homebrew and WinGet installations do not auto-update. Use `brew upgrade claude-code` or `winget upgrade Anthropic.ClaudeCode` to update manually.
>
> **Known issue:** Claude Code may notify you of updates before the new version is available in these package managers. If an upgrade fails, wait and try again later.
>
> Homebrew keeps old versions on disk after upgrades. Run `brew cleanup claude-code` periodically to reclaim disk space.

Configure release channel

Control which release channel Claude Code follows for auto-updates and `claude update` with the `autoUpdatesChannel` setting:

- `"latest"`, the default: receive new features as soon as they're released
- `"stable"`: use a version that is typically about one week old, skipping releases with major regressions

Configure this via `/config` → **Auto-update channel**, or add it to your settings.json file:

```
{
  "autoUpdatesChannel": "stable"
}
```

For enterprise deployments, you can enforce a consistent release channel across your organization using managed settings.

Disable auto-updates

Set `DISABLE_AUTOUPDATER` to `"1"` in the `env` key of your `settings.json` file:

```
{
  "env": {
    "DISABLE_AUTOUPDATER": "1"
  }
}
```

Update manually

To apply an update immediately without waiting for the next background check, run:

```
claude update
```

Advanced installation options

These options are for version pinning, migrating from npm, and verifying binary integrity.

Install a specific version

The native installer accepts either a specific version number or a release channel (`latest` or `stable`). The channel you choose at install time becomes your default for auto-updates. See configure release channel for more information.

To install the latest version (default):

macOS, Linux, WSL

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

Windows PowerShell

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

Windows CMD

```
curl -fsSL https://claude.ai/install.cmd -o install.cmd && install.cmd && del
install.cmd
```

To install the stable version:

macOS, Linux, WSL

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

Windows PowerShell

```
& ([scriptblock]::Create((irm https://claude.ai/install.ps1))) stable
```

Windows CMD

```
curl -fsSL https://claude.ai/install.cmd -o install.cmd && install.cmd stable &&
del install.cmd
```

To install a specific version number:

macOS, Linux, WSL

```
curl -fsSL https://claude.ai/install.sh | bash -s 1.0.58
```

Windows PowerShell

```
& ([scriptblock]::Create((irm https://claude.ai/install.ps1))) 1.0.58
```

Windows CMD

```
curl -fsSL https://claude.ai/install.cmd -o install.cmd && install.cmd 1.0.58 &&
del install.cmd
```

Deprecated npm installation

npm installation is deprecated. The native installer is faster, requires no dependencies, and auto-updates in the background. Use the native installation method when possible.

Migrate from npm to native

If you previously installed Claude Code with npm, switch to the native installer:

```
## Install the native binary
curl -fsSL https://claude.ai/install.sh | bash

## Remove the old npm installation
npm uninstall -g @anthropic-ai/claude-code
```

You can also run `claude install` from an existing npm installation to install the native binary alongside it, then remove the npm version.

Install with npm

If you need npm installation for compatibility reasons, you must have Node.js 18+ installed. Install the package globally:

```
npm install -g @anthropic-ai/claude-code
```

> **Warning:**
>
> Do NOT use `sudo npm install -g` as this can lead to permission issues and security risks. If you encounter permission errors, see troubleshooting permission errors.

Binary integrity and code signing

You can verify the integrity of Claude Code binaries using SHA256 checksums and code signatures.

- SHA256 checksums for all platforms are published in the release manifests at `https://storage.googleapis.com/claude-code-dist-86c565f3-f756-42ad-8dfa-d59b1c096819/claude-code-releases/{VERSION}/manifest.json`. Replace `{VERSION}` with a version number such as `2.0.30`.
- Signed binaries are distributed for the following platforms:
  - **macOS**: signed by "Anthropic PBC" and notarized by Apple
  - **Windows**: signed by "Anthropic, PBC"

Uninstall Claude Code

To remove Claude Code, follow the instructions for your installation method.

Native installation

Remove the Claude Code binary and version files:

macOS, Linux, WSL

```
rm -f ~/.local/bin/claude
rm -rf ~/.local/share/claude
```

Windows PowerShell

```
Remove-Item -Path "$env:USERPROFILE\.local\bin\claude.exe" -Force
Remove-Item -Path "$env:USERPROFILE\.local\share\claude" -Recurse -Force
```

Homebrew installation

Remove the Homebrew cask:

```
brew uninstall --cask claude-code
```

WinGet installation

Remove the WinGet package:

```
winget uninstall Anthropic.ClaudeCode
```

npm

Remove the global npm package:

```
npm uninstall -g @anthropic-ai/claude-code
```

Remove configuration files

> **Warning:**
>
> Removing configuration files will delete all your settings, allowed tools, MCP server configurations, and session history.

To remove Claude Code settings and cached data:

macOS, Linux, WSL

```
## Remove user settings and state
rm -rf ~/.claude
rm ~/.claude.json

## Remove project-specific settings (run from your project directory)
rm -rf .claude
rm -f .mcp.json
```

Windows PowerShell

```
## Remove user settings and state
Remove-Item -Path "$env:USERPROFILE\.claude" -Recurse -Force
Remove-Item -Path "$env:USERPROFILE\.claude.json" -Force

## Remove project-specific settings (run from your project directory)
Remove-Item -Path ".claude" -Recurse -Force
Remove-Item -Path ".mcp.json" -Force
```

How Claude Code works

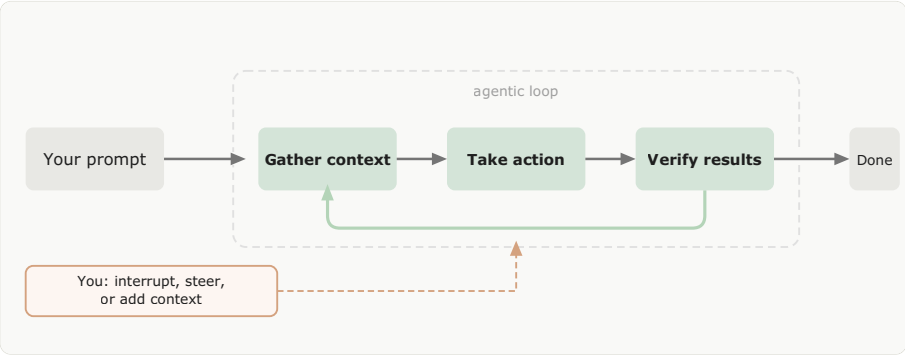
> *Understand the agentic loop, built-in tools, and how Claude Code interacts with your project.*

Claude Code is an agentic assistant that runs in your terminal. While it excels at coding, it can help with anything you can do from the command line: writing docs, running builds, searching files, researching topics, and more.

This guide covers the core architecture, built-in capabilities, and tips for working effectively. For step-by-step walkthroughs, see Common workflows. For extensibility features like skills, MCP, and hooks, see Extend Claude Code.

The agentic loop

When you give Claude a task, it works through three phases: **gather context**, **take action**, and **verify results**. These phases blend together. Claude uses tools throughout, whether searching files to understand your code, editing to make changes, or running tests to check its work.



The agentic loop: Your prompt leads to Claude gathering context, taking action, verifying results, and repeating until task complete. You can interrupt at any point.

The loop adapts to what you ask. A question about your codebase might only need context gathering. A bug fix cycles through all three phases repeatedly. A refactor might involve extensive verification. Claude decides what each step requires based on what it learned from the previous step, chaining dozens of actions together and course-correcting along the way.

You're part of this loop too. You can interrupt at any point to steer Claude in a different direction, provide additional context, or ask it to try a different approach. Claude works autonomously but stays responsive to your input.

The agentic loop is powered by two components: models that reason and tools that act. Claude Code serves as the **agentic harness** around Claude: it provides the tools, context management, and execution environment that turn a language model into a capable coding agent.

Models

Claude Code uses Claude models to understand your code and reason about tasks. Claude can read code in any language, understand how components connect, and figure out what needs to change to accomplish your goal. For complex tasks, it breaks work into steps, executes them, and adjusts based on what it learns.

Multiple models are available with different tradeoffs. Sonnet handles most coding tasks well. Opus provides stronger reasoning for complex architectural decisions. Switch with `/model` during a session or start with `claude --model <name>`.

When this guide says "Claude chooses" or "Claude decides," it's the model doing the reasoning.

Tools

Tools are what make Claude Code agentic. Without tools, Claude can only respond with text. With tools, Claude can act: read your code, edit files, run commands, search the web, and interact with external services. Each tool use returns information that feeds back into the loop, informing Claude's next decision.

The built-in tools generally fall into five categories, each representing a different kind of agency.

Category	What Claude can do
File operations	Read files, edit code, create new files, rename and reorganize
Search	Find files by pattern, search content with regex, explore codebases
Execution	Run shell commands, start servers, run tests, use git

Cat-egory	What Claude can do
Web	Search the web, fetch documentation, look up error messages
Code intelli-gence	See type errors and warnings after edits, jump to definitions, find references (requires code intelligence plugins)

These are the primary capabilities. Claude also has tools for spawning subagents, asking you questions, and other orchestration tasks. See Tools available to Claude for the complete list.

Claude chooses which tools to use based on your prompt and what it learns along the way. When you say "fix the failing tests," Claude might:

1. Run the test suite to see what's failing
2. Read the error output
3. Search for the relevant source files
4. Read those files to understand the code
5. Edit the files to fix the issue
6. Run the tests again to verify

Each tool use gives Claude new information that informs the next step. This is the agentic loop in action.

Extending the base capabilities: The built-in tools are the foundation. You can extend what Claude knows with skills, connect to external services with MCP, automate workflows with hooks, and offload tasks to subagents. These extensions form a layer on top of the core agentic loop. See Extend Claude Code for guidance on choosing the right extension for your needs.

What Claude can access

This guide focuses on the terminal. Claude Code also runs in VS Code, JetBrains IDEs, and other environments.

When you run `claude` in a directory, Claude Code gains access to:

- **Your project.** Files in your directory and subdirectories, plus files elsewhere with your permission.
- **Your terminal.** Any command you could run: build tools, git, package managers, system utilities, scripts. If you can do it from the command line, Claude can too.

- **Your git state.** Current branch, uncommitted changes, and recent commit history.
- **Your CLAUDE.md.** A markdown file where you store project-specific instructions, conventions, and context that Claude should know every session.
- **Auto memory.** Learnings Claude saves automatically as you work, like project patterns and your preferences. The first 200 lines of MEMORY.md are loaded at the start of each session.
- **Extensions you configure.** MCP servers for external services, skills for workflows, subagents for delegated work, and Claude in Chrome for browser interaction.

Because Claude sees your whole project, it can work across it. When you ask Claude to "fix the authentication bug," it searches for relevant files, reads multiple files to understand context, makes coordinated edits across them, runs tests to verify the fix, and commits the changes if you ask. This is different from inline code assistants that only see the current file.

Environments and interfaces

The agentic loop, tools, and capabilities described above are the same everywhere you use Claude Code. What changes is where the code executes and how you interact with it.

Execution environments

Claude Code runs in three environments, each with different tradeoffs for where your code executes.

Environment	Where code runs	Use case
Local	Your machine	Default. Full access to your files, tools, and environment
Cloud	Anthropic-managed VMs	Offload tasks, work on repos you don't have locally
Remote Control	Your machine, controlled from a browser	Use the web UI while keeping everything local

Interfaces

You can access Claude Code through the terminal, the desktop app, IDE extensions, claude.ai/code, Remote Control, Slack, and CI/CD pipelines. The interface determines how you see and interact with Claude, but the underlying agentic loop is identical. See Use Claude Code everywhere for the full list.

Work with sessions

Claude Code saves your conversation locally as you work. Each message, tool use, and result is stored, which enables rewinding, resuming, and forking sessions. Before Claude makes code changes, it also snapshots the affected files so you can revert if needed.

Sessions are independent. Each new session starts with a fresh context window, without the conversation history from previous sessions. Claude can persist learnings across sessions using auto memory, and you can add your own persistent instructions in CLAUDE.md.

Work across branches

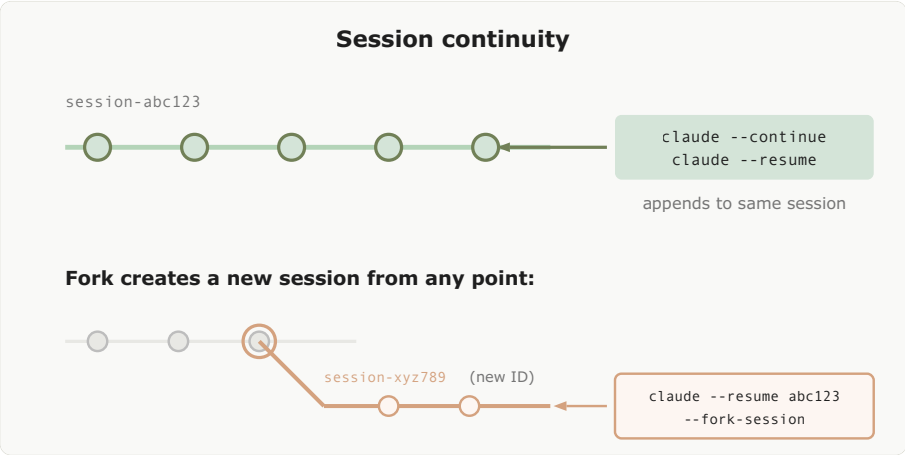
Each Claude Code conversation is a session tied to your current directory. When you resume, you only see sessions from that directory.

Claude sees your current branch's files. When you switch branches, Claude sees the new branch's files, but your conversation history stays the same. Claude remembers what you discussed even after switching.

Since sessions are tied to directories, you can run parallel Claude sessions by using git worktrees, which create separate directories for individual branches.

Resume or fork sessions

When you resume a session with `claude --continue` or `claude --resume`, you pick up where you left off using the same session ID. New messages append to the existing conversation. Your full conversation history is restored, but session-scoped permissions are not. You'll need to re-approve those.



Session continuity: resume continues the same session, fork creates a new branch with a new ID.

To branch off and try a different approach without affecting the original session, use the `--fork-session` flag:

```
claude --continue --fork-session
```

This creates a new session ID while preserving the conversation history up to that point. The original session remains unchanged. Like resume, forked sessions don't inherit session-scoped permissions.

Same session in multiple terminals: If you resume the same session in multiple terminals, both terminals write to the same session file. Messages from both get interleaved, like two people writing in the same notebook. Nothing corrupts, but the conversation becomes jumbled. Each terminal only sees its own messages during the session, but if you resume that session later, you'll see everything interleaved. For parallel work from the same starting point, use `--fork-session` to give each terminal its own clean session.

The context window

Claude's context window holds your conversation history, file contents, command outputs, CLAUDE.md, loaded skills, and system instructions. As you work, context fills up. Claude compacts automatically, but instructions from early in the conversation can get lost. Put persistent rules in CLAUDE.md, and run `/context` to see what's using space.

When context fills up

Claude Code manages context automatically as you approach the limit. It clears older tool outputs first, then summarizes the conversation if needed. Your requests and key code snippets are preserved; detailed instructions from early in the conversation may be lost. Put persistent rules in CLAUDE.md rather than relying on conversation history.

To control what's preserved during compaction, add a "Compact Instructions" section to CLAUDE.md or run `/compact` with a focus (like `/compact focus on the API changes`).

Run `/context` to see what's using space. MCP servers add tool definitions to every request, so a few servers can consume significant context before you start working. Run `/mcp` to check per-server costs.

Manage context with skills and subagents

Beyond compaction, you can use other features to control what loads into context.

Skills load on demand. Claude sees skill descriptions at session start, but the full content only loads when a skill is used. For skills you invoke manually, set `disable-model-invocation: true` to keep descriptions out of context until you need them.

Subagents get their own fresh context, completely separate from your main conversation. Their work doesn't bloat your context. When done, they return a summary. This isolation is why subagents help with long sessions.

See context costs for what each feature costs, and reduce token usage for tips on managing context.

Stay safe with checkpoints and permissions

Claude has two safety mechanisms: checkpoints let you undo file changes, and permissions control what Claude can do without asking.

Undo changes with checkpoints

Every file edit is reversible. Before Claude edits any file, it snapshots the current contents. If something goes wrong, press `Esc` twice to rewind to a previous state, or ask Claude to undo.

Checkpoints are local to your session, separate from git. They only cover file changes. Actions that affect remote systems (databases, APIs, deployments) can't be checkpointed, which is why Claude asks before running commands with external side effects.

Control what Claude can do

Press `Shift+Tab` to cycle through permission modes:

- **Default**: Claude asks before file edits and shell commands
- **Auto-accept edits**: Claude edits files without asking, still asks for commands
- **Plan mode**: Claude uses read-only tools only, creating a plan you can approve before execution

You can also allow specific commands in `.claude/settings.json` so Claude doesn't ask each time. This is useful for trusted commands like `npm test` or `git status`. Settings can be scoped from organization-wide policies down to personal preferences. See Permissions for details.

Work effectively with Claude Code

These tips help you get better results from Claude Code.

Ask Claude Code for help

Claude Code can teach you how to use it. Ask questions like “how do I set up hooks?” or “what’s the best way to structure my CLAUDE.md?” and Claude will explain.

Built-in commands also guide you through setup:

- `/init` walks you through creating a CLAUDE.md for your project
- `/agents` helps you configure custom subagents
- `/doctor` diagnoses common issues with your installation

It’s a conversation

Claude Code is conversational. You don’t need perfect prompts. Start with what you want, then refine:

```
Fix the login bug
```

[Claude investigates, tries something]

```
That's not quite right. The issue is in the session handling.
```

[Claude adjusts approach]

When the first attempt isn’t right, you don’t start over. You iterate.

Interrupt and steer

You can interrupt Claude at any point. If it’s going down the wrong path, just type your correction and press Enter. Claude will stop what it’s doing and adjust its approach based on your input. You don’t have to wait for it to finish or start over.

Be specific upfront

The more precise your initial prompt, the fewer corrections you’ll need. Reference specific files, mention constraints, and point to example patterns.

```
The checkout flow is broken for users with expired cards.
Check src/payments/ for the issue, especially token refresh.
Write a failing test first, then fix it.
```

Vague prompts work, but you'll spend more time steering. Specific prompts like the one above often succeed on the first attempt.

Give Claude something to verify against

Claude performs better when it can check its own work. Include test cases, paste screenshots of expected UI, or define the output you want.

```
Implement validateEmail. Test cases: 'user@example.com' → true,
'invalid' → false, 'user@.com' → false. Run the tests after.
```

For visual work, paste a screenshot of the design and ask Claude to compare its implementation against it.

Explore before implementing

For complex problems, separate research from coding. Use plan mode (`Shift+Tab` twice) to analyze the codebase first:

```
Read src/auth/ and understand how we handle sessions.
Then create a plan for adding OAuth support.
```

Review the plan, refine it through conversation, then let Claude implement. This two-phase approach produces better results than jumping straight to code.

Delegate, don't dictate

Think of delegating to a capable colleague. Give context and direction, then trust Claude to figure out the details:

```
The checkout flow is broken for users with expired cards.
The relevant code is in src/payments/. Can you investigate and fix it?
```

You don't need to specify which files to read or what commands to run. Claude figures that out.

What's next

- Extend with features Add Skills, MCP connections, and custom commands
- Common workflows Step-by-step guides for typical tasks

Part 2: Core Usage

Extend Claude Code

> *Understand when to use CLAUDE.md, Skills, subagents, hooks, MCP, and plugins.*

Claude Code combines a model that reasons about your code with built-in tools for file operations, search, execution, and web access. The built-in tools cover most coding tasks. This guide covers the extension layer: features you add to customize what Claude knows, connect it to external services, and automate workflows.

> **Note:**
>
> For how the core agentic loop works, see How Claude Code works.

New to Claude Code? Start with CLAUDE.md for project conventions. Add other extensions as you need them.

Overview

Extensions plug into different parts of the agentic loop:

- **CLAUDE.md** adds persistent context Claude sees every session
- **Skills** add reusable knowledge and invocable workflows
- **MCP** connects Claude to external services and tools
- **Subagents** run their own loops in isolated context, returning summaries
- **Agent teams** coordinate multiple independent sessions with shared tasks and peer-to-peer messaging
- **Hooks** run outside the loop entirely as deterministic scripts
- **Plugins** and **marketplaces** package and distribute these features

Skills are the most flexible extension. A skill is a markdown file containing knowledge, workflows, or instructions. You can invoke skills with a command like `/deploy`, or Claude can load them automatically when relevant. Skills can run in your current conversation or in an isolated context via subagents.

Match features to your goal

Features range from always-on context that Claude sees every session, to on-demand capabilities you or Claude can invoke, to background automation that runs on specific events. The table below shows what's available and when each one makes sense.

Feature	What it does	When to use it	Example
CLAUDE.md	Persistent context loaded every conversation	Project conventions, "always do X" rules	"Use pnpm, not npm. Run tests before committing."
Skill	Instructions, knowledge, and workflows Claude can use	Reusable content, reference docs, repeatable tasks	`/deploy` runs your deployment checklist; API docs skill with endpoint patterns
Subagent	Isolated execution context that returns summarized results	Context isolation, parallel tasks, specialized workers	Research task that reads many files but returns only key findings
Agent teams	Coordinate multiple independent Claude Code sessions	Parallel research, new feature development, debugging with competing hypotheses	Spawn reviewers to check security, performance, and tests simultaneously
MCP	Connect to external services	External data or actions	Query your database, post to Slack, control a browser
Hook	Deterministic script that runs on events	Predictable automation, no LLM involved	Run ESLint after every file edit

Plugins are the packaging layer. A plugin bundles skills, hooks, subagents, and MCP servers into a single installable unit. Plugin skills are namespaced (like `/my-plugin:review`) so multiple plugins can coexist. Use plugins when you want to reuse the same setup across multiple repositories or distribute to others via a **marketplace**.

Compare similar features

Some features can seem similar. Here's how to tell them apart.

Skill vs Subagent

Skills and subagents solve different problems:

- **Skills** are reusable content you can load into any context
- **Subagents** are isolated workers that run separately from your main conversation

Aspect	Skill	Subagent
What it is	Reusable instructions, knowledge, or workflows	Isolated worker with its own context
Key benefit	Share content across contexts	Context isolation. Work happens separately, only summary returns
Best for	Reference material, invocable workflows	Tasks that read many files, parallel work, specialized workers

Skills can be reference or action. Reference skills provide knowledge Claude uses throughout your session (like your API style guide). Action skills tell Claude to do something specific (like `/deploy` that runs your deployment workflow).

Use a subagent when you need context isolation or when your context window is getting full. The subagent might read dozens of files or run extensive searches, but your main conversation only receives a summary. Since subagent work doesn't consume your main context, this is also useful when you don't need the intermediate work to remain visible. Custom subagents can have their own instructions and can preload skills.

They can combine. A subagent can preload specific skills (`skills:` field). A skill can run in isolated context using `context: fork` . See Skills for details.

CLAUDE.md vs Skill

Both store instructions, but they load differently and serve different purposes.

Aspect	CLAUDE.md	Skill
Loads	Every session, automatically	On demand
Can include files	Yes, with `@path` imports	Yes, with `@path` imports
Can trigger workflows	No	Yes, with `/<name>`
Best for	"Always do X" rules	Reference material, invocable workflows

Put it in CLAUDE.md if Claude should always know it: coding conventions, build commands, project structure, "never do X" rules.

Put it in a skill if it's reference material Claude needs sometimes (API docs, style guides) or a workflow you trigger with `/<name>` (deploy, review, release).

Rule of thumb: Keep CLAUDE.md under 200 lines. If it's growing, move reference content to skills or split into `.claude/rules/` files.

CLAUDE.md vs Rules vs Skills

All three store instructions, but they load differently:

Aspect	CLAUDE.md	`.claude/rules/`	Skill
Loads	Every session	Every session, or when matching files are opened	On demand, when invoked or relevant
Scope	Whole project	Can be scoped to file paths	Task-specific
Best for	Core conventions and build commands	Language-specific or directory-specific guidelines	Reference material, repeatable workflows

Use CLAUDE.md for instructions every session needs: build commands, test conventions, project architecture.

Use rules to keep CLAUDE.md focused. Rules with `paths` frontmatter only load when Claude works with matching files, saving context.

Use skills for content Claude only needs sometimes, like API documentation or a deployment checklist you trigger with `/<name>`.

Subagent vs Agent team

Both parallelize work, but they're architecturally different:

- **Subagents** run inside your session and report results back to your main context
- **Agent teams** are independent Claude Code sessions that communicate with each other

Aspect	Subagent	Agent team
Context	Own context window; results return to the caller	Own context window; fully independent

Aspect	Subagent	Agent team
Communication	Reports results back to the main agent only	Teammates message each other directly
Coordination	Main agent manages all work	Shared task list with self-coordination
Best for	Focused tasks where only the result matters	Complex work requiring discussion and collaboration
Token cost	Lower: results summarized back to main context	Higher: each teammate is a separate Claude instance

Use a subagent when you need a quick, focused worker: research a question, verify a claim, review a file. The subagent does the work and returns a summary. Your main conversation stays clean.

Use an agent team when teammates need to share findings, challenge each other, and coordinate independently. Agent teams are best for research with competing hypotheses, parallel code review, and new feature development where each teammate owns a separate piece.

Transition point: If you're running parallel subagents but hitting context limits, or if your subagents need to communicate with each other, agent teams are the natural next step.

> **Note:**
>
> Agent teams are experimental and disabled by default. See agent teams for setup and current limitations.

MCP vs Skill

MCP connects Claude to external services. Skills extend what Claude knows, including how to use those services effectively.

Aspect	MCP	Skill
What it is	Protocol for connecting to external services	Knowledge, workflows, and reference material

Aspect	MCP	Skill
Provides	Tools and data access	Knowledge, workflows, reference material
Examples	Slack integration, database queries, browser control	Code review checklist, deploy workflow, API style guide

These solve different problems and work well together:

MCP gives Claude the ability to interact with external systems. Without MCP, Claude can't query your database or post to Slack.

Skills give Claude knowledge about how to use those tools effectively, plus workflows you can trigger with `/<name>`. A skill might include your team's database schema and query patterns, or a `/post-to-slack` workflow with your team's message formatting rules.

Example: An MCP server connects Claude to your database. A skill teaches Claude your data model, common query patterns, and which tables to use for different tasks.

Understand how features layer

Features can be defined at multiple levels: user-wide, per-project, via plugins, or through managed policies. You can also nest CLAUDE.md files in subdirectories or place skills in specific packages of a monorepo. When the same feature exists at multiple levels, here's how they layer:

- **CLAUDE.md files** are additive: all levels contribute content to Claude's context simultaneously. Files from your working directory and above load at launch; subdirectories load as you work in them. When instructions conflict, Claude uses judgment to reconcile them, with more specific instructions typically taking precedence. See how CLAUDE.md files load.
- **Skills and subagents** override by name: when the same name exists at multiple levels, one definition wins based on priority (managed > user > project for skills; managed > CLI flag > project > user > plugin for subagents). Plugin skills are namespaced to avoid conflicts. See skill discovery and subagent scope.
- **MCP servers** override by name: local > project > user. See MCP scope.
- **Hooks** merge: all registered hooks fire for their matching events regardless of source. See hooks.

Combine features

Each extension solves a different problem: CLAUDE.md handles always-on context, skills handle on-demand knowledge and workflows, MCP handles external connections, subagents handle isolation, and hooks handle automation. Real setups combine them based on your workflow.

For example, you might use CLAUDE.md for project conventions, a skill for your deployment workflow, MCP to connect to your database, and a hook to run linting after every edit. Each feature handles what it's best at.

Pattern	How it works	Example
Skill + MCP	MCP provides the connection; a skill teaches Claude how to use it well	MCP connects to your database, a skill documents your schema and query patterns
Skill + Subagent	A skill spawns subagents for parallel work	`/audit` skill kicks off security, performance, and style subagents that work in isolated context
CLAUDE.md + Skills	CLAUDE.md holds always-on rules; skills hold reference material loaded on demand	CLAUDE.md says "follow our API conventions," a skill contains the full API style guide
Hook + MCP	A hook triggers external actions through MCP	Post-edit hook sends a Slack notification when Claude modifies critical files

Understand context costs

Every feature you add consumes some of Claude's context. Too much can fill up your context window, but it can also add noise that makes Claude less effective; skills may not trigger correctly, or Claude may lose track of your conventions. Understanding these trade-offs helps you build an effective setup.

Context cost by feature

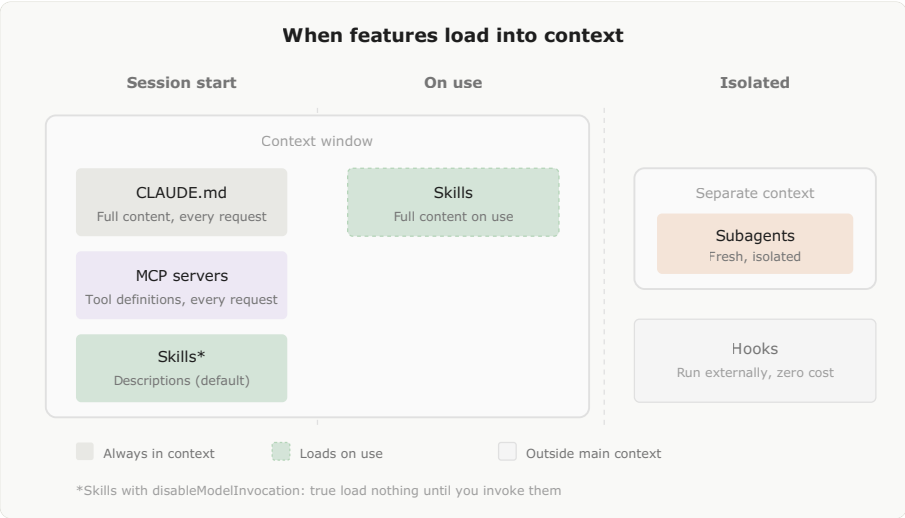
Each feature has a different loading strategy and context cost:

Feature	When it loads	What loads	Context cost
CLAUDE.md	Session start	Full content	Every request
Skills	Session start + when used	Descriptions at start, full content when used	Low (descriptions every request)*
MCP servers	Session start	All tool definitions and schemas	Every request
Subagents	When spawned	Fresh context with specified skills	Isolated from main session
Hooks	On trigger	Nothing (runs externally)	Zero, unless hook returns additional context

*By default, skill descriptions load at session start so Claude can decide when to use them. Set `disable-model-invocation: true` in a skill's frontmatter to hide it from Claude entirely until you invoke it manually. This reduces context cost to zero for skills you only trigger yourself.

Understand how features load

Each feature loads at different points in your session. The tabs below explain when each one loads and what goes into context.



Context loading: CLAUDE.md and MCP load at session start and stay in every request. Skills load descriptions at start, full content on invocation. Subagents get isolated context. Hooks run externally.

CLAUDE.md

When: Session start

What loads: Full content of all CLAUDE.md files (managed, user, and project levels).

Inheritance: Claude reads CLAUDE.md files from your working directory up to the root, and discovers nested ones in subdirectories as it accesses those files. See How CLAUDE.md files load for details.

> **Tip:**
>
> Keep CLAUDE.md under ~500 lines. Move reference material to skills, which load on-demand.

Skills

Skills are extra capabilities in Claude's toolkit. They can be reference material (like an API style guide) or invocable workflows you trigger with `/<name>` (like `/deploy`). Claude Code ships with bundled skills like `/simplify`, `/batch`, and `/debug` that work out of the box. You can also create your own. Claude uses skills when appropriate, or you can invoke one directly.

When: Depends on the skill's configuration. By default, descriptions load at session start and full content loads when used. For user-only skills (`disable-model-invocation: true`), nothing loads until you invoke them.

What loads: For model-invocable skills, Claude sees names and descriptions in every request. When you invoke a skill with `/<name>` or Claude loads it automatically, the full content loads into your conversation.

How Claude chooses skills: Claude matches your task against skill descriptions to decide which are relevant. If descriptions are vague or overlap, Claude may load the wrong skill or miss one that would help. To tell Claude to use a specific skill, invoke it with `/<name>`. Skills with `disable-model-invocation: true` are invisible to Claude until you invoke them.

Context cost: Low until used. User-only skills have zero cost until invoked.

In subagents: Skills work differently in subagents. Instead of on-demand loading, skills passed to a subagent are fully preloaded into its context at launch. Subagents don't inherit skills from the main session; you must specify them explicitly.

> **Tip:**
>
> Use `disable-model-invocation: true` for skills with side effects. This saves context and ensures only you trigger them.

MCP servers

When: Session start.

What loads: All tool definitions and JSON schemas from connected servers.

Context cost: Tool search (enabled by default) loads MCP tools up to 10% of context and defers the rest until needed.

Reliability note: MCP connections can fail silently mid-session. If a server disconnects, its tools disappear without warning. Claude may try to use a tool that no longer exists. If you notice Claude failing to use an MCP tool it previously could access, check the connection with `/mcp`.

> **Tip:**
>
> Run `/mcp` to see token costs per server. Disconnect servers you're not actively using.

Subagents

When: On demand, when you or Claude spawns one for a task.

What loads: Fresh, isolated context containing:

- The system prompt (shared with parent for cache efficiency)
- Full content of skills listed in the agent's `skills:` field
- CLAUDE.md and git status (inherited from parent)
- Whatever context the lead agent passes in the prompt

Context cost: Isolated from main session. Subagents don't inherit your conversation history or invoked skills.

> **Tip:**
>
> Use subagents for work that doesn't need your full conversation context. Their isolation prevents bloating your main session.

Hooks

When: On trigger. Hooks fire at specific lifecycle events like tool execution, session boundaries, prompt submission, permission requests, and compaction. See Hooks for the full list.

What loads: Nothing by default. Hooks run as external scripts.

Context cost: Zero, unless the hook returns output that gets added as messages to your conversation.

> **Tip:**
>
> Hooks are ideal for side effects (linting, logging) that don't need to affect Claude's context.

Learn more

Each feature has its own guide with setup instructions, examples, and configuration options.

- CLAUDE.md Store project context, conventions, and instructions
- Skills Give Claude domain expertise and reusable workflows
- Subagents Offload work to isolated context
- Agent teams Coordinate multiple sessions working in parallel
- MCP Connect Claude to external services

- Hooks Automate workflows with hooks
- Plugins Bundle and share feature sets
- Marketplaces Host and distribute plugin collections

Interactive mode

> *Complete reference for keyboard shortcuts, input modes, and interactive features in Claude Code sessions.*

Keyboard shortcuts

Note:

Keyboard shortcuts may vary by platform and terminal. Press `?` to see available shortcuts for your environment.

macOS users: Option/Alt key shortcuts (`Alt+B` , `Alt+F` , `Alt+Y` , `Alt+M` , `Alt+P`) require configuring Option as Meta in your terminal:

- **iTerm2**: settings → Profiles → Keys → set Left/Right Option key to "Esc+"
- **Terminal.app**: settings → Profiles → Keyboard → check "Use Option as Meta Key"
- **VS Code**: settings → Profiles → Keys → set Left/Right Option key to "Esc+"

See Terminal configuration for details.

General controls

Shortcut	Description	Context
`Ctrl+C`	Cancel current input or generation	Standard interrupt
`Ctrl+F`	Kill all background agents. Press twice within 3 seconds to confirm	Background agent control
`Ctrl+D`	Exit Claude Code session	EOF signal
`Ctrl+G`	Open in default text editor	Edit your prompt or custom response in your default text editor
`Ctrl+L`	Clear terminal screen	Keeps conversation history
`Ctrl+O`	Toggle verbose output	Shows detailed tool usage and execution

Shortcut	Description	Context
`Ctrl+R`	Reverse search command history	Search through previous commands interactively
`Ctrl+V` or `Cmd+V` (iTerm2) or `Alt+V` (Windows)	Paste image from clipboard	Pastes an image or path to an image file
`Ctrl+B`	Background running tasks	Backgrounds bash commands and agents. Tmux users press twice
`Ctrl+T`	Toggle task list	Show or hide the task list in the terminal status area
`Left/Right arrows`	Cycle through dialog tabs	Navigate between tabs in permission dialogs and menus
`Up/Down arrows`	Navigate command history	Recall previous inputs
`Esc` + `Esc`	Rewind or summarize	Restore code and/or conversation to a previous point, or summarize from a selected message
`Shift+Tab` or `Alt+M` (some configurations)	Toggle permission modes	Switch between Auto-Accept Mode, Plan Mode, and normal mode.
`Option+P` (macOS) or `Alt+P` (Windows/Linux)	Switch model	Switch models without clearing your prompt
`Option+T` (macOS) or `Alt+T` (Windows/Linux)	Toggle extended thinking	Enable or disable extended thinking mode. Run `/terminal-setup` first to enable this shortcut

Text editing

Shortcut	Description	Context
`Ctrl+K`	Delete to end of line	Stores deleted text for pasting

Shortcut	Description	Context
`Ctrl+U`	Delete entire line	Stores deleted text for pasting
`Ctrl+Y`	Paste deleted text	Paste text deleted with `Ctrl+K` or `Ctrl+U`
`Alt+Y` (after `Ctrl+Y`)	Cycle paste history	After pasting, cycle through previously deleted text. Requires Option as Meta on macOS
`Alt+B`	Move cursor back one word	Word navigation. Requires Option as Meta on macOS
`Alt+F`	Move cursor forward one word	Word navigation. Requires Option as Meta on macOS

Theme and display

Shortcut	Description	Context
`Ctrl+T`	Toggle syntax highlighting for code blocks	Only works inside the `/theme` picker menu. Controls whether code in Claude's responses uses syntax coloring

> **Note:**
>
> Syntax highlighting is only available in the native build of Claude Code.

Multiline input

Method	Shortcut	Context
Quick escape	`\` + `Enter`	Works in all terminals
macOS default	`Option+Enter`	Default on macOS
Shift+Enter	`Shift+Enter`	Works out of the box in iTerm2, WezTerm, Ghostty, Kitty

Method	Shortcut	Context
Control sequence	`Ctrl+J`	Line feed character for multiline
Paste mode	Paste directly	For code blocks, logs

> **Tip:**
>
> Shift+Enter works without configuration in iTerm2, WezTerm, Ghostty, and Kitty. For other terminals (VS Code, Alacritty, Zed, Warp), run `/terminal-setup` to install the binding.

Quick commands

Shortcut	Description	Notes
`/` at start	Command or skill	See built-in commands and skills
`!` at start	Bash mode	Run commands directly and add execution output to the session
`@`	File path mention	Trigger file path autocomplete

Built-in commands

Type `/` in Claude Code to see all available commands, or type `/` followed by any letters to filter. The `/` menu shows both built-in commands and bundled skills like `/simplify`. Not all commands are visible to every user since some depend on your platform or plan.

See the commands reference for the full list of built-in commands. To create your own commands, see skills.

Vim editor mode

Enable vim-style editing with `/vim` command or configure permanently via `/config`.

Mode switching

Command	Action	From mode
`Esc`	Enter NORMAL mode	INSERT
`i`	Insert before cursor	NORMAL
`I`	Insert at beginning of line	NORMAL
`a`	Insert after cursor	NORMAL
`A`	Insert at end of line	NORMAL
`o`	Open line below	NORMAL
`O`	Open line above	NORMAL

Navigation (NORMAL mode)

Command	Action
`h` / `j` / `k` / `l`	Move left/down/up/right
`w`	Next word
`e`	End of word
`b`	Previous word
`0`	Beginning of line
`$`	End of line
`^`	First non-blank character
`gg`	Beginning of input
`G`	End of input
`f{char}`	Jump to next occurrence of character
`F{char}`	Jump to previous occurrence of character
`t{char}`	Jump to just before next occurrence of character
`T{char}`	Jump to just after previous occurrence of character

Command	Action
`;`	Repeat last f/F/t/T motion
`,`	Repeat last f/F/t/T motion in reverse

> **Note:**
>
> In vim normal mode, if the cursor is at the beginning or end of input and cannot move further, the arrow keys navigate command history instead.

Editing (NORMAL mode)

Command	Action
`x`	Delete character
`dd`	Delete line
`D`	Delete to end of line
`dw` / `de` / `db`	Delete word/to end/back
`cc`	Change line
`C`	Change to end of line
`cw` / `ce` / `cb`	Change word/to end/back
`yy` / `Y`	Yank (copy) line
`yw` / `ye` / `yb`	Yank word/to end/back
`p`	Paste after cursor
`P`	Paste before cursor
`>>`	Indent line
`<<`	Dedent line
`J`	Join lines
`.`	Repeat last change

Text objects (NORMAL mode)

Text objects work with operators like `d`, `c`, and `y`:

Command	Action
`iw` / `aw`	Inner/around word
`iW` / `aW`	Inner/around WORD (whitespace-delimited)
`i"` / `a"`	Inner/around double quotes
`i'` / `a'`	Inner/around single quotes
`i(` / `a(`	Inner/around parentheses
`i[` / `a[`	Inner/around brackets
`i{` / `a{`	Inner/around braces

Command history

Claude Code maintains command history for the current session:

- Input history is stored per working directory
- Input history resets when you run `/clear` to start a new session. The previous session's conversation is preserved and can be resumed.
- Use Up/Down arrows to navigate (see keyboard shortcuts above)
- **Note**: history expansion (`!`) is disabled by default

Reverse search with Ctrl+R

Press `Ctrl+R` to interactively search through your command history:

1. **Start search**: press `Ctrl+R` to activate reverse history search
2. **Type query**: enter text to search for in previous commands. The search term is highlighted in matching results
3. **Navigate matches**: press `Ctrl+R` again to cycle through older matches
4. **Accept match**:
   - Press `Tab` or `Esc` to accept the current match and continue editing
   - Press `Enter` to accept and execute the command immediately
5. **Cancel search**:
   - Press `Ctrl+C` to cancel and restore your original input

- Press `Backspace` on empty search to cancel

The search displays matching commands with the search term highlighted, so you can find and reuse previous inputs.

Background bash commands

Claude Code supports running bash commands in the background, allowing you to continue working while long-running processes execute.

How backgrounding works

When Claude Code runs a command in the background, it runs the command asynchronously and immediately returns a background task ID. Claude Code can respond to new prompts while the command continues executing in the background.

To run commands in the background, you can either:

- Prompt Claude Code to run a command in the background
- Press Ctrl+B to move a regular Bash tool invocation to the background. (Tmux users must press Ctrl+B twice due to tmux's prefix key.)

Key features:

- Output is buffered and Claude can retrieve it using the TaskOutput tool
- Background tasks have unique IDs for tracking and output retrieval
- Background tasks are automatically cleaned up when Claude Code exits

To disable all background task functionality, set the `CLAUDE_CODE_DISABLE_BACKGROUND_TASKS` environment variable to `1`. See Environment variables for details.

Common backgrounded commands:

- Build tools (webpack, vite, make)
- Package managers (npm, yarn, pnpm)
- Test runners (jest, pytest)
- Development servers
- Long-running processes (docker, terraform)

Bash mode with `!` prefix

Run bash commands directly without going through Claude by prefixing your input with `!`:

```
! npm test
! git status
! ls -la
```

Bash mode:

- Adds the command and its output to the conversation context
- Shows real-time progress and output
- Supports the same `Ctrl+B` backgrounding for long-running commands
- Does not require Claude to interpret or approve the command
- Supports history-based autocomplete: type a partial command and press **Tab** to complete from previous `!` commands in the current project
- Exit with `Escape`, `Backspace`, or `Ctrl+U` on an empty prompt

This is useful for quick shell operations while maintaining conversation context.

Prompt suggestions

When you first open a session, a grayed-out example command appears in the prompt input to help you get started. Claude Code picks this from your project's git history, so it reflects files you've been working on recently.

After Claude responds, suggestions continue to appear based on your conversation history, such as a follow-up step from a multi-part request or a natural continuation of your workflow.

- Press **Tab** to accept the suggestion, or press **Enter** to accept and submit
- Start typing to dismiss it

The suggestion runs as a background request that reuses the parent conversation's prompt cache, so the additional cost is minimal. Claude Code skips suggestion generation when the cache is cold to avoid unnecessary cost.

Suggestions are automatically skipped after the first turn of a conversation, in non-interactive mode, and in plan mode.

To disable prompt suggestions entirely, set the environment variable or toggle the setting in `/config`:

```
export CLAUDE_CODE_ENABLE_PROMPT_SUGGESTION=false
```

Side questions with /btw

Use `/btw` to ask a quick question about your current work without adding to the conversation history. This is useful when you want a fast answer but don't want to clutter the main context or derail Claude from a long-running task.

```
/btw what was the name of that config file again?
```

Side questions have full visibility into the current conversation, so you can ask about code Claude has already read, decisions it made earlier, or anything else from the session. The question and answer are ephemeral: they appear in a dismissible overlay and never enter the conversation history.

- **Available while Claude is working**: you can run `/btw` even while Claude is processing a response. The side question runs independently and does not interrupt the main turn.
- **No tool access**: side questions answer only from what is already in context. Claude cannot read files, run commands, or search when answering a side question.
- **Single response**: there are no follow-up turns. If you need a back-and-forth, use a normal prompt instead.
- **Low cost**: the side question reuses the parent conversation's prompt cache, so the additional cost is minimal.

Press **Space**, **Enter**, or **Escape** to dismiss the answer and return to the prompt.

`/btw` is the inverse of a subagent: it sees your full conversation but has no tools, while a subagent has full tools but starts with an empty context. Use `/btw` to ask about what Claude already knows from this session; use a subagent to go find out something new.

Task list

When working on complex, multi-step work, Claude creates a task list to track progress. Tasks appear in the status area of your terminal with indicators showing what's pending, in progress, or complete.

- Press `Ctrl+T` to toggle the task list view. The display shows up to 10 tasks at a time
- To see all tasks or clear them, ask Claude directly: "show me all tasks" or "clear all tasks"
- Tasks persist across context compactions, helping Claude stay organized on larger projects

- To share a task list across sessions, set `CLAUDE_CODE_TASK_LIST_ID` to use a named directory in `~/.claude/tasks/`: `CLAUDE_CODE_TASK_LIST_ID=my-project claude`

PR review status

When working on a branch with an open pull request, Claude Code displays a clickable PR link in the footer (for example, "PR #446"). The link has a colored underline indicating the review state:

- Green: approved
- Yellow: pending review
- Red: changes requested
- Gray: draft
- Purple: merged

`Cmd+click` (Mac) or `Ctrl+click` (Windows/Linux) the link to open the pull request in your browser. The status updates automatically every 60 seconds.

> **Note:**
>
> PR status requires the `gh` CLI to be installed and authenticated (`gh auth login`).

See also

- Skills - Custom prompts and workflows
- Checkpointing - Rewind Claude's edits and restore previous states
- CLI reference - Command-line flags and options
- Settings - Configuration options
- Memory management - Managing CLAUDE.md files

Best Practices for Claude Code

> *Tips and patterns for getting the most out of Claude Code, from configuring your environment to scaling across parallel sessions.*

Claude Code is an agentic coding environment. Unlike a chatbot that answers questions and waits, Claude Code can read your files, run commands, make changes, and autonomously work through problems while you watch, redirect, or step away entirely.

This changes how you work. Instead of writing code yourself and asking Claude to review it, you describe what you want and Claude figures out how to build it. Claude explores, plans, and implements.

But this autonomy still comes with a learning curve. Claude works within certain constraints you need to understand.

This guide covers patterns that have proven effective across Anthropic's internal teams and for engineers using Claude Code across various codebases, languages, and environments. For how the agentic loop works under the hood, see How Claude Code works.

Most best practices are based on one constraint: Claude's context window fills up fast, and performance degrades as it fills.

Claude's context window holds your entire conversation, including every message, every file Claude reads, and every command output. However, this can fill up fast. A single debugging session or codebase exploration might generate and consume tens of thousands of tokens.

This matters since LLM performance degrades as context fills. When the context window is getting full, Claude may start "forgetting" earlier instructions or making more mistakes. The context window is the most important resource to manage. Track context usage continuously with a custom status line, and see Reduce token usage for strategies on reducing token usage.

Give Claude a way to verify its work

> **Tip:**
>
> Include tests, screenshots, or expected outputs so Claude can check itself. This is the single highest-leverage thing you can do.

Claude performs dramatically better when it can verify its own work, like run tests, compare screenshots, and validate outputs.

Without clear success criteria, it might produce something that looks right but actually doesn't work. You become the only feedback loop, and every mistake requires your attention.

Strategy	Before	After
Provide verification criteria	*"implement a function that validates email addresses"*	*"write a validateEmail function. example test cases: user@example.com is true, invalid is false, user@.com is false. run the tests after implementing"*
Verify UI changes visually	*"make the dashboard look better"*	*"[paste screenshot] implement this design. take a screenshot of the result and compare it to the original. list differences and fix them"*
Address root causes, not symptoms	*"the build is failing"*	*"the build fails with this error: [paste error]. fix it and verify the build succeeds. address the root cause, don't suppress the error"*

UI changes can be verified using the Claude in Chrome extension. It opens new tabs in your browser, tests the UI, and iterates until the code works.

Your verification can also be a test suite, a linter, or a Bash command that checks output. Invest in making your verification rock-solid.

Explore first, then plan, then code

> **Tip:**
>
> Separate research and planning from implementation to avoid solving the wrong problem.

Letting Claude jump straight to coding can produce code that solves the wrong problem. Use Plan Mode to separate exploration from execution.

The recommended workflow has four phases:

Step 1: Explore

Enter Plan Mode. Claude reads files and answers questions without making changes.

```
read /src/auth and understand how we handle sessions and login.
also look at how we manage environment variables for secrets.
```

Step 2: Plan

Ask Claude to create a detailed implementation plan.

```
I want to add Google OAuth. What files need to change?
What's the session flow? Create a plan.
```

Press `Ctrl+G` to open the plan in your text editor for direct editing before Claude proceeds.

Step 3: Implement

Switch back to Normal Mode and let Claude code, verifying against its plan.

```
implement the OAuth flow from your plan. write tests for the
callback handler, run the test suite and fix any failures.
```

Step 4: Commit

Ask Claude to commit with a descriptive message and create a PR.

```
commit with a descriptive message and open a PR
```

Plan Mode is useful, but also adds overhead.

For tasks where the scope is clear and the fix is small (like fixing a typo, adding a log line, or renaming a variable) ask Claude to do it directly.

Planning is most useful when you're uncertain about the approach, when the change modifies multiple files, or when you're unfamiliar with the code being modified. If you could describe the diff in one sentence, skip the plan.

Provide specific context in your prompts

> **Tip:**
>
> The more precise your instructions, the fewer corrections you'll need.

Claude can infer intent, but it can't read your mind. Reference specific files, mention constraints, and point to example patterns.

Strategy	Before	After
Scope the task. Specify which file, what scenario, and testing preferences.	*"add tests for foo.py"*	*"write a test for foo.py covering the edge case where the user is logged out. avoid mocks."*
Point to sources. Direct Claude to the source that can answer a question.	*"why does ExecutionFactory have such a weird api?"*	*"look through ExecutionFactory's git history and summarize how its api came to be"*

Strategy	Before	After
Reference existing patterns. Point Claude to patterns in your codebase.	*"add a calendar widget"*	*"look at how existing widgets are implemented on the home page to understand the patterns. HotDogWidget.php is a good example. follow the pattern to implement a new calendar widget that lets the user select a month and paginate forwards/backwards to pick a year. build from scratch without libraries other than the ones already used in the codebase."*
Describe the symptom. Provide the symptom, the likely location, and what "fixed" looks like.	*"fix the login bug"*	*"users report that login fails after session timeout. check the auth flow in src/auth/, especially token refresh. write a failing test that reproduces the issue, then fix it"*

Vague prompts can be useful when you're exploring and can afford to course-correct. A prompt like `"what would you improve in this file?"` can surface things you wouldn't have thought to ask about.

Provide rich content

> **Tip:**
>
> Use `@` to reference files, paste screenshots/images, or pipe data directly.

You can provide rich data to Claude in several ways:

- **Reference files with `@`** instead of describing where code lives. Claude reads the file before responding.
- **Paste images directly**. Copy/paste or drag and drop images into the prompt.
- **Give URLs** for documentation and API references. Use `/permissions` to allowlist frequently-used domains.

- **Pipe in data** by running `cat error.log | claude` to send file contents directly.
- **Let Claude fetch what it needs**. Tell Claude to pull context itself using Bash commands, MCP tools, or by reading files.

Configure your environment

A few setup steps make Claude Code significantly more effective across all your sessions. For a full overview of extension features and when to use each one, see Extend Claude Code.

Write an effective CLAUDE.md

> **Tip:**
>
> Run `/init` to generate a starter CLAUDE.md file based on your current project structure, then refine over time.

CLAUDE.md is a special file that Claude reads at the start of every conversation. Include Bash commands, code style, and workflow rules. This gives Claude persistent context it can't infer from code alone.

The `/init` command analyzes your codebase to detect build systems, test frameworks, and code patterns, giving you a solid foundation to refine.

There's no required format for CLAUDE.md files, but keep it short and human-readable. For example:

```
## Code style
- Use ES modules (import/export) syntax, not CommonJS (require)
- Destructure imports when possible (eg. import { foo } from 'bar')

## Workflow
- Be sure to typecheck when you're done making a series of code changes
- Prefer running single tests, and not the whole test suite, for performance
```

CLAUDE.md is loaded every session, so only include things that apply broadly. For domain knowledge or workflows that are only relevant sometimes, use skills instead. Claude loads them on demand without bloating every conversation.

Keep it concise. For each line, ask: *"Would removing this cause Claude to make mistakes?"* If not, cut it. Bloated CLAUDE.md files cause Claude to ignore your actual instructions!

✅ Include	❌ Exclude
Bash commands Claude can't guess	Anything Claude can figure out by reading code
Code style rules that differ from defaults	Standard language conventions Claude already knows
Testing instructions and preferred test runners	Detailed API documentation (link to docs instead)
Repository etiquette (branch naming, PR conventions)	Information that changes frequently
Architectural decisions specific to your project	Long explanations or tutorials
Developer environment quirks (required env vars)	File-by-file descriptions of the codebase
Common gotchas or non-obvious behaviors	Self-evident practices like "write clean code"

If Claude keeps doing something you don't want despite having a rule against it, the file is probably too long and the rule is getting lost. If Claude asks you questions that are answered in CLAUDE.md, the phrasing might be ambiguous. Treat CLAUDE.md like code: review it when things go wrong, prune it regularly, and test changes by observing whether Claude's behavior actually shifts.

You can tune instructions by adding emphasis (e.g., "IMPORTANT" or "YOU MUST") to improve adherence. Check CLAUDE.md into git so your team can contribute. The file compounds in value over time.

CLAUDE.md files can import additional files using `@path/to/import` syntax:

```
See @README.md for project overview and @package.json for available npm commands.

## Additional Instructions
- Git workflow: @docs/git-instructions.md
- Personal overrides: @~/.claude/my-project-instructions.md
```

You can place CLAUDE.md files in several locations:

- **Home folder (`~/.claude/CLAUDE.md`)**: applies to all Claude sessions
- **Project root (`./CLAUDE.md`)**: check into git to share with your team
- **Parent directories**: useful for monorepos where both `root/CLAUDE.md` and `root/foo/CLAUDE.md` are pulled in automatically
- **Child directories**: Claude pulls in child CLAUDE.md files on demand when working with files in those directories

Configure permissions

> **Tip:**
>
> Use `/permissions` to allowlist safe commands or `/sandbox` for OS-level isolation. This reduces interruptions while keeping you in control.

By default, Claude Code requests permission for actions that might modify your system: file writes, Bash commands, MCP tools, etc. This is safe but tedious. After the tenth approval you're not really reviewing anymore, you're just clicking through. There are two ways to reduce these interruptions:

- **Permission allowlists**: permit specific tools you know are safe (like `npm run lint` or `git commit`)
- **Sandboxing**: enable OS-level isolation that restricts filesystem and network access, allowing Claude to work more freely within defined boundaries

Alternatively, use `--dangerously-skip-permissions` to bypass all permission checks for contained workflows like fixing lint errors or generating boilerplate.

> **Warning:**
>
> Letting Claude run arbitrary commands can result in data loss, system corruption, or data exfiltration via prompt injection. Only use `--dangerously-skip-permissions` in a sandbox without internet access.

Read more about configuring permissions and enabling sandboxing.

Use CLI tools

> **Tip:**

> Tell Claude Code to use CLI tools like `gh`, `aws`, `gcloud`, and `sentry-cli` when interacting with external services.

CLI tools are the most context-efficient way to interact with external services. If you use GitHub, install the `gh` CLI. Claude knows how to use it for creating issues, opening pull requests, and reading comments. Without `gh`, Claude can still use the GitHub API, but unauthenticated requests often hit rate limits.

Claude is also effective at learning CLI tools it doesn't already know. Try prompts like `Use 'foo-cli-tool --help' to learn about foo tool, then use it to solve A, B, C.`

Connect MCP servers

> **Tip:**
>
> Run `claude mcp add` to connect external tools like Notion, Figma, or your database.

With MCP servers, you can ask Claude to implement features from issue trackers, query databases, analyze monitoring data, integrate designs from Figma, and automate workflows.

Set up hooks

> **Tip:**
>
> Use hooks for actions that must happen every time with zero exceptions.

Hooks run scripts automatically at specific points in Claude's workflow. Unlike CLAUDE.md instructions which are advisory, hooks are deterministic and guarantee the action happens.

Claude can write hooks for you. Try prompts like *"Write a hook that runs eslint after every file edit"* or *"Write a hook that blocks writes to the migrations folder."* Edit `.claude/settings.json` directly to configure hooks by hand, and run `/hooks` to browse what's configured.

Create skills

> **Tip:**

> Create `SKILL.md` files in `.claude/skills/` to give Claude domain knowledge and reusable workflows.

Skills extend Claude's knowledge with information specific to your project, team, or domain. Claude applies them automatically when relevant, or you can invoke them directly with `/skill-name`.

Create a skill by adding a directory with a `SKILL.md` to `.claude/skills/`:

```
---
name: api-conventions
description: REST API design conventions for our services
---
## API Conventions
- Use kebab-case for URL paths
- Use camelCase for JSON properties
- Always include pagination for list endpoints
- Version APIs in the URL path (/v1/, /v2/)
```

Skills can also define repeatable workflows you invoke directly:

```
---
name: fix-issue
description: Fix a GitHub issue
disable-model-invocation: true
---
Analyze and fix the GitHub issue: $ARGUMENTS.

1. Use `gh issue view` to get the issue details
2. Understand the problem described in the issue
3. Search the codebase for relevant files
4. Implement the necessary changes to fix the issue
5. Write and run tests to verify the fix
6. Ensure code passes linting and type checking
7. Create a descriptive commit message
8. Push and create a PR
```

Run `/fix-issue 1234` to invoke it. Use `disable-model-invocation: true` for workflows with side effects that you want to trigger manually.

Create custom subagents

> **Tip:**
>
> Define specialized assistants in `.claude/agents/` that Claude can delegate to for isolated tasks.

Subagents run in their own context with their own set of allowed tools. They're useful for tasks that read many files or need specialized focus without cluttering your main conversation.

```
---
name: security-reviewer
description: Reviews code for security vulnerabilities
tools: Read, Grep, Glob, Bash
model: opus
---
You are a senior security engineer. Review code for:
- Injection vulnerabilities (SQL, XSS, command injection)
- Authentication and authorization flaws
- Secrets or credentials in code
- Insecure data handling

Provide specific line references and suggested fixes.
```

Tell Claude to use subagents explicitly: *"Use a subagent to review this code for security issues."*

Install plugins

> **Tip:**
>
> Run `/plugin` to browse the marketplace. Plugins add skills, tools, and integrations without configuration.

Plugins bundle skills, hooks, subagents, and MCP servers into a single installable unit from the community and Anthropic. If you work with a typed language, install a code intelligence plugin to give Claude precise symbol navigation and automatic error detection after edits.

For guidance on choosing between skills, subagents, hooks, and MCP, see Extend Claude Code.

Communicate effectively

The way you communicate with Claude Code significantly impacts the quality of results.

Ask codebase questions

> **Tip:**
>
> Ask Claude questions you'd ask a senior engineer.

When onboarding to a new codebase, use Claude Code for learning and exploration. You can ask Claude the same sorts of questions you would ask another engineer:

- How does logging work?
- How do I make a new API endpoint?
- What does `async move { ... }` do on line 134 of `foo.rs`?
- What edge cases does `CustomerOnboardingFlowImpl` handle?
- Why does this code call `foo()` instead of `bar()` on line 333?

Using Claude Code this way is an effective onboarding workflow, improving ramp-up time and reducing load on other engineers. No special prompting required: ask questions directly.

Let Claude interview you

> **Tip:**
>
> For larger features, have Claude interview you first. Start with a minimal prompt and ask Claude to interview you using the `AskUserQuestion` tool.

Claude asks about things you might not have considered yet, including technical implementation, UI/UX, edge cases, and tradeoffs.

```
I want to build [brief description]. Interview me in detail using the
AskUserQuestion tool.

Ask about technical implementation, UI/UX, edge cases, concerns, and tradeoffs.
Don't ask obvious questions, dig into the hard parts I might not have considered.

Keep interviewing until we've covered everything, then write a complete spec to
SPEC.md.
```

Once the spec is complete, start a fresh session to execute it. The new session has clean context focused entirely on implementation, and you have a written spec to reference.

Manage your session

Conversations are persistent and reversible. Use this to your advantage!

Course-correct early and often

> **Tip:**
>
> Correct Claude as soon as you notice it going off track.

The best results come from tight feedback loops. Though Claude occasionally solves problems perfectly on the first attempt, correcting it quickly generally produces better solutions faster.

- `Esc` : stop Claude mid-action with the `Esc` key. Context is preserved, so you can redirect.
- `Esc + Esc` **or** `/rewind` : press `Esc` twice or run `/rewind` to open the rewind menu and restore previous conversation and code state, or summarize from a selected message.
- `"Undo that"` : have Claude revert its changes.
- `/clear` : reset context between unrelated tasks. Long sessions with irrelevant context can reduce performance.

If you've corrected Claude more than twice on the same issue in one session, the context is cluttered with failed approaches. Run `/clear` and start fresh with a more specific prompt that incorporates what you learned. A clean session with a better prompt almost always outperforms a long session with accumulated corrections.

Manage context aggressively

> **Tip:**
>
> Run `/clear` between unrelated tasks to reset context.

Claude Code automatically compacts conversation history when you approach context limits, which preserves important code and decisions while freeing space.

During long sessions, Claude's context window can fill with irrelevant conversation, file contents, and commands. This can reduce performance and sometimes distract Claude.

- Use `/clear` frequently between tasks to reset the context window entirely
- When auto compaction triggers, Claude summarizes what matters most, including code patterns, file states, and key decisions
- For more control, run `/compact <instructions>`, like `/compact Focus on the API changes`
- To compact only part of the conversation, use `Esc + Esc` or `/rewind`, select a message checkpoint, and choose **Summarize from here**. This condenses messages from that point forward while keeping earlier context intact.
- Customize compaction behavior in CLAUDE.md with instructions like `"When compacting, always preserve the full list of modified files and any test commands"` to ensure critical context survives summarization
- For quick questions that don't need to stay in context, use `/btw`. The answer appears in a dismissible overlay and never enters conversation history, so you can check a detail without growing context.

Use subagents for investigation

> **Tip:**
>
> Delegate research with `"use subagents to investigate X"`. They explore in a separate context, keeping your main conversation clean for implementation.

Since context is your fundamental constraint, subagents are one of the most powerful tools available. When Claude researches a codebase it reads lots of files, all of which consume your context. Subagents run in separate context windows and report back summaries:

```
Use subagents to investigate how our authentication system handles token
refresh, and whether we have any existing OAuth utilities I should reuse.
```

The subagent explores the codebase, reads relevant files, and reports back with findings, all without cluttering your main conversation.

You can also use subagents for verification after Claude implements something:

```
use a subagent to review this code for edge cases
```

Rewind with checkpoints

> **Tip:**
>
> Every action Claude makes creates a checkpoint. You can restore conversation, code, or both to any previous checkpoint.

Claude automatically checkpoints before changes. Double-tap `Escape` or run `/rewind` to open the rewind menu. You can restore conversation only, restore code only, restore both, or summarize from a selected message. See Checkpointing for details.

Instead of carefully planning every move, you can tell Claude to try something risky. If it doesn't work, rewind and try a different approach. Checkpoints persist across sessions, so you can close your terminal and still rewind later.

> **Warning:**
>
> Checkpoints only track changes made *by Claude*, not external processes. This isn't a replacement for git.

Resume conversations

> **Tip:**

> Run `claude --continue` to pick up where you left off, or `--resume` to choose from recent sessions.

Claude Code saves conversations locally. When a task spans multiple sessions, you don't have to re-explain the context:

```
claude --continue    # Resume the most recent conversation
claude --resume      # Select from recent conversations
```

Use `/rename` to give sessions descriptive names like `"oauth-migration"` or `"debugging-memory-leak"` so you can find them later. Treat sessions like branches: different workstreams can have separate, persistent contexts.

Automate and scale

Once you're effective with one Claude, multiply your output with parallel sessions, non-interactive mode, and fan-out patterns.

Everything so far assumes one human, one Claude, and one conversation. But Claude Code scales horizontally. The techniques in this section show how you can get more done.

Run non-interactive mode

> **Tip:**
>
> Use `claude -p "prompt"` in CI, pre-commit hooks, or scripts. Add `--output-format stream-json` for streaming JSON output.

With `claude -p "your prompt"`, you can run Claude non-interactively, without a session. Non-interactive mode is how you integrate Claude into CI pipelines, pre-commit hooks, or any automated workflow. The output formats let you parse results programmatically: plain text, JSON, or streaming JSON.

```
## One-off queries
claude -p "Explain what this project does"

## Structured output for scripts
claude -p "List all API endpoints" --output-format json

## Streaming for real-time processing
claude -p "Analyze this log file" --output-format stream-json
```

Run multiple Claude sessions

> **Tip:**
>
> Run multiple Claude sessions in parallel to speed up development, run isolated experiments, or start complex workflows.

There are three main ways to run parallel sessions:

- Claude Code desktop app: Manage multiple local sessions visually. Each session gets its own isolated worktree.
- Claude Code on the web: Run on Anthropic's secure cloud infrastructure in isolated VMs.
- Agent teams: Automated coordination of multiple sessions with shared tasks, messaging, and a team lead.

Beyond parallelizing work, multiple sessions enable quality-focused workflows. A fresh context improves code review since Claude won't be biased toward code it just wrote.

For example, use a Writer/Reviewer pattern:

Session A (Writer)	Session B (Reviewer)
`Implement a rate limiter for our API endpoints`	
	`Review the rate limiter implementation in @src/middleware/rateLimiter.ts. Look for edge cases, race conditions, and consistency with our existing middleware patterns.`

Session A (Writer)	Session B (Reviewer)
`Here's the review feedback: [Session B output]. Address these issues.`	

You can do something similar with tests: have one Claude write tests, then another write code to pass them.

Fan out across files

> **Tip:**
>
> Loop through tasks calling `claude -p` for each. Use `--allowedTools` to scope permissions for batch operations.

For large migrations or analyses, you can distribute work across many parallel Claude invocations:

Step 1: Generate a task list

Have Claude list all files that need migrating (e.g., `list all 2,000 Python files that need migrating`)

Step 2: Write a script to loop through the list

```
for file in $(cat files.txt); do
  claude -p "Migrate $file from React to Vue. Return OK or FAIL." \
    --allowedTools "Edit,Bash(git commit *)"
done
```

Step 3: Test on a few files, then run at scale

Refine your prompt based on what goes wrong with the first 2-3 files, then run on the full set. The `--allowedTools` flag restricts what Claude can do, which matters when you're running unattended.

You can also integrate Claude into existing data/processing pipelines:

```
claude -p "<your prompt>" --output-format json | your_command
```

Use `--verbose` for debugging during development, and turn it off in production.

Avoid common failure patterns

These are common mistakes. Recognizing them early saves time:

- **The kitchen sink session.** You start with one task, then ask Claude something unrelated, then go back to the first task. Context is full of irrelevant information. > **Fix**: `/clear` between unrelated tasks.
- **Correcting over and over.** Claude does something wrong, you correct it, it's still wrong, you correct again. Context is polluted with failed approaches. > **Fix**: After two failed corrections, `/clear` and write a better initial prompt incorporating what you learned.
- **The over-specified CLAUDE.md.** If your CLAUDE.md is too long, Claude ignores half of it because important rules get lost in the noise. > **Fix**: Ruthlessly prune. If Claude already does something correctly without the instruction, delete it or convert it to a hook.
- **The trust-then-verify gap.** Claude produces a plausible-looking implementation that doesn't handle edge cases. > **Fix**: Always provide verification (tests, scripts, screenshots). If you can't verify it, don't ship it.
- **The infinite exploration.** You ask Claude to "investigate" something without scoping it. Claude reads hundreds of files, filling the context. > **Fix**: Scope investigations narrowly or use subagents so the exploration doesn't consume your main context.

Develop your intuition

The patterns in this guide aren't set in stone. They're starting points that work well in general, but might not be optimal for every situation.

Sometimes you *should* let context accumulate because you're deep in one complex problem and the history is valuable. Sometimes you should skip planning and let Claude figure it out because the task is exploratory. Sometimes a vague prompt is exactly right because you want to see how Claude interprets the problem before constraining it.

Pay attention to what works. When Claude produces great output, notice what you did: the prompt structure, the context you provided, the mode you were in. When Claude struggles, ask why. Was the context too noisy? The prompt too vague? The task too big for one pass?

Over time, you'll develop intuition that no guide can capture. You'll know when to be specific and when to be open-ended, when to plan and when to explore, when to clear context and when to let it accumulate.

Related resources

- How Claude Code works: the agentic loop, tools, and context management
- Extend Claude Code: skills, hooks, MCP, subagents, and plugins
- Common workflows: step-by-step recipes for debugging, testing, PRs, and more
- CLAUDE.md: store project conventions and persistent context

Common workflows

> *Step-by-step guides for exploring codebases, fixing bugs, refactoring, testing, and other everyday tasks with Claude Code.*

This page covers practical workflows for everyday development: exploring unfamiliar code, debugging, refactoring, writing tests, creating PRs, and managing sessions. Each section includes example prompts you can adapt to your own projects. For higher-level patterns and tips, see Best practices.

Understand new codebases

Get a quick codebase overview

Suppose you've just joined a new project and need to understand its structure quickly.

Step 1: Navigate to the project root directory

```
cd /path/to/project
```

Step 2: Start Claude Code

```
claude
```

Step 3: Ask for a high-level overview

```
give me an overview of this codebase
```

Step 4: Dive deeper into specific components

```
explain the main architecture patterns used here
```

```
what are the key data models?
```

```
how is authentication handled?
```

> **Tip:**
>
> Tips:
>
> - Start with broad questions, then narrow down to specific areas
> - Ask about coding conventions and patterns used in the project
> - Request a glossary of project-specific terms

Find relevant code

Suppose you need to locate code related to a specific feature or functionality.

Step 1: Ask Claude to find relevant files

```
find the files that handle user authentication
```

Step 2: Get context on how components interact

```
how do these authentication files work together?
```

Step 3: Understand the execution flow

```
trace the login process from front-end to database
```

> **Tip:**
>
> Tips:
>
> - Be specific about what you're looking for
> - Use domain language from the project
> - Install a code intelligence plugin for your language to give Claude precise "go to definition" and "find references" navigation

---

Fix bugs efficiently

Suppose you've encountered an error message and need to find and fix its source.

Step 1: Share the error with Claude

```
I'm seeing an error when I run npm test
```

Step 2: Ask for fix recommendations

```
suggest a few ways to fix the @ts-ignore in user.ts
```

Step 3: Apply the fix

```
update user.ts to add the null check you suggested
```

> **Tip:**
>
> Tips:
>
> - Tell Claude the command to reproduce the issue and get a stack trace
> - Mention any steps to reproduce the error
> - Let Claude know if the error is intermittent or consistent

---

Refactor code

Suppose you need to update old code to use modern patterns and practices.

Step 1: Identify legacy code for refactoring

```
find deprecated API usage in our codebase
```

Step 2: Get refactoring recommendations

```
suggest how to refactor utils.js to use modern JavaScript features
```

Step 3: Apply the changes safely

```
refactor utils.js to use ES2024 features while maintaining the same behavior
```

Step 4: Verify the refactoring

```
run tests for the refactored code
```

> **Tip:**
>
> Tips:
>
> - Ask Claude to explain the benefits of the modern approach
> - Request that changes maintain backward compatibility when needed
> - Do refactoring in small, testable increments

Use specialized subagents

Suppose you want to use specialized AI subagents to handle specific tasks more effectively.

Step 1: View available subagents

```
/agents
```

This shows all available subagents and lets you create new ones.

Step 2: Use subagents automatically

Claude Code automatically delegates appropriate tasks to specialized subagents:

```
review my recent code changes for security issues
```

```
run all tests and fix any failures
```

Step 3: Explicitly request specific subagents

```
use the code-reviewer subagent to check the auth module
```

```
have the debugger subagent investigate why users can't log in
```

Step 4: Create custom subagents for your workflow

```
/agents
```

Then select “Create New subagent” and follow the prompts to define:

- A unique identifier that describes the subagent’s purpose (for example, `code-reviewer`, `api-designer`).
- When Claude should use this agent
- Which tools it can access
- A system prompt describing the agent’s role and behavior

> **Tip:**
>
> Tips:
>
> - Create project-specific subagents in `.claude/agents/` for team sharing
> - Use descriptive `description` fields to enable automatic delegation
> - Limit tool access to what each subagent actually needs
> - Check the subagents documentation for detailed examples

Use Plan Mode for safe code analysis

Plan Mode instructs Claude to create a plan by analyzing the codebase with read-only operations, perfect for exploring codebases, planning complex changes, or reviewing code safely. In Plan Mode, Claude uses `AskUserQuestion` to gather requirements and clarify your goals before proposing a plan.

When to use Plan Mode

- **Multi-step implementation**: When your feature requires making edits to many files
- **Code exploration**: When you want to research the codebase thoroughly before changing anything
- **Interactive development**: When you want to iterate on the direction with Claude

How to use Plan Mode

Turn on Plan Mode during a session

You can switch into Plan Mode during a session using **Shift+Tab** to cycle through permission modes.

If you are in Normal Mode, **Shift+Tab** first switches into Auto-Accept Mode, indicated by `▸▸ accept edits on` at the bottom of the terminal. A subsequent **Shift+Tab** will switch into Plan Mode, indicated by `⏸ plan mode on`.

Start a new session in Plan Mode

To start a new session in Plan Mode, use the `--permission-mode plan` flag:

```
claude --permission-mode plan
```

Run "headless" queries in Plan Mode

You can also run a query in Plan Mode directly with `-p` (that is, in "headless mode"):

```
claude --permission-mode plan -p "Analyze the authentication system and suggest improvements"
```

Example: Planning a complex refactor

```
claude --permission-mode plan
```

```
I need to refactor our authentication system to use OAuth2. Create a detailed migration plan.
```

Claude analyzes the current implementation and create a comprehensive plan. Refine with follow-ups:

```
What about backward compatibility?
```

```
How should we handle database migration?
```

> **Tip:**

> Press `Ctrl+G` to open the plan in your default text editor, where you can edit it directly before Claude proceeds.

Configure Plan Mode as default

```
// .claude/settings.json
{
  "permissions": {
    "defaultMode": "plan"
  }
}
```

See settings documentation for more configuration options.

Work with tests

Suppose you need to add tests for uncovered code.

Step 1: Identify untested code

```
find functions in NotificationsService.swift that are not covered by tests
```

Step 2: Generate test scaffolding

```
add tests for the notification service
```

Step 3: Add meaningful test cases

```
add test cases for edge conditions in the notification service
```

Step 4: Run and verify tests

```
run the new tests and fix any failures
```

Claude can generate tests that follow your project's existing patterns and conventions. When asking for tests, be specific about what behavior you want to verify. Claude examines your existing test files to match the style, frameworks, and assertion patterns already in use.

For comprehensive coverage, ask Claude to identify edge cases you might have missed. Claude can analyze your code paths and suggest tests for error conditions, boundary values, and unexpected inputs that are easy to overlook.

Create pull requests

You can create pull requests by asking Claude directly ("create a pr for my changes"), or guide Claude through it step-by-step:

Step 1: Summarize your changes

```
summarize the changes I've made to the authentication module
```

Step 2: Generate a pull request

```
create a pr
```

Step 3: Review and refine

```
enhance the PR description with more context about the security improvements
```

When you create a PR using `gh pr create`, the session is automatically linked to that PR. You can resume it later with `claude --from-pr <number>`.

> **Tip:**
>
> Review Claude's generated PR before submitting and ask Claude to highlight potential risks or considerations.

Handle documentation

Suppose you need to add or update documentation for your code.

Step 1: Identify undocumented code

```
find functions without proper JSDoc comments in the auth module
```

Step 2: Generate documentation

```
add JSDoc comments to the undocumented functions in auth.js
```

Step 3: Review and enhance

```
improve the generated documentation with more context and examples
```

Step 4: Verify documentation

```
check if the documentation follows our project standards
```

> **Tip:**
>
> Tips:
>
> - Specify the documentation style you want (JSDoc, docstrings, etc.)
> - Ask for examples in the documentation
> - Request documentation for public APIs, interfaces, and complex logic

---

Work with images

Suppose you need to work with images in your codebase, and you want Claude's help analyzing image content.

Step 1: Add an image to the conversation

You can use any of these methods:

1. Drag and drop an image into the Claude Code window
2. Copy an image and paste it into the CLI with ctrl+v (Do not use cmd+v)
3. Provide an image path to Claude. E.g., "Analyze this image: /path/to/your/image.png"

Step 2: Ask Claude to analyze the image

```
What does this image show?
```

```
Describe the UI elements in this screenshot
```

```
Are there any problematic elements in this diagram?
```

Step 3: Use images for context

```
Here's a screenshot of the error. What's causing it?
```

```
This is our current database schema. How should we modify it for the new feature?
```

Step 4: Get code suggestions from visual content

```
Generate CSS to match this design mockup
```

```
What HTML structure would recreate this component?
```

Tip:

Tips:

- Use images when text descriptions would be unclear or cumbersome
- Include screenshots of errors, UI designs, or diagrams for better context
- You can work with multiple images in a conversation
- Image analysis works with diagrams, screenshots, mockups, and more
- When Claude references images (for example, `[Image #1]`), `Cmd+Click` (Mac) or `Ctrl+Click` (Windows/Linux) the link to open the image in your default viewer

---

Reference files and directories

Use @ to quickly include files or directories without waiting for Claude to read them.

Step 1: Reference a single file

```
Explain the logic in @src/utils/auth.js
```

This includes the full content of the file in the conversation.

Step 2: Reference a directory

```
What's the structure of @src/components?
```

This provides a directory listing with file information.

Step 3: Reference MCP resources

```
Show me the data from @github:repos/owner/repo/issues
```

This fetches data from connected MCP servers using the format @server:resource. See MCP resources for details.

> **Tip:**
>
> Tips:
>
> - File paths can be relative or absolute
> - @ file references add `CLAUDE.md` in the file's directory and parent directories to context
> - Directory references show file listings, not contents
> - You can reference multiple files in a single message (for example, "@file1.js and @file2.js")

Use extended thinking (thinking mode)

Extended thinking is enabled by default, giving Claude space to reason through complex problems step-by-step before responding. This reasoning is visible in verbose mode, which you can toggle on with `Ctrl+O`.

Additionally, Opus 4.6 and Sonnet 4.6 support adaptive reasoning: instead of a fixed thinking token budget, the model dynamically allocates thinking based on your effort level setting. Extended thinking and adaptive reasoning work together to give you control over how deeply Claude reasons before responding.

Extended thinking is particularly valuable for complex architectural decisions, challenging bugs, multi-step implementation planning, and evaluating tradeoffs between different approaches.

> **Note:**
>
> Phrases like “think”, “think hard”, and “think more” are interpreted as regular prompt instructions and don’t allocate thinking tokens.

Configure thinking mode

Thinking is enabled by default, but you can adjust or disable it.

Scope	How to configure	Details
Effort level	Run `/effort`, adjust in `/model`, or set `CLAUDE_CODE_EFFORT_LEVEL`	Control thinking depth for Opus 4.6 and Sonnet 4.6. See Adjust effort level
`ultrathink` **keyword**	Include “ultrathink” anywhere in your prompt	Sets effort to high for that turn on Opus 4.6 and Sonnet 4.6. Useful for one-off tasks requiring deep reasoning without permanently changing your effort setting
Toggle shortcut	Press `Option+T` (macOS) or `Alt+T` (Windows/Linux)	Toggle thinking on/off for the current session (all models). May require terminal configuration to enable Option key shortcuts
Global default	Use `/config` to toggle thinking mode	Sets your default across all projects (all models). Saved as `alwaysThinkingEnabled` in `~/.claude/settings.json`

Scope	How to configure	Details
Limit token budget	Set `MAX_THINKING_TOKENS` environment variable	Limit the thinking budget to a specific number of tokens (ignored on Opus 4.6 and Sonnet 4.6 unless set to 0). Example: `export MAX_THINKING_TOKENS=10000`

To view Claude's thinking process, press `Ctrl+O` to toggle verbose mode and see the internal reasoning displayed as gray italic text.

How extended thinking works

Extended thinking controls how much internal reasoning Claude performs before responding. More thinking provides more space to explore solutions, analyze edge cases, and self-correct mistakes.

With Opus 4.6 and Sonnet 4.6, thinking uses adaptive reasoning: the model dynamically allocates thinking tokens based on the effort level you select. This is the recommended way to tune the tradeoff between speed and reasoning depth.

With older models, thinking uses a fixed budget of up to 31,999 tokens from your output budget. You can limit this with the `MAX_THINKING_TOKENS` environment variable, or disable thinking entirely via `/config` or the `Option+T` / `Alt+T` toggle.

`MAX_THINKING_TOKENS` is ignored on Opus 4.6 and Sonnet 4.6, since adaptive reasoning controls thinking depth instead. The one exception: setting `MAX_THINKING_TOKENS=0` still disables thinking entirely on any model. To disable adaptive thinking and revert to the fixed thinking budget, set `CLAUDE_CODE_DISABLE_ADAPTIVE_THINKING=1`. See environment variables.

> **Warning:**
>
> You're charged for all thinking tokens used, even though Claude 4 models show summarized thinking

Resume previous conversations

When starting Claude Code, you can resume a previous session:

- `claude --continue` continues the most recent conversation in the current directory
- `claude --resume` opens a conversation picker or resumes by name
- `claude --from-pr 123` resumes sessions linked to a specific pull request

From inside an active session, use `/resume` to switch to a different conversation.

Sessions are stored per project directory. The `/resume` picker shows sessions from the same git repository, including worktrees.

Name your sessions

Give sessions descriptive names to find them later. This is a best practice when working on multiple tasks or features.

Step 1: Name the session

Name a session at startup with `-n`:

```
claude -n auth-refactor
```

Or use `/rename` during a session, which also shows the name on the prompt bar:

```
/rename auth-refactor
```

You can also rename any session from the picker: run `/resume`, navigate to a session, and press `R`.

Step 2: Resume by name later

From the command line:

```
claude --resume auth-refactor
```

Or from inside an active session:

```
/resume auth-refactor
```

Use the session picker

The `/resume` command (or `claude --resume` without arguments) opens an interactive session picker with these features:

Keyboard shortcuts in the picker:

Shortcut	Action
`↑` / `↓`	Navigate between sessions
`→` / `←`	Expand or collapse grouped sessions
`Enter`	Select and resume the highlighted session
`P`	Preview the session content
`R`	Rename the highlighted session
`/`	Search to filter sessions
`A`	Toggle between current directory and all projects
`B`	Filter to sessions from your current git branch
`Esc`	Exit the picker or search mode

Session organization:

The picker displays sessions with helpful metadata:

- Session name or initial prompt
- Time elapsed since last activity
- Message count
- Git branch (if applicable)

Forked sessions (created with `/rewind` or `--fork-session`) are grouped together under their root session, making it easier to find related conversations.

> **Tip:**
>
> Tips:
>
> - **Name sessions early**: Use `/rename` when starting work on a distinct task—it's much easier to find "payment-integration" than "explain this function" later
> - Use `--continue` for quick access to your most recent conversation in the current directory

- Use `--resume session-name` when you know which session you need
- Use `--resume` (without a name) when you need to browse and select
- For scripts, use `claude --continue --print "prompt"` to resume in non-interactive mode
- Press `P` in the picker to preview a session before resuming it
- The resumed conversation starts with the same model and configuration as the original

How it works:

1. **Conversation Storage**: All conversations are automatically saved locally with their full message history
2. **Message Deserialization**: When resuming, the entire message history is restored to maintain context
3. **Tool State**: Tool usage and results from the previous conversation are preserved
4. **Context Restoration**: The conversation resumes with all previous context intact

---

Run parallel Claude Code sessions with Git worktrees

When working on multiple tasks at once, you need each Claude session to have its own copy of the codebase so changes don't collide. Git worktrees solve this by creating separate working directories that each have their own files and branch, while sharing the same repository history and remote connections. This means you can have Claude working on a feature in one worktree while fixing a bug in another, without either session interfering with the other.

Use the `--worktree` (`-w`) flag to create an isolated worktree and start Claude in it. The value you pass becomes the worktree directory name and branch name:

```
## Start Claude in a worktree named "feature-auth"
## Creates .claude/worktrees/feature-auth/ with a new branch
claude --worktree feature-auth

## Start another session in a separate worktree
claude --worktree bugfix-123
```

If you omit the name, Claude generates a random one automatically:

```
## Auto-generates a name like "bright-running-fox"
claude --worktree
```

Worktrees are created at `<repo>/.claude/worktrees/<name>` and branch from the default remote branch. The worktree branch is named `worktree-<name>`.

You can also ask Claude to "work in a worktree" or "start a worktree" during a session, and it will create one automatically.

Subagent worktrees

Subagents can also use worktree isolation to work in parallel without conflicts. Ask Claude to "use worktrees for your agents" or configure it in a custom subagent by adding `isolation: worktree` to the agent's frontmatter. Each subagent gets its own worktree that is automatically cleaned up when the subagent finishes without changes.

Worktree cleanup

When you exit a worktree session, Claude handles cleanup based on whether you made changes:

- **No changes**: the worktree and its branch are removed automatically
- **Changes or commits exist**: Claude prompts you to keep or remove the worktree. Keeping preserves the directory and branch so you can return later. Removing deletes the worktree directory and its branch, discarding all uncommitted changes and commits

To clean up worktrees outside of a Claude session, use manual worktree management.

> **Tip:**
>
> Add `.claude/worktrees/` to your `.gitignore` to prevent worktree contents from appearing as untracked files in your main repository.

Manage worktrees manually

For more control over worktree location and branch configuration, create worktrees with Git directly. This is useful when you need to check out a specific existing branch or place the worktree outside the repository.

```
## Create a worktree with a new branch
git worktree add ../project-feature-a -b feature-a

## Create a worktree with an existing branch
git worktree add ../project-bugfix bugfix-123

## Start Claude in the worktree
cd ../project-feature-a && claude

## Clean up when done
git worktree list
git worktree remove ../project-feature-a
```

Learn more in the official Git worktree documentation.

> **Tip:**
>
> Remember to initialize your development environment in each new worktree according to your project's setup. Depending on your stack, this might include running dependency installation (`npm install`, `yarn`), setting up virtual environments, or following your project's standard setup process.

Non-git version control

Worktree isolation works with git by default. For other version control systems like SVN, Perforce, or Mercurial, configure WorktreeCreate and WorktreeRemove hooks to provide custom worktree creation and cleanup logic. When configured, these hooks replace the default git behavior when you use `--worktree`.

For automated coordination of parallel sessions with shared tasks and messaging, see agent teams.

Get notified when Claude needs your attention

When you kick off a long-running task and switch to another window, you can set up desktop notifications so you know when Claude finishes or needs your input. This uses the `Notification` hook event, which fires whenever Claude is waiting for permission, idle and ready for a new prompt, or completing authentication.

Step 1: Add the hook to your settings

Open `~/.claude/settings.json` and add a `Notification` hook that calls your platform's native notification command:

macOS

```
{
  "hooks": {
    "Notification": [
      {
        "matcher": "",
        "hooks": [
          {
            "type": "command",
            "command": "osascript -e 'display notification \"Claude Code needs your attention\" with title \"Claude Code\"'"
          }
        ]
      }
    ]
  }
}
```

Linux

```
{
  "hooks": {
    "Notification": [
      {
        "matcher": "",
        "hooks": [
          {
            "type": "command",
            "command": "notify-send 'Claude Code' 'Claude Code needs your
attention'"
          }
        ]
      }
    ]
  }
}
```

Windows

```
{
  "hooks": {
    "Notification": [
      {
        "matcher": "",
        "hooks": [
          {
            "type": "command",
            "command": "powershell.exe -Command \"[System.Reflection.Assembly]::Lo
adWithPartialName('System.Windows.Forms');
[System.Windows.Forms.MessageBox]::Show('Claude Code needs your attention',
'Claude Code')\""
          }
        ]
      }
    ]
  }
}
```

If your settings file already has a `hooks` key, merge the `Notification` entry into it rather than overwriting. You can also ask Claude to write the hook for you by describing what you want in the CLI.

Step 2: Optionally narrow the matcher

By default the hook fires on all notification types. To fire only for specific events, set the `matcher` field to one of these values:

Matcher	Fires when
`permission_prompt`	Claude needs you to approve a tool use
`idle_prompt`	Claude is done and waiting for your next prompt
`auth_success`	Authentication completes
`elicitation_dialog`	Claude is asking you a question

Step 3: Verify the hook

Type `/hooks` and select `Notification` to confirm the hook appears. Selecting it shows the command that will run. To test it end-to-end, ask Claude to run a command that requires permission and switch away from the terminal, or ask Claude to trigger a notification directly.

For the complete event schema and notification types, see the Notification reference.

Use Claude as a unix-style utility

Add Claude to your verification process

Suppose you want to use Claude Code as a linter or code reviewer.

Add Claude to your build script:

```
// package.json
{
    ...
    "scripts": {
        ...
        "lint:claude": "claude -p 'you are a linter. please look at the changes vs. main and report any issues related to typos. report the filename and line number on one line, and a description of the issue on the second line. do not return any other text.'"
    }
}
```

> **Tip:**
>
> Tips:
>
> - Use Claude for automated code review in your CI/CD pipeline
> - Customize the prompt to check for specific issues relevant to your project
> - Consider creating multiple scripts for different types of verification

Pipe in, pipe out

Suppose you want to pipe data into Claude, and get back data in a structured format.

Pipe data through Claude:

```
cat build-error.txt | claude -p 'concisely explain the root cause of this build error' > output.txt
```

Tip:

Tips:

- Use pipes to integrate Claude into existing shell scripts
- Combine with other Unix tools for powerful workflows
- Consider using –output-format for structured output

Control output format

Suppose you need Claude's output in a specific format, especially when integrating Claude Code into scripts or other tools.

Step 1: Use text format (default)

```
cat data.txt | claude -p 'summarize this data' --output-format text > summary.txt
```

This outputs just Claude's plain text response (default behavior).

Step 2: Use JSON format

```
cat code.py | claude -p 'analyze this code for bugs' --output-format json > analysis.json
```

This outputs a JSON array of messages with metadata including cost and duration.

Step 3: Use streaming JSON format

```
cat log.txt | claude -p 'parse this log file for errors' --output-format stream-json
```

This outputs a series of JSON objects in real-time as Claude processes the request. Each message is a valid JSON object, but the entire output is not valid JSON if concatenated.

Tip:

Tips:

- Use `--output-format text` for simple integrations where you just need Claude's response
- Use `--output-format json` when you need the full conversation log
- Use `--output-format stream-json` for real-time output of each conversation turn

Ask Claude about its capabilities

Claude has built-in access to its documentation and can answer questions about its own features and limitations.

Example questions

```
can Claude Code create pull requests?
```

```
how does Claude Code handle permissions?
```

```
what skills are available?
```

```
how do I use MCP with Claude Code?
```

```
how do I configure Claude Code for Amazon Bedrock?
```

```
what are the limitations of Claude Code?
```

Note:

Claude provides documentation-based answers to these questions. For executable examples and hands-on demonstrations, refer to the specific workflow sections above.

Tip:

Tips:

- Claude always has access to the latest Claude Code documentation, regardless of the version you're using
- Ask specific questions to get detailed answers
- Claude can explain complex features like MCP integration, enterprise configurations, and advanced workflows

Next steps

- Best practices Patterns for getting the most out of Claude Code
- How Claude Code works Understand the agentic loop and context management
- Extend Claude Code Add skills, hooks, MCP, subagents, and plugins
- Reference implementation Clone our development container reference implementation

Part 3: Commands & Reference

Built-in commands

> *Complete reference for built-in commands available in Claude Code.*

Type `/` in Claude Code to see all available commands, or type `/` followed by any letters to filter. Not all commands are visible to every user. Some depend on your platform, plan, or environment. For example, `/desktop` only appears on macOS and Windows, `/upgrade` and `/privacy-settings` are only available on Pro and Max plans, and `/terminal-setup` is hidden when your terminal natively supports its keybindings.

Claude Code also includes bundled skills like `/simplify`, `/batch`, and `/debug` that appear alongside built-in commands when you type `/`. To create your own commands, see skills.

In the table below, `<arg>` indicates a required argument and `[arg]` indicates an optional one.

Command	Purpose
`/add-dir <path>`	Add a new working directory to the current session
`/agents`	Manage agent configurations
`/btw <question>`	Ask a quick side question without adding to the conversation
`/chrome`	Configure Claude in Chrome settings
`/clear`	Clear conversation history and free up context. Aliases: `/reset`, `/new`
`/color [color|default]`	Set the prompt bar color for the current session. Available colors: `red`, `blue`, `green`, `yellow`, `purple`, `orange`, `pink`, `cyan`. Use `default` to reset
`/compact [instructions]`	Compact conversation with optional focus instructions

Command	Purpose
`/config`	Open the Settings interface to adjust theme, model, output style, and other preferences. Alias: `/settings`
`/context`	Visualize current context usage as a colored grid. Shows optimization suggestions for context-heavy tools, memory bloat, and capacity warnings
`/copy`	Copy the last assistant response to clipboard. When code blocks are present, shows an interactive picker to select individual blocks or the full response
`/cost`	Show token usage statistics. See cost tracking guide for subscription-specific details
`/desktop`	Continue the current session in the Claude Code Desktop app. macOS and Windows only. Alias: `/app`
`/diff`	Open an interactive diff viewer showing uncommitted changes and per-turn diffs. Use left/right arrows to switch between the current git diff and individual Claude turns, and up/down to browse files
`/doctor`	Diagnose and verify your Claude Code installation and settings
`/effort [low|medium|high|max|auto]`	Set the model effort level. `low`, `medium`, and `high` persist across sessions. `max` applies to the current session only and requires Opus 4.6. `auto` resets to the model default. Without an argument, shows the current level. Takes effect immediately without waiting for the current response to finish
`/exit`	Exit the CLI. Alias: `/quit`
`/export [filename]`	Export the current conversation as plain text. With a filename, writes directly to that file. Without, opens a dialog to copy to clipboard or save to a file
`/extra-usage`	Configure extra usage to keep working when rate limits are hit
`/fast [on|off]`	Toggle fast mode on or off
`/feedback [report]`	Submit feedback about Claude Code. Alias: `/bug`
`/fork [name]`	Create a fork of the current conversation at this point
`/help`	Show help and available commands
`/hooks`	View hook configurations for tool events

Command	Purpose
`/ide`	Manage IDE integrations and show status
`/init`	Initialize project with `CLAUDE.md` guide
`/insights`	Generate a report analyzing your Claude Code sessions, including project areas, interaction patterns, and friction points
`/install-github-app`	Set up the Claude GitHub Actions app for a repository. Walks you through selecting a repo and configuring the integration
`/install-slack-app`	Install the Claude Slack app. Opens a browser to complete the OAuth flow
`/keybindings`	Open or create your keybindings configuration file
`/login`	Sign in to your Anthropic account
`/logout`	Sign out from your Anthropic account
`/mcp`	Manage MCP server connections and OAuth authentication
`/memory`	Edit `CLAUDE.md` memory files, enable or disable auto-memory, and view auto-memory entries
`/mobile`	Show QR code to download the Claude mobile app. Aliases: `/ios`, `/android`
`/model [model]`	Select or change the AI model. For models that support it, use left/right arrows to adjust effort level. The change takes effect immediately without waiting for the current response to finish
`/passes`	Share a free week of Claude Code with friends. Only visible if your account is eligible
`/permissions`	View or update permissions. Alias: `/allowed-tools`
`/plan`	Enter plan mode directly from the prompt
`/plugin`	Manage Claude Code plugins
`/pr-comments [PR]`	Fetch and display comments from a GitHub pull request. Automatically detects the PR for the current branch, or pass a PR URL or number. Requires the `gh` CLI

Command	Purpose
`/privacy-settings`	View and update your privacy settings. Only available for Pro and Max plan subscribers
`/release-notes`	View the full changelog, with the most recent version closest to your prompt
`/reload-plugins`	Reload all active plugins to apply pending changes without restarting. Reports counts for each reloaded component and flags any load errors
`/remote-control`	Make this session available for remote control from claude.ai. Alias: `/rc`
`/remote-env`	Configure the default remote environment for web sessions started with `--remote`
`/rename [name]`	Rename the current session and show the name on the prompt bar. Without a name, auto-generates one from conversation history
`/resume [session]`	Resume a conversation by ID or name, or open the session picker. Alias: `/continue`
`/review`	Deprecated. Install the `code-review` plugin instead: `claude plugin install code-review@claude-code-marketplace`
`/rewind`	Rewind the conversation and/or code to a previous point, or summarize from a selected message. See checkpointing. Alias: `/checkpoint`
`/sandbox`	Toggle sandbox mode. Available on supported platforms only
`/security-review`	Analyze pending changes on the current branch for security vulnerabilities. Reviews the git diff and identifies risks like injection, auth issues, and data exposure
`/skills`	List available skills
`/stats`	Visualize daily usage, session history, streaks, and model preferences
`/status`	Open the Settings interface (Status tab) showing version, model, account, and connectivity
`/statusline`	Configure Claude Code's status line. Describe what you want, or run without arguments to auto-configure from your shell prompt
`/stickers`	Order Claude Code stickers
`/tasks`	List and manage background tasks

Command	Purpose
`/terminal-setup`	Configure terminal keybindings for Shift+Enter and other shortcuts. Only visible in terminals that need it, like VS Code, Alacritty, or Warp
`/theme`	Change the color theme. Includes light and dark variants, colorblind-accessible (daltonized) themes, and ANSI themes that use your terminal's color palette
`/upgrade`	Open the upgrade page to switch to a higher plan tier
`/usage`	Show plan usage limits and rate limit status
`/vim`	Toggle between Vim and Normal editing modes

MCP prompts

MCP servers can expose prompts that appear as commands. These use the format `/mcp__<server>__<prompt>` and are dynamically discovered from connected servers. See MCP prompts for details.

See also

- Skills: create your own commands
- Interactive mode: keyboard shortcuts, Vim mode, and command history
- CLI reference: launch-time flags

CLI reference

> *Complete reference for Claude Code command-line interface, including commands and flags.*

CLI commands

You can start sessions, pipe content, resume conversations, and manage updates with these commands:

Command	Description	Example
`claude`	Start interactive session	`claude`
`claude "query"`	Start interactive session with initial prompt	`claude "explain this project"`
`claude -p "query"`	Query via SDK, then exit	`claude -p "explain this function"`
`cat file | claude -p "query"`	Process piped content	`cat logs.txt | claude -p "explain"`
`claude -c`	Continue most recent conversation in current directory	`claude -c`
`claude -c -p "query"`	Continue via SDK	`claude -c -p "Check for type errors"`
`claude -r "<session>" "query"`	Resume session by ID or name	`claude -r "auth-refactor" "Finish this PR"`
`claude update`	Update to latest version	`claude update`
`claude auth login`	Sign in to your Anthropic account. Use `--email` to pre-fill your email address and `--sso` to force SSO authentication	`claude auth login --email user@example.com --sso`
`claude auth logout`	Log out from your Anthropic account	`claude auth logout`

Command	Description	Example
`claude auth status`	Show authentication status as JSON. Use `--text` for human-readable output. Exits with code 0 if logged in, 1 if not	`claude auth status`
`claude agents`	List all configured subagents, grouped by source	`claude agents`
`claude mcp`	Configure Model Context Protocol (MCP) servers	See the Claude Code MCP documentation.
`claude remote-control`	Start a Remote Control server to control Claude Code from Claude.ai or the Claude app. Runs in server mode (no local interactive session). See Server mode flags	`claude remote-control --name "My Project"`

CLI flags

Customize Claude Code's behavior with these command-line flags:

Flag	Description	Example
`--add-dir`	Add additional working directories for Claude to access (validates each path exists as a directory)	`claude --add-dir ../apps ../lib`
`--agent`	Specify an agent for the current session (overrides the `agent` setting)	`claude --agent my-custom-agent`
`--agents`	Define custom subagents dynamically via JSON. Uses the same field names as subagent frontmatter, plus a `prompt` field for the agent's instructions	`claude --agents '{"reviewer": {"description":"Reviews code","prompt":"You are a code reviewer"}}'`

Flag	Description	Example
`--allow-dangerously-skip-permissions`	Enable permission bypassing as an option without immediately activating it. Allows composing with `--permission-mode` (use with caution)	`claude --permission-mode plan --allow-dangerously-skip-permissions`
`--allowedTools`	Tools that execute without prompting for permission. See permission rule syntax for pattern matching. To restrict which tools are available, use `--tools` instead	`"Bash(git log *)" "Bash(git diff *)" "Read"`
`--append-system-prompt`	Append custom text to the end of the default system prompt	`claude --append-system-prompt "Always use TypeScript"`
`--append-system-prompt-file`	Load additional system prompt text from a file and append to the default prompt	`claude --append-system-prompt-file ./extra-rules.txt`
`--betas`	Beta headers to include in API requests (API key users only)	`claude --betas interleaved-thinking`
`--chrome`	Enable Chrome browser integration for web automation and testing	`claude --chrome`
`--continue`, `-c`	Load the most recent conversation in the current directory	`claude --continue`
`--dangerously-skip-permissions`	Skip all permission prompts (use with caution)	`claude --dangerously-skip-permissions`
`--debug`	Enable debug mode with optional category filtering (for example, `"api,hooks"` or `"!statsig,!file"`)	`claude --debug "api,mcp"`
`--disable-slash-commands`	Disable all skills and commands for this session	`claude --disable-slash-commands`
`--disallowedTools`	Tools that are removed from the model's context and cannot be used	`"Bash(git log *)" "Bash(git diff *)" "Edit"`

Flag	Description	Example
`--effort`	Set the effort level for the current session. Options: `low`, `medium`, `high`, `max` (Opus 4.6 only). Session-scoped and does not persist to settings	`claude --effort high`
`--fallback-model`	Enable automatic fallback to specified model when default model is overloaded (print mode only)	`claude -p --fallback-model sonnet "query"`
`--fork-session`	When resuming, create a new session ID instead of reusing the original (use with `--resume` or `--continue`)	`claude --resume abc123 --fork-session`
`--from-pr`	Resume sessions linked to a specific GitHub PR. Accepts a PR number or URL. Sessions are automatically linked when created via `gh pr create`	`claude --from-pr 123`
`--ide`	Automatically connect to IDE on startup if exactly one valid IDE is available	`claude --ide`
`--init`	Run initialization hooks and start interactive mode	`claude --init`
`--init-only`	Run initialization hooks and exit (no interactive session)	`claude --init-only`
`--include-partial-messages`	Include partial streaming events in output (requires `--print` and `--output-format=stream-json`)	`claude -p --output-format stream-json --include-partial-messages "query"`
`--input-format`	Specify input format for print mode (options: `text`, `stream-json`)	`claude -p --output-format json --input-format stream-json`

Flag	Description	Example
`--json-schema`	Get validated JSON output matching a JSON Schema after agent completes its workflow (print mode only, see structured outputs)	`claude -p --json-schema '{"type":"object","properties":{...}}' "query"`
`--maintenance`	Run maintenance hooks and exit	`claude --maintenance`
`--max-budget-usd`	Maximum dollar amount to spend on API calls before stopping (print mode only)	`claude -p --max-budget-usd 5.00 "query"`
`--max-turns`	Limit the number of agentic turns (print mode only). Exits with an error when the limit is reached. No limit by default	`claude -p --max-turns 3 "query"`
`--mcp-config`	Load MCP servers from JSON files or strings (space-separated)	`claude --mcp-config ./mcp.json`
`--model`	Sets the model for the current session with an alias for the latest model (`sonnet` or `opus`) or a model's full name	`claude --model claude-sonnet-4-6`
`--name` , `-n`	Set a display name for the session, shown in `/resume` and the terminal title. You can resume a named session with `claude --resume <name>` . `/rename` changes the name mid-session and also shows it on the prompt bar	`claude -n "my-feature-work"`
`--no-chrome`	Disable Chrome browser integration for this session	`claude --no-chrome`
`--no-session-persistence`	Disable session persistence so sessions are not saved to disk and cannot be resumed (print mode only)	`claude -p --no-session-persistence "query"`
`--output-format`	Specify output format for print mode (options: `text` , `json` , `stream-json`)	`claude -p "query" --output-format json`

Flag	Description	Example
`--permission-mode`	Begin in a specified permission mode	`claude --permission-mode plan`
`--permission-prompt-tool`	Specify an MCP tool to handle permission prompts in non-interactive mode	`claude -p --permission-prompt-tool mcp_auth_tool "query"`
`--plugin-dir`	Load plugins from a directory for this session only. Each flag takes one path. Repeat the flag for multiple directories: `--plugin-dir A --plugin-dir B`	`claude --plugin-dir ./my-plugins`
`--print`, `-p`	Print response without interactive mode (see Agent SDK documentation for programmatic usage details)	`claude -p "query"`
`--remote`	Create a new web session on claude.ai with the provided task description	`claude --remote "Fix the login bug"`
`--remote-control`, `--rc`	Start an interactive session with Remote Control enabled so you can also control it from claude.ai or the Claude app. Optionally pass a name for the session	`claude --remote-control "My Project"`
`--resume`, `-r`	Resume a specific session by ID or name, or show an interactive picker to choose a session	`claude --resume auth-refactor`
`--session-id`	Use a specific session ID for the conversation (must be a valid UUID)	`claude --session-id "550e8400-e29b-41d4-a716-446655440000"`
`--setting-sources`	Comma-separated list of setting sources to load (`user`, `project`, `local`)	`claude --setting-sources user,project`
`--settings`	Path to a settings JSON file or a JSON string to load additional settings from	`claude --settings ./settings.json`

Flag	Description	Example
`--strict-mcp-config`	Only use MCP servers from `--mcp-config`, ignoring all other MCP configurations	`claude --strict-mcp-config --mcp-config ./mcp.json`
`--system-prompt`	Replace the entire system prompt with custom text	`claude --system-prompt "You are a Python expert"`
`--system-prompt-file`	Load system prompt from a file, replacing the default prompt	`claude --system-prompt-file ./custom-prompt.txt`
`--teleport`	Resume a web session in your local terminal	`claude --teleport`
`--teammate-mode`	Set how agent team teammates display: `auto` (default), `in-process`, or `tmux`. See set up agent teams	`claude --teammate-mode in-process`
`--tools`	Restrict which built-in tools Claude can use. Use `""` to disable all, `"default"` for all, or tool names like `"Bash,Edit,Read"`	`claude --tools "Bash,Edit,Read"`
`--verbose`	Enable verbose logging, shows full turn-by-turn output	`claude --verbose`
`--version`, `-v`	Output the version number	`claude -v`
`--worktree`, `-w`	Start Claude in an isolated git worktree at `<repo>/.claude/worktrees/<name>`. If no name is given, one is auto-generated	`claude -w feature-auth`

System prompt flags

Claude Code provides four flags for customizing the system prompt. All four work in both interactive and non-interactive modes.

Flag	Behavior	Example
`--system-prompt`	Replaces the entire default prompt	`claude --system-prompt "You are a Python expert"`

Flag	Behavior	Example
`--system-prompt-file`	Replaces with file contents	`claude --system-prompt-file ./prompts/review.txt`
`--append-system-prompt`	Appends to the default prompt	`claude --append-system-prompt "Always use TypeScript"`
`--append-system-prompt-file`	Appends file contents to the default prompt	`claude --append-system-prompt-file ./style-rules.txt`

`--system-prompt` and `--system-prompt-file` are mutually exclusive. The append flags can be combined with either replacement flag.

For most use cases, use an append flag. Appending preserves Claude Code's built-in capabilities while adding your requirements. Use a replacement flag only when you need complete control over the system prompt.

See also

- Chrome extension - Browser automation and web testing
- Interactive mode - Shortcuts, input modes, and interactive features
- Quickstart guide - Getting started with Claude Code
- Common workflows - Advanced workflows and patterns
- Settings - Configuration options
- Agent SDK documentation - Programmatic usage and integrations

Tools reference

> *Complete reference for the tools Claude Code can use, including permission requirements.*

Claude Code has access to a set of tools that help it understand and modify your codebase. The tool names below are the exact strings you use in permission rules, subagent tool lists, and hook matchers.

Tool	Description	Permission Required
`Agent`	Spawns a subagent with its own context window to handle a task	No
`AskUserQuestion`	Asks multiple-choice questions to gather requirements or clarify ambiguity	No
`Bash`	Executes shell commands in your environment. See Bash tool behavior	Yes
`CronCreate`	Schedules a recurring or one-shot prompt within the current session (gone when Claude exits). See scheduled tasks	No
`CronDelete`	Cancels a scheduled task by ID	No
`CronList`	Lists all scheduled tasks in the session	No
`Edit`	Makes targeted edits to specific files	Yes
`EnterPlanMode`	Switches to plan mode to design an approach before coding	No
`EnterWorktree`	Creates an isolated git worktree and switches into it	No
`ExitPlanMode`	Presents a plan for approval and exits plan mode	Yes

Tool	Description	Permission Required
`ExitWorktree`	Exits a worktree session and returns to the original directory	No
`Glob`	Finds files based on pattern matching	No
`Grep`	Searches for patterns in file contents	No
`ListMcpResourcesTool`	Lists resources exposed by connected MCP servers	No
`LSP`	Code intelligence via language servers. Reports type errors and warnings automatically after file edits. Also supports navigation operations: jump to definitions, find references, get type info, list symbols, find implementations, trace call hierarchies. Requires a code intelligence plugin and its language server binary	No
`NotebookEdit`	Modifies Jupyter notebook cells	Yes
`Read`	Reads the contents of files	No
`ReadMcpResourceTool`	Reads a specific MCP resource by URI	No
`Skill`	Executes a skill within the main conversation	Yes
`TaskCreate`	Creates a new task in the task list	No
`TaskGet`	Retrieves full details for a specific task	No
`TaskList`	Lists all tasks with their current status	No
`TaskOutput`	Retrieves output from a background task	No
`TaskStop`	Kills a running background task by ID	No

Tool	Description	Permission Required
`TaskUpdate`	Updates task status, dependencies, details, or deletes tasks	No
`TodoWrite`	Manages the session task checklist. Available in non-interactive mode and the Agent SDK; interactive sessions use TaskCreate, TaskGet, TaskList, and TaskUpdate instead	No
`ToolSearch`	Searches for and loads deferred tools when tool search is enabled	No
`WebFetch`	Fetches content from a specified URL	Yes
`WebSearch`	Performs web searches	Yes
`Write`	Creates or overwrites files	Yes

Permission rules can be configured using `/permissions` or in permission settings. Also see Tool-specific permission rules.

Bash tool behavior

The Bash tool runs each command in a separate process with the following persistence behavior:

- Working directory persists across commands. Set `CLAUDE_BASH_MAINTAIN_PROJECT_WORKING_DIR=1` to reset to the project directory after each command.
- Environment variables do not persist. An `export` in one command will not be available in the next.

Activate your virtualenv or conda environment before launching Claude Code. To make environment variables persist across Bash commands, set `CLAUDE_ENV_FILE` to a shell script before launching Claude Code, or use a SessionStart hook to populate it dynamically.

See also

- Permissions: permission system, rule syntax, and tool-specific patterns
- Subagents: configure tool access for subagents

- Hooks: run custom commands before or after tool execution

Customize keyboard shortcuts

> *Customize keyboard shortcuts in Claude Code with a keybindings configuration file.*

> **Note:**
>
> Customizable keyboard shortcuts require Claude Code v2.1.18 or later. Check your version with `claude --version`.

Claude Code supports customizable keyboard shortcuts. Run `/keybindings` to create or open your configuration file at `~/.claude/keybindings.json`.

Configuration file

The keybindings configuration file is an object with a `bindings` array. Each block specifies a context and a map of keystrokes to actions.

> **Note:**
>
> Changes to the keybindings file are automatically detected and applied without restarting Claude Code.

Field	Description
`$schema`	Optional JSON Schema URL for editor autocompletion
`$docs`	Optional documentation URL
`bindings`	Array of binding blocks by context

This example binds `Ctrl+E` to open an external editor in the chat context, and unbinds `Ctrl+U`:

```
{
  "$schema": "https://www.schemastore.org/claude-code-keybindings.json",
  "$docs": "https://code.claude.com/docs/en/keybindings",
  "bindings": [
    {
      "context": "Chat",
      "bindings": {
        "ctrl+e": "chat:externalEditor",
        "ctrl+u": null
      }
    }
  ]
}
```

Contexts

Each binding block specifies a **context** where the bindings apply:

Context	Description
`Global`	Applies everywhere in the app
`Chat`	Main chat input area
`Autocomplete`	Autocomplete menu is open
`Settings`	Settings menu (escape-only dismiss)
`Confirmation`	Permission and confirmation dialogs
`Tabs`	Tab navigation components
`Help`	Help menu is visible
`Transcript`	Transcript viewer
`HistorySearch`	History search mode (Ctrl+R)
`Task`	Background task is running
`ThemePicker`	Theme picker dialog
`Attachments`	Image/attachment bar navigation

Context	Description
`Footer`	Footer indicator navigation (tasks, teams, diff)
`MessageSelector`	Rewind and summarize dialog message selection
`DiffDialog`	Diff viewer navigation
`ModelPicker`	Model picker effort level
`Select`	Generic select/list components
`Plugin`	Plugin dialog (browse, discover, manage)

Available actions

Actions follow a `namespace:action` format, such as `chat:submit` to send a message or `app:toggleTodos` to show the task list. Each context has specific actions available.

App actions

Actions available in the `Global` context:

Action	Default	Description
`app:interrupt`	Ctrl+C	Cancel current operation
`app:exit`	Ctrl+D	Exit Claude Code
`app:toggleTodos`	Ctrl+T	Toggle task list visibility
`app:toggleTranscript`	Ctrl+O	Toggle verbose transcript

History actions

Actions for navigating command history:

Action	Default	Description
`history:search`	Ctrl+R	Open history search
`history:previous`	Up	Previous history item
`history:next`	Down	Next history item

Chat actions

Actions available in the `Chat` context:

Action	Default	Description
`chat:cancel`	Escape	Cancel current input
`chat:cycleMode`	Shift+Tab*	Cycle permission modes
`chat:modelPicker`	Cmd+P / Meta+P	Open model picker
`chat:thinkingToggle`	Cmd+T / Meta+T	Toggle extended thinking
`chat:submit`	Enter	Submit message
`chat:undo`	Ctrl+_	Undo last action
`chat:externalEditor`	Ctrl+G	Open in external editor
`chat:stash`	Ctrl+S	Stash current prompt
`chat:imagePaste`	Ctrl+V (Alt+V on Windows)	Paste image

*On Windows without VT mode (Node <24.2.0/<22.17.0, Bun <1.2.23), defaults to Meta+M.

Autocomplete actions

Actions available in the `Autocomplete` context:

Action	Default	Description
`autocomplete:accept`	Tab	Accept suggestion
`autocomplete:dismiss`	Escape	Dismiss menu
`autocomplete:previous`	Up	Previous suggestion
`autocomplete:next`	Down	Next suggestion

Confirmation actions

Actions available in the `Confirmation` context:

Action	Default	Description
`confirm:yes`	Y, Enter	Confirm action

Action	Default	Description
`confirm:no`	N, Escape	Decline action
`confirm:previous`	Up	Previous option
`confirm:next`	Down	Next option
`confirm:nextField`	Tab	Next field
`confirm:previousField`	(unbound)	Previous field
`confirm:cycleMode`	Shift+Tab	Cycle permission modes
`confirm:toggleExplanation`	Ctrl+E	Toggle permission explanation

Permission actions

Actions available in the `Confirmation` context for permission dialogs:

Action	Default	Description
`permission:toggleDebug`	Ctrl+D	Toggle permission debug info

Transcript actions

Actions available in the `Transcript` context:

Action	Default	Description
`transcript:toggleShowAll`	Ctrl+E	Toggle show all content
`transcript:exit`	Ctrl+C, Escape	Exit transcript view

History search actions

Actions available in the `HistorySearch` context:

Action	Default	Description
`historySearch:next`	Ctrl+R	Next match
`historySearch:accept`	Escape, Tab	Accept selection

Action	Default	Description
`historySearch:cancel`	Ctrl+C	Cancel search
`historySearch:execute`	Enter	Execute selected command

Task actions

Actions available in the `Task` context:

Action	Default	Description
`task:background`	Ctrl+B	Background current task

Theme actions

Actions available in the `ThemePicker` context:

Action	Default	Description
`theme:toggleSyntaxHighlighting`	Ctrl+T	Toggle syntax highlighting

Help actions

Actions available in the `Help` context:

Action	Default	Description
`help:dismiss`	Escape	Close help menu

Tabs actions

Actions available in the `Tabs` context:

Action	Default	Description
`tabs:next`	Tab, Right	Next tab
`tabs:previous`	Shift+Tab, Left	Previous tab

Attachments actions

Actions available in the `Attachments` context:

Action	Default	Description
`attachments:next`	Right	Next attachment
`attachments:previous`	Left	Previous attachment
`attachments:remove`	Backspace, Delete	Remove selected attachment
`attachments:exit`	Down, Escape	Exit attachment bar

Footer actions

Actions available in the `Footer` context:

Action	Default	Description
`footer:next`	Right	Next footer item
`footer:previous`	Left	Previous footer item
`footer:openSelected`	Enter	Open selected footer item
`footer:clearSelection`	Escape	Clear footer selection

Message selector actions

Actions available in the `MessageSelector` context:

Action	Default	Description
`messageSelector:up`	Up, K, Ctrl+P	Move up in list
`messageSelector:down`	Down, J, Ctrl+N	Move down in list
`messageSelector:top`	Ctrl+Up, Shift+Up, Meta+Up, Shift+K	Jump to top
`messageSelector:bottom`	Ctrl+Down, Shift+Down, Meta+Down, Shift+J	Jump to bottom
`messageSelector:select`	Enter	Select message

Diff actions

Actions available in the `DiffDialog` context:

Action	Default	Description
`diff:dismiss`	Escape	Close diff viewer
`diff:previousSource`	Left	Previous diff source
`diff:nextSource`	Right	Next diff source
`diff:previousFile`	Up	Previous file in diff
`diff:nextFile`	Down	Next file in diff
`diff:viewDetails`	Enter	View diff details
`diff:back`	(context-specific)	Go back in diff viewer

Model picker actions

Actions available in the `ModelPicker` context:

Action	Default	Description
`modelPicker:decreaseEffort`	Left	Decrease effort level
`modelPicker:increaseEffort`	Right	Increase effort level

Select actions

Actions available in the `Select` context:

Action	Default	Description
`select:next`	Down, J, Ctrl+N	Next option
`select:previous`	Up, K, Ctrl+P	Previous option
`select:accept`	Enter	Accept selection
`select:cancel`	Escape	Cancel selection

Plugin actions

Actions available in the `Plugin` context:

Action	Default	Description
`plugin:toggle`	Space	Toggle plugin selection

Action	Default	Description
`plugin:install`	I	Install selected plugins

Settings actions

Actions available in the `Settings` context:

Action	Default	Description
`settings:search`	/	Enter search mode
`settings:retry`	R	Retry loading usage data (on error)

Keystroke syntax

Modifiers

Use modifier keys with the `+` separator:

- `ctrl` or `control` - Control key
- `alt`, `opt`, or `option` - Alt/Option key
- `shift` - Shift key
- `meta`, `cmd`, or `command` - Meta/Command key

For example:

```
ctrl+k          Single key with modifier
shift+tab       Shift + Tab
meta+p          Command/Meta + P
ctrl+shift+c    Multiple modifiers
```

Uppercase letters

A standalone uppercase letter implies Shift. For example, `K` is equivalent to `shift+k`. This is useful for vim-style bindings where uppercase and lowercase keys have different meanings.

Uppercase letters with modifiers (e.g., `ctrl+K`) are treated as stylistic and do **not** imply Shift — `ctrl+K` is the same as `ctrl+k`.

Chords

Chords are sequences of keystrokes separated by spaces:

```
ctrl+k ctrl+s   Press Ctrl+K, release, then Ctrl+S
```

Special keys

- `escape` or `esc` - Escape key
- `enter` or `return` - Enter key
- `tab` - Tab key
- `space` - Space bar
- `up`, `down`, `left`, `right` - Arrow keys
- `backspace`, `delete` - Delete keys

Unbind default shortcuts

Set an action to `null` to unbind a default shortcut:

```
{
  "bindings": [
    {
      "context": "Chat",
      "bindings": {
        "ctrl+s": null
      }
    }
  ]
}
```

Reserved shortcuts

These shortcuts cannot be rebound:

Shortcut	Reason
Ctrl+C	Hardcoded interrupt/cancel
Ctrl+D	Hardcoded exit

Terminal conflicts

Some shortcuts may conflict with terminal multiplexers:

Shortcut	Conflict
Ctrl+B	tmux prefix (press twice to send)
Ctrl+A	GNU screen prefix
Ctrl+Z	Unix process suspend (SIGTSTP)

Vim mode interaction

When vim mode is enabled (`/vim`), keybindings and vim mode operate independently:

- **Vim mode** handles input at the text input level (cursor movement, modes, motions)
- **Keybindings** handle actions at the component level (toggle todos, submit, etc.)
- The Escape key in vim mode switches INSERT to NORMAL mode; it does not trigger `chat:cancel`
- Most Ctrl+key shortcuts pass through vim mode to the keybinding system
- In vim NORMAL mode, `?` shows the help menu (vim behavior)

Validation

Claude Code validates your keybindings and shows warnings for:

- Parse errors (invalid JSON or structure)
- Invalid context names
- Reserved shortcut conflicts
- Terminal multiplexer conflicts
- Duplicate bindings in the same context

Run `/doctor` to see any keybinding warnings.

Part 4: Configuration

Configure permissions

> *Control what Claude Code can access and do with fine-grained permission rules, modes, and managed policies.*

Claude Code supports fine-grained permissions so that you can specify exactly what the agent is allowed to do and what it cannot. Permission settings can be checked into version control and distributed to all developers in your organization, as well as customized by individual developers.

Permission system

Claude Code uses a tiered permission system to balance power and safety:

Tool type	Example	Approval required	"Yes, don't ask again" behavior
Read-only	File reads, Grep	No	N/A
Bash commands	Shell execution	Yes	Permanently per project directory and command
File modification	Edit/write files	Yes	Until session end

Manage permissions

You can view and manage Claude Code's tool permissions with `/permissions`. This UI lists all permission rules and the settings.json file they are sourced from.

- **Allow** rules let Claude Code use the specified tool without manual approval.
- **Ask** rules prompt for confirmation whenever Claude Code tries to use the specified tool.
- **Deny** rules prevent Claude Code from using the specified tool.

Rules are evaluated in order: **deny -> ask -> allow**. The first matching rule wins, so deny rules always take precedence.

Permission modes

Claude Code supports several permission modes that control how tools are approved. Set the `defaultMode` in your settings files:

Mode	Description
`default`	Standard behavior: prompts for permission on first use of each tool
`acceptEdits`	Automatically accepts file edit permissions for the session
`plan`	Plan Mode: Claude can analyze but not modify files or execute commands
`dontAsk`	Auto-denies tools unless pre-approved via `/permissions` or `permissions.allow` rules
`bypassPermissions`	Skips all permission prompts (requires safe environment, see warning below)

> **Warning:**
>
> `bypassPermissions` mode disables all permission checks. Only use this in isolated environments like containers or VMs where Claude Code cannot cause damage. Administrators can prevent this mode by setting `disableBypassPermissionsMode` to `"disable"` in managed settings.

Permission rule syntax

Permission rules follow the format `Tool` or `Tool(specifier)`.

Match all uses of a tool

To match all uses of a tool, use just the tool name without parentheses:

Rule	Effect
`Bash`	Matches all Bash commands
`WebFetch`	Matches all web fetch requests
`Read`	Matches all file reads

`Bash(*)` is equivalent to `Bash` and matches all Bash commands.

Use specifiers for fine-grained control

Add a specifier in parentheses to match specific tool uses:

Rule	Effect
`Bash(npm run build)`	Matches the exact command `npm run build`
`Read(./.env)`	Matches reading the `.env` file in the current directory
`WebFetch(domain:example.com)`	Matches fetch requests to example.com

Wildcard patterns

Bash rules support glob patterns with `*`. Wildcards can appear at any position in the command. This configuration allows npm and git commit commands while blocking git push:

```
{
  "permissions": {
    "allow": [
      "Bash(npm run *)",
      "Bash(git commit *)",
      "Bash(git * main)",
      "Bash(* --version)",
      "Bash(* --help *)"
    ],
    "deny": [
      "Bash(git push *)"
    ]
  }
}
```

The space before `*` matters: `Bash(ls *)` matches `ls -la` but not `lsof`, while `Bash(ls*)` matches both. The legacy `:*` suffix syntax is equivalent to `*` but is deprecated.

Tool-specific permission rules

Bash

Bash permission rules support wildcard matching with `*`. Wildcards can appear at any position in the command, including at the beginning, middle, or end:

- `Bash(npm run build)` matches the exact Bash command `npm run build`
- `Bash(npm run test *)` matches Bash commands starting with `npm run test`
- `Bash(npm *)` matches any command starting with `npm`
- `Bash(* install)` matches any command ending with `install`
- `Bash(git * main)` matches commands like `git checkout main`, `git merge main`

When `*` appears at the end with a space before it (like `Bash(ls *)`), it enforces a word boundary, requiring the prefix to be followed by a space or end-of-string. For example, `Bash(ls *)` matches `ls -la` but not `lsof`. In contrast, `Bash(ls*)` without a space matches both `ls -la` and `lsof` because there's no word boundary constraint.

Tip:

Claude Code is aware of shell operators (like `&&`) so a prefix match rule like `Bash(safe-cmd *)` won't give it permission to run the command `safe-cmd && other-cmd`.

Warning:

Bash permission patterns that try to constrain command arguments are fragile. For example, `Bash(curl http://github.com/ *)` intends to restrict curl to GitHub URLs, but won't match variations like:

- Options before URL: `curl -X GET http://github.com/...`
- Different protocol: `curl https://github.com/...`
- Redirects: `curl -L http://bit.ly/xyz` (redirects to github)
- Variables: `URL=http://github.com && curl $URL`
- Extra spaces: `curl http://github.com`

For more reliable URL filtering, consider:

- **Restrict Bash network tools**: use deny rules to block `curl`, `wget`, and similar commands, then use the WebFetch tool with `WebFetch(domain:github.com)` permission for allowed domains
- **Use PreToolUse hooks**: implement a hook that validates URLs in Bash commands and blocks disallowed domains
- Instructing Claude Code about your allowed curl patterns via CLAUDE.md

> Note that using WebFetch alone does not prevent network access. If Bash is allowed, Claude can still use `curl`, `wget`, or other tools to reach any URL.

Read and Edit

`Edit` rules apply to all built-in tools that edit files. Claude makes a best-effort attempt to apply `Read` rules to all built-in tools that read files like Grep and Glob.

Read and Edit rules both follow the gitignore specification with four distinct pattern types:

Pattern	Meaning	Example	Matches
`//path`	**Absolute** path from filesystem root	`Read(//Users/alice/secrets/**)`	`/Users/alice/secrets/**`
`~/path`	Path from **home** directory	`Read(~/Documents/*.pdf)`	`/Users/alice/Documents/*.pdf`
`/path`	Path **relative to project root**	`Edit(/src/**/*.ts)`	`<project root>/src/**/*.ts`
`path` or `./path`	Path **relative to current directory**	`Read(*.env)`	`<cwd>/*.env`

> **Warning:**
>
> A pattern like `/Users/alice/file` is NOT an absolute path. It's relative to the project root. Use `//Users/alice/file` for absolute paths.

Examples:

- `Edit(/docs/**)`: edits in `<project>/docs/` (NOT `/docs/` and NOT `<project>/.claude/docs/`)
- `Read(~/.zshrc)`: reads your home directory's `.zshrc`
- `Edit(//tmp/scratch.txt)`: edits the absolute path `/tmp/scratch.txt`
- `Read(src/**)`: reads from `<current-directory>/src/`

> **Note:**

> In gitignore patterns, `*` matches files in a single directory while `**` matches recursively across directories. To allow all file access, use just the tool name without parentheses: `Read`, `Edit`, or `Write`.

WebFetch

- `WebFetch(domain:example.com)` matches fetch requests to example.com

MCP

- `mcp__puppeteer` matches any tool provided by the `puppeteer` server (name configured in Claude Code)
- `mcp__puppeteer__*` wildcard syntax that also matches all tools from the `puppeteer` server
- `mcp__puppeteer__puppeteer_navigate` matches the `puppeteer_navigate` tool provided by the `puppeteer` server

Agent (subagents)

Use `Agent(AgentName)` rules to control which subagents Claude can use:

- `Agent(Explore)` matches the Explore subagent
- `Agent(Plan)` matches the Plan subagent
- `Agent(my-custom-agent)` matches a custom subagent named `my-custom-agent`

Add these rules to the `deny` array in your settings or use the `--disallowedTools` CLI flag to disable specific agents. To disable the Explore agent:

```
{
  "permissions": {
    "deny": ["Agent(Explore)"]
  }
}
```

Extend permissions with hooks

Claude Code hooks provide a way to register custom shell commands to perform permission evaluation at runtime. When Claude Code makes a tool call, PreToolUse hooks run before the permission system, and the hook output can determine whether to approve or deny the tool call in place of the permission system.

Working directories

By default, Claude has access to files in the directory where it was launched. You can extend this access:

- **During startup**: use `--add-dir <path>` CLI argument
- **During session**: use `/add-dir` command
- **Persistent configuration**: add to `additionalDirectories` in settings files

Files in additional directories follow the same permission rules as the original working directory: they become readable without prompts, and file editing permissions follow the current permission mode.

How permissions interact with sandboxing

Permissions and sandboxing are complementary security layers:

- **Permissions** control which tools Claude Code can use and which files or domains it can access. They apply to all tools (Bash, Read, Edit, WebFetch, MCP, and others).
- **Sandboxing** provides OS-level enforcement that restricts the Bash tool's filesystem and network access. It applies only to Bash commands and their child processes.

Use both for defense-in-depth:

- Permission deny rules block Claude from even attempting to access restricted resources
- Sandbox restrictions prevent Bash commands from reaching resources outside defined boundaries, even if a prompt injection bypasses Claude's decision-making
- Filesystem restrictions in the sandbox use Read and Edit deny rules, not separate sandbox configuration
- Network restrictions combine WebFetch permission rules with the sandbox's `allowedDomains` list

Managed settings

For organizations that need centralized control over Claude Code configuration, administrators can deploy managed settings that cannot be overridden by user or project settings. These policy settings follow the same format as regular settings files and can be delivered through MDM/OS-level policies, managed settings files, or server-managed settings. See settings files for delivery mechanisms and file locations.

Managed-only settings

Some settings are only effective in managed settings:

Setting	Description
`disableBypassPermissionsMode`	Set to `"disable"` to prevent `bypassPermissions` mode and the `--dangerously-skip-permissions` flag
`allowManagedPermissionRulesOnly`	When `true`, prevents user and project settings from defining `allow`, `ask`, or `deny` permission rules. Only rules in managed settings apply
`allowManagedHooksOnly`	When `true`, prevents loading of user, project, and plugin hooks. Only managed hooks and SDK hooks are allowed
`allowManagedMcpServersOnly`	When `true`, only `allowedMcpServers` from managed settings are respected. `deniedMcpServers` still merges from all sources. See Managed MCP configuration
`blockedMarketplaces`	Blocklist of marketplace sources. Blocked sources are checked before downloading, so they never touch the filesystem. See managed marketplace restrictions
`sandbox.network.allowManagedDomainsOnly`	When `true`, only `allowedDomains` and `WebFetch(domain:...)` allow rules from managed settings are respected. Non-allowed domains are blocked automatically without prompting the user. Denied domains still merge from all sources
`strictKnownMarketplaces`	Controls which plugin marketplaces users can add. See managed marketplace restrictions
`allow_remote_sessions`	When `true`, allows users to start Remote Control and web sessions. Defaults to `true`. Set to `false` to prevent remote session access

Settings precedence

Permission rules follow the same settings precedence as all other Claude Code settings:

1. **Managed settings**: cannot be overridden by any other level, including command line arguments
2. **Command line arguments**: temporary session overrides
3. **Local project settings** (`.claude/settings.local.json`)
4. **Shared project settings** (`.claude/settings.json`)
5. **User settings** (`~/.claude/settings.json`)

If a tool is denied at any level, no other level can allow it. For example, a managed settings deny cannot be overridden by `--allowedTools` , and `--disallowedTools` can add restrictions beyond what managed settings define.

If a permission is allowed in user settings but denied in project settings, the project setting takes precedence and the permission is blocked.

Example configurations

This repository includes starter settings configurations for common deployment scenarios. Use these as starting points and adjust them to fit your needs.

See also

- Settings: complete configuration reference including the permission settings table
- Sandboxing: OS-level filesystem and network isolation for Bash commands
- Authentication: set up user access to Claude Code
- Security: security safeguards and best practices
- Hooks: automate workflows and extend permission evaluation

How Claude remembers your project

> *Give Claude persistent instructions with CLAUDE.md files, and let Claude accumulate learnings automatically with auto memory.*

Each Claude Code session begins with a fresh context window. Two mechanisms carry knowledge across sessions:

- **CLAUDE.md files**: instructions you write to give Claude persistent context
- **Auto memory**: notes Claude writes itself based on your corrections and preferences

This page covers how to:

- Write and organize CLAUDE.md files
- Scope rules to specific file types with `.claude/rules/`
- Configure auto memory so Claude takes notes automatically
- Troubleshoot when instructions aren't being followed

CLAUDE.md vs auto memory

Claude Code has two complementary memory systems. Both are loaded at the start of every conversation. Claude treats them as context, not enforced configuration. The more specific and concise your instructions, the more consistently Claude follows them.

	CLAUDE.md files	Auto memory
Who writes it	You	Claude
What it contains	Instructions and rules	Learnings and patterns
Scope	Project, user, or org	Per working tree
Loaded into	Every session	Every session (first 200 lines)
Use for	Coding standards, workflows, project architecture	Build commands, debugging insights, preferences Claude discovers

Use CLAUDE.md files when you want to guide Claude's behavior. Auto memory lets Claude learn from your corrections without manual effort.

Subagents can also maintain their own auto memory. See subagent configuration for details.

CLAUDE.md files

CLAUDE.md files are markdown files that give Claude persistent instructions for a project, your personal workflow, or your entire organization. You write these files in plain text; Claude reads them at the start of every session.

Choose where to put CLAUDE.md files

CLAUDE.md files can live in several locations, each with a different scope. More specific locations take precedence over broader ones.

Scope	Location	Purpose	Use case examples	Shared with
Managed policy	• macOS: `/Library/Application Support/ClaudeCode/CLAUDE.md` • Linux and WSL: `/etc/claude-code/CLAUDE.md` • Windows: `C:\Program Files\ClaudeCode\CLAUDE.md`	Organization-wide instructions managed by IT/DevOps	Company coding standards, security policies, compliance requirements	All users in organization
Project instructions	`./CLAUDE.md` or `./.claude/CLAUDE.md`	Team-shared instructions for the project	Project architecture, coding standards, common workflows	Team members via source control
User instructions	`~/.claude/CLAUDE.md`	Personal preferences for all projects	Code styling preferences, personal tooling shortcuts	Just you (all projects)

CLAUDE.md files in the directory hierarchy above the working directory are loaded in full at launch. CLAUDE.md files in subdirectories load on demand when Claude reads files in those directories. See How CLAUDE.md files load for the full resolution order.

For large projects, you can break instructions into topic-specific files using project rules. Rules let you scope instructions to specific file types or subdirectories.

Set up a project CLAUDE.md

A project CLAUDE.md can be stored in either `./CLAUDE.md` or `./.claude/CLAUDE.md`. Create this file and add instructions that apply to anyone working on the project: build and test commands, coding standards, architectural decisions, naming conventions, and common workflows. These instructions are shared with your team through version control, so focus on project-level standards rather than personal preferences.

> **Tip:**
>
> Run `/init` to generate a starting CLAUDE.md automatically. Claude analyzes your codebase and creates a file with build commands, test instructions, and project conventions it discovers. If a CLAUDE.md already exists, `/init` suggests improvements rather than overwriting it. Refine from there with instructions Claude wouldn't discover on its own.

Write effective instructions

CLAUDE.md files are loaded into the context window at the start of every session, consuming tokens alongside your conversation. Because they're context rather than enforced configuration, how you write instructions affects how reliably Claude follows them. Specific, concise, well-structured instructions work best.

Size: target under 200 lines per CLAUDE.md file. Longer files consume more context and reduce adherence. If your instructions are growing large, split them using imports or `.claude/rules/` files.

Structure: use markdown headers and bullets to group related instructions. Claude scans structure the same way readers do: organized sections are easier to follow than dense paragraphs.

Specificity: write instructions that are concrete enough to verify. For example:

- "Use 2-space indentation" instead of "Format code properly"
- "Run `npm test` before committing" instead of "Test your changes"
- "API handlers live in `src/api/handlers/`" instead of "Keep files organized"

Consistency: if two rules contradict each other, Claude may pick one arbitrarily. Review your CLAUDE.md files, nested CLAUDE.md files in subdirectories, and `.claude/rules/` periodically to remove outdated or conflicting instructions. In monorepos, use `claudeMdExcludes` to skip CLAUDE.md files from other teams that aren't relevant to your work.

Import additional files

CLAUDE.md files can import additional files using `@path/to/import` syntax. Imported files are expanded and loaded into context at launch alongside the CLAUDE.md that references them.

Both relative and absolute paths are allowed. Relative paths resolve relative to the file containing the import, not the working directory. Imported files can recursively import other files, with a maximum depth of five hops.

To pull in a README, package.json, and a workflow guide, reference them with `@` syntax anywhere in your CLAUDE.md:

```
See @README for project overview and @package.json for available npm commands for
this project.

## Additional Instructions
- git workflow @docs/git-instructions.md
```

For personal preferences you don't want to check in, import a file from your home directory. The import goes in the shared CLAUDE.md, but the file it points to stays on your machine:

```
## Individual Preferences
- @~/.claude/my-project-instructions.md
```

> **Warning:**
>
> The first time Claude Code encounters external imports in a project, it shows an approval dialog listing the files. If you decline, the imports stay disabled and the dialog does not appear again.

For a more structured approach to organizing instructions, see `.claude/rules/`.

How CLAUDE.md files load

Claude Code reads CLAUDE.md files by walking up the directory tree from your current working directory, checking each directory along the way. This means if you run Claude Code in `foo/bar/`, it loads instructions from both `foo/bar/CLAUDE.md` and `foo/CLAUDE.md`.

Claude also discovers CLAUDE.md files in subdirectories under your current working directory. Instead of loading them at launch, they are included when Claude reads files in those subdirectories.

If you work in a large monorepo where other teams' CLAUDE.md files get picked up, use `claudeMdExcludes` to skip them.

Load from additional directories

The `--add-dir` flag gives Claude access to additional directories outside your main working directory. By default, CLAUDE.md files from these directories are not loaded.

To also load CLAUDE.md files from additional directories, including `CLAUDE.md`, `.claude/CLAUDE.md`, and `.claude/rules/*.md`, set the `CLAUDE_CODE_ADDITIONAL_DIRECTORIES_CLAUDE_MD` environment variable:

```
CLAUDE_CODE_ADDITIONAL_DIRECTORIES_CLAUDE_MD=1 claude --add-dir ../shared-config
```

Organize rules with `.claude/rules/`

For larger projects, you can organize instructions into multiple files using the `.claude/rules/` directory. This keeps instructions modular and easier for teams to maintain. Rules can also be scoped to specific file paths, so they only load into context when Claude works with matching files, reducing noise and saving context space.

> **Note:**
>
> Rules load into context every session or when matching files are opened. For task-specific instructions that don't need to be in context all the time, use skills instead, which only load when you invoke them or when Claude determines they're relevant to your prompt.

Set up rules

Place markdown files in your project's `.claude/rules/` directory. Each file should cover one topic, with a descriptive filename like `testing.md` or `api-design.md`. All `.md` files are discovered recursively, so you can organize rules into subdirectories like `frontend/` or `backend/`:

```
your-project/
├── .claude/
│   ├── CLAUDE.md           # Main project instructions
│   └── rules/
│       ├── code-style.md   # Code style guidelines
│       ├── testing.md      # Testing conventions
│       └── security.md     # Security requirements
```

Rules without `paths` frontmatter are loaded at launch with the same priority as `.claude/CLAUDE.md`.

Path-specific rules

Rules can be scoped to specific files using YAML frontmatter with the `paths` field. These conditional rules only apply when Claude is working with files matching the specified patterns.

```
---
paths:
  - "src/api/**/*.ts"
---

## API Development Rules

- All API endpoints must include input validation
- Use the standard error response format
- Include OpenAPI documentation comments
```

Rules without a `paths` field are loaded unconditionally and apply to all files. Path-scoped rules trigger when Claude reads files matching the pattern, not on every tool use.

Use glob patterns in the `paths` field to match files by extension, directory, or any combination:

Pattern	Matches
`**/*.ts`	All TypeScript files in any directory
`src/**/*`	All files under `src/` directory

Pattern	Matches
`*.md`	Markdown files in the project root
`src/components/*.tsx`	React components in a specific directory

You can specify multiple patterns and use brace expansion to match multiple extensions in one pattern:

```
---
paths:
  - "src/**/*.{ts,tsx}"
  - "lib/**/*.ts"
  - "tests/**/*.test.ts"
---
```

Share rules across projects with symlinks

The `.claude/rules/` directory supports symlinks, so you can maintain a shared set of rules and link them into multiple projects. Symlinks are resolved and loaded normally, and circular symlinks are detected and handled gracefully.

This example links both a shared directory and an individual file:

```
ln -s ~/shared-claude-rules .claude/rules/shared
ln -s ~/company-standards/security.md .claude/rules/security.md
```

User-level rules

Personal rules in `~/.claude/rules/` apply to every project on your machine. Use them for preferences that aren't project-specific:

```
~/.claude/rules/
├── preferences.md    # Your personal coding preferences
└── workflows.md      # Your preferred workflows
```

User-level rules are loaded before project rules, giving project rules higher priority.

Manage CLAUDE.md for large teams

For organizations deploying Claude Code across teams, you can centralize instructions and control which CLAUDE.md files are loaded.

Deploy organization-wide CLAUDE.md

Organizations can deploy a centrally managed CLAUDE.md that applies to all users on a machine. This file cannot be excluded by individual settings.

Step 1: Create the file at the managed policy location

- macOS: `/Library/Application Support/ClaudeCode/CLAUDE.md`
- Linux and WSL: `/etc/claude-code/CLAUDE.md`
- Windows: `C:\Program Files\ClaudeCode\CLAUDE.md`

Step 2: Deploy with your configuration management system

Use MDM, Group Policy, Ansible, or similar tools to distribute the file across developer machines. See managed settings for other organization-wide configuration options.

Exclude specific CLAUDE.md files

In large monorepos, ancestor CLAUDE.md files may contain instructions that aren't relevant to your work. The `claudeMdExcludes` setting lets you skip specific files by path or glob pattern.

This example excludes a top-level CLAUDE.md and a rules directory from a parent folder. Add it to `.claude/settings.local.json` so the exclusion stays local to your machine:

```
{
  "claudeMdExcludes": [
    "**/monorepo/CLAUDE.md",
    "/home/user/monorepo/other-team/.claude/rules/**"
  ]
}
```

Patterns are matched against absolute file paths using glob syntax. You can configure `claudeMdExcludes` at any settings layer: user, project, local, or managed policy. Arrays merge across layers.

Managed policy CLAUDE.md files cannot be excluded. This ensures organization-wide instructions always apply regardless of individual settings.

Auto memory

Auto memory lets Claude accumulate knowledge across sessions without you writing anything. Claude saves notes for itself as it works: build commands, debugging insights, architecture notes, code style preferences, and workflow habits. Claude doesn't save something every session. It decides what's worth remembering based on whether the information would be useful in a future conversation.

> **Note:**
>
> Auto memory requires Claude Code v2.1.59 or later. Check your version with `claude --version`.

Enable or disable auto memory

Auto memory is on by default. To toggle it, open `/memory` in a session and use the auto memory toggle, or set `autoMemoryEnabled` in your project settings:

```
{
  "autoMemoryEnabled": false
}
```

To disable auto memory via environment variable, set `CLAUDE_CODE_DISABLE_AUTO_MEMORY=1`.

Storage location

Each project gets its own memory directory at `~/.claude/projects/<project>/memory/`. The `<project>` path is derived from the git repository, so all worktrees and subdirectories within the same repo share one auto memory directory. Outside a git repo, the project root is used instead.

To store auto memory in a different location, set `autoMemoryDirectory` in your user or local settings:

```
{
  "autoMemoryDirectory": "~/my-custom-memory-dir"
}
```

This setting is accepted from policy, local, and user settings. It is not accepted from project settings (`.claude/settings.json`) to prevent a shared project from redirecting auto memory writes to sensitive locations.

The directory contains a `MEMORY.md` entrypoint and optional topic files:

```
~/.claude/projects/<project>/memory/
├── MEMORY.md          # Concise index, loaded into every session
├── debugging.md       # Detailed notes on debugging patterns
├── api-conventions.md # API design decisions
└── ...                # Any other topic files Claude creates
```

`MEMORY.md` acts as an index of the memory directory. Claude reads and writes files in this directory throughout your session, using `MEMORY.md` to keep track of what's stored where.

Auto memory is machine-local. All worktrees and subdirectories within the same git repository share one auto memory directory. Files are not shared across machines or cloud environments.

How it works

The first 200 lines of `MEMORY.md` are loaded at the start of every conversation. Content beyond line 200 is not loaded at session start. Claude keeps `MEMORY.md` concise by moving detailed notes into separate topic files.

This 200-line limit applies only to `MEMORY.md` . CLAUDE.md files are loaded in full regardless of length, though shorter files produce better adherence.

Topic files like `debugging.md` or `patterns.md` are not loaded at startup. Claude reads them on demand using its standard file tools when it needs the information.

Claude reads and writes memory files during your session. When you see "Writing memory" or "Recalled memory" in the Claude Code interface, Claude is actively updating or reading from `~/.claude/projects/<project>/memory/` .

Audit and edit your memory

Auto memory files are plain markdown you can edit or delete at any time. Run `/memory` to browse and open memory files from within a session.

View and edit with `/memory`

The `/memory` command lists all CLAUDE.md and rules files loaded in your current session, lets you toggle auto memory on or off, and provides a link to open the auto memory folder. Select any file to open it in your editor.

When you ask Claude to remember something, like “always use pnpm, not npm” or “remember that the API tests require a local Redis instance,” Claude saves it to auto memory. To add instructions to CLAUDE.md instead, ask Claude directly, like “add this to CLAUDE.md,” or edit the file yourself via `/memory`.

Troubleshoot memory issues

These are the most common issues with CLAUDE.md and auto memory, along with steps to debug them.

Claude isn’t following my CLAUDE.md

CLAUDE.md content is delivered as a user message after the system prompt, not as part of the system prompt itself. Claude reads it and tries to follow it, but there’s no guarantee of strict compliance, especially for vague or conflicting instructions.

To debug:

- Run `/memory` to verify your CLAUDE.md files are being loaded. If a file isn’t listed, Claude can’t see it.
- Check that the relevant CLAUDE.md is in a location that gets loaded for your session (see Choose where to put CLAUDE.md files).
- Make instructions more specific. “Use 2-space indentation” works better than “format code nicely.”
- Look for conflicting instructions across CLAUDE.md files. If two files give different guidance for the same behavior, Claude may pick one arbitrarily.

For instructions you want at the system prompt level, use `--append-system-prompt`. This must be passed every invocation, so it’s better suited to scripts and automation than interactive use.

> **Tip:**
>
> Use the `InstructionsLoaded` hook to log exactly which instruction files are loaded, when they load, and why. This is useful for debugging path-specific rules or lazy-loaded files in subdirectories.

I don't know what auto memory saved

Run `/memory` and select the auto memory folder to browse what Claude has saved. Everything is plain markdown you can read, edit, or delete.

My CLAUDE.md is too large

Files over 200 lines consume more context and may reduce adherence. Move detailed content into separate files referenced with `@path` imports (see Import additional files), or split your instructions across `.claude/rules/` files.

Instructions seem lost after `/compact`

CLAUDE.md fully survives compaction. After `/compact`, Claude re-reads your CLAUDE.md from disk and re-injects it fresh into the session. If an instruction disappeared after compaction, it was given only in conversation, not written to CLAUDE.md. Add it to CLAUDE.md to make it persist across sessions.

See Write effective instructions for guidance on size, structure, and specificity.

Related resources

- Skills: package repeatable workflows that load on demand
- Settings: configure Claude Code behavior with settings files
- Manage sessions: manage context, resume conversations, and run parallel sessions
- Subagent memory: let subagents maintain their own auto memory

Claude Code settings

> *Configure Claude Code with global and project-level settings, and environment variables.*

Claude Code offers a variety of settings to configure its behavior to meet your needs. You can configure Claude Code by running the `/config` command when using the interactive REPL, which opens a tabbed Settings interface where you can view status information and modify configuration options.

Configuration scopes

Claude Code uses a **scope system** to determine where configurations apply and who they're shared with. Understanding scopes helps you decide how to configure Claude Code for personal use, team collaboration, or enterprise deployment.

Available scopes

Scope	Location	Who it affects	Shared with team?
Managed	Server-managed settings, plist / registry, or system-level `managed-settings.json`	All users on the machine	Yes (deployed by IT)
User	`~/.claude/` directory	You, across all projects	No
Project	`.claude/` in repository	All collaborators on this repository	Yes (committed to git)
Local	`.claude/settings.local.json`	You, in this repository only	No (gitignored)

When to use each scope

Managed scope is for:

- Security policies that must be enforced organization-wide
- Compliance requirements that can't be overridden
- Standardized configurations deployed by IT/DevOps

User scope is best for:

- Personal preferences you want everywhere (themes, editor settings)
- Tools and plugins you use across all projects
- API keys and authentication (stored securely)

Project scope is best for:

- Team-shared settings (permissions, hooks, MCP servers)
- Plugins the whole team should have
- Standardizing tooling across collaborators

Local scope is best for:

- Personal overrides for a specific project
- Testing configurations before sharing with the team
- Machine-specific settings that won't work for others

How scopes interact

When the same setting is configured in multiple scopes, more specific scopes take precedence:

1. **Managed** (highest) - can't be overridden by anything
2. **Command line arguments** - temporary session overrides
3. **Local** - overrides project and user settings
4. **Project** - overrides user settings
5. **User** (lowest) - applies when nothing else specifies the setting

For example, if a permission is allowed in user settings but denied in project settings, the project setting takes precedence and the permission is blocked.

What uses scopes

Scopes apply to many Claude Code features:

Feature	User location	Project location	Local location
Settings	`~/.claude/settings.json`	`.claude/settings.json`	`.claude/settings.local.json`
Subagents	`~/.claude/agents/`	`.claude/agents/`	None

Feature	User location	Project location	Local location
MCP servers	`~/.claude.json`	`.mcp.json`	`~/.claude.json` (per-project)
Plugins	`~/.claude/settings.json`	`.claude/settings.json`	`.claude/settings.local.json`
CLAUDE.md	`~/.claude/CLAUDE.md`	`CLAUDE.md` or `.claude/CLAUDE.md`	None

Settings files

The `settings.json` file is the official mechanism for configuring Claude Code through hierarchical settings:

- **User settings** are defined in `~/.claude/settings.json` and apply to all projects.
- **Project settings** are saved in your project directory:
  - `.claude/settings.json` for settings that are checked into source control and shared with your team
  - `.claude/settings.local.json` for settings that are not checked in, useful for personal preferences and experimentation. Claude Code will configure git to ignore `.claude/settings.local.json` when it is created.
- **Managed settings**: For organizations that need centralized control, Claude Code supports multiple delivery mechanisms for managed settings. All use the same JSON format and cannot be overridden by user or project settings:
  - **Server-managed settings**: delivered from Anthropic's servers via the Claude.ai admin console. See server-managed settings.
  - **MDM/OS-level policies**: delivered through native device management on macOS and Windows:
  - macOS: `com.anthropic.claudecode` managed preferences domain (deployed via configuration profiles in Jamf, Kandji, or other MDM tools)
  - Windows: `HKLM\SOFTWARE\Policies\ClaudeCode` registry key with a `Settings` value (REG_SZ or REG_EXPAND_SZ) containing JSON (deployed via Group Policy or Intune)

- Windows (user-level): `HKCU\SOFTWARE\Policies\ClaudeCode` (lowest policy priority, only used when no admin-level source exists)
- **File-based**: `managed-settings.json` and `managed-mcp.json` deployed to system directories:
- macOS: `/Library/Application Support/ClaudeCode/`
- Linux and WSL: `/etc/claude-code/`
- Windows: `C:\Program Files\ClaudeCode\`

> **Warning:**
>
> The legacy Windows path `C:\ProgramData\ClaudeCode\managed-settings.json` is no longer supported as of v2.1.75. Administrators who deployed settings to that location must migrate files to `C:\Program Files\ClaudeCode\managed-settings.json`.

See managed settings and Managed MCP configuration for details.

> **Note:**
>
> Managed deployments can also restrict **plugin marketplace additions** using `strictKnownMarketplaces`. For more information, see Managed marketplace restrictions.

- **Other configuration** is stored in `~/.claude.json`. This file contains your preferences (theme, notification settings, editor mode), OAuth session, MCP server configurations for user and local scopes, per-project state (allowed tools, trust settings), and various caches. Project-scoped MCP servers are stored separately in `.mcp.json`.

> **Note:**
>
> Claude Code automatically creates timestamped backups of configuration files and retains the five most recent backups to prevent data loss.

```
{
  "$schema": "https://json.schemastore.org/claude-code-settings.json",
  "permissions": {
    "allow": [
      "Bash(npm run lint)",
      "Bash(npm run test *)",
      "Read(~/.zshrc)"
    ],
    "deny": [
      "Bash(curl *)",
      "Read(./.env)",
      "Read(./.env.*)",
      "Read(./secrets/**)"
    ]
  },
  "env": {
    "CLAUDE_CODE_ENABLE_TELEMETRY": "1",
    "OTEL_METRICS_EXPORTER": "otlp"
  },
  "companyAnnouncements": [
    "Welcome to Acme Corp! Review our code guidelines at docs.acme.com",
    "Reminder: Code reviews required for all PRs",
    "New security policy in effect"
  ]
}
```

The `$schema` line in the example above points to the official JSON schema for Claude Code settings. Adding it to your `settings.json` enables autocomplete and inline validation in VS Code, Cursor, and any other editor that supports JSON schema validation.

Available settings

`settings.json` supports a number of options:

Key	Description	Example
`apiKeyHelper`	Custom script, to be executed in `/bin/sh`, to generate an auth value. This value will be sent as `X-Api-Key` and `Authorization: Bearer` headers for model requests	`/bin/generate_temp_api_key.sh`
`autoMemoryDirectory`	Custom directory for auto memory storage. Accepts `~/`-expanded paths. Not accepted in project settings (`.claude/settings.json`) to prevent shared repos from redirecting memory writes to sensitive locations. Accepted from policy, local, and user settings	`"~/my-memory-dir"`
`cleanupPeriodDays`	Sessions inactive for longer than this period are deleted at startup (default: 30 days). Setting to `0` deletes all existing transcripts at startup and disables session persistence entirely. No new `.jsonl` files are written, `/resume` shows no conversations, and hooks receive an empty `transcript_path`.	`20`
`companyAnnouncements`	Announcement to display to users at startup. If multiple announcements are provided, they will be cycled through at random.	`["Welcome to Acme Corp! Review our code guidelines at docs.acme.com"]`
`env`	Environment variables that will be applied to every session	`{"FOO": "bar"}`
`attribution`	Customize attribution for git commits and pull requests. See Attribution settings	`{"commit": "🤖 Generated with Claude Code", "pr": ""}`
`includeCoAuthoredBy`	**Deprecated**: Use `attribution` instead. Whether to include the `co-authored-by Claude` byline in git commits and pull requests (default: `true`)	`false`

Key	Description	Example
`includeGitInstructions`	Include built-in commit and PR workflow instructions in Claude's system prompt (default: `true`). Set to `false` to remove these instructions, for example when using your own git workflow skills. The `CLAUDE_CODE_DISABLE_GIT_INSTRUCTIONS` environment variable takes precedence over this setting when set	`false`
`permissions`	See table below for structure of permissions.	
`hooks`	Configure custom commands to run at lifecycle events. See hooks documentation for format	See hooks
`disableAllHooks`	Disable all hooks and any custom status line	`true`
`allowManagedHooksOnly`	(Managed settings only) Prevent loading of user, project, and plugin hooks. Only allows managed hooks and SDK hooks. See Hook configuration	`true`
`allowedHttpHookUrls`	Allowlist of URL patterns that HTTP hooks may target. Supports `*` as a wildcard. When set, hooks with non-matching URLs are blocked. Undefined = no restriction, empty array = block all HTTP hooks. Arrays merge across settings sources. See Hook configuration	`["https://hooks.example.com/*"]`
`httpHookAllowedEnvVars`	Allowlist of environment variable names HTTP hooks may interpolate into headers. When set, each hook's effective `allowedEnvVars` is the intersection with this list. Undefined = no restriction. Arrays merge across settings sources. See Hook configuration	`["MY_TOKEN", "HOOK_SECRET"]`

Key	Description	Example
`allowManagedPermissionRulesOnly`	(Managed settings only) Prevent user and project settings from defining `allow`, `ask`, or `deny` permission rules. Only rules in managed settings apply. See Managed-only settings	`true`
`allowManagedMcpServersOnly`	(Managed settings only) Only `allowedMcpServers` from managed settings are respected. `deniedMcpServers` still merges from all sources. Users can still add MCP servers, but only the admin-defined allowlist applies. See Managed MCP configuration	`true`
`model`	Override the default model to use for Claude Code	`"claude-sonnet-4-6"`
`availableModels`	Restrict which models users can select via `/model`, `--model`, Config tool, or `ANTHROPIC_MODEL`. Does not affect the Default option. See Restrict model selection	`["sonnet", "haiku"]`
`modelOverrides`	Map Anthropic model IDs to provider-specific model IDs such as Bedrock inference profile ARNs. Each model picker entry uses its mapped value when calling the provider API. See Override model IDs per version	`{"claude-opus-4-6": "arn:aws:bedrock:..."}`
`effortLevel`	Persist the effort level across sessions. Accepts `"low"`, `"medium"`, or `"high"`. Written automatically when you run `/effort low`, `/effort medium`, or `/effort high`. Supported on Opus 4.6 and Sonnet 4.6	`"medium"`
`otelHeadersHelper`	Script to generate dynamic OpenTelemetry headers. Runs at startup and periodically (see Dynamic headers)	`/bin/generate_otel_headers.sh`

Key	Description	Example
`statusLine`	Configure a custom status line to display context. See `statusLine` documentation	`{"type": "command", "command": "~/.claude/statusline.sh"}`
`fileSuggestion`	Configure a custom script for `@` file autocomplete. See File suggestion settings	`{"type": "command", "command": "~/.claude/file-suggestion.sh"}`
`respectGitignore`	Control whether the `@` file picker respects `.gitignore` patterns. When `true` (default), files matching `.gitignore` patterns are excluded from suggestions	`false`
`outputStyle`	Configure an output style to adjust the system prompt. See output styles documentation	`"Explanatory"`
`forceLoginMethod`	Use `claudeai` to restrict login to Claude.ai accounts, `console` to restrict login to Claude Console (API usage billing) accounts	`claudeai`
`forceLoginOrgUUID`	Specify the UUID of an organization to automatically select it during login, bypassing the organization selection step. Requires `forceLoginMethod` to be set	`"xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx"`
`enableAllProjectMcpServers`	Automatically approve all MCP servers defined in project `.mcp.json` files	`true`
`enabledMcpjsonServers`	List of specific MCP servers from `.mcp.json` files to approve	`["memory", "github"]`
`disabledMcpjsonServers`	List of specific MCP servers from `.mcp.json` files to reject	`["filesystem"]`

Key	Description	Example
`allowedMcpServers`	When set in managed-settings.json, allowlist of MCP servers users can configure. Undefined = no restrictions, empty array = lockdown. Applies to all scopes. Denylist takes precedence. See Managed MCP configuration	`[{ "serverName": "github" }]`
`deniedMcpServers`	When set in managed-settings.json, denylist of MCP servers that are explicitly blocked. Applies to all scopes including managed servers. Denylist takes precedence over allowlist. See Managed MCP configuration	`[{ "serverName": "filesystem" }]`
`strictKnownMarketplaces`	When set in managed-settings.json, allowlist of plugin marketplaces users can add. Undefined = no restrictions, empty array = lockdown. Applies to marketplace additions only. See Managed marketplace restrictions	`[{ "source": "github", "repo": "acme-corp/plugins" }]`
`blockedMarketplaces`	(Managed settings only) Blocklist of marketplace sources. Blocked sources are checked before downloading, so they never touch the filesystem. See Managed marketplace restrictions	`[{ "source": "github", "repo": "untrusted/plugins" }]`
`pluginTrustMessage`	(Managed settings only) Custom message appended to the plugin trust warning shown before installation. Use this to add organization-specific context, for example to confirm that plugins from your internal marketplace are vetted.	`"All plugins from our marketplace are approved by IT"`
`awsAuthRefresh`	Custom script that modifies the `.aws` directory (see advanced credential configuration)	`aws sso login --profile myprofile`

Key	Description	Example
`awsCredentialExport`	Custom script that outputs JSON with AWS credentials (see advanced credential configuration)	`/bin/generate_aws_grant.sh`
`alwaysThinkingEnabled`	Enable extended thinking by default for all sessions. Typically configured via the `/config` command rather than editing directly	`true`
`plansDirectory`	Customize where plan files are stored. Path is relative to project root. Default: `~/.claude/plans`	`"./plans"`
`showTurnDuration`	Show turn duration messages after responses (e.g., “Cooked for 1m 6s”). Set to `false` to hide these messages	`true`
`spinnerVerbs`	Customize the action verbs shown in the spinner and turn duration messages. Set `mode` to `"replace"` to use only your verbs, or `"append"` to add them to the defaults	`{"mode": "append", "verbs": ["Pondering", "Crafting"]}`
`language`	Configure Claude’s preferred response language (e.g., `"japanese"`, `"spanish"`, `"french"`). Claude will respond in this language by default	`"japanese"`
`autoUpdatesChannel`	Release channel to follow for updates. Use `"stable"` for a version that is typically about one week old and skips versions with major regressions, or `"latest"` (default) for the most recent release	`"stable"`
`spinnerTipsEnabled`	Show tips in the spinner while Claude is working. Set to `false` to disable tips (default: `true`)	`false`

Key	Description	Example
`spinnerTipsOverride`	Override spinner tips with custom strings. `tips` : array of tip strings. `excludeDefault` : if `true` , only show custom tips; if `false` or absent, custom tips are merged with built-in tips	`{ "excludeDefault": true, "tips": ["Use our internal tool X"] }`
`terminalProgressBarEnabled`	Enable the terminal progress bar that shows progress in supported terminals like Windows Terminal and iTerm2 (default: `true`)	`false`
`prefersReducedMotion`	Reduce or disable UI animations (spinners, shimmer, flash effects) for accessibility	`true`
`fastModePerSessionOptIn`	When `true` , fast mode does not persist across sessions. Each session starts with fast mode off, requiring users to enable it with `/fast` . The user's fast mode preference is still saved. See Require per-session opt-in	`true`
`teammateMode`	How agent team teammates display: `auto` (picks split panes in tmux or iTerm2, in-process otherwise), `in-process` , or `tmux` . See set up agent teams	`"in-process"`
`feedbackSurveyRate`	Probability (0–1) that the session quality survey appears when eligible. Set to `0` to suppress entirely. Useful when using Bedrock, Vertex, or Foundry where the default sample rate does not apply	`0.05`

Worktree settings

Configure how `--worktree` creates and manages git worktrees. Use these settings to reduce disk usage and startup time in large monorepos.

Key	Description	Example
`worktree.symlinkDirectories`	Directories to symlink from the main repository into each worktree to avoid duplicating large directories on disk. No directories are symlinked by default	`["node_modules", ".cache"]`
`worktree.sparsePaths`	Directories to check out in each worktree via git sparse-checkout (cone mode). Only the listed paths are written to disk, which is faster in large monorepos	`["packages/my-app", "shared/utils"]`

Permission settings

Keys	Description	Example
`allow`	Array of permission rules to allow tool use. See Permission rule syntax below for pattern matching details	`[ "Bash(git diff *)" ]`
`ask`	Array of permission rules to ask for confirmation upon tool use. See Permission rule syntax below	`[ "Bash(git push *)" ]`
`deny`	Array of permission rules to deny tool use. Use this to exclude sensitive files from Claude Code access. See Permission rule syntax and Bash permission limitations	`[ "WebFetch", "Bash(curl *)", "Read(./.env)", "Read(./secrets/**)" ]`
`additionalDirectories`	Additional working directories that Claude has access to	`[ "../docs/" ]`
`defaultMode`	Default permission mode when opening Claude Code	`"acceptEdits"`
`disableBypassPermissionsMode`	Set to `"disable"` to prevent `bypassPermissions` mode from being activated. This disables the `--dangerously-skip-permissions` command-line flag. See managed settings	`"disable"`

Permission rule syntax

Permission rules follow the format `Tool` or `Tool(specifier)`. Rules are evaluated in order: deny rules first, then ask, then allow. The first matching rule wins.

Quick examples:

Rule	Effect
`Bash`	Matches all Bash commands
`Bash(npm run *)`	Matches commands starting with `npm run`
`Read(./.env)`	Matches reading the `.env` file
`WebFetch(domain:example.com)`	Matches fetch requests to example.com

For the complete rule syntax reference, including wildcard behavior, tool-specific patterns for Read, Edit, WebFetch, MCP, and Agent rules, and security limitations of Bash patterns, see Permission rule syntax.

Sandbox settings

Configure advanced sandboxing behavior. Sandboxing isolates bash commands from your filesystem and network. See Sandboxing for details.

Keys	Description	Example
`enabled`	Enable bash sandboxing (macOS, Linux, and WSL2). Default: false	`true`
`autoAllowBashIfSandboxed`	Auto-approve bash commands when sandboxed. Default: true	`true`
`excludedCommands`	Commands that should run outside of the sandbox	`["git", "docker"]`

Keys	Description	Example
`allowUnsandboxedCommands`	Allow commands to run outside the sandbox via the `dangerouslyDisableSandbox` parameter. When set to `false`, the `dangerouslyDisableSandbox` escape hatch is completely disabled and all commands must run sandboxed (or be in `excludedCommands`). Useful for enterprise policies that require strict sandboxing. Default: true	`false`
`filesystem.allowWrite`	Additional paths where sandboxed commands can write. Arrays are merged across all settings scopes: user, project, and managed paths are combined, not replaced. Also merged with paths from `Edit(...)` allow permission rules. See path prefixes below.	`["//tmp/build", "~/.kube"]`
`filesystem.denyWrite`	Paths where sandboxed commands cannot write. Arrays are merged across all settings scopes. Also merged with paths from `Edit(...)` deny permission rules.	`["//etc", "//usr/local/bin"]`
`filesystem.denyRead`	Paths where sandboxed commands cannot read. Arrays are merged across all settings scopes. Also merged with paths from `Read(...)` deny permission rules.	`["~/.aws/credentials"]`
`network.allowUnixSockets`	Unix socket paths accessible in sandbox (for SSH agents, etc.)	`["~/.ssh/agent-socket"]`
`network.allowAllUnixSockets`	Allow all Unix socket connections in sandbox. Default: false	`true`
`network.allowLocalBinding`	Allow binding to localhost ports (macOS only). Default: false	`true`

Keys	Description	Example
`network.allowedDomains`	Array of domains to allow for outbound network traffic. Supports wildcards (e.g., `*.example.com`).	`["github.com", "*.npmjs.org"]`
`network.allowManagedDomainsOnly`	(Managed settings only) Only `allowedDomains` and `WebFetch(domain:...)` allow rules from managed settings are respected. Domains from user, project, and local settings are ignored. Non-allowed domains are blocked automatically without prompting the user. Denied domains are still respected from all sources. Default: false	`true`
`network.httpProxyPort`	HTTP proxy port used if you wish to bring your own proxy. If not specified, Claude will run its own proxy.	`8080`
`network.socksProxyPort`	SOCKS5 proxy port used if you wish to bring your own proxy. If not specified, Claude will run its own proxy.	`8081`
`enableWeakerNestedSandbox`	Enable weaker sandbox for unprivileged Docker environments (Linux and WSL2 only). **Reduces security.** Default: false	`true`
`enableWeakerNetworkIsolation`	(macOS only) Allow access to the system TLS trust service (`com.apple.trustd.agent`) in the sandbox. Required for Go-based tools like `gh`, `gcloud`, and `terraform` to verify TLS certificates when using `httpProxyPort` with a MITM proxy and custom CA. **Reduces security** by opening a potential data exfiltration path. Default: false	`true`

Sandbox path prefixes

Paths in `filesystem.allowWrite`, `filesystem.denyWrite`, and `filesystem.denyRead` support these prefixes:

Prefix	Meaning	Example
`//`	Absolute path from filesystem root	`//tmp/build` becomes `/tmp/build`
`~/`	Relative to home directory	`~/.kube` becomes `$HOME/.kube`
`/`	Relative to the settings file's directory	`/build` becomes `$SETTINGS_DIR/build`
`./` or no prefix	Relative path (resolved by sandbox runtime)	`./output`

Configuration example:

```
{
  "sandbox": {
    "enabled": true,
    "autoAllowBashIfSandboxed": true,
    "excludedCommands": ["docker"],
    "filesystem": {
      "allowWrite": ["//tmp/build", "~/.kube"],
      "denyRead": ["~/.aws/credentials"]
    },
    "network": {
      "allowedDomains": ["github.com", "*.npmjs.org", "registry.yarnpkg.com"],
      "allowUnixSockets": [
        "/var/run/docker.sock"
      ],
      "allowLocalBinding": true
    }
  }
}
```

Filesystem and network restrictions can be configured in two ways that are merged together:

- `sandbox.filesystem` **settings** (shown above): Control paths at the OS-level sandbox boundary. These restrictions apply to all subprocess commands (e.g., `kubectl`, `terraform`, `npm`), not just Claude's file tools.
- **Permission rules**: Use `Edit` allow/deny rules to control Claude's file tool access, `Read` deny rules to block reads, and `WebFetch` allow/deny rules to control network domains. Paths from these rules are also merged into the sandbox configuration.

Attribution settings

Claude Code adds attribution to git commits and pull requests. These are configured separately:

- Commits use git trailers (like `Co-Authored-By`) by default, which can be customized or disabled
- Pull request descriptions are plain text

Keys	Description
`commit`	Attribution for git commits, including any trailers. Empty string hides commit attribution
`pr`	Attribution for pull request descriptions. Empty string hides pull request attribution

Default commit attribution:

```
🤖 Generated with [Claude Code](https://claude.com/claude-code)

   Co-Authored-By: Claude Sonnet 4.6 <noreply@anthropic.com>
```

Default pull request attribution:

```
🤖 Generated with [Claude Code](https://claude.com/claude-code)
```

Example:

```
{
  "attribution": {
    "commit": "Generated with AI\n\nCo-Authored-By: AI <ai@example.com>",
    "pr": ""
  }
}
```

> **Note:**
>
> The `attribution` setting takes precedence over the deprecated `includeCoAuthoredBy` setting. To hide all attribution, set `commit` and `pr` to empty strings.

File suggestion settings

Configure a custom command for `@` file path autocomplete. The built-in file suggestion uses fast filesystem traversal, but large monorepos may benefit from project-specific indexing such as a pre-built file index or custom tooling.

```
{
  "fileSuggestion": {
    "type": "command",
    "command": "~/.claude/file-suggestion.sh"
  }
}
```

The command runs with the same environment variables as hooks, including `CLAUDE_PROJECT_DIR`. It receives JSON via stdin with a `query` field:

```
{"query": "src/comp"}
```

Output newline-separated file paths to stdout (currently limited to 15):

```
src/components/Button.tsx
src/components/Modal.tsx
src/components/Form.tsx
```

Example:

```
#!/bin/bash
query=$(cat | jq -r '.query')
your-repo-file-index --query "$query" | head -20
```

Hook configuration

These settings control which hooks are allowed to run and what HTTP hooks can access. The `allowManagedHooksOnly` setting can only be configured in managed settings. The URL and env var allowlists can be set at any settings level and merge across sources.

Behavior when `allowManagedHooksOnly` is `true`:

- Managed hooks and SDK hooks are loaded
- User hooks, project hooks, and plugin hooks are blocked

Restrict HTTP hook URLs:

Limit which URLs HTTP hooks can target. Supports `*` as a wildcard for matching. When the array is defined, HTTP hooks targeting non-matching URLs are silently blocked.

```
{
  "allowedHttpHookUrls": ["https://hooks.example.com/*", "http://localhost:*"]
}
```

Restrict HTTP hook environment variables:

Limit which environment variable names HTTP hooks can interpolate into header values. Each hook's effective `allowedEnvVars` is the intersection of its own list and this setting.

```
{
  "httpHookAllowedEnvVars": ["MY_TOKEN", "HOOK_SECRET"]
}
```

Settings precedence

Settings apply in order of precedence. From highest to lowest:

1. **Managed settings** (server-managed, MDM/OS-level policies, or managed settings)
   - Policies deployed by IT through server delivery, MDM configuration profiles, registry policies, or managed settings files
   - Cannot be overridden by any other level, including command line arguments

    - Within the managed tier, precedence is: server-managed > MDM/OS-level policies > `managed-settings.json` > HKCU registry (Windows only). Only one managed source is used; sources do not merge.

2. **Command line arguments**
    - Temporary overrides for a specific session
3. **Local project settings** (`.claude/settings.local.json`)
    - Personal project-specific settings
4. **Shared project settings** (`.claude/settings.json`)
    - Team-shared project settings in source control
5. **User settings** (`~/.claude/settings.json`)
    - Personal global settings

This hierarchy ensures that organizational policies are always enforced while still allowing teams and individuals to customize their experience.

For example, if your user settings allow `Bash(npm run *)` but a project's shared settings deny it, the project setting takes precedence and the command is blocked.

> **Note:**
>
> **Array settings merge across scopes.** When the same array-valued setting (such as `sandbox.filesystem.allowWrite` or `permissions.allow`) appears in multiple scopes, the arrays are **concatenated and deduplicated**, not replaced. This means lower-priority scopes can add entries without overriding those set by higher-priority scopes, and vice versa. For example, if managed settings set `allowWrite` to `["//opt/company-tools"]` and a user adds `["~/.kube"]` , both paths are included in the final configuration.

Verify active settings

Run `/status` inside Claude Code to see which settings sources are active and where they come from. The output shows each configuration layer (managed, user, project) along with its origin, such as `Enterprise managed settings (remote)` , `Enterprise managed settings (plist)` , `Enterprise managed settings (HKLM)` , or `Enterprise managed settings (file)` . If a settings file contains errors, `/status` reports the issue so you can fix it.

Key points about the configuration system

- **Memory files (`CLAUDE.md`)**: Contain instructions and context that Claude loads at startup

- **Settings files (JSON)**: Configure permissions, environment variables, and tool behavior
- **Skills**: Custom prompts that can be invoked with `/skill-name` or loaded by Claude automatically
- **MCP servers**: Extend Claude Code with additional tools and integrations
- **Precedence**: Higher-level configurations (Managed) override lower-level ones (User/Project)
- **Inheritance**: Settings are merged, with more specific settings adding to or overriding broader ones

System prompt

Claude Code's internal system prompt is not published. To add custom instructions, use `CLAUDE.md` files or the `--append-system-prompt` flag.

Excluding sensitive files

To prevent Claude Code from accessing files containing sensitive information like API keys, secrets, and environment files, use the `permissions.deny` setting in your `.claude/settings.json` file:

```
{
  "permissions": {
    "deny": [
      "Read(./.env)",
      "Read(./.env.*)",
      "Read(./secrets/**)",
      "Read(./config/credentials.json)",
      "Read(./build)"
    ]
  }
}
```

This replaces the deprecated `ignorePatterns` configuration. Files matching these patterns are excluded from file discovery and search results, and read operations on these files are denied.

Subagent configuration

Claude Code supports custom AI subagents that can be configured at both user and project levels. These subagents are stored as Markdown files with YAML frontmatter:

- **User subagents**: `~/.claude/agents/` - Available across all your projects
- **Project subagents**: `.claude/agents/` - Specific to your project and can be shared with your team

Subagent files define specialized AI assistants with custom prompts and tool permissions. Learn more about creating and using subagents in the subagents documentation.

Plugin configuration

Claude Code supports a plugin system that lets you extend functionality with skills, agents, hooks, and MCP servers. Plugins are distributed through marketplaces and can be configured at both user and repository levels.

Plugin settings

Plugin-related settings in `settings.json`:

```
{
  "enabledPlugins": {
    "formatter@acme-tools": true,
    "deployer@acme-tools": true,
    "analyzer@security-plugins": false
  },
  "extraKnownMarketplaces": {
    "acme-tools": {
      "source": "github",
      "repo": "acme-corp/claude-plugins"
    }
  }
}
```

`enabledPlugins`

Controls which plugins are enabled. Format: `"plugin-name@marketplace-name": true/false`

Scopes:

- **User settings** (`~/.claude/settings.json`): Personal plugin preferences
- **Project settings** (`.claude/settings.json`): Project-specific plugins shared with team
- **Local settings** (`.claude/settings.local.json`): Per-machine overrides (not committed)

Example:

```
{
  "enabledPlugins": {
    "code-formatter@team-tools": true,
    "deployment-tools@team-tools": true,
    "experimental-features@personal": false
  }
}
```

`extraKnownMarketplaces`

Defines additional marketplaces that should be made available for the repository. Typically used in repository-level settings to ensure team members have access to required plugin sources.

When a repository includes `extraKnownMarketplaces` :

1. Team members are prompted to install the marketplace when they trust the folder
2. Team members are then prompted to install plugins from that marketplace
3. Users can skip unwanted marketplaces or plugins (stored in user settings)
4. Installation respects trust boundaries and requires explicit consent

Example:

```
{
  "extraKnownMarketplaces": {
    "acme-tools": {
      "source": {
        "source": "github",
        "repo": "acme-corp/claude-plugins"
      }
    },
    "security-plugins": {
      "source": {
        "source": "git",
        "url": "https://git.example.com/security/plugins.git"
      }
    }
  }
}
```

Marketplace source types:

- `github`: GitHub repository (uses `repo`)
- `git`: Any git URL (uses `url`)
- `directory`: Local filesystem path (uses `path`, for development only)
- `hostPattern`: regex pattern to match marketplace hosts (uses `hostPattern`)

`strictKnownMarketplaces`

Managed settings only: Controls which plugin marketplaces users are allowed to add. This setting can only be configured in managed settings and provides administrators with strict control over marketplace sources.

Managed settings file locations:

- **macOS**: `/Library/Application Support/ClaudeCode/managed-settings.json`
- **Linux and WSL**: `/etc/claude-code/managed-settings.json`
- **Windows**: `C:\Program Files\ClaudeCode\managed-settings.json`

Key characteristics:

- Only available in managed settings (`managed-settings.json`)
- Cannot be overridden by user or project settings (highest precedence)

- Enforced BEFORE network/filesystem operations (blocked sources never execute)
- Uses exact matching for source specifications (including `ref`, `path` for git sources), except `hostPattern`, which uses regex matching

Allowlist behavior:

- `undefined` (default): No restrictions - users can add any marketplace
- Empty array `[]`: Complete lockdown - users cannot add any new marketplaces
- List of sources: Users can only add marketplaces that match exactly

All supported source types:

The allowlist supports seven marketplace source types. Most sources use exact matching, while `hostPattern` uses regex matching against the marketplace host.

1. **GitHub repositories**:

```
{ "source": "github", "repo": "acme-corp/approved-plugins" }
{ "source": "github", "repo": "acme-corp/security-tools", "ref": "v2.0" }
{ "source": "github", "repo": "acme-corp/plugins", "ref": "main", "path": "marketp
lace" }
```

Fields: `repo` (required), `ref` (optional: branch/tag/SHA), `path` (optional: subdirectory)

1. **Git repositories**:

```
{ "source": "git", "url": "https://gitlab.example.com/tools/plugins.git" }
{ "source": "git", "url": "https://bitbucket.org/acme-corp/plugins.git", "ref": "p
roduction" }
{ "source": "git", "url": "ssh://git@git.example.com/plugins.git", "ref": "v3.1",
"path": "approved" }
```

Fields: `url` (required), `ref` (optional: branch/tag/SHA), `path` (optional: subdirectory)

1. **URL-based marketplaces**:

```
{ "source": "url", "url": "https://plugins.example.com/marketplace.json" }
{ "source": "url", "url": "https://cdn.example.com/marketplace.json", "headers":
{ "Authorization": "Bearer ${TOKEN}" } }
```

Fields: `url` (required), `headers` (optional: HTTP headers for authenticated access)

> **Note:**
>
> URL-based marketplaces only download the `marketplace.json` file. They do not download plugin files from the server. Plugins in URL-based marketplaces must use external sources (GitHub, npm, or git URLs) rather than relative paths. For plugins with relative paths, use a Git-based marketplace instead. See Troubleshooting for details.

1. **NPM packages**:

```
{ "source": "npm", "package": "@acme-corp/claude-plugins" }
{ "source": "npm", "package": "@acme-corp/approved-marketplace" }
```

Fields: `package` (required, supports scoped packages)

1. **File paths**:

```
{ "source": "file", "path": "/usr/local/share/claude/acme-marketplace.json" }
{ "source": "file", "path": "/opt/acme-corp/plugins/marketplace.json" }
```

Fields: `path` (required: absolute path to marketplace.json file)

1. **Directory paths**:

```
{ "source": "directory", "path": "/usr/local/share/claude/acme-plugins" }
{ "source": "directory", "path": "/opt/acme-corp/approved-marketplaces" }
```

Fields: `path` (required: absolute path to directory containing `.claude-plugin/marketplace.json`)

1. **Host pattern matching**:

```
{ "source": "hostPattern", "hostPattern": "^github\\.example\\.com$" }
{ "source": "hostPattern", "hostPattern": "^gitlab\\.internal\\.example\\.com$" }
```

Fields: `hostPattern` (required: regex pattern to match against the marketplace host)

Use host pattern matching when you want to allow all marketplaces from a specific host without enumerating each repository individually. This is useful for organizations with internal GitHub Enterprise or GitLab servers where developers create their own marketplaces.

Host extraction by source type:

- `github` : always matches against `github.com`
- `git` : extracts hostname from the URL (supports both HTTPS and SSH formats)
- `url` : extracts hostname from the URL
- `npm` , `file` , `directory` : not supported for host pattern matching

Configuration examples:

Example: allow specific marketplaces only:

```
{
  "strictKnownMarketplaces": [
    {
      "source": "github",
      "repo": "acme-corp/approved-plugins"
    },
    {
      "source": "github",
      "repo": "acme-corp/security-tools",
      "ref": "v2.0"
    },
    {
      "source": "url",
      "url": "https://plugins.example.com/marketplace.json"
    },
    {
      "source": "npm",
      "package": "@acme-corp/compliance-plugins"
    }
  ]
}
```

Example - Disable all marketplace additions:

```
{
  "strictKnownMarketplaces": []
}
```

Example: allow all marketplaces from an internal git server:

```
{
  "strictKnownMarketplaces": [
    {
      "source": "hostPattern",
      "hostPattern": "^github\\.example\\.com$"
    }
  ]
}
```

Exact matching requirements:

Marketplace sources must match **exactly** for a user's addition to be allowed. For git-based sources (`github` and `git`), this includes all optional fields:

- The `repo` or `url` must match exactly
- The `ref` field must match exactly (or both be undefined)
- The `path` field must match exactly (or both be undefined)

Examples of sources that **do NOT match**:

```
// These are DIFFERENT sources:
{ "source": "github", "repo": "acme-corp/plugins" }
{ "source": "github", "repo": "acme-corp/plugins", "ref": "main" }

// These are also DIFFERENT:
{ "source": "github", "repo": "acme-corp/plugins", "path": "marketplace" }
{ "source": "github", "repo": "acme-corp/plugins" }
```

Comparison with `extraKnownMarketplaces`:

Aspect	`strictKnownMarketplaces`	`extraKnownMarketplaces`
Purpose	Organizational policy enforcement	Team convenience
Settings file	`managed-settings.json` only	Any settings file
Behavior	Blocks non-allowlisted additions	Auto-installs missing marketplaces
When enforced	Before network/filesystem operations	After user trust prompt
Can be overridden	No (highest precedence)	Yes (by higher precedence settings)
Source format	Direct source object	Named marketplace with nested source
Use case	Compliance, security restrictions	Onboarding, standardization

Format difference:

`strictKnownMarketplaces` uses direct source objects:

```
{
  "strictKnownMarketplaces": [
    { "source": "github", "repo": "acme-corp/plugins" }
  ]
}
```

`extraKnownMarketplaces` requires named marketplaces:

```
{
  "extraKnownMarketplaces": {
    "acme-tools": {
      "source": { "source": "github", "repo": "acme-corp/plugins" }
    }
  }
}
```

Using both together:

`strictKnownMarketplaces` is a policy gate: it controls what users may add but does not register any marketplaces. To both restrict and pre-register a marketplace for all users, set both in `managed-settings.json`:

```
{
  "strictKnownMarketplaces": [
    { "source": "github", "repo": "acme-corp/plugins" }
  ],
  "extraKnownMarketplaces": {
    "acme-tools": {
      "source": { "source": "github", "repo": "acme-corp/plugins" }
    }
  }
}
```

With only `strictKnownMarketplaces` set, users can still add the allowed marketplace manually via `/plugin marketplace add`, but it is not available automatically.

Important notes:

- Restrictions are checked BEFORE any network requests or filesystem operations
- When blocked, users see clear error messages indicating the source is blocked by managed policy
- The restriction applies only to adding NEW marketplaces; previously installed marketplaces remain accessible
- Managed settings have the highest precedence and cannot be overridden

See Managed marketplace restrictions for user-facing documentation.

Managing plugins

Use the `/plugin` command to manage plugins interactively:

- Browse available plugins from marketplaces
- Install/uninstall plugins
- Enable/disable plugins
- View plugin details (commands, agents, hooks provided)
- Add/remove marketplaces

Learn more about the plugin system in the plugins documentation.

Environment variables

Environment variables let you control Claude Code behavior without editing settings files. Any variable can also be configured in `settings.json` under the `env` key to apply it to every session or roll it out to your team.

See the environment variables reference for the full list.

Tools available to Claude

Claude Code has access to a set of tools for reading, editing, searching, running commands, and orchestrating subagents. Tool names are the exact strings you use in permission rules and hook matchers.

See the tools reference for the full list and Bash tool behavior details.

See also

- Permissions: permission system, rule syntax, tool-specific patterns, and managed policies
- Authentication: set up user access to Claude Code
- Troubleshooting: solutions for common configuration issues

Environment variables

> *Complete reference for environment variables that control Claude Code behavior.*

Claude Code supports the following environment variables to control its behavior. Set them in your shell before launching `claude`, or configure them in `settings.json` under the `env` key to apply them to every session or roll them out across your team.

Variable	Purpose
`ANTHROPIC_API_KEY`	API key sent as `X-Api-Key` header, typically for the Claude SDK (for interactive usage, run `/login`)
`ANTHROPIC_AUTH_TOKEN`	Custom value for the `Authorization` header (the value you set here will be prefixed with `Bearer`)
`ANTHROPIC_CUSTOM_HEADERS`	Custom headers to add to requests (`Name: Value` format, newline-separated for multiple headers)
`ANTHROPIC_DEFAULT_HAIKU_MODEL`	See Model configuration
`ANTHROPIC_DEFAULT_OPUS_MODEL`	See Model configuration
`ANTHROPIC_DEFAULT_SONNET_MODEL`	See Model configuration
`ANTHROPIC_FOUNDRY_API_KEY`	API key for Microsoft Foundry authentication (see Microsoft Foundry)
`ANTHROPIC_FOUNDRY_BASE_URL`	Full base URL for the Foundry resource (for example, `https://my-resource.services.ai.azure.com/anthropic`). Alternative to `ANTHROPIC_FOUNDRY_RESOURCE` (see Microsoft Foundry)
`ANTHROPIC_FOUNDRY_RESOURCE`	Foundry resource name (for example, `my-resource`). Required if `ANTHROPIC_FOUNDRY_BASE_URL` is not set (see Microsoft Foundry)

Variable	Purpose
`ANTHROPIC_MODEL`	Name of the model setting to use (see Model Configuration)
`ANTHROPIC_SMALL_FAST_MODEL`	[DEPRECATED] Name of Haiku-class model for background tasks
`ANTHROPIC_SMALL_FAST_MODEL_AWS_REGION`	Override AWS region for the Haiku-class model when using Bedrock
`AWS_BEARER_TOKEN_BEDROCK`	Bedrock API key for authentication (see Bedrock API keys)
`BASH_DEFAULT_TIMEOUT_MS`	Default timeout for long-running bash commands
`BASH_MAX_OUTPUT_LENGTH`	Maximum number of characters in bash outputs before they are middle-truncated
`BASH_MAX_TIMEOUT_MS`	Maximum timeout the model can set for long-running bash commands
`CLAUDECODE`	Set to `1` in shell environments Claude Code spawns (Bash tool, tmux sessions). Not set in hooks or status line commands. Use to detect when a script is running inside a shell spawned by Claude Code
`CLAUDE_AUTOCOMPACT_PCT_OVERRIDE`	Set the percentage of context capacity (1-100) at which auto-compaction triggers. By default, auto-compaction triggers at approximately 95% capacity. Use lower values like `50` to compact earlier. Values above the default threshold have no effect. Applies to both main conversations and subagents. This percentage aligns with the `context_window.used_percentage` field available in status line

Variable	Purpose
`CLAUDE_BASH_MAINTAIN_PROJECT_WORKING_DIR`	Return to the original working directory after each Bash command
`CLAUDE_CODE_ACCOUNT_UUID`	Account UUID for the authenticated user. Used by SDK callers to provide account information synchronously, avoiding a race condition where early telemetry events lack account metadata. Requires `CLAUDE_CODE_USER_EMAIL` and `CLAUDE_CODE_ORGANIZATION_UUID` to also be set
`CLAUDE_CODE_ADDITIONAL_DIRECTORIES_CLAUDE_MD`	Set to `1` to load CLAUDE.md files from directories specified with `--add-dir`. By default, additional directories do not load memory files
`CLAUDE_CODE_AUTO_COMPACT_WINDOW`	Set the context capacity in tokens used for auto-compaction calculations. Defaults to the model's context window: 200K for standard models or 1M for extended context models. Use a lower value like `500000` on a 1M model to treat the window as 500K for compaction purposes. The value is capped at the model's actual context window. `CLAUDE_AUTOCOMPACT_PCT_OVERRIDE` is applied as a percentage of this value. Setting this variable decouples the compaction threshold from the status line's `used_percentage`, which always uses the model's full context window
`CLAUDE_CODE_API_KEY_HELPER_TTL_MS`	Interval in milliseconds at which credentials should be refreshed (when using `apiKeyHelper`)
`CLAUDE_CODE_CLIENT_CERT`	Path to client certificate file for mTLS authentication

Variable	Purpose
`CLAUDE_CODE_CLIENT_KEY`	Path to client private key file for mTLS authentication
`CLAUDE_CODE_CLIENT_KEY_PASSPHRASE`	Passphrase for encrypted CLAUDE_CODE_CLIENT_KEY (optional)
`CLAUDE_CODE_DISABLE_1M_CONTEXT`	Set to `1` to disable 1M context window support. When set, 1M model variants are unavailable in the model picker. Useful for enterprise environments with compliance requirements
`CLAUDE_CODE_DISABLE_ADAPTIVE_THINKING`	Set to `1` to disable adaptive reasoning for Opus 4.6 and Sonnet 4.6. When disabled, these models fall back to the fixed thinking budget controlled by `MAX_THINKING_TOKENS`
`CLAUDE_CODE_DISABLE_AUTO_MEMORY`	Set to `1` to disable auto memory. Set to `0` to force auto memory on during the gradual rollout. When disabled, Claude does not create or load auto memory files
`CLAUDE_CODE_DISABLE_GIT_INSTRUCTIONS`	Set to `1` to remove built-in commit and PR workflow instructions from Claude's system prompt. Useful when using your own git workflow skills. Takes precedence over the `includeGitInstructions` setting when set
`CLAUDE_CODE_DISABLE_BACKGROUND_TASKS`	Set to `1` to disable all background task functionality, including the `run_in_background` parameter on Bash and subagent tools, auto-backgrounding, and the Ctrl+B shortcut

Variable	Purpose
`CLAUDE_CODE_DISABLE_CRON`	Set to `1` to disable scheduled tasks. The `/loop` skill and cron tools become unavailable and any already-scheduled tasks stop firing, including tasks that are already running mid-session
`CLAUDE_CODE_DISABLE_EXPERIMENTAL_BETAS`	Set to `1` to disable Anthropic API-specific `anthropic-beta` headers. Use this if experiencing issues like "Unexpected value(s) for the `anthropic-beta` header" when using an LLM gateway with third-party providers
`CLAUDE_CODE_DISABLE_FAST_MODE`	Set to `1` to disable fast mode
`CLAUDE_CODE_DISABLE_FEEDBACK_SURVEY`	Set to `1` to disable the "How is Claude doing?" session quality surveys. Surveys are also disabled when `DISABLE_TELEMETRY` or `CLAUDE_CODE_DISABLE_NONESSENTIAL_TRAFFIC` is set. See Session quality surveys
`CLAUDE_CODE_DISABLE_NONESSENTIAL_TRAFFIC`	Equivalent of setting `DISABLE_AUTOUPDATER`, `DISABLE_BUG_COMMAND`, `DISABLE_ERROR_REPORTING`, and `DISABLE_TELEMETRY`
`CLAUDE_CODE_DISABLE_TERMINAL_TITLE`	Set to `1` to disable automatic terminal title updates based on conversation context
`CLAUDE_CODE_EFFORT_LEVEL`	Set the effort level for supported models. Values: `low`, `medium`, `high`, `max` (Opus 4.6 only), or `auto` to use the model default. Takes precedence over `/effort` and the `effortLevel` setting. See Adjust effort level

Variable	Purpose
`CLAUDE_CODE_ENABLE_PROMPT_SUGGESTION`	Set to `false` to disable prompt suggestions (the "Prompt suggestions" toggle in `/config`). These are the grayed-out predictions that appear in your prompt input after Claude responds. See Prompt suggestions
`CLAUDE_CODE_ENABLE_TASKS`	Set to `true` to enable the task tracking system in non-interactive mode (the `-p` flag). Tasks are on by default in interactive mode. See Task list
`CLAUDE_CODE_ENABLE_TELEMETRY`	Set to `1` to enable OpenTelemetry data collection for metrics and logging. Required before configuring OTel exporters. See Monitoring
`CLAUDE_CODE_EXIT_AFTER_STOP_DELAY`	Time in milliseconds to wait after the query loop becomes idle before automatically exiting. Useful for automated workflows and scripts using SDK mode
`CLAUDE_CODE_EXPERIMENTAL_AGENT_TEAMS`	Set to `1` to enable agent teams. Agent teams are experimental and disabled by default
`CLAUDE_CODE_FILE_READ_MAX_OUTPUT_TOKENS`	Override the default token limit for file reads. Useful when you need to read larger files in full
`CLAUDE_CODE_IDE_SKIP_AUTO_INSTALL`	Skip auto-installation of IDE extensions
`CLAUDE_CODE_MAX_OUTPUT_TOKENS`	Set the maximum number of output tokens for most requests. Default: 32,000. Maximum: 64,000. Increasing this value reduces the effective context window available before auto-compaction triggers.

Variable	Purpose
`CLAUDE_CODE_ORGANIZATION_UUID`	Organization UUID for the authenticated user. Used by SDK callers to provide account information synchronously. Requires `CLAUDE_CODE_ACCOUNT_UUID` and `CLAUDE_CODE_USER_EMAIL` to also be set
`CLAUDE_CODE_OTEL_HEADERS_HELPER_DEBOUNCE_MS`	Interval for refreshing dynamic OpenTelemetry headers in milliseconds (default: 1740000 / 29 minutes). See Dynamic headers
`CLAUDE_CODE_PLAN_MODE_REQUIRED`	Auto-set to `true` on agent team teammates that require plan approval. Read-only: set by Claude Code when spawning teammates. See require plan approval
`CLAUDE_CODE_PLUGIN_GIT_TIMEOUT_MS`	Timeout in milliseconds for git operations when installing or updating plugins (default: 120000). Increase this value for large repositories or slow network connections. See Git operations time out
`CLAUDE_CODE_PROXY_RESOLVES_HOSTS`	Set to `true` to allow the proxy to perform DNS resolution instead of the caller. Opt-in for environments where the proxy should handle hostname resolution
`CLAUDE_CODE_SESSIONEND_HOOKS_TIMEOUT_MS`	Maximum time in milliseconds for SessionEnd hooks to complete (default: `1500`). Applies to both session exit and `/clear`. Per-hook `timeout` values are also capped by this budget
`CLAUDE_CODE_SHELL`	Override automatic shell detection. Useful when your login shell differs from your preferred working shell (for example, `bash` vs `zsh`)

Variable	Purpose
`CLAUDE_CODE_SHELL_PREFIX`	Command prefix to wrap all bash commands (for example, for logging or auditing). Example: `/path/to/logger.sh` will execute `/path/to/logger.sh <command>`
`CLAUDE_CODE_SIMPLE`	Set to `1` to run with a minimal system prompt and only the Bash, file read, and file edit tools. Disables MCP tools, attachments, hooks, and CLAUDE.md files
`CLAUDE_CODE_SKIP_BEDROCK_AUTH`	Skip AWS authentication for Bedrock (for example, when using an LLM gateway)
`CLAUDE_CODE_SKIP_FAST_MODE_NETWORK_ERRORS`	Set to `1` to allow fast mode when the organization status check fails due to a network error. Useful when a corporate proxy blocks the status endpoint. The API still enforces organization-level disable separately
`CLAUDE_CODE_SKIP_FOUNDRY_AUTH`	Skip Azure authentication for Microsoft Foundry (for example, when using an LLM gateway)
`CLAUDE_CODE_SKIP_VERTEX_AUTH`	Skip Google authentication for Vertex (for example, when using an LLM gateway)
`CLAUDE_CODE_SUBAGENT_MODEL`	See Model configuration
`CLAUDE_CODE_TASK_LIST_ID`	Share a task list across sessions. Set the same ID in multiple Claude Code instances to coordinate on a shared task list. See Task list
`CLAUDE_CODE_TEAM_NAME`	Name of the agent team this teammate belongs to. Set automatically on agent team members

Variable	Purpose
`CLAUDE_CODE_TMPDIR`	Override the temp directory used for internal temp files. Claude Code appends `/claude/` to this path. Default: `/tmp` on Unix/macOS, `os.tmpdir()` on Windows
`CLAUDE_CODE_USER_EMAIL`	Email address for the authenticated user. Used by SDK callers to provide account information synchronously. Requires `CLAUDE_CODE_ACCOUNT_UUID` and `CLAUDE_CODE_ORGANIZATION_UUID` to also be set
`CLAUDE_CODE_USE_BEDROCK`	Use Bedrock
`CLAUDE_CODE_USE_FOUNDRY`	Use Microsoft Foundry
`CLAUDE_CODE_USE_VERTEX`	Use Vertex
`CLAUDE_CONFIG_DIR`	Customize where Claude Code stores its configuration and data files
`CLAUDE_ENV_FILE`	Path to a shell script that Claude Code sources before each Bash command. Use to persist virtualenv or conda activation across commands. Also populated dynamically by SessionStart hooks
`DISABLE_AUTOUPDATER`	Set to `1` to disable automatic updates.
`DISABLE_BUG_COMMAND`	Set to `1` to disable the `/bug` command
`DISABLE_COST_WARNINGS`	Set to `1` to disable cost warning messages
`DISABLE_ERROR_REPORTING`	Set to `1` to opt out of Sentry error reporting

Variable	Purpose
`DISABLE_INSTALLATION_CHECKS`	Set to `1` to disable installation warnings. Use only when manually managing the installation location, as this can mask issues with standard installations
`DISABLE_PROMPT_CACHING`	Set to `1` to disable prompt caching for all models (takes precedence over per-model settings)
`DISABLE_PROMPT_CACHING_HAIKU`	Set to `1` to disable prompt caching for Haiku models
`DISABLE_PROMPT_CACHING_OPUS`	Set to `1` to disable prompt caching for Opus models
`DISABLE_PROMPT_CACHING_SONNET`	Set to `1` to disable prompt caching for Sonnet models
`DISABLE_TELEMETRY`	Set to `1` to opt out of Statsig telemetry (note that Statsig events do not include user data like code, file paths, or bash commands)
`ENABLE_CLAUDEAI_MCP_SERVERS`	Set to `false` to disable claude.ai MCP servers in Claude Code. Enabled by default for logged-in users
`ENABLE_TOOL_SEARCH`	Controls MCP tool search. Unset: enabled by default, but disabled when `ANTHROPIC_BASE_URL` points to a non-first-party host. Values: `true` (always on including proxies), `auto` (enables at 10% context), `auto:N` (custom threshold, e.g., `auto:5` for 5%), `false` (disabled)
`FORCE_AUTOUPDATE_PLUGINS`	Set to `true` to force plugin auto-updates even when the main auto-updater is disabled via `DISABLE_AUTOUPDATER`

Variable	Purpose
`HTTP_PROXY`	Specify HTTP proxy server for network connections
`HTTPS_PROXY`	Specify HTTPS proxy server for network connections
`IS_DEMO`	Set to `true` to enable demo mode: hides email and organization from the UI, skips onboarding, and hides internal commands. Useful for streaming or recording sessions
`MAX_MCP_OUTPUT_TOKENS`	Maximum number of tokens allowed in MCP tool responses. Claude Code displays a warning when output exceeds 10,000 tokens (default: 25000)
`MAX_THINKING_TOKENS`	Override the extended thinking token budget. Thinking is enabled at max budget (31,999 tokens) by default. Use this to limit the budget (for example, `MAX_THINKING_TOKENS=10000`) or disable thinking entirely (`MAX_THINKING_TOKENS=0`). For Opus 4.6, thinking depth is controlled by effort level instead, and this variable is ignored unless set to `0` to disable thinking.
`MCP_CLIENT_SECRET`	OAuth client secret for MCP servers that require pre-configured credentials. Avoids the interactive prompt when adding a server with `--client-secret`
`MCP_OAUTH_CALLBACK_PORT`	Fixed port for the OAuth redirect callback, as an alternative to `--callback-port` when adding an MCP server with pre-configured credentials
`MCP_TIMEOUT`	Timeout in milliseconds for MCP server startup

Variable	Purpose
`MCP_TOOL_TIMEOUT`	Timeout in milliseconds for MCP tool execution
`NO_PROXY`	List of domains and IPs to which requests will be directly issued, bypassing proxy
`SLASH_COMMAND_TOOL_CHAR_BUDGET`	Override the character budget for skill metadata shown to the Skill tool. The budget scales dynamically at 2% of the context window, with a fallback of 16,000 characters. Legacy name kept for backwards compatibility
`USE_BUILTIN_RIPGREP`	Set to `0` to use system-installed `rg` instead of `rg` included with Claude Code
`VERTEX_REGION_CLAUDE_3_5_HAIKU`	Override region for Claude 3.5 Haiku when using Vertex AI
`VERTEX_REGION_CLAUDE_3_7_SONNET`	Override region for Claude 3.7 Sonnet when using Vertex AI
`VERTEX_REGION_CLAUDE_4_0_OPUS`	Override region for Claude 4.0 Opus when using Vertex AI
`VERTEX_REGION_CLAUDE_4_0_SONNET`	Override region for Claude 4.0 Sonnet when using Vertex AI
`VERTEX_REGION_CLAUDE_4_1_OPUS`	Override region for Claude 4.1 Opus when using Vertex AI

See also

- Settings: configure environment variables in `settings.json` so they apply to every session
- CLI reference: launch-time flags
- Network configuration: proxy and TLS setup

Model configuration

> *Learn about the Claude Code model configuration, including model aliases like* `opusplan`

Available models

For the `model` setting in Claude Code, you can configure either:

- A **model alias**
- A **model name**
  - Anthropic API: A full **model name**
  - Bedrock: an inference profile ARN
  - Foundry: a deployment name
  - Vertex: a version name

Model aliases

Model aliases provide a convenient way to select model settings without remembering exact version numbers:

Model alias	Behavior
`default`	Recommended model setting, depending on your account type
`sonnet`	Uses the latest Sonnet model (currently Sonnet 4.6) for daily coding tasks
`opus`	Uses the latest Opus model (currently Opus 4.6) for complex reasoning tasks
`haiku`	Uses the fast and efficient Haiku model for simple tasks
`sonnet[1m]`	Uses Sonnet with a 1 million token context window for long sessions
`opus[1m]`	Uses Opus with a 1 million token context window for long sessions
`opusplan`	Special mode that uses `opus` during plan mode, then switches to `sonnet` for execution

Aliases always point to the latest version. To pin to a specific version, use the full model name (for example, `claude-opus-4-6`) or set the corresponding environment variable like `ANTHROPIC_DEFAULT_OPUS_MODEL`.

Setting your model

You can configure your model in several ways, listed in order of priority:

1. **During session** - Use `/model <alias|name>` to switch models mid-session
2. **At startup** - Launch with `claude --model <alias|name>`
3. **Environment variable** - Set `ANTHROPIC_MODEL=<alias|name>`
4. **Settings** - Configure permanently in your settings file using the `model` field.

Example usage:

```
## Start with Opus
claude --model opus

## Switch to Sonnet during session
/model sonnet
```

Example settings file:

```
{
    "permissions": {
        ...
    },
    "model": "opus"
}
```

Restrict model selection

Enterprise administrators can use `availableModels` in managed or policy settings to restrict which models users can select.

When `availableModels` is set, users cannot switch to models not in the list via `/model`, `--model` flag, Config tool, or `ANTHROPIC_MODEL` environment variable.

```
{
  "availableModels": ["sonnet", "haiku"]
}
```

Default model behavior

The Default option in the model picker is not affected by `availableModels`. It always remains available and represents the system's runtime default based on the user's subscription tier.

Even with `availableModels: []`, users can still use Claude Code with the Default model for their tier.

Control the model users run on

To fully control the model experience, use `availableModels` together with the `model` setting:

- **availableModels**: restricts what users can switch to
- **model**: sets the explicit model override, taking precedence over the Default

This example ensures all users run Sonnet 4.6 and can only choose between Sonnet and Haiku:

```
{
  "model": "sonnet",
  "availableModels": ["sonnet", "haiku"]
}
```

Merge behavior

When `availableModels` is set at multiple levels, such as user settings and project settings, arrays are merged and deduplicated. To enforce a strict allowlist, set `availableModels` in managed or policy settings which take highest priority.

Special model behavior

`default` model setting

The behavior of `default` depends on your account type:

- **Max and Team Premium**: defaults to Opus 4.6

- **Pro and Team Standard**: defaults to Sonnet 4.6
- **Enterprise**: Opus 4.6 is available but not the default

Claude Code may automatically fall back to Sonnet if you hit a usage threshold with Opus.

`opusplan` model setting

The `opusplan` model alias provides an automated hybrid approach:

- **In plan mode** - Uses `opus` for complex reasoning and architecture decisions
- **In execution mode** - Automatically switches to `sonnet` for code generation and implementation

This gives you the best of both worlds: Opus's superior reasoning for planning, and Sonnet's efficiency for execution.

Adjust effort level

Effort levels control adaptive reasoning, which dynamically allocates thinking based on task complexity. Lower effort is faster and cheaper for straightforward tasks, while higher effort provides deeper reasoning for complex problems.

Three levels persist across sessions: **low**, **medium**, and **high**. A fourth level, **max**, provides the deepest reasoning with no constraint on token spending, so responses are slower and cost more than at `high`. `max` is available on Opus 4.6 only and applies to the current session without persisting. Opus 4.6 defaults to medium effort for Max and Team subscribers.

Setting effort:

- `/effort`: run `/effort low`, `/effort medium`, `/effort high`, or `/effort max` to change the level, or `/effort auto` to reset to the model default
- **In `/model`**: use left/right arrow keys to adjust the effort slider when selecting a model
- **`--effort` flag**: pass `low`, `medium`, `high`, or `max` to set the level for a single session when launching Claude Code
- **Environment variable**: set `CLAUDE_CODE_EFFORT_LEVEL` to `low`, `medium`, `high`, `max`, or `auto`
- **Settings**: set `effortLevel` in your settings file to `"low"`, `"medium"`, or `"high"`

The environment variable takes precedence, then your configured level, then the model default.

Effort is supported on Opus 4.6 and Sonnet 4.6. The effort slider appears in `/model` when a supported model is selected. The current effort level is also displayed next to the logo and spinner, for example "with low effort", so you can confirm which setting is active without opening `/model`.

To disable adaptive reasoning on Opus 4.6 and Sonnet 4.6 and revert to the previous fixed thinking budget, set `CLAUDE_CODE_DISABLE_ADAPTIVE_THINKING=1`. When disabled, these models use the fixed budget controlled by `MAX_THINKING_TOKENS`. See environment variables.

Extended context

Opus 4.6 and Sonnet 4.6 support a 1 million token context window for long sessions with large codebases.

Availability varies by model and plan. On Max, Team, and Enterprise plans, Opus is automatically upgraded to 1M context with no additional configuration. This applies to both Team Standard and Team Premium seats.

Plan	Opus 4.6 with 1M context	Sonnet 4.6 with 1M context
Max, Team, and Enterprise	Included with subscription	Requires extra usage
Pro	Requires extra usage	Requires extra usage
API and pay-as-you-go	Full access	Full access

To disable 1M context entirely, set `CLAUDE_CODE_DISABLE_1M_CONTEXT=1`. This removes 1M model variants from the model picker. See environment variables.

The 1M context window uses standard model pricing with no premium for tokens beyond 200K. For plans where extended context is included with your subscription, usage remains covered by your subscription. For plans that access extended context through extra usage, tokens are billed to extra usage.

If your account supports 1M context, the option appears in the model picker (`/model`) in the latest versions of Claude Code. If you don't see it, try restarting your session.

You can also use the `[1m]` suffix with model aliases or full model names:

```
## Use the opus[1m] or sonnet[1m] alias
/model opus[1m]
/model sonnet[1m]

## Or append [1m] to a full model name
/model claude-opus-4-6[1m]
```

Checking your current model

You can see which model you're currently using in several ways:

1. In status line (if configured)
2. In `/status`, which also displays your account information.

Environment variables

You can use the following environment variables, which must be full **model names** (or equivalent for your API provider), to control the model names that the aliases map to.

Environment variable	Description
`ANTHROPIC_DEFAULT_OPUS_MODEL`	The model to use for `opus`, or for `opusplan` when Plan Mode is active.
`ANTHROPIC_DEFAULT_SONNET_MODEL`	The model to use for `sonnet`, or for `opusplan` when Plan Mode is not active.
`ANTHROPIC_DEFAULT_HAIKU_MODEL`	The model to use for `haiku`, or background functionality
`CLAUDE_CODE_SUBAGENT_MODEL`	The model to use for subagents

Note: `ANTHROPIC_SMALL_FAST_MODEL` is deprecated in favor of `ANTHROPIC_DEFAULT_HAIKU_MODEL`.

Pin models for third-party deployments

When deploying Claude Code through Bedrock, Vertex AI, or Foundry, pin model versions before rolling out to users.

Without pinning, Claude Code uses model aliases (`sonnet`, `opus`, `haiku`) that resolve to the latest version. When Anthropic releases a new model, users whose accounts don't have the new version enabled will break silently.

> **Warning:**
>
> Set all three model environment variables to specific version IDs as part of your initial setup. Skipping this step means a Claude Code update can break your users without any action on your part.

Use the following environment variables with version-specific model IDs for your provider:

Provider	Example
Bedrock	`export ANTHROPIC_DEFAULT_OPUS_MODEL='us.anthropic.claude-opus-4-6-v1'`
Vertex AI	`export ANTHROPIC_DEFAULT_OPUS_MODEL='claude-opus-4-6'`
Foundry	`export ANTHROPIC_DEFAULT_OPUS_MODEL='claude-opus-4-6'`

Apply the same pattern for `ANTHROPIC_DEFAULT_SONNET_MODEL` and `ANTHROPIC_DEFAULT_HAIKU_MODEL`. For current and legacy model IDs across all providers, see Models overview. To upgrade users to a new model version, update these environment variables and redeploy.

To enable extended context for a pinned model, append `[1m]` to the model ID in `ANTHROPIC_DEFAULT_OPUS_MODEL` or `ANTHROPIC_DEFAULT_SONNET_MODEL`:

```
export ANTHROPIC_DEFAULT_OPUS_MODEL='claude-opus-4-6[1m]'
```

The `[1m]` suffix applies the 1M context window to all usage of that alias, including `opusplan`. Claude Code strips the suffix before sending the model ID to your provider. Only append `[1m]` when the underlying model supports 1M context, such as Opus 4.6 or Sonnet 4.6.

> **Note:**
>
> The `settings.availableModels` allowlist still applies when using third-party providers. Filtering matches on the model alias (`opus` , `sonnet` , `haiku`), not the provider-specific model ID.

Override model IDs per version

The family-level environment variables above configure one model ID per family alias. If you need to map several versions within the same family to distinct provider IDs, use the `modelOverrides` setting instead.

`modelOverrides` maps individual Anthropic model IDs to the provider-specific strings that Claude Code sends to your provider's API. When a user selects a mapped model in the `/model` picker, Claude Code uses your configured value instead of the built-in default.

This lets enterprise administrators route each model version to a specific Bedrock inference profile ARN, Vertex AI version name, or Foundry deployment name for governance, cost allocation, or regional routing.

Set `modelOverrides` in your settings file:

```
{
  "modelOverrides": {
    "claude-opus-4-6": "arn:aws:bedrock:us-east-2:123456789012:application-inference-profile/opus-prod",
    "claude-opus-4-5-20251101": "arn:aws:bedrock:us-east-2:123456789012:application-inference-profile/opus-45-prod",
    "claude-sonnet-4-6": "arn:aws:bedrock:us-east-2:123456789012:application-inference-profile/sonnet-prod"
  }
}
```

Keys must be Anthropic model IDs as listed in the Models overview. For dated model IDs, include the date suffix exactly as it appears there. Unknown keys are ignored.

Overrides replace the built-in model IDs that back each entry in the `/model` picker. On Bedrock, overrides take precedence over any inference profiles that Claude Code discovers automatically at startup. Values you supply directly through `ANTHROPIC_MODEL`, `--model`, or the `ANTHROPIC_DEFAULT_*_MODEL` environment variables are passed to the provider as-is and are not transformed by `modelOverrides`.

`modelOverrides` works alongside `availableModels`. The allowlist is evaluated against the Anthropic model ID, not the override value, so an entry like `"opus"` in `availableModels` continues to match even when Opus versions are mapped to ARNs.

Prompt caching configuration

Claude Code automatically uses prompt caching to optimize performance and reduce costs. You can disable prompt caching globally or for specific model tiers:

Environment variable	Description
`DISABLE_PROMPT_CACHING`	Set to `1` to disable prompt caching for all models (takes precedence over per-model settings)
`DISABLE_PROMPT_CACHING_HAIKU`	Set to `1` to disable prompt caching for Haiku models only
`DISABLE_PROMPT_CACHING_SONNET`	Set to `1` to disable prompt caching for Sonnet models only
`DISABLE_PROMPT_CACHING_OPUS`	Set to `1` to disable prompt caching for Opus models only

These environment variables give you fine-grained control over prompt caching behavior. The global `DISABLE_PROMPT_CACHING` setting takes precedence over the model-specific settings, allowing you to quickly disable all caching when needed. The per-model settings are useful for selective control, such as when debugging specific models or working with cloud providers that may have different caching implementations.

Output styles

> *Adapt Claude Code for uses beyond software engineering*

Output styles allow you to use Claude Code as any type of agent while keeping its core capabilities, such as running local scripts, reading/writing files, and tracking TODOs.

Built-in output styles

Claude Code's **Default** output style is the existing system prompt, designed to help you complete software engineering tasks efficiently.

There are two additional built-in output styles focused on teaching you the codebase and how Claude operates:

- **Explanatory**: Provides educational "Insights" in between helping you complete software engineering tasks. Helps you understand implementation choices and codebase patterns.
- **Learning**: Collaborative, learn-by-doing mode where Claude will not only share "Insights" while coding, but also ask you to contribute small, strategic pieces of code yourself. Claude Code will add `TODO(human)` markers in your code for you to implement.

How output styles work

Output styles directly modify Claude Code's system prompt.

- All output styles exclude instructions for efficient output (such as responding concisely).
- Custom output styles exclude instructions for coding (such as verifying code with tests), unless `keep-coding-instructions` is true.
- All output styles have their own custom instructions added to the end of the system prompt.
- All output styles trigger reminders for Claude to adhere to the output style instructions during the conversation.

Change your output style

Run `/config` and select **Output style** to pick a style from a menu. Your selection is saved to `.claude/settings.local.json` at the local project level.

To set a style without the menu, edit the `outputStyle` field directly in a settings file:

```
{
  "outputStyle": "Explanatory"
}
```

Because the output style is set in the system prompt at session start, changes take effect the next time you start a new session. This keeps the system prompt stable throughout a conversation so prompt caching can reduce latency and cost.

Create a custom output style

Custom output styles are Markdown files with frontmatter and the text that will be added to the system prompt:

```
---
name: My Custom Style
description:
  A brief description of what this style does, to be displayed to the user
---

## Custom Style Instructions

You are an interactive CLI tool that helps users with software engineering
tasks. [Your custom instructions here...]

### Specific Behaviors

[Define how the assistant should behave in this style...]
```

You can save these files at the user level (`~/.claude/output-styles`) or project level (`.claude/output-styles`).

Frontmatter

Output style files support frontmatter for specifying metadata:

Frontmatter	Purpose	Default
`name`	Name of the output style, if not the file name	Inherits from file name
`description`	Description of the output style, shown in the `/config` picker	None
`keep-coding-instructions`	Whether to keep the parts of Claude Code's system prompt related to coding.	false

Comparisons to related features

Output Styles vs. CLAUDE.md vs. –append-system-prompt

Output styles completely "turn off" the parts of Claude Code's default system prompt specific to software engineering. Neither CLAUDE.md nor `--append-system-prompt` edit Claude Code's default system prompt. CLAUDE.md adds the contents as a user message *following* Claude Code's default system prompt. `--append-system-prompt` appends the content to the system prompt.

Output Styles vs. Agents

Output styles directly affect the main agent loop and only affect the system prompt. Agents are invoked to handle specific tasks and can include additional settings like the model to use, the tools they have available, and some context about when to use the agent.

Output Styles vs. Skills

Output styles modify how Claude responds (formatting, tone, structure) and are always active once selected. Skills are task-specific prompts that you invoke with `/skill-name` or that Claude loads automatically when relevant. Use output styles for consistent formatting preferences; use skills for reusable workflows and tasks.

Speed up responses with fast mode

> *Get faster Opus 4.6 responses in Claude Code by toggling fast mode.*

Note:

Fast mode is in research preview. The feature, pricing, and availability may change based on feedback.

Fast mode is a high-speed configuration for Claude Opus 4.6, making the model 2.5x faster at a higher cost per token. Toggle it on with `/fast` when you need speed for interactive work like rapid iteration or live debugging, and toggle it off when cost matters more than latency.

Fast mode is not a different model. It uses the same Opus 4.6 with a different API configuration that prioritizes speed over cost efficiency. You get identical quality and capabilities, just faster responses.

Note:

Fast mode requires Claude Code v2.1.36 or later. Check your version with `claude --version`.

What to know:

- Use `/fast` to toggle on fast mode in Claude Code CLI. Also available via `/fast` in Claude Code VS Code Extension.
- Fast mode for Opus 4.6 pricing is $30/150 MTok.
- Available to all Claude Code users on subscription plans (Pro/Max/Team/Enterprise) and Claude Console.
- For Claude Code users on subscription plans (Pro/Max/Team/Enterprise), fast mode is available via extra usage only and not included in the subscription rate limits.

This page covers how to toggle fast mode, its cost tradeoff, when to use it, requirements, per-session opt-in, and rate limit behavior.

Toggle fast mode

Toggle fast mode in either of these ways:

- Type `/fast` and press Tab to toggle on or off
- Set `"fastMode": true` in your user settings file

By default, fast mode persists across sessions. Administrators can configure fast mode to reset each session. See require per-session opt-in for details.

For the best cost efficiency, enable fast mode at the start of a session rather than switching mid-conversation. See understand the cost tradeoff for details.

When you enable fast mode:

- If you're on a different model, Claude Code automatically switches to Opus 4.6
- You'll see a confirmation message: "Fast mode ON"
- A small `↯` icon appears next to the prompt while fast mode is active
- Run `/fast` again at any time to check whether fast mode is on or off

When you disable fast mode with `/fast` again, you remain on Opus 4.6. The model does not revert to your previous model. To switch to a different model, use `/model`.

Understand the cost tradeoff

Fast mode has higher per-token pricing than standard Opus 4.6:

Mode	Input (MTok)	Output (MTok)
Fast mode on Opus 4.6	$30	$150

Fast mode pricing is flat across the full 1M token context window.

When you switch into fast mode mid-conversation, you pay the full fast mode uncached input token price for the entire conversation context. This costs more than if you had enabled fast mode from the start.

Decide when to use fast mode

Fast mode is best for interactive work where response latency matters more than cost:

- Rapid iteration on code changes
- Live debugging sessions
- Time-sensitive work with tight deadlines

Standard mode is better for:

- Long autonomous tasks where speed matters less
- Batch processing or CI/CD pipelines
- Cost-sensitive workloads

Fast mode vs effort level

Fast mode and effort level both affect response speed, but differently:

Setting	Effect
Fast mode	Same model quality, lower latency, higher cost
Lower effort level	Less thinking time, faster responses, potentially lower quality on complex tasks

You can combine both: use fast mode with a lower effort level for maximum speed on straightforward tasks.

Requirements

Fast mode requires all of the following:

- **Not available on third-party cloud providers**: fast mode is not available on Amazon Bedrock, Google Vertex AI, or Microsoft Azure Foundry. Fast mode is available through the Anthropic Console API and for Claude subscription plans using extra usage.
- **Extra usage enabled**: your account must have extra usage enabled, which allows billing beyond your plan's included usage. For individual accounts, enable this in your Console billing settings. For Teams and Enterprise, an admin must enable extra usage for the organization.

> **Note:**
>
> Fast mode usage is billed directly to extra usage, even if you have remaining usage on your plan. This means fast mode tokens do not count against your plan's included usage and are charged at the fast mode rate from the first token.

- **Admin enablement for Teams and Enterprise**: fast mode is disabled by default for Teams and Enterprise organizations. An admin must explicitly enable fast mode before users can access it.

> **Note:**
>
> If your admin has not enabled fast mode for your organization, the `/fast` command will show "Fast mode has been disabled by your organization."

Enable fast mode for your organization

Admins can enable fast mode in:

- **Console** (API customers): Claude Code preferences
- **Claude AI** (Teams and Enterprise): Admin Settings > Claude Code

Another option to disable fast mode entirely is to set `CLAUDE_CODE_DISABLE_FAST_MODE=1`. See Environment variables.

Require per-session opt-in

By default, fast mode persists across sessions: if a user enables fast mode, it stays on in future sessions. Administrators on Teams or Enterprise plans can prevent this by setting `fastModePerSessionOptIn` to `true` in managed settings or server-managed settings. This causes each session to start with fast mode off, requiring users to explicitly enable it with `/fast`.

```
{
  "fastModePerSessionOptIn": true
}
```

This is useful for controlling costs in organizations where users run multiple concurrent sessions. Users can still enable fast mode with `/fast` when they need speed, but it resets at the start of each new session. The user's fast mode preference is still saved, so removing this setting restores the default persistent behavior.

Handle rate limits

Fast mode has separate rate limits from standard Opus 4.6. When you hit the fast mode rate limit or run out of extra usage credits:

1. Fast mode automatically falls back to standard Opus 4.6
2. The `↯` icon turns gray to indicate cooldown
3. You continue working at standard speed and pricing
4. When the cooldown expires, fast mode automatically re-enables

To disable fast mode manually instead of waiting for cooldown, run `/fast` again.

Research preview

Fast mode is a research preview feature. This means:

- The feature may change based on feedback
- Availability and pricing are subject to change
- The underlying API configuration may evolve

Report issues or feedback through your usual Anthropic support channels.

See also

- Model configuration: switch models and adjust effort levels
- Manage costs effectively: track token usage and reduce costs
- Status line configuration: display model and context information

Manage costs effectively

> *Track token usage, set team spend limits, and reduce Claude Code costs with context management, model selection, extended thinking settings, and preprocessing hooks.*

Claude Code consumes tokens for each interaction. Costs vary based on codebase size, query complexity, and conversation length. The average cost is $6 per developer per day, with daily costs remaining below $12 for 90% of users.

For team usage, Claude Code charges by API token consumption. On average, Claude Code costs ~$100-200/developer per month with Sonnet 4.6 though there is large variance depending on how many instances users are running and whether they're using it in automation.

This page covers how to track your costs, manage costs for teams, and reduce token usage.

Track your costs

Using the `/cost` command

> **Note:**
>
> The `/cost` command shows API token usage and is intended for API users. Claude Max and Pro subscribers have usage included in their subscription, so `/cost` data isn't relevant for billing purposes. Subscribers can use `/stats` to view usage patterns.

The `/cost` command provides detailed token usage statistics for your current session:

```
Total cost:            $0.55
Total duration (API):  6m 19.7s
Total duration (wall): 6h 33m 10.2s
Total code changes:    0 lines added, 0 lines removed
```

Managing costs for teams

When using Claude API, you can set workspace spend limits on the total Claude Code workspace spend. Admins can view cost and usage reporting in the Console.

> **Note:**
>
> When you first authenticate Claude Code with your Claude Console account, a workspace called "Claude Code" is automatically created for you. This workspace provides centralized cost tracking and management for all Claude Code usage in your organization. You cannot create API keys for this workspace; it is exclusively for Claude Code authentication and usage.

On Bedrock, Vertex, and Foundry, Claude Code does not send metrics from your cloud. To get cost metrics, several large enterprises reported using LiteLLM, which is an open-source tool that helps companies track spend by key. This project is unaffiliated with Anthropic and has not been audited for security.

Rate limit recommendations

When setting up Claude Code for teams, consider these Token Per Minute (TPM) and Request Per Minute (RPM) per-user recommendations based on your organization size:

Team size	TPM per user	RPM per user
1-5 users	200k-300k	5-7
5-20 users	100k-150k	2.5-3.5
20-50 users	50k-75k	1.25-1.75
50-100 users	25k-35k	0.62-0.87
100-500 users	15k-20k	0.37-0.47
500+ users	10k-15k	0.25-0.35

For example, if you have 200 users, you might request 20k TPM for each user, or 4 million total TPM (200*20,000 = 4 million).

The TPM per user decreases as team size grows because fewer users tend to use Claude Code concurrently in larger organizations. These rate limits apply at the organization level, not per individual user, which means individual users can temporarily consume more than their calculated share when others aren't actively using the service.

> **Note:**
>
> If you anticipate scenarios with unusually high concurrent usage (such as live training sessions with large groups), you may need higher TPM allocations per user.

Agent team token costs

Agent teams spawn multiple Claude Code instances, each with its own context window. Token usage scales with the number of active teammates and how long each one runs.

To keep agent team costs manageable:

- Use Sonnet for teammates. It balances capability and cost for coordination tasks.
- Keep teams small. Each teammate runs its own context window, so token usage is roughly proportional to team size.
- Keep spawn prompts focused. Teammates load CLAUDE.md, MCP servers, and skills automatically, but everything in the spawn prompt adds to their context from the start.
- Clean up teams when work is done. Active teammates continue consuming tokens even if idle.
- Agent teams are disabled by default. Set `CLAUDE_CODE_EXPERIMENTAL_AGENT_TEAMS=1` in your settings.json or environment to enable them. See enable agent teams.

Reduce token usage

Token costs scale with context size: the more context Claude processes, the more tokens you use. Claude Code automatically optimizes costs through prompt caching (which reduces costs for repeated content like system prompts) and auto-compaction (which summarizes conversation history when approaching context limits).

The following strategies help you keep context small and reduce per-message costs.

Manage context proactively

Use `/cost` to check your current token usage, or configure your status line to display it continuously.

- **Clear between tasks**: Use `/clear` to start fresh when switching to unrelated work. Stale context wastes tokens on every subsequent message. Use `/rename` before clearing so you can easily find the session later, then `/resume` to return to it.
- **Add custom compaction instructions**: `/compact Focus on code samples and API usage` tells Claude what to preserve during summarization.

You can also customize compaction behavior in your CLAUDE.md:

```
## Compact instructions

When you are using compact, please focus on test output and code changes
```

Choose the right model

Sonnet handles most coding tasks well and costs less than Opus. Reserve Opus for complex architectural decisions or multi-step reasoning. Use `/model` to switch models mid-session, or set a default in `/config`. For simple subagent tasks, specify `model: haiku` in your subagent configuration.

Reduce MCP server overhead

Each MCP server adds tool definitions to your context, even when idle. Run `/context` to see what's consuming space.

- **Prefer CLI tools when available**: Tools like `gh`, `aws`, `gcloud`, and `sentry-cli` are more context-efficient than MCP servers because they don't add persistent tool definitions. Claude can run CLI commands directly without the overhead.
- **Disable unused servers**: Run `/mcp` to see configured servers and disable any you're not actively using.
- **Tool search is automatic**: When MCP tool descriptions exceed 10% of your context window, Claude Code automatically defers them and loads tools on-demand via tool search. Since deferred tools only enter context when actually used, a lower threshold means fewer idle tool definitions consuming space. Set a lower threshold with `ENABLE_TOOL_SEARCH=auto:<N>` (for example, `auto:5` triggers when tools exceed 5% of your context window).

Install code intelligence plugins for typed languages

Code intelligence plugins give Claude precise symbol navigation instead of text-based search, reducing unnecessary file reads when exploring unfamiliar code. A single "go to definition" call replaces what might otherwise be a grep followed by reading multiple candidate files. Installed language servers also report type errors automatically after edits, so Claude catches mistakes without running a compiler.

Offload processing to hooks and skills

Custom hooks can preprocess data before Claude sees it. Instead of Claude reading a 10,000-line log file to find errors, a hook can grep for `ERROR` and return only matching lines, reducing context from tens of thousands of tokens to hundreds.

A skill can give Claude domain knowledge so it doesn't have to explore. For example, a "codebase-overview" skill could describe your project's architecture, key directories, and naming conventions. When Claude invokes the skill, it gets this context immediately instead of spending tokens reading multiple files to understand the structure.

For example, this PreToolUse hook filters test output to show only failures:

settings.json

Add this to your settings.json to run the hook before every Bash command:

```
{
  "hooks": {
    "PreToolUse": [
      {
        "matcher": "Bash",
        "hooks": [
          {
            "type": "command",
            "command": "~/.claude/hooks/filter-test-output.sh"
          }
        ]
      }
    ]
  }
}
```

filter-test-output.sh

The hook calls this script, which checks if the command is a test runner and modifies it to show only failures:

```
#!/bin/bash
input=$(cat)
cmd=$(echo "$input" | jq -r '.tool_input.command')

## If running tests, filter to show only failures
if [[ "$cmd" =~ ^(npm test|pytest|go test) ]]; then
  filtered_cmd="$cmd 2>&1 | grep -A 5 -E '(FAIL|ERROR|error:)' | head -100"
  echo "{\"hookSpecificOutput\":{\"hookEventName\":\"PreToolUse\",\"permissionDeci
sion\":\"allow\",\"updatedInput\":{\"command\":\"$filtered_cmd\"}}}"
else
  echo "{}"
fi
```

Move instructions from CLAUDE.md to skills

Your CLAUDE.md file is loaded into context at session start. If it contains detailed instructions for specific workflows (like PR reviews or database migrations), those tokens are present even when you're doing unrelated work. Skills load on-demand only when invoked, so moving specialized instructions into skills keeps your base context smaller. Aim to keep CLAUDE.md under ~500 lines by including only essentials.

Adjust extended thinking

Extended thinking is enabled by default with a budget of 31,999 tokens because it significantly improves performance on complex planning and reasoning tasks. However, thinking tokens are billed as output tokens, so for simpler tasks where deep reasoning isn't needed, you can reduce costs by lowering the effort level with `/effort` or in `/model`, disabling thinking in `/config`, or lowering the budget (for example, `MAX_THINKING_TOKENS=8000`).

Delegate verbose operations to subagents

Running tests, fetching documentation, or processing log files can consume significant context. Delegate these to subagents so the verbose output stays in the subagent's context while only a summary returns to your main conversation.

Manage agent team costs

Agent teams use approximately 7x more tokens than standard sessions when teammates run in plan mode, because each teammate maintains its own context window and runs as a separate Claude instance. Keep team tasks small and self-contained to limit per-teammate token usage. See agent teams for details.

Write specific prompts

Vague requests like “improve this codebase” trigger broad scanning. Specific requests like “add input validation to the login function in auth.ts” let Claude work efficiently with minimal file reads.

Work efficiently on complex tasks

For longer or more complex work, these habits help avoid wasted tokens from going down the wrong path:

- **Use plan mode for complex tasks**: Press Shift+Tab to enter plan mode before implementation. Claude explores the codebase and proposes an approach for your approval, preventing expensive re-work when the initial direction is wrong.
- **Course-correct early**: If Claude starts heading the wrong direction, press Escape to stop immediately. Use `/rewind` or double-tap Escape to restore conversation and code to a previous checkpoint.
- **Give verification targets**: Include test cases, paste screenshots, or define expected output in your prompt. When Claude can verify its own work, it catches issues before you need to request fixes.
- **Test incrementally**: Write one file, test it, then continue. This catches issues early when they’re cheap to fix.

Background token usage

Claude Code uses tokens for some background functionality even when idle:

- **Conversation summarization**: Background jobs that summarize previous conversations for the `claude --resume` feature
- **Command processing**: Some commands like `/cost` may generate requests to check status

These background processes consume a small amount of tokens (typically under $0.04 per session) even without active interaction.

Understanding changes in Claude Code behavior

Claude Code regularly receives updates that may change how features work, including cost reporting. Run `claude --version` to check your current version. For specific billing questions, contact Anthropic support through your Console account. For team deployments, start with a small pilot group to establish usage patterns before wider rollout.

Part 5: Extensibility

Hooks reference

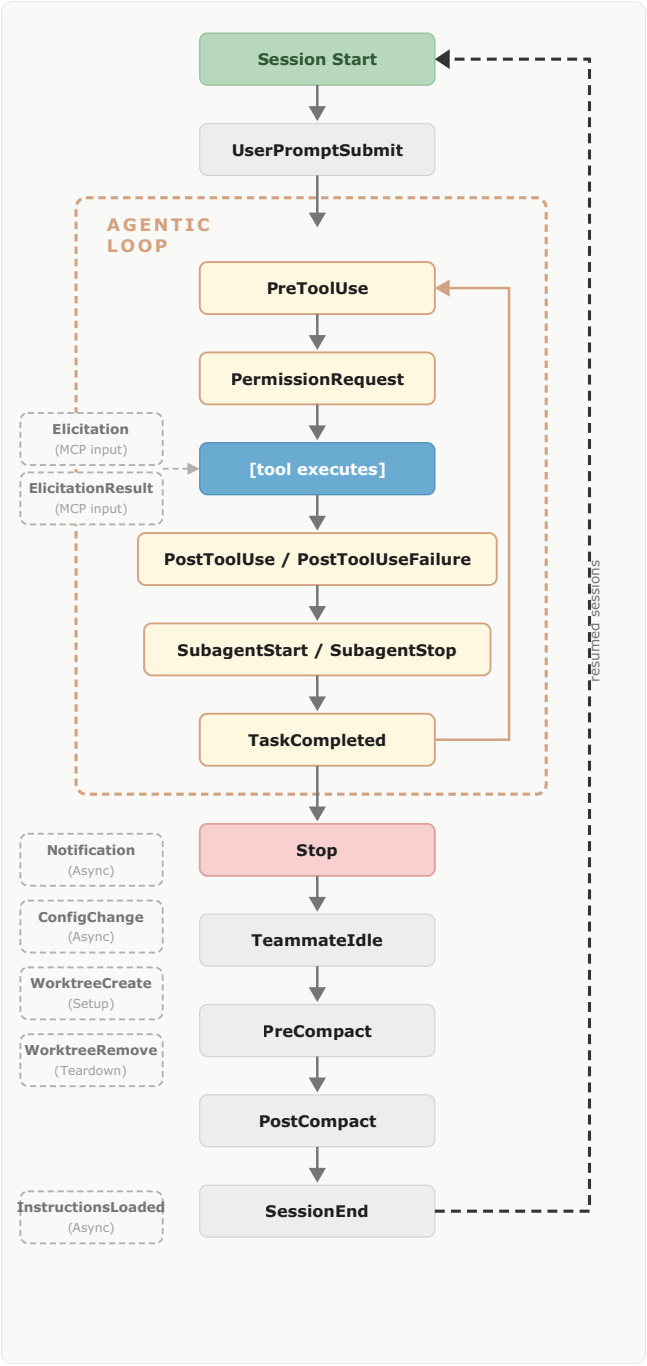
> *Reference for Claude Code hook events, configuration schema, JSON input/output formats, exit codes, async hooks, HTTP hooks, prompt hooks, and MCP tool hooks.*

> **Tip:**
>
> For a quickstart guide with examples, see Automate workflows with hooks.

Hooks are user-defined shell commands, HTTP endpoints, or LLM prompts that execute automatically at specific points in Claude Code's lifecycle. Use this reference to look up event schemas, configuration options, JSON input/output formats, and advanced features like async hooks, HTTP hooks, and MCP tool hooks. If you're setting up hooks for the first time, start with the guide instead.

Hook lifecycle

Hooks fire at specific points during a Claude Code session. When an event fires and a matcher matches, Claude Code passes JSON context about the event to your hook handler. For command hooks, input arrives on stdin. For HTTP hooks, it arrives as the POST request body. Your handler can then inspect the input, take action, and optionally return a decision. Some events fire once per session, while others fire repeatedly inside the agentic loop:


Session Start
UserPromptSubmit
AGENTIC LOOP
PreToolUse
PermissionRequest
Elicitation
(MCP input)
[tool executes]
ElicitationResult
(MCP input)
PostToolUse / PostToolUseFailure
SubagentStart / SubagentStop
TaskCompleted
Stop
Notification
(Async)
ConfigChange
(Async)
TeammateIdle
WorktreeCreate
(Setup)
PreCompact
WorktreeRemove
(Teardown)
PostCompact
InstructionsLoaded
(Async)
SessionEnd
resumed sessions

Hook lifecycle diagram showing the sequence of hooks from SessionStart through the agentic loop (PreToolUse, PermissionRequest, PostToolUse, SubagentStart/Stop, TaskCompleted) to PostCompact and SessionEnd, with Elicitation and ElicitationResult nested inside MCP tool execution and WorktreeCreate, WorktreeRemove, Notification, ConfigChange, and InstructionsLoaded as standalone async events

The table below summarizes when each event fires. The Hook events section documents the full input schema and decision control options for each one.

Event	When it fires
`SessionStart`	When a session begins or resumes
`UserPromptSubmit`	When you submit a prompt, before Claude processes it
`PreToolUse`	Before a tool call executes. Can block it
`PermissionRequest`	When a permission dialog appears
`PostToolUse`	After a tool call succeeds
`PostToolUseFailure`	After a tool call fails
`Notification`	When Claude Code sends a notification
`SubagentStart`	When a subagent is spawned
`SubagentStop`	When a subagent finishes
`Stop`	When Claude finishes responding
`TeammateIdle`	When an agent team teammate is about to go idle
`TaskCompleted`	When a task is being marked as completed
`InstructionsLoaded`	When a CLAUDE.md or `.claude/rules/*.md` file is loaded into context. Fires at session start and when files are lazily loaded during a session
`ConfigChange`	When a configuration file changes during a session
`WorktreeCreate`	When a worktree is being created via `--worktree` or `isolation: "worktree"`. Replaces default git behavior
`WorktreeRemove`	When a worktree is being removed, either at session exit or when a subagent finishes
`PreCompact`	Before context compaction

Event	When it fires
`PostCompact`	After context compaction completes
`Elicitation`	When an MCP server requests user input during a tool call
`ElicitationResult`	After a user responds to an MCP elicitation, before the response is sent back to the server
`SessionEnd`	When a session terminates

How a hook resolves

To see how these pieces fit together, consider this `PreToolUse` hook that blocks destructive shell commands. The hook runs `block-rm.sh` before every Bash tool call:

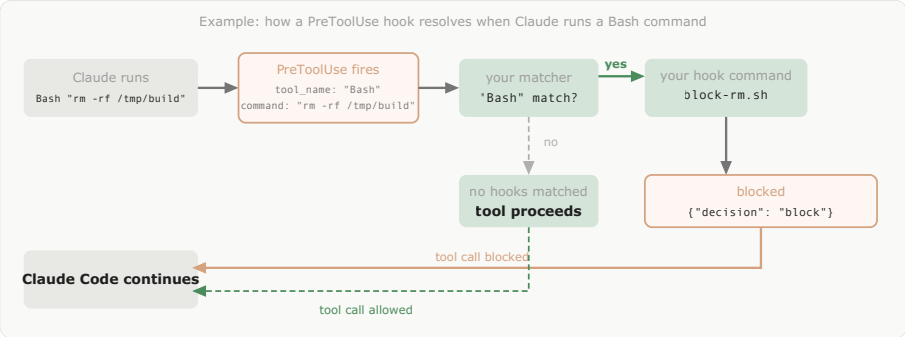
```
{
  "hooks": {
    "PreToolUse": [
      {
        "matcher": "Bash",
        "hooks": [
          {
            "type": "command",
            "command": ".claude/hooks/block-rm.sh"
          }
        ]
      }
    ]
  }
}
```

The script reads the JSON input from stdin, extracts the command, and returns a `permissionDecision` of `"deny"` if it contains `rm -rf`:

```
#!/bin/bash
## .claude/hooks/block-rm.sh
COMMAND=$(jq -r '.tool_input.command')

if echo "$COMMAND" | grep -q 'rm -rf'; then
  jq -n '{
    hookSpecificOutput: {
      hookEventName: "PreToolUse",
      permissionDecision: "deny",
      permissionDecisionReason: "Destructive command blocked by hook"
    }
  }'
else
  exit 0  # allow the command
fi
```

Now suppose Claude Code decides to run `Bash "rm -rf /tmp/build"`. Here's what happens:



Hook resolution flow: PreToolUse event fires, matcher checks for Bash match, hook handler runs, result returns to Claude Code

Step 1: Event fires

The `PreToolUse` event fires. Claude Code sends the tool input as JSON on stdin to the hook:

```
{ "tool_name": "Bash", "tool_input": { "command": "rm -rf /tmp/build" }, ... }
```

Step 2: Matcher checks

The matcher `"Bash"` matches the tool name, so `block-rm.sh` runs. If you omit the matcher or use `"*"`, the hook runs on every occurrence of the event. Hooks only skip when a matcher is defined and doesn't match.

Step 3: Hook handler runs

The script extracts `"rm -rf /tmp/build"` from the input and finds `rm -rf`, so it prints a decision to stdout:

```
{
  "hookSpecificOutput": {
    "hookEventName": "PreToolUse",
    "permissionDecision": "deny",
    "permissionDecisionReason": "Destructive command blocked by hook"
  }
}
```

If the command had been safe (like `npm test`), the script would hit `exit 0` instead, which tells Claude Code to allow the tool call with no further action.

Step 4: Claude Code acts on the result

Claude Code reads the JSON decision, blocks the tool call, and shows Claude the reason.

The Configuration section below documents the full schema, and each hook event section documents what input your command receives and what output it can return.

Configuration

Hooks are defined in JSON settings files. The configuration has three levels of nesting:

1. Choose a hook event to respond to, like `PreToolUse` or `Stop`
2. Add a matcher group to filter when it fires, like "only for the Bash tool"
3. Define one or more hook handlers to run when matched

See How a hook resolves above for a complete walkthrough with an annotated example.

> **Note:**

> This page uses specific terms for each level: **hook event** for the lifecycle point, **matcher group** for the filter, and **hook handler** for the shell command, HTTP endpoint, prompt, or agent that runs. "Hook" on its own refers to the general feature.

Hook locations

Where you define a hook determines its scope:

Location	Scope	Shareable
`~/.claude/settings.json`	All your projects	No, local to your machine
`.claude/settings.json`	Single project	Yes, can be committed to the repo
`.claude/settings.local.json`	Single project	No, gitignored
Managed policy settings	Organization-wide	Yes, admin-controlled
Plugin `hooks/hooks.json`	When plugin is enabled	Yes, bundled with the plugin
Skill or agent frontmatter	While the component is active	Yes, defined in the component file

For details on settings file resolution, see settings. Enterprise administrators can use `allowManagedHooksOnly` to block user, project, and plugin hooks. See Hook configuration.

Matcher patterns

The `matcher` field is a regex string that filters when hooks fire. Use `"*"`, `""`, or omit `matcher` entirely to match all occurrences. Each event type matches on a different field:

Event	What the matcher filters	Example matcher values
`PreToolUse`, `PostToolUse`, `PostToolUseFailure`, `PermissionRequest`	tool name	`Bash`, `Edit|Write`, `mcp__.*`
`SessionStart`	how the session started	`startup`, `resume`, `clear`, `compact`

Event	What the matcher filters	Example matcher values
`SessionEnd`	why the session ended	`clear`, `logout`, `prompt_input_exit`, `bypass_permissions_disabled`, `other`
`Notification`	notification type	`permission_prompt`, `idle_prompt`, `auth_success`, `elicitation_dialog`
`SubagentStart`	agent type	`Bash`, `Explore`, `Plan`, or custom agent names
`PreCompact`	what triggered compaction	`manual`, `auto`
`SubagentStop`	agent type	same values as `SubagentStart`
`ConfigChange`	configuration source	`user_settings`, `project_settings`, `local_settings`, `policy_settings`, `skills`
`UserPromptSubmit`, `Stop`, `TeammateIdle`, `TaskCompleted`, `WorktreeCreate`, `WorktreeRemove`, `InstructionsLoaded`	no matcher support	always fires on every occurrence

The matcher is a regex, so `Edit|Write` matches either tool and `Notebook.*` matches any tool starting with Notebook. The matcher runs against a field from the JSON input that Claude Code sends to your hook on stdin. For tool events, that field is `tool_name`. Each hook event section lists the full set of matcher values and the input schema for that event.

This example runs a linting script only when Claude writes or edits a file:

```
{
  "hooks": {
    "PostToolUse": [
      {
        "matcher": "Edit|Write",
        "hooks": [
          {
            "type": "command",
            "command": "/path/to/lint-check.sh"
          }
        ]
      }
    ]
  }
}
```

`UserPromptSubmit`, `Stop`, `TeammateIdle`, `TaskCompleted`, `WorktreeCreate`, `WorktreeRemove`, and `InstructionsLoaded` don't support matchers and always fire on every occurrence. If you add a `matcher` field to these events, it is silently ignored.

Match MCP tools

MCP server tools appear as regular tools in tool events (`PreToolUse`, `PostToolUse`, `PostToolUseFailure`, `PermissionRequest`), so you can match them the same way you match any other tool name.

MCP tools follow the naming pattern `mcp__<server>__<tool>`, for example:

- `mcp__memory__create_entities`: Memory server's create entities tool
- `mcp__filesystem__read_file`: Filesystem server's read file tool
- `mcp__github__search_repositories`: GitHub server's search tool

Use regex patterns to target specific MCP tools or groups of tools:

- `mcp__memory__.*` matches all tools from the `memory` server
- `mcp__.*__write.*` matches any tool containing "write" from any server

This example logs all memory server operations and validates write operations from any MCP server:

```
{
  "hooks": {
    "PreToolUse": [
      {
        "matcher": "mcp__memory__.*",
        "hooks": [
          {
            "type": "command",
            "command": "echo 'Memory operation initiated' >> ~/mcp-operations.log"
          }
        ]
      },
      {
        "matcher": "mcp__.*__write.*",
        "hooks": [
          {
            "type": "command",
            "command": "/home/user/scripts/validate-mcp-write.py"
          }
        ]
      }
    ]
  }
}
```

Hook handler fields

Each object in the inner `hooks` array is a hook handler: the shell command, HTTP endpoint, LLM prompt, or agent that runs when the matcher matches. There are four types:

- **Command hooks** (`type: "command"`): run a shell command. Your script receives the event's JSON input on stdin and communicates results back through exit codes and stdout.
- **HTTP hooks** (`type: "http"`): send the event's JSON input as an HTTP POST request to a URL. The endpoint communicates results back through the response body using the same JSON output format as command hooks.
- **Prompt hooks** (`type: "prompt"`): send a prompt to a Claude model for single-turn evaluation. The model returns a yes/no decision as JSON. See Prompt-based hooks.

- **Agent hooks** (`type: "agent"`): spawn a subagent that can use tools like Read, Grep, and Glob to verify conditions before returning a decision. See Agent-based hooks.

Common fields

These fields apply to all hook types:

Field	Required	Description
`type`	yes	`"command"` , `"http"` , `"prompt"` , or `"agent"`
`timeout`	no	Seconds before canceling. Defaults: 600 for command, 30 for prompt, 60 for agent
`statusMessage`	no	Custom spinner message displayed while the hook runs
`once`	no	If `true` , runs only once per session then is removed. Skills only, not agents. See Hooks in skills and agents

Command hook fields

In addition to the common fields, command hooks accept these fields:

Field	Required	Description
`command`	yes	Shell command to execute
`async`	no	If `true` , runs in the background without blocking. See Run hooks in the background

HTTP hook fields

In addition to the common fields, HTTP hooks accept these fields:

Field	Required	Description
`url`	yes	URL to send the POST request to
`headers`	no	Additional HTTP headers as key-value pairs. Values support environment variable interpolation using `$VAR_NAME` or `${VAR_NAME}` syntax. Only variables listed in `allowedEnvVars` are resolved
`allowedEnvVars`	no	List of environment variable names that may be interpolated into header values. References to unlisted variables are replaced with empty strings. Required for any env var interpolation to work

Claude Code sends the hook's JSON input as the POST request body with `Content-Type: application/json`. The response body uses the same JSON output format as command hooks.

Error handling differs from command hooks: non-2xx responses, connection failures, and timeouts all produce non-blocking errors that allow execution to continue. To block a tool call or deny a permission, return a 2xx response with a JSON body containing `decision: "block"` or a `hookSpecificOutput` with `permissionDecision: "deny"`.

This example sends `PreToolUse` events to a local validation service, authenticating with a token from the `MY_TOKEN` environment variable:

```
{
  "hooks": {
    "PreToolUse": [
      {
        "matcher": "Bash",
        "hooks": [
          {
            "type": "http",
            "url": "http://localhost:8080/hooks/pre-tool-use",
            "timeout": 30,
            "headers": {
              "Authorization": "Bearer $MY_TOKEN"
            },
            "allowedEnvVars": ["MY_TOKEN"]
          }
        ]
      }
    ]
  }
}
```

Prompt and agent hook fields

In addition to the common fields, prompt and agent hooks accept these fields:

Field	Required	Description
`prompt`	yes	Prompt text to send to the model. Use `$ARGUMENTS` as a placeholder for the hook input JSON
`model`	no	Model to use for evaluation. Defaults to a fast model

All matching hooks run in parallel, and identical handlers are deduplicated automatically. Command hooks are deduplicated by command string, and HTTP hooks are deduplicated by URL. Handlers run in the current directory with Claude Code's environment. The `$CLAUDE_CODE_REMOTE` environment variable is set to `"true"` in remote web environments and not set in the local CLI.

Reference scripts by path

Use environment variables to reference hook scripts relative to the project or plugin root, regardless of the working directory when the hook runs:

- `$CLAUDE_PROJECT_DIR`: the project root. Wrap in quotes to handle paths with spaces.
- `${CLAUDE_PLUGIN_ROOT}`: the plugin's root directory, for scripts bundled with a plugin.

Project scripts

This example uses `$CLAUDE_PROJECT_DIR` to run a style checker from the project's `.claude/hooks/` directory after any `Write` or `Edit` tool call:

```
{
  "hooks": {
    "PostToolUse": [
      {
        "matcher": "Write|Edit",
        "hooks": [
          {
            "type": "command",
            "command": "\"$CLAUDE_PROJECT_DIR\"/.claude/hooks/check-style.sh"
          }
        ]
      }
    ]
  }
}
```

Plugin scripts

Define plugin hooks in `hooks/hooks.json` with an optional top-level `description` field. When a plugin is enabled, its hooks merge with your user and project hooks.

This example runs a formatting script bundled with the plugin:

```
{
  "description": "Automatic code formatting",
  "hooks": {
    "PostToolUse": [
      {
        "matcher": "Write|Edit",
        "hooks": [
          {
            "type": "command",
            "command": "${CLAUDE_PLUGIN_ROOT}/scripts/format.sh",
            "timeout": 30
          }
        ]
      }
    ]
  }
}
```

See the plugin components reference for details on creating plugin hooks.

Hooks in skills and agents

In addition to settings files and plugins, hooks can be defined directly in skills and subagents using frontmatter. These hooks are scoped to the component's lifecycle and only run when that component is active.

All hook events are supported. For subagents, `Stop` hooks are automatically converted to `SubagentStop` since that is the event that fires when a subagent completes.

Hooks use the same configuration format as settings-based hooks but are scoped to the component's lifetime and cleaned up when it finishes.

This skill defines a `PreToolUse` hook that runs a security validation script before each `Bash` command:

```
---
name: secure-operations
description: Perform operations with security checks
hooks:
  PreToolUse:
    - matcher: "Bash"
      hooks:
        - type: command
          command: "./scripts/security-check.sh"
---
```

Agents use the same format in their YAML frontmatter.

The `/hooks` menu

Type `/hooks` in Claude Code to open a read-only browser for your configured hooks. The menu shows every hook event with a count of configured hooks, lets you drill into matchers, and shows the full details of each hook handler. Use it to verify configuration, check which settings file a hook came from, or inspect a hook's command, prompt, or URL.

The menu displays all four hook types: `command`, `prompt`, `agent`, and `http`. Each hook is labeled with a `[type]` prefix and a source indicating where it was defined:

- `User`: from `~/.claude/settings.json`
- `Project`: from `.claude/settings.json`
- `Local`: from `.claude/settings.local.json`
- `Plugin`: from a plugin's `hooks/hooks.json`
- `Session`: registered in memory for the current session
- `Built-in`: registered internally by Claude Code

Selecting a hook opens a detail view showing its event, matcher, type, source file, and the full command, prompt, or URL. The menu is read-only: to add, modify, or remove hooks, edit the settings JSON directly or ask Claude to make the change.

Disable or remove hooks

To remove a hook, delete its entry from the settings JSON file.

To temporarily disable all hooks without removing them, set `"disableAllHooks": true` in your settings file. There is no way to disable an individual hook while keeping it in the configuration.

The `disableAllHooks` setting respects the managed settings hierarchy. If an administrator has configured hooks through managed policy settings, `disableAllHooks` set in user, project, or local settings cannot disable those managed hooks. Only `disableAllHooks` set at the managed settings level can disable managed hooks.

Direct edits to hooks in settings files are normally picked up automatically by the file watcher.

Hook input and output

Command hooks receive JSON data via stdin and communicate results through exit codes, stdout, and stderr. HTTP hooks receive the same JSON as the POST request body and communicate results through the HTTP response body. This section covers fields and behavior common to all events. Each event's section under Hook events includes its specific input schema and decision control options.

Common input fields

All hook events receive these fields as JSON, in addition to event-specific fields documented in each hook event section. For command hooks, this JSON arrives via stdin. For HTTP hooks, it arrives as the POST request body.

Field	Description
`session_id`	Current session identifier
`transcript_path`	Path to conversation JSON
`cwd`	Current working directory when the hook is invoked
`permission_mode`	Current permission mode: `"default"`, `"plan"`, `"acceptEdits"`, `"dontAsk"`, or `"bypassPermissions"`
`hook_event_name`	Name of the event that fired

When running with `--agent` or inside a subagent, two additional fields are included:

Field	Description
`agent_id`	Unique identifier for the subagent. Present only when the hook fires inside a subagent call. Use this to distinguish subagent hook calls from main-thread calls.
`agent_type`	Agent name (for example, `"Explore"` or `"security-reviewer"`). Present when the session uses `--agent` or the hook fires inside a subagent. For subagents, the subagent's type takes precedence over the session's `--agent` value.

For example, a `PreToolUse` hook for a Bash command receives this on stdin:

```
{
  "session_id": "abc123",
  "transcript_path": "/home/user/.claude/projects/.../transcript.jsonl",
  "cwd": "/home/user/my-project",
  "permission_mode": "default",
  "hook_event_name": "PreToolUse",
  "tool_name": "Bash",
  "tool_input": {
    "command": "npm test"
  }
}
```

The `tool_name` and `tool_input` fields are event-specific. Each hook event section documents the additional fields for that event.

Exit code output

The exit code from your hook command tells Claude Code whether the action should proceed, be blocked, or be ignored.

Exit 0 means success. Claude Code parses stdout for JSON output fields. JSON output is only processed on exit 0. For most events, stdout is only shown in verbose mode (`Ctrl+O`). The exceptions are `UserPromptSubmit` and `SessionStart`, where stdout is added as context that Claude can see and act on.

Exit 2 means a blocking error. Claude Code ignores stdout and any JSON in it. Instead, stderr text is fed back to Claude as an error message. The effect depends on the event: `PreToolUse` blocks the tool call, `UserPromptSubmit` rejects the prompt, and so on. See exit code 2 behavior for the full list.

Any other exit code is a non-blocking error. stderr is shown in verbose mode (`Ctrl+O`) and execution continues.

For example, a hook command script that blocks dangerous Bash commands:

```
#!/bin/bash
## Reads JSON input from stdin, checks the command
command=$(jq -r '.tool_input.command' < /dev/stdin)

if [[ "$command" == rm* ]]; then
  echo "Blocked: rm commands are not allowed" >&2
  exit 2  # Blocking error: tool call is prevented
fi

exit 0  # Success: tool call proceeds
```

Exit code 2 behavior per event

Exit code 2 is the way a hook signals "stop, don't do this." The effect depends on the event, because some events represent actions that can be blocked (like a tool call that hasn't happened yet) and others represent things that already happened or can't be prevented.

Hook event	Can block?	What happens on exit 2
`PreToolUse`	Yes	Blocks the tool call
`PermissionRequest`	Yes	Denies the permission
`UserPromptSubmit`	Yes	Blocks prompt processing and erases the prompt
`Stop`	Yes	Prevents Claude from stopping, continues the conversation
`SubagentStop`	Yes	Prevents the subagent from stopping
`TeammateIdle`	Yes	Prevents the teammate from going idle (teammate continues working)
`TaskCompleted`	Yes	Prevents the task from being marked as completed

Hook event	Can block?	What happens on exit 2
`ConfigChange`	Yes	Blocks the configuration change from taking effect (except `policy_settings`)
`PostToolUse`	No	Shows stderr to Claude (tool already ran)
`PostToolUseFailure`	No	Shows stderr to Claude (tool already failed)
`Notification`	No	Shows stderr to user only
`SubagentStart`	No	Shows stderr to user only
`SessionStart`	No	Shows stderr to user only
`SessionEnd`	No	Shows stderr to user only
`PreCompact`	No	Shows stderr to user only
`PostCompact`	No	Shows stderr to user only
`Elicitation`	Yes	Denies the elicitation
`ElicitationResult`	Yes	Blocks the response (action becomes decline)
`WorktreeCreate`	Yes	Any non-zero exit code causes worktree creation to fail
`WorktreeRemove`	No	Failures are logged in debug mode only
`InstructionsLoaded`	No	Exit code is ignored

HTTP response handling

HTTP hooks use HTTP status codes and response bodies instead of exit codes and stdout:

- **2xx with an empty body**: success, equivalent to exit code 0 with no output
- **2xx with a plain text body**: success, the text is added as context

- **2xx with a JSON body**: success, parsed using the same JSON output schema as command hooks
- **Non-2xx status**: non-blocking error, execution continues
- **Connection failure or timeout**: non-blocking error, execution continues

Unlike command hooks, HTTP hooks cannot signal a blocking error through status codes alone. To block a tool call or deny a permission, return a 2xx response with a JSON body containing the appropriate decision fields.

JSON output

Exit codes let you allow or block, but JSON output gives you finer-grained control. Instead of exiting with code 2 to block, exit 0 and print a JSON object to stdout. Claude Code reads specific fields from that JSON to control behavior, including decision control for blocking, allowing, or escalating to the user.

> **Note:**
>
> You must choose one approach per hook, not both: either use exit codes alone for signaling, or exit 0 and print JSON for structured control. Claude Code only processes JSON on exit 0. If you exit 2, any JSON is ignored.

Your hook's stdout must contain only the JSON object. If your shell profile prints text on startup, it can interfere with JSON parsing. See JSON validation failed in the troubleshooting guide.

The JSON object supports three kinds of fields:

- **Universal fields** like `continue` work across all events. These are listed in the table below.
- **Top-level `decision` and `reason`** are used by some events to block or provide feedback.
- `hookSpecificOutput` is a nested object for events that need richer control. It requires a `hookEventName` field set to the event name.

Field	Default	Description
`continue`	`true`	If `false`, Claude stops processing entirely after the hook runs. Takes precedence over any event-specific decision fields

Field	Default	Description
`stopReason`	none	Message shown to the user when `continue` is `false`. Not shown to Claude
`suppressOutput`	`false`	If `true`, hides stdout from verbose mode output
`systemMessage`	none	Warning message shown to the user

To stop Claude entirely regardless of event type:

```
{ "continue": false, "stopReason": "Build failed, fix errors before continuing" }
```

Decision control

Not every event supports blocking or controlling behavior through JSON. The events that do each use a different set of fields to express that decision. Use this table as a quick reference before writing a hook:

Events	Decision pattern	Key fields
UserPromptSubmit, PostToolUse, PostToolUseFailure, Stop, SubagentStop, ConfigChange	Top-level `decision`	`decision: "block"`, `reason`
TeammateIdle, TaskCompleted	Exit code or `continue: false`	Exit code 2 blocks the action with stderr feedback. JSON `{"continue": false, "stopReason": "..."}` also stops the teammate entirely, matching `Stop` hook behavior
PreToolUse	`hookSpecificOutput`	`permissionDecision` (allow/deny/ask), `permissionDecisionReason`
PermissionRequest	`hookSpecificOutput`	`decision.behavior` (allow/deny)

Events	Decision pattern	Key fields
WorktreeCreate	stdout path	Hook prints absolute path to created worktree. Non-zero exit fails creation
Elicitation	`hookSpecificOutput`	`action` (accept/decline/cancel), `content` (form field values for accept)
ElicitationResult	`hookSpecificOutput`	`action` (accept/decline/cancel), `content` (form field values override)
WorktreeRemove, Notification, SessionEnd, PreCompact, PostCompact, InstructionsLoaded	None	No decision control. Used for side effects like logging or cleanup

Here are examples of each pattern in action:

Top-level decision

Used by `UserPromptSubmit`, `PostToolUse`, `PostToolUseFailure`, `Stop`, `SubagentStop`, and `ConfigChange`. The only value is `"block"`. To allow the action to proceed, omit `decision` from your JSON, or exit 0 without any JSON at all:

```
{
  "decision": "block",
  "reason": "Test suite must pass before proceeding"
}
```

PreToolUse

Uses `hookSpecificOutput` for richer control: allow, deny, or escalate to the user. You can also modify tool input before it runs or inject additional context for Claude. See Pre-ToolUse decision control for the full set of options.

```
{
  "hookSpecificOutput": {
    "hookEventName": "PreToolUse",
    "permissionDecision": "deny",
    "permissionDecisionReason": "Database writes are not allowed"
  }
}
```

PermissionRequest

Uses `hookSpecificOutput` to allow or deny a permission request on behalf of the user. When allowing, you can also modify the tool's input or apply permission rules so the user isn't prompted again. See PermissionRequest decision control for the full set of options.

```
{
  "hookSpecificOutput": {
    "hookEventName": "PermissionRequest",
    "decision": {
      "behavior": "allow",
      "updatedInput": {
        "command": "npm run lint"
      }
    }
  }
}
```

For extended examples including Bash command validation, prompt filtering, and auto-approval scripts, see What you can automate in the guide and the Bash command validator reference implementation.

Hook events

Each event corresponds to a point in Claude Code's lifecycle where hooks can run. The sections below are ordered to match the lifecycle: from session setup through the agentic loop to session end. Each section describes when the event fires, what matchers it supports, the JSON input it receives, and how to control behavior through output.

SessionStart

Runs when Claude Code starts a new session or resumes an existing session. Useful for loading development context like existing issues or recent changes to your codebase, or setting up environment variables. For static context that does not require a script, use CLAUDE.md instead.

SessionStart runs on every session, so keep these hooks fast. Only `type: "command"` hooks are supported.

The matcher value corresponds to how the session was initiated:

Matcher	When it fires
`startup`	New session
`resume`	`--resume`, `--continue`, or `/resume`
`clear`	`/clear`
`compact`	Auto or manual compaction

SessionStart input

In addition to the common input fields, SessionStart hooks receive `source`, `model`, and optionally `agent_type`. The `source` field indicates how the session started: `"startup"` for new sessions, `"resume"` for resumed sessions, `"clear"` after `/clear`, or `"compact"` after compaction. The `model` field contains the model identifier. If you start Claude Code with `claude --agent <name>`, an `agent_type` field contains the agent name.

```
{
  "session_id": "abc123",
  "transcript_path": "/Users/.../.claude/projects/.../
00893aaf-19fa-41d2-8238-13269b9b3ca0.jsonl",
  "cwd": "/Users/...",
  "permission_mode": "default",
  "hook_event_name": "SessionStart",
  "source": "startup",
  "model": "claude-sonnet-4-6"
}
```

SessionStart decision control

Any text your hook script prints to stdout is added as context for Claude. In addition to the JSON output fields available to all hooks, you can return these event-specific fields:

Field	Description
`additionalContext`	String added to Claude's context. Multiple hooks' values are concatenated

```
{
  "hookSpecificOutput": {
    "hookEventName": "SessionStart",
    "additionalContext": "My additional context here"
  }
}
```

Persist environment variables

SessionStart hooks have access to the `CLAUDE_ENV_FILE` environment variable, which provides a file path where you can persist environment variables for subsequent Bash commands.

To set individual environment variables, write `export` statements to `CLAUDE_ENV_FILE`. Use append (`>>`) to preserve variables set by other hooks:

```
#!/bin/bash

if [ -n "$CLAUDE_ENV_FILE" ]; then
  echo 'export NODE_ENV=production' >> "$CLAUDE_ENV_FILE"
  echo 'export DEBUG_LOG=true' >> "$CLAUDE_ENV_FILE"
  echo 'export PATH="$PATH:./node_modules/.bin"' >> "$CLAUDE_ENV_FILE"
fi

exit 0
```

To capture all environment changes from setup commands, compare the exported variables before and after:

```
#!/bin/bash

ENV_BEFORE=$(export -p | sort)

## Run your setup commands that modify the environment
source ~/.nvm/nvm.sh
nvm use 20

if [ -n "$CLAUDE_ENV_FILE" ]; then
  ENV_AFTER=$(export -p | sort)
  comm -13 <(echo "$ENV_BEFORE") <(echo "$ENV_AFTER") >> "$CLAUDE_ENV_FILE"
fi

exit 0
```

Any variables written to this file will be available in all subsequent Bash commands that Claude Code executes during the session.

> **Note:**
>
> `CLAUDE_ENV_FILE` is available for SessionStart hooks. Other hook types do not have access to this variable.

InstructionsLoaded

Fires when a `CLAUDE.md` or `.claude/rules/*.md` file is loaded into context. This event fires at session start for eagerly-loaded files and again later when files are lazily loaded, for example when Claude accesses a subdirectory that contains a nested `CLAUDE.md` or when conditional rules with `paths:` frontmatter match. The hook does not support blocking or decision control. It runs asynchronously for observability purposes.

InstructionsLoaded does not support matchers and fires on every load occurrence.

InstructionsLoaded input

In addition to the common input fields, InstructionsLoaded hooks receive these fields:

Field	Description
`file_path`	Absolute path to the instruction file that was loaded

Field	Description
`memory_type`	Scope of the file: `"User"`, `"Project"`, `"Local"`, or `"Managed"`
`load_reason`	Why the file was loaded: `"session_start"`, `"nested_traversal"`, `"path_glob_match"`, or `"include"`
`globs`	Path glob patterns from the file's `paths:` frontmatter, if any. Present only for `path_glob_match` loads
`trigger_file_path`	Path to the file whose access triggered this load, for lazy loads
`parent_file_path`	Path to the parent instruction file that included this one, for `include` loads

```
{
  "session_id": "abc123",
  "transcript_path": "/Users/.../.claude/projects/.../transcript.jsonl",
  "cwd": "/Users/my-project",
  "permission_mode": "default",
  "hook_event_name": "InstructionsLoaded",
  "file_path": "/Users/my-project/CLAUDE.md",
  "memory_type": "Project",
  "load_reason": "session_start"
}
```

InstructionsLoaded decision control

InstructionsLoaded hooks have no decision control. They cannot block or modify instruction loading. Use this event for audit logging, compliance tracking, or observability.

UserPromptSubmit

Runs when the user submits a prompt, before Claude processes it. This allows you to add additional context based on the prompt/conversation, validate prompts, or block certain types of prompts.

UserPromptSubmit input

In addition to the common input fields, UserPromptSubmit hooks receive the `prompt` field containing the text the user submitted.

```
{
  "session_id": "abc123",
  "transcript_path": "/Users/.../.claude/projects/.../
00893aaf-19fa-41d2-8238-13269b9b3ca0.jsonl",
  "cwd": "/Users/...",
  "permission_mode": "default",
  "hook_event_name": "UserPromptSubmit",
  "prompt": "Write a function to calculate the factorial of a number"
}
```

UserPromptSubmit decision control

`UserPromptSubmit` hooks can control whether a user prompt is processed and add context. All JSON output fields are available.

There are two ways to add context to the conversation on exit code 0:

- **Plain text stdout**: any non-JSON text written to stdout is added as context
- **JSON with `additionalContext`**: use the JSON format below for more control. The `additionalContext` field is added as context

Plain stdout is shown as hook output in the transcript. The `additionalContext` field is added more discretely.

To block a prompt, return a JSON object with `decision` set to `"block"`:

Field	Description
`decision`	`"block"` prevents the prompt from being processed and erases it from context. Omit to allow the prompt to proceed
`reason`	Shown to the user when `decision` is `"block"`. Not added to context
`additionalContext`	String added to Claude's context

```
{
  "decision": "block",
  "reason": "Explanation for decision",
  "hookSpecificOutput": {
    "hookEventName": "UserPromptSubmit",
    "additionalContext": "My additional context here"
  }
}
```

> **Note:**
>
> The JSON format isn't required for simple use cases. To add context, you can print plain text to stdout with exit code 0. Use JSON when you need to block prompts or want more structured control.

PreToolUse

Runs after Claude creates tool parameters and before processing the tool call. Matches on tool name: `Bash`, `Edit`, `Write`, `Read`, `Glob`, `Grep`, `Agent`, `WebFetch`, `WebSearch`, and any MCP tool names.

Use PreToolUse decision control to allow, deny, or ask for permission to use the tool.

PreToolUse input

In addition to the common input fields, PreToolUse hooks receive `tool_name`, `tool_input`, and `tool_use_id`. The `tool_input` fields depend on the tool:

Bash

Executes shell commands.

Field	Type	Example	Description
`command`	string	`"npm test"`	The shell command to execute
`description`	string	`"Run test suite"`	Optional description of what the command does
`timeout`	number	`120000`	Optional timeout in milliseconds

Field	Type	Example	Description
`run_in_background`	boolean	`false`	Whether to run the command in background

Write

Creates or overwrites a file.

Field	Type	Example	Description
`file_path`	string	`"/path/to/file.txt"`	Absolute path to the file to write
`content`	string	`"file content"`	Content to write to the file

Edit

Replaces a string in an existing file.

Field	Type	Example	Description
`file_path`	string	`"/path/to/file.txt"`	Absolute path to the file to edit
`old_string`	string	`"original text"`	Text to find and replace
`new_string`	string	`"replacement text"`	Replacement text
`replace_all`	boolean	`false`	Whether to replace all occurrences

Read

Reads file contents.

Field	Type	Example	Description
`file_path`	string	`"/path/to/file.txt"`	Absolute path to the file to read

Field	Type	Example	Description
`offset`	number	`10`	Optional line number to start reading from
`limit`	number	`50`	Optional number of lines to read

Glob

Finds files matching a glob pattern.

Field	Type	Example	Description
`pattern`	string	`"**/*.ts"`	Glob pattern to match files against
`path`	string	`"/path/to/dir"`	Optional directory to search in. Defaults to current working directory

Grep

Searches file contents with regular expressions.

Field	Type	Example	Description
`pattern`	string	`"TODO.*fix"`	Regular expression pattern to search for
`path`	string	`"/path/to/dir"`	Optional file or directory to search in
`glob`	string	`"*.ts"`	Optional glob pattern to filter files
`output_mode`	string	`"content"`	`"content"`, `"files_with_matches"`, or `"count"`. Defaults to `"files_with_matches"`
`-i`	boolean	`true`	Case insensitive search

Field	Type	Example	Description
`multiline`	boolean	`false`	Enable multiline matching

WebFetch

Fetches and processes web content.

Field	Type	Example	Description
`url`	string	`"https://example.com/api"`	URL to fetch content from
`prompt`	string	`"Extract the API endpoints"`	Prompt to run on the fetched content

WebSearch

Searches the web.

Field	Type	Example	Description
`query`	string	`"react hooks best practices"`	Search query
`allowed_domains`	array	`["docs.example.com"]`	Optional: only include results from these domains
`blocked_domains`	array	`["spam.example.com"]`	Optional: exclude results from these domains

Agent

Spawns a subagent.

Field	Type	Example	Description
`prompt`	string	`"Find all API endpoints"`	The task for the agent to perform
`description`	string	`"Find API endpoints"`	Short description of the task

Field	Type	Example	Description
`subagent_type`	string	`"Explore"`	Type of specialized agent to use
`model`	string	`"sonnet"`	Optional model alias to override the default

PreToolUse decision control

`PreToolUse` hooks can control whether a tool call proceeds. Unlike other hooks that use a top-level `decision` field, PreToolUse returns its decision inside a `hookSpecificOutput` object. This gives it richer control: three outcomes (allow, deny, or ask) plus the ability to modify tool input before execution.

Field	Description
`permissionDecision`	`"allow"` bypasses the permission system, `"deny"` prevents the tool call, `"ask"` prompts the user to confirm
`permissionDecisionReason`	For `"allow"` and `"ask"`, shown to the user but not Claude. For `"deny"`, shown to Claude
`updatedInput`	Modifies the tool's input parameters before execution. Combine with `"allow"` to auto-approve, or `"ask"` to show the modified input to the user
`additionalContext`	String added to Claude's context before the tool executes

When a hook returns `"ask"`, the permission prompt displayed to the user includes a label identifying where the hook came from: for example, `[User]`, `[Project]`, `[Plugin]`, or `[Local]`. This helps users understand which configuration source is requesting confirmation.

```
{
  "hookSpecificOutput": {
    "hookEventName": "PreToolUse",
    "permissionDecision": "allow",
    "permissionDecisionReason": "My reason here",
    "updatedInput": {
      "field_to_modify": "new value"
    },
    "additionalContext": "Current environment: production. Proceed with caution."
  }
}
```

Note:

PreToolUse previously used top-level `decision` and `reason` fields, but these are deprecated for this event. Use `hookSpecificOutput.permissionDecision` and `hookSpecificOutput.permissionDecisionReason` instead. The deprecated values `"approve"` and `"block"` map to `"allow"` and `"deny"` respectively. Other events like PostToolUse and Stop continue to use top-level `decision` and `reason` as their current format.

PermissionRequest

Runs when the user is shown a permission dialog. Use PermissionRequest decision control to allow or deny on behalf of the user.

Matches on tool name, same values as PreToolUse.

PermissionRequest input

PermissionRequest hooks receive `tool_name` and `tool_input` fields like PreToolUse hooks, but without `tool_use_id`. An optional `permission_suggestions` array contains the "always allow" options the user would normally see in the permission dialog. The difference is when the hook fires: PermissionRequest hooks run when a permission dialog is about to be shown to the user, while PreToolUse hooks run before tool execution regardless of permission status.

```
{
  "session_id": "abc123",
  "transcript_path": "/Users/.../.claude/projects/.../
00893aaf-19fa-41d2-8238-13269b9b3ca0.jsonl",
  "cwd": "/Users/...",
  "permission_mode": "default",
  "hook_event_name": "PermissionRequest",
  "tool_name": "Bash",
  "tool_input": {
    "command": "rm -rf node_modules",
    "description": "Remove node_modules directory"
  },
  "permission_suggestions": [
    {
      "type": "addRules",
      "rules": [{ "toolName": "Bash", "ruleContent": "rm -rf node_modules" }],
      "behavior": "allow",
      "destination": "localSettings"
    }
  ]
}
```

PermissionRequest decision control

`PermissionRequest` hooks can allow or deny permission requests. In addition to the JSON output fields available to all hooks, your hook script can return a `decision` object with these event-specific fields:

Field	Description
`behavior`	`"allow"` grants the permission, `"deny"` denies it
`updatedInput`	For `"allow"` only: modifies the tool's input parameters before execution
`updatedPermissions`	For `"allow"` only: array of permission update entries to apply, such as adding an allow rule or changing the session permission mode
`message`	For `"deny"` only: tells Claude why the permission was denied
`interrupt`	For `"deny"` only: if `true`, stops Claude

```
{
  "hookSpecificOutput": {
    "hookEventName": "PermissionRequest",
    "decision": {
      "behavior": "allow",
      "updatedInput": {
        "command": "npm run lint"
      }
    }
  }
}
```

Permission update entries

The `updatedPermissions` output field and the `permission_suggestions` input field both use the same array of entry objects. Each entry has a `type` that determines its other fields, and a `destination` that controls where the change is written.

`type`	Fields	Effect
`addRules`	`rules`, `behavior`, `destination`	Adds permission rules. `rules` is an array of `{toolName, ruleContent?}` objects. Omit `ruleContent` to match the whole tool. `behavior` is `"allow"`, `"deny"`, or `"ask"`
`replaceRules`	`rules`, `behavior`, `destination`	Replaces all rules of the given `behavior` at the `destination` with the provided `rules`
`removeRules`	`rules`, `behavior`, `destination`	Removes matching rules of the given `behavior`

type	Fields	Effect
setMode	mode, destination	Changes the permission mode. Valid modes are default, acceptEdits, dontAsk, bypassPermissions, and plan
addDirectories	directories, destination	Adds working directories. directories is an array of path strings
removeDirectories	directories, destination	Removes working directories

The destination field on every entry determines whether the change stays in memory or persists to a settings file.

destination	Writes to
session	in-memory only, discarded when the session ends
localSettings	.claude/settings.local.json
projectSettings	.claude/settings.json
userSettings	~/.claude/settings.json

A hook can echo one of the permission_suggestions it received as its own updatedPermissions output, which is equivalent to the user selecting that "always allow" option in the dialog.

PostToolUse

Runs immediately after a tool completes successfully.

Matches on tool name, same values as PreToolUse.

PostToolUse input

PostToolUse hooks fire after a tool has already executed successfully. The input includes both tool_input, the arguments sent to the tool, and tool_response, the result it returned. The exact schema for both depends on the tool.

```
{
  "session_id": "abc123",
  "transcript_path": "/Users/.../.claude/projects/.../
00893aaf-19fa-41d2-8238-13269b9b3ca0.jsonl",
  "cwd": "/Users/...",
  "permission_mode": "default",
  "hook_event_name": "PostToolUse",
  "tool_name": "Write",
  "tool_input": {
    "file_path": "/path/to/file.txt",
    "content": "file content"
  },
  "tool_response": {
    "filePath": "/path/to/file.txt",
    "success": true
  },
  "tool_use_id": "toolu_01ABC123..."
}
```

PostToolUse decision control

`PostToolUse` hooks can provide feedback to Claude after tool execution. In addition to the JSON output fields available to all hooks, your hook script can return these event-specific fields:

Field	Description
`decision`	`"block"` prompts Claude with the `reason`. Omit to allow the action to proceed
`reason`	Explanation shown to Claude when `decision` is `"block"`
`additionalContext`	Additional context for Claude to consider
`updatedMCPToolOutput`	For MCP tools only: replaces the tool's output with the provided value

```
{
  "decision": "block",
  "reason": "Explanation for decision",
  "hookSpecificOutput": {
    "hookEventName": "PostToolUse",
    "additionalContext": "Additional information for Claude"
  }
}
```

PostToolUseFailure

Runs when a tool execution fails. This event fires for tool calls that throw errors or return failure results. Use this to log failures, send alerts, or provide corrective feedback to Claude.

Matches on tool name, same values as PreToolUse.

PostToolUseFailure input

PostToolUseFailure hooks receive the same `tool_name` and `tool_input` fields as PostToolUse, along with error information as top-level fields:

```
{
  "session_id": "abc123",
  "transcript_path": "/Users/.../.claude/projects/.../
00893aaf-19fa-41d2-8238-13269b9b3ca0.jsonl",
  "cwd": "/Users/...",
  "permission_mode": "default",
  "hook_event_name": "PostToolUseFailure",
  "tool_name": "Bash",
  "tool_input": {
    "command": "npm test",
    "description": "Run test suite"
  },
  "tool_use_id": "toolu_01ABC123...",
  "error": "Command exited with non-zero status code 1",
  "is_interrupt": false
}
```

Field	Description
`error`	String describing what went wrong
`is_interrupt`	Optional boolean indicating whether the failure was caused by user interruption

PostToolUseFailure decision control

`PostToolUseFailure` hooks can provide context to Claude after a tool failure. In addition to the JSON output fields available to all hooks, your hook script can return these event-specific fields:

Field	Description
`additionalContext`	Additional context for Claude to consider alongside the error

```
{
  "hookSpecificOutput": {
    "hookEventName": "PostToolUseFailure",
    "additionalContext": "Additional information about the failure for Claude"
  }
}
```

Notification

Runs when Claude Code sends notifications. Matches on notification type: `permission_prompt`, `idle_prompt`, `auth_success`, `elicitation_dialog`. Omit the matcher to run hooks for all notification types.

Use separate matchers to run different handlers depending on the notification type. This configuration triggers a permission-specific alert script when Claude needs permission approval and a different notification when Claude has been idle:

```
{
  "hooks": {
    "Notification": [
      {
        "matcher": "permission_prompt",
        "hooks": [
          {
            "type": "command",
            "command": "/path/to/permission-alert.sh"
          }
        ]
      },
      {
        "matcher": "idle_prompt",
        "hooks": [
          {
            "type": "command",
            "command": "/path/to/idle-notification.sh"
          }
        ]
      }
    ]
  }
}
```

Notification input

In addition to the common input fields, Notification hooks receive `message` with the notification text, an optional `title`, and `notification_type` indicating which type fired.

```
{
  "session_id": "abc123",
  "transcript_path": "/Users/.../.claude/projects/.../
00893aaf-19fa-41d2-8238-13269b9b3ca0.jsonl",
  "cwd": "/Users/...",
  "permission_mode": "default",
  "hook_event_name": "Notification",
  "message": "Claude needs your permission to use Bash",
  "title": "Permission needed",
  "notification_type": "permission_prompt"
}
```

Notification hooks cannot block or modify notifications. In addition to the JSON output fields available to all hooks, you can return `additionalContext` to add context to the conversation:

Field	Description
`additionalContext`	String added to Claude's context

SubagentStart

Runs when a Claude Code subagent is spawned via the Agent tool. Supports matchers to filter by agent type name (built-in agents like `Bash`, `Explore`, `Plan`, or custom agent names from `.claude/agents/`).

SubagentStart input

In addition to the common input fields, SubagentStart hooks receive `agent_id` with the unique identifier for the subagent and `agent_type` with the agent name (built-in agents like `"Bash"`, `"Explore"`, `"Plan"`, or custom agent names).

```
{
  "session_id": "abc123",
  "transcript_path": "/Users/.../.claude/projects/.../
00893aaf-19fa-41d2-8238-13269b9b3ca0.jsonl",
  "cwd": "/Users/...",
  "permission_mode": "default",
  "hook_event_name": "SubagentStart",
  "agent_id": "agent-abc123",
  "agent_type": "Explore"
}
```

SubagentStart hooks cannot block subagent creation, but they can inject context into the subagent. In addition to the JSON output fields available to all hooks, you can return:

Field	Description
`additionalContext`	String added to the subagent's context

```
{
  "hookSpecificOutput": {
    "hookEventName": "SubagentStart",
    "additionalContext": "Follow security guidelines for this task"
  }
}
```

SubagentStop

Runs when a Claude Code subagent has finished responding. Matches on agent type, same values as SubagentStart.

SubagentStop input

In addition to the common input fields, SubagentStop hooks receive `stop_hook_active`, `agent_id`, `agent_type`, `agent_transcript_path`, and `last_assistant_message`. The `agent_type` field is the value used for matcher filtering. The `transcript_path` is the main session's transcript, while `agent_transcript_path` is the subagent's own transcript stored in a nested `subagents/` folder. The `last_assistant_message` field contains the text content of the subagent's final response, so hooks can access it without parsing the transcript file.

```
{
  "session_id": "abc123",
  "transcript_path": "~/.claude/projects/.../abc123.jsonl",
  "cwd": "/Users/...",
  "permission_mode": "default",
  "hook_event_name": "SubagentStop",
  "stop_hook_active": false,
  "agent_id": "def456",
  "agent_type": "Explore",
  "agent_transcript_path": "~/.claude/projects/.../abc123/subagents/agent-
def456.jsonl",
  "last_assistant_message": "Analysis complete. Found 3 potential issues..."
}
```

SubagentStop hooks use the same decision control format as Stop hooks.

Stop

Runs when the main Claude Code agent has finished responding. Does not run if the stoppage occurred due to a user interrupt.

Stop input

In addition to the common input fields, Stop hooks receive `stop_hook_active` and `last_assistant_message`. The `stop_hook_active` field is `true` when Claude Code is already continuing as a result of a stop hook. Check this value or process the transcript to prevent Claude Code from running indefinitely. The `last_assistant_message` field contains the text content of Claude's final response, so hooks can access it without parsing the transcript file.

```
{
  "session_id": "abc123",
  "transcript_path": "~/.claude/projects/.../
00893aaf-19fa-41d2-8238-13269b9b3ca0.jsonl",
  "cwd": "/Users/...",
  "permission_mode": "default",
  "hook_event_name": "Stop",
  "stop_hook_active": true,
  "last_assistant_message": "I've completed the refactoring. Here's a summary..."
}
```

Stop decision control

`Stop` and `SubagentStop` hooks can control whether Claude continues. In addition to the JSON output fields available to all hooks, your hook script can return these event-specific fields:

Field	Description
`decision`	`"block"` prevents Claude from stopping. Omit to allow Claude to stop
`reason`	Required when `decision` is `"block"`. Tells Claude why it should continue

```
{
  "decision": "block",
  "reason": "Must be provided when Claude is blocked from stopping"
}
```

TeammateIdle

Runs when an agent team teammate is about to go idle after finishing its turn. Use this to enforce quality gates before a teammate stops working, such as requiring passing lint checks or verifying that output files exist.

When a `TeammateIdle` hook exits with code 2, the teammate receives the stderr message as feedback and continues working instead of going idle. To stop the teammate entirely instead of re-running it, return JSON with `{"continue": false, "stopReason": "..."}`. TeammateIdle hooks do not support matchers and fire on every occurrence.

TeammateIdle input

In addition to the common input fields, TeammateIdle hooks receive `teammate_name` and `team_name`.

```
{
  "session_id": "abc123",
  "transcript_path": "/Users/.../.claude/projects/.../
00893aaf-19fa-41d2-8238-13269b9b3ca0.jsonl",
  "cwd": "/Users/...",
  "permission_mode": "default",
  "hook_event_name": "TeammateIdle",
  "teammate_name": "researcher",
  "team_name": "my-project"
}
```

Field	Description
`teammate_name`	Name of the teammate that is about to go idle
`team_name`	Name of the team

TeammateIdle decision control

TeammateIdle hooks support two ways to control teammate behavior:

- **Exit code 2**: the teammate receives the stderr message as feedback and continues working instead of going idle.
- **JSON** `{"continue": false, "stopReason": "..."}` : stops the teammate entirely, matching `Stop` hook behavior. The `stopReason` is shown to the user.

This example checks that a build artifact exists before allowing a teammate to go idle:

```
#!/bin/bash

if [ ! -f "./dist/output.js" ]; then
  echo "Build artifact missing. Run the build before stopping." >&2
  exit 2
fi

exit 0
```

TaskCompleted

Runs when a task is being marked as completed. This fires in two situations: when any agent explicitly marks a task as completed through the TaskUpdate tool, or when an agent team teammate finishes its turn with in-progress tasks. Use this to enforce completion criteria like passing tests or lint checks before a task can close.

When a `TaskCompleted` hook exits with code 2, the task is not marked as completed and the stderr message is fed back to the model as feedback. To stop the teammate entirely instead of re-running it, return JSON with `{"continue": false, "stopReason": "..."}`. TaskCompleted hooks do not support matchers and fire on every occurrence.

TaskCompleted input

In addition to the common input fields, TaskCompleted hooks receive `task_id`, `task_subject`, and optionally `task_description`, `teammate_name`, and `team_name`.

```
{
  "session_id": "abc123",
  "transcript_path": "/Users/.../.claude/projects/.../
00893aaf-19fa-41d2-8238-13269b9b3ca0.jsonl",
  "cwd": "/Users/...",
  "permission_mode": "default",
  "hook_event_name": "TaskCompleted",
  "task_id": "task-001",
  "task_subject": "Implement user authentication",
  "task_description": "Add login and signup endpoints",
  "teammate_name": "implementer",
  "team_name": "my-project"
}
```

Field	Description
`task_id`	Identifier of the task being completed
`task_subject`	Title of the task
`task_description`	Detailed description of the task. May be absent
`teammate_name`	Name of the teammate completing the task. May be absent
`team_name`	Name of the team. May be absent

TaskCompleted decision control

TaskCompleted hooks support two ways to control task completion:

- **Exit code 2**: the task is not marked as completed and the stderr message is fed back to the model as feedback.
- **JSON** `{"continue": false, "stopReason": "..."}` : stops the teammate entirely, matching `Stop` hook behavior. The `stopReason` is shown to the user.

This example runs tests and blocks task completion if they fail:

```
#!/bin/bash
INPUT=$(cat)
TASK_SUBJECT=$(echo "$INPUT" | jq -r '.task_subject')

## Run the test suite
if ! npm test 2>&1; then
  echo "Tests not passing. Fix failing tests before completing: $TASK_SUBJECT" >&2
  exit 2
fi

exit 0
```

ConfigChange

Runs when a configuration file changes during a session. Use this to audit settings changes, enforce security policies, or block unauthorized modifications to configuration files.

ConfigChange hooks fire for changes to settings files, managed policy settings, and skill files. The `source` field in the input tells you which type of configuration changed, and the optional `file_path` field provides the path to the changed file.

The matcher filters on the configuration source:

Matcher	When it fires
`user_settings`	`~/.claude/settings.json` changes
`project_settings`	`.claude/settings.json` changes
`local_settings`	`.claude/settings.local.json` changes
`policy_settings`	Managed policy settings change
`skills`	A skill file in `.claude/skills/` changes

This example logs all configuration changes for security auditing:

```
{
  "hooks": {
    "ConfigChange": [
      {
        "hooks": [
          {
            "type": "command",
            "command": "\"$CLAUDE_PROJECT_DIR\"/.claude/hooks/audit-config-
change.sh"
          }
        ]
      }
    ]
  }
}
```

ConfigChange input

In addition to the common input fields, ConfigChange hooks receive `source` and optionally `file_path`. The `source` field indicates which configuration type changed, and `file_path` provides the path to the specific file that was modified.

```
{
  "session_id": "abc123",
  "transcript_path": "/Users/.../.claude/projects/.../
00893aaf-19fa-41d2-8238-13269b9b3ca0.jsonl",
  "cwd": "/Users/...",
  "permission_mode": "default",
  "hook_event_name": "ConfigChange",
  "source": "project_settings",
  "file_path": "/Users/.../my-project/.claude/settings.json"
}
```

ConfigChange decision control

ConfigChange hooks can block configuration changes from taking effect. Use exit code 2 or a JSON `decision` to prevent the change. When blocked, the new settings are not applied to the running session.

Field	Description
`decision`	`"block"` prevents the configuration change from being applied. Omit to allow the change
`reason`	Explanation shown to the user when `decision` is `"block"`

```
{
  "decision": "block",
  "reason": "Configuration changes to project settings require admin approval"
}
```

`policy_settings` changes cannot be blocked. Hooks still fire for `policy_settings` sources, so you can use them for audit logging, but any blocking decision is ignored. This ensures enterprise-managed settings always take effect.

WorktreeCreate

When you run `claude --worktree` or a subagent uses `isolation: "worktree"`, Claude Code creates an isolated working copy using `git worktree`. If you configure a WorktreeCreate hook, it replaces the default git behavior, letting you use a different version control system like SVN, Perforce, or Mercurial.

The hook must print the absolute path to the created worktree directory on stdout. Claude Code uses this path as the working directory for the isolated session.

This example creates an SVN working copy and prints the path for Claude Code to use. Replace the repository URL with your own:

```
{
  "hooks": {
    "WorktreeCreate": [
      {
        "hooks": [
          {
            "type": "command",
            "command": "bash -c 'NAME=$(jq -r .name); DIR=\"$HOME/.claude/
worktrees/$NAME\"; svn checkout https://svn.example.com/repo/trunk
\"$DIR\" >&2 && echo \"$DIR\"'"
          }
        ]
      }
    ]
  }
}
```

The hook reads the worktree `name` from the JSON input on stdin, checks out a fresh copy into a new directory, and prints the directory path. The `echo` on the last line is what Claude Code reads as the worktree path. Redirect any other output to stderr so it doesn't interfere with the path.

WorktreeCreate input

In addition to the common input fields, WorktreeCreate hooks receive the `name` field. This is a slug identifier for the new worktree, either specified by the user or auto-generated (for example, `bold-oak-a3f2`).

```
{
  "session_id": "abc123",
  "transcript_path": "/Users/.../.claude/projects/.../
00893aaf-19fa-41d2-8238-13269b9b3ca0.jsonl",
  "cwd": "/Users/...",
  "hook_event_name": "WorktreeCreate",
  "name": "feature-auth"
}
```

WorktreeCreate output

The hook must print the absolute path to the created worktree directory on stdout. If the hook fails or produces no output, worktree creation fails with an error.

WorktreeCreate hooks do not use the standard allow/block decision model. Instead, the hook's success or failure determines the outcome. Only `type: "command"` hooks are supported.

WorktreeRemove

The cleanup counterpart to WorktreeCreate. This hook fires when a worktree is being removed, either when you exit a `--worktree` session and choose to remove it, or when a subagent with `isolation: "worktree"` finishes. For git-based worktrees, Claude handles cleanup automatically with `git worktree remove`. If you configured a WorktreeCreate hook for a non-git version control system, pair it with a WorktreeRemove hook to handle cleanup. Without one, the worktree directory is left on disk.

Claude Code passes the path that WorktreeCreate printed on stdout as `worktree_path` in the hook input. This example reads that path and removes the directory:

```
{
  "hooks": {
    "WorktreeRemove": [
      {
        "hooks": [
          {
            "type": "command",
            "command": "bash -c 'jq -r .worktree_path | xargs rm -rf'"
          }
        ]
      }
    ]
  }
}
```

WorktreeRemove input

In addition to the common input fields, WorktreeRemove hooks receive the `worktree_path` field, which is the absolute path to the worktree being removed.

```
{
  "session_id": "abc123",
  "transcript_path": "/Users/.../.claude/projects/.../
00893aaf-19fa-41d2-8238-13269b9b3ca0.jsonl",
  "cwd": "/Users/...",
  "hook_event_name": "WorktreeRemove",
  "worktree_path": "/Users/.../my-project/.claude/worktrees/feature-auth"
}
```

WorktreeRemove hooks have no decision control. They cannot block worktree removal but can perform cleanup tasks like removing version control state or archiving changes. Hook failures are logged in debug mode only. Only `type: "command"` hooks are supported.

PreCompact

Runs before Claude Code is about to run a compact operation.

The matcher value indicates whether compaction was triggered manually or automatically:

Matcher	When it fires
`manual`	`/compact`
`auto`	Auto-compact when the context window is full

PreCompact input

In addition to the common input fields, PreCompact hooks receive `trigger` and `custom_instructions`. For `manual`, `custom_instructions` contains what the user passes into `/compact`. For `auto`, `custom_instructions` is empty.

```
{
  "session_id": "abc123",
  "transcript_path": "/Users/.../.claude/projects/.../
00893aaf-19fa-41d2-8238-13269b9b3ca0.jsonl",
  "cwd": "/Users/...",
  "permission_mode": "default",
  "hook_event_name": "PreCompact",
  "trigger": "manual",
  "custom_instructions": ""
}
```

PostCompact

Runs after Claude Code completes a compact operation. Use this event to react to the new compacted state, for example to log the generated summary or update external state.

The same matcher values apply as for `PreCompact`:

Matcher	When it fires
`manual`	After `/compact`
`auto`	After auto-compact when the context window is full

PostCompact input

In addition to the common input fields, PostCompact hooks receive `trigger` and `compact_summary`. The `compact_summary` field contains the conversation summary generated by the compact operation.

```
{
  "session_id": "abc123",
  "transcript_path": "/Users/.../.claude/projects/.../
00893aaf-19fa-41d2-8238-13269b9b3ca0.jsonl",
  "cwd": "/Users/...",
  "permission_mode": "default",
  "hook_event_name": "PostCompact",
  "trigger": "manual",
  "compact_summary": "Summary of the compacted conversation..."
}
```

PostCompact hooks have no decision control. They cannot affect the compaction result but can perform follow-up tasks.

SessionEnd

Runs when a Claude Code session ends. Useful for cleanup tasks, logging session statistics, or saving session state. Supports matchers to filter by exit reason.

The `reason` field in the hook input indicates why the session ended:

Reason	Description
`clear`	Session cleared with `/clear` command
`logout`	User logged out
`prompt_input_exit`	User exited while prompt input was visible
`bypass_permissions_disabled`	Bypass permissions mode was disabled
`other`	Other exit reasons

SessionEnd input

In addition to the common input fields, SessionEnd hooks receive a `reason` field indicating why the session ended. See the reason table above for all values.

```
{
  "session_id": "abc123",
  "transcript_path": "/Users/.../.claude/projects/.../
00893aaf-19fa-41d2-8238-13269b9b3ca0.jsonl",
  "cwd": "/Users/...",
  "permission_mode": "default",
  "hook_event_name": "SessionEnd",
  "reason": "other"
}
```

SessionEnd hooks have no decision control. They cannot block session termination but can perform cleanup tasks.

SessionEnd hooks have a default timeout of 1.5 seconds. This applies to both session exit and `/clear`. If your hooks need more time, set the `CLAUDE_CODE_SESSIONEND_HOOKS_TIMEOUT_MS` environment variable to a higher value in milliseconds. Any per-hook `timeout` setting is also capped by this value.

```
CLAUDE_CODE_SESSIONEND_HOOKS_TIMEOUT_MS=5000 claude
```

Elicitation

Runs when an MCP server requests user input mid-task. By default, Claude Code shows an interactive dialog for the user to respond. Hooks can intercept this request and respond programmatically, skipping the dialog entirely.

The matcher field matches against the MCP server name.

Elicitation input

In addition to the common input fields, Elicitation hooks receive `mcp_server_name`, `message`, and optional `mode`, `url`, `elicitation_id`, and `requested_schema` fields.

For form-mode elicitation (the most common case):

```
{
  "session_id": "abc123",
  "transcript_path": "/Users/.../.claude/projects/.../
00893aaf-19fa-41d2-8238-13269b9b3ca0.jsonl",
  "cwd": "/Users/...",
  "permission_mode": "default",
  "hook_event_name": "Elicitation",
  "mcp_server_name": "my-mcp-server",
  "message": "Please provide your credentials",
  "mode": "form",
  "requested_schema": {
    "type": "object",
    "properties": {
      "username": { "type": "string", "title": "Username" }
    }
  }
}
```

For URL-mode elicitation (browser-based authentication):

```
{
  "session_id": "abc123",
  "transcript_path": "/Users/.../.claude/projects/.../
00893aaf-19fa-41d2-8238-13269b9b3ca0.jsonl",
  "cwd": "/Users/...",
  "permission_mode": "default",
  "hook_event_name": "Elicitation",
  "mcp_server_name": "my-mcp-server",
  "message": "Please authenticate",
  "mode": "url",
  "url": "https://auth.example.com/login"
}
```

Elicitation output

To respond programmatically without showing the dialog, return a JSON object with `hookSpecificOutput`:

```
{
  "hookSpecificOutput": {
    "hookEventName": "Elicitation",
    "action": "accept",
    "content": {
      "username": "alice"
    }
  }
}
```

Field	Values	Description
`action`	`accept`, `decline`, `cancel`	Whether to accept, decline, or cancel the request
`content`	object	Form field values to submit. Only used when `action` is `accept`

Exit code 2 denies the elicitation and shows stderr to the user.

ElicitationResult

Runs after a user responds to an MCP elicitation. Hooks can observe, modify, or block the response before it is sent back to the MCP server.

The matcher field matches against the MCP server name.

ElicitationResult input

In addition to the common input fields, ElicitationResult hooks receive `mcp_server_name`, `action`, and optional `mode`, `elicitation_id`, and `content` fields.

```
{
  "session_id": "abc123",
  "transcript_path": "/Users/.../.claude/projects/.../
00893aaf-19fa-41d2-8238-13269b9b3ca0.jsonl",
  "cwd": "/Users/...",
  "permission_mode": "default",
  "hook_event_name": "ElicitationResult",
  "mcp_server_name": "my-mcp-server",
  "action": "accept",
  "content": { "username": "alice" },
  "mode": "form",
  "elicitation_id": "elicit-123"
}
```

ElicitationResult output

To override the user's response, return a JSON object with `hookSpecificOutput`:

```
{
  "hookSpecificOutput": {
    "hookEventName": "ElicitationResult",
    "action": "decline",
    "content": {}
  }
}
```

Field	Values	Description
`action`	`accept`, `decline`, `cancel`	Overrides the user's action
`content`	object	Overrides form field values. Only meaningful when `action` is `accept`

Exit code 2 blocks the response, changing the effective action to `decline`.

Prompt-based hooks

In addition to command and HTTP hooks, Claude Code supports prompt-based hooks (`type: "prompt"`) that use an LLM to evaluate whether to allow or block an action, and agent hooks (`type: "agent"`) that spawn an agentic verifier with tool access. Not all events support every hook type.

Events that support all four hook types (`command` , `http` , `prompt` , and `agent`):

- `PermissionRequest`
- `PostToolUse`
- `PostToolUseFailure`
- `PreToolUse`
- `Stop`
- `SubagentStop`
- `TaskCompleted`
- `UserPromptSubmit`

Events that only support `type: "command"` hooks:

- `ConfigChange`
- `Elicitation`
- `ElicitationResult`
- `InstructionsLoaded`
- `Notification`
- `PostCompact`
- `PreCompact`
- `SessionEnd`
- `SessionStart`
- `SubagentStart`
- `TeammateIdle`
- `WorktreeCreate`
- `WorktreeRemove`

How prompt-based hooks work

Instead of executing a Bash command, prompt-based hooks:

1. Send the hook input and your prompt to a Claude model, Haiku by default

2. The LLM responds with structured JSON containing a decision
3. Claude Code processes the decision automatically

Prompt hook configuration

Set `type` to `"prompt"` and provide a `prompt` string instead of a `command`. Use the `$ARGUMENTS` placeholder to inject the hook's JSON input data into your prompt text. Claude Code sends the combined prompt and input to a fast Claude model, which returns a JSON decision.

This `Stop` hook asks the LLM to evaluate whether all tasks are complete before allowing Claude to finish:

```
{
  "hooks": {
    "Stop": [
      {
        "hooks": [
          {
            "type": "prompt",
            "prompt": "Evaluate if Claude should stop: $ARGUMENTS. Check if all
tasks are complete."
          }
        ]
      }
    ]
  }
}
```

Field	Required	Description
`type`	yes	Must be `"prompt"`
`prompt`	yes	The prompt text to send to the LLM. Use `$ARGUMENTS` as a placeholder for the hook input JSON. If `$ARGUMENTS` is not present, input JSON is appended to the prompt

Field	Required	Description
`model`	no	Model to use for evaluation. Defaults to a fast model
`timeout`	no	Timeout in seconds. Default: 30

Response schema

The LLM must respond with JSON containing:

```
{
  "ok": true | false,
  "reason": "Explanation for the decision"
}
```

Field	Description
`ok`	`true` allows the action, `false` prevents it
`reason`	Required when `ok` is `false`. Explanation shown to Claude

Example: Multi-criteria Stop hook

This `Stop` hook uses a detailed prompt to check three conditions before allowing Claude to stop. If `"ok"` is `false`, Claude continues working with the provided reason as its next instruction. `SubagentStop` hooks use the same format to evaluate whether a subagent should stop:

```
{
  "hooks": {
    "Stop": [
      {
        "hooks": [
          {
            "type": "prompt",
            "prompt": "You are evaluating whether Claude should stop working.
Context: $ARGUMENTS\n\nAnalyze the conversation and determine if:\n1. All user-
requested tasks are complete\n2. Any errors need to be addressed\n3. Follow-up
work is needed\n\nRespond with JSON: {\"ok\": true} to allow stopping, or
{\"ok\": false, \"reason\": \"your explanation\"} to continue working.",
            "timeout": 30
          }
        ]
      }
    ]
  }
}
```

Agent-based hooks

Agent-based hooks (`type: "agent"`) are like prompt-based hooks but with multi-turn tool access. Instead of a single LLM call, an agent hook spawns a subagent that can read files, search code, and inspect the codebase to verify conditions. Agent hooks support the same events as prompt-based hooks.

How agent hooks work

When an agent hook fires:

1. Claude Code spawns a subagent with your prompt and the hook's JSON input
2. The subagent can use tools like Read, Grep, and Glob to investigate
3. After up to 50 turns, the subagent returns a structured `{ "ok": true/false }` decision
4. Claude Code processes the decision the same way as a prompt hook

Agent hooks are useful when verification requires inspecting actual files or test output, not just evaluating the hook input data alone.

Agent hook configuration

Set `type` to `"agent"` and provide a `prompt` string. The configuration fields are the same as prompt hooks, with a longer default timeout:

Field	Required	Description
`type`	yes	Must be `"agent"`
`prompt`	yes	Prompt describing what to verify. Use `$ARGUMENTS` as a placeholder for the hook input JSON
`model`	no	Model to use. Defaults to a fast model
`timeout`	no	Timeout in seconds. Default: 60

The response schema is the same as prompt hooks: `{ "ok": true }` to allow or `{ "ok": false, "reason": "..." }` to block.

This `Stop` hook verifies that all unit tests pass before allowing Claude to finish:

```
{
  "hooks": {
    "Stop": [
      {
        "hooks": [
          {
            "type": "agent",
            "prompt": "Verify that all unit tests pass. Run the test suite and
check the results. $ARGUMENTS",
            "timeout": 120
          }
        ]
      }
    ]
  }
}
```

Run hooks in the background

By default, hooks block Claude's execution until they complete. For long-running tasks like deployments, test suites, or external API calls, set `"async": true` to run the hook in the background while Claude continues working. Async hooks cannot block or control Claude's behavior: response fields like `decision`, `permissionDecision`, and `continue` have no effect, because the action they would have controlled has already completed.

Configure an async hook

Add `"async": true` to a command hook's configuration to run it in the background without blocking Claude. This field is only available on `type: "command"` hooks.

This hook runs a test script after every `Write` tool call. Claude continues working immediately while `run-tests.sh` executes for up to 120 seconds. When the script finishes, its output is delivered on the next conversation turn:

```
{
  "hooks": {
    "PostToolUse": [
      {
        "matcher": "Write",
        "hooks": [
          {
            "type": "command",
            "command": "/path/to/run-tests.sh",
            "async": true,
            "timeout": 120
          }
        ]
      }
    ]
  }
}
```

The `timeout` field sets the maximum time in seconds for the background process. If not specified, async hooks use the same 10-minute default as sync hooks.

How async hooks execute

When an async hook fires, Claude Code starts the hook process and immediately continues without waiting for it to finish. The hook receives the same JSON input via stdin as a synchronous hook.

After the background process exits, if the hook produced a JSON response with a `systemMessage` or `additionalContext` field, that content is delivered to Claude as context on the next conversation turn.

Async hook completion notifications are suppressed by default. To see them, enable verbose mode with `Ctrl+O` or start Claude Code with `--verbose`.

Example: run tests after file changes

This hook starts a test suite in the background whenever Claude writes a file, then reports the results back to Claude when the tests finish. Save this script to `.claude/hooks/run-tests-async.sh` in your project and make it executable with `chmod +x`:

```
#!/bin/bash
## run-tests-async.sh

## Read hook input from stdin
INPUT=$(cat)
FILE_PATH=$(echo "$INPUT" | jq -r '.tool_input.file_path // empty')

## Only run tests for source files
if [[ "$FILE_PATH" != *.ts && "$FILE_PATH" != *.js ]]; then
  exit 0
fi

## Run tests and report results via systemMessage
RESULT=$(npm test 2>&1)
EXIT_CODE=$?

if [ $EXIT_CODE -eq 0 ]; then
  echo "{\"systemMessage\": \"Tests passed after editing $FILE_PATH\"}"
else
  echo "{\"systemMessage\": \"Tests failed after editing $FILE_PATH: $RESULT\"}"
fi
```

Then add this configuration to `.claude/settings.json` in your project root. The `async: true` flag lets Claude keep working while tests run:

```
{
  "hooks": {
    "PostToolUse": [
      {
        "matcher": "Write|Edit",
        "hooks": [
          {
            "type": "command",
            "command": "\"$CLAUDE_PROJECT_DIR\"/.claude/hooks/run-tests-async.sh",
            "async": true,
            "timeout": 300
          }
        ]
      }
    ]
  }
}
```

Limitations

Async hooks have several constraints compared to synchronous hooks:

- Only `type: "command"` hooks support `async`. Prompt-based hooks cannot run asynchronously.
- Async hooks cannot block tool calls or return decisions. By the time the hook completes, the triggering action has already proceeded.
- Hook output is delivered on the next conversation turn. If the session is idle, the response waits until the next user interaction.
- Each execution creates a separate background process. There is no deduplication across multiple firings of the same async hook.

Security considerations

Disclaimer

Command hooks run with your system user's full permissions.

> **Warning:**
>
> Command hooks execute shell commands with your full user permissions. They can modify, delete, or access any files your user account can access. Review and test all hook commands before adding them to your configuration.

Security best practices

Keep these practices in mind when writing hooks:

- **Validate and sanitize inputs**: never trust input data blindly
- **Always quote shell variables**: use `"$VAR"` not `$VAR`
- **Block path traversal**: check for `..` in file paths
- **Use absolute paths**: specify full paths for scripts, using `"$CLAUDE_PROJECT_DIR"` for the project root
- **Skip sensitive files**: avoid `.env`, `.git/`, keys, etc.

Debug hooks

Run `claude --debug` to see hook execution details, including which hooks matched, their exit codes, and output. Toggle verbose mode with `Ctrl+O` to see hook progress in the transcript.

```
[DEBUG] Executing hooks for PostToolUse:Write
[DEBUG] Getting matching hook commands for PostToolUse with query: Write
[DEBUG] Found 1 hook matchers in settings
[DEBUG] Matched 1 hooks for query "Write"
[DEBUG] Found 1 hook commands to execute
[DEBUG] Executing hook command: <Your command> with timeout 600000ms
[DEBUG] Hook command completed with status 0: <Your stdout>
```

For troubleshooting common issues like hooks not firing, infinite Stop hook loops, or configuration errors, see Limitations and troubleshooting in the guide.

Automate workflows with hooks

> *Run shell commands automatically when Claude Code edits files, finishes tasks, or needs input. Format code, send notifications, validate commands, and enforce project rules.*

Hooks are user-defined shell commands that execute at specific points in Claude Code's lifecycle. They provide deterministic control over Claude Code's behavior, ensuring certain actions always happen rather than relying on the LLM to choose to run them. Use hooks to enforce project rules, automate repetitive tasks, and integrate Claude Code with your existing tools.

For decisions that require judgment rather than deterministic rules, you can also use prompt-based hooks or agent-based hooks that use a Claude model to evaluate conditions.

For other ways to extend Claude Code, see skills for giving Claude additional instructions and executable commands, subagents for running tasks in isolated contexts, and plugins for packaging extensions to share across projects.

> **Tip:**
>
> This guide covers common use cases and how to get started. For full event schemas, JSON input/output formats, and advanced features like async hooks and MCP tool hooks, see the Hooks reference.

Set up your first hook

To create a hook, add a `hooks` block to a settings file. This walkthrough creates a desktop notification hook, so you get alerted whenever Claude is waiting for your input instead of watching the terminal.

Step 1: Add the hook to your settings

Open `~/.claude/settings.json` and add a `Notification` hook. The example below uses `osascript` for macOS; see Get notified when Claude needs input for Linux and Windows commands.

```
{
  "hooks": {
    "Notification": [
      {
        "matcher": "",
        "hooks": [
          {
            "type": "command",
            "command": "osascript -e 'display notification \"Claude Code needs
your attention\" with title \"Claude Code\"'"
          }
        ]
      }
    ]
  }
}
```

If your settings file already has a `hooks` key, merge the `Notification` entry into it rather than replacing the whole object. You can also ask Claude to write the hook for you by describing what you want in the CLI.

Step 2: Verify the configuration

Type `/hooks` to open the hooks browser. You'll see a list of all available hook events, with a count next to each event that has hooks configured. Select `Notification` to confirm your new hook appears in the list. Selecting the hook shows its details: the event, matcher, type, source file, and command.

Step 3: Test the hook

Press `Esc` to return to the CLI. Ask Claude to do something that requires permission, then switch away from the terminal. You should receive a desktop notification.

> **Tip:**
>
> The `/hooks` menu is read-only. To add, modify, or remove hooks, edit your settings JSON directly or ask Claude to make the change.

What you can automate

Hooks let you run code at key points in Claude Code's lifecycle: format files after edits, block commands before they execute, send notifications when Claude needs input, inject context at session start, and more. For the full list of hook events, see the Hooks reference.

Each example includes a ready-to-use configuration block that you add to a settings file. The most common patterns:

- Get notified when Claude needs input
- Auto-format code after edits
- Block edits to protected files
- Re-inject context after compaction
- Audit configuration changes
- Auto-approve specific permission prompts

Get notified when Claude needs input

Get a desktop notification whenever Claude finishes working and needs your input, so you can switch to other tasks without checking the terminal.

This hook uses the `Notification` event, which fires when Claude is waiting for input or permission. Each tab below uses the platform's native notification command. Add this to `~/.claude/settings.json`:

macOS

```
{
  "hooks": {
    "Notification": [
      {
        "matcher": "",
        "hooks": [
          {
            "type": "command",
            "command": "osascript -e 'display notification \"Claude Code needs
your attention\" with title \"Claude Code\"'"
          }
        ]
      }
    ]
  }
}
```

Linux

```
{
  "hooks": {
    "Notification": [
      {
        "matcher": "",
        "hooks": [
          {
            "type": "command",
            "command": "notify-send 'Claude Code' 'Claude Code needs your
attention'"
          }
        ]
      }
    ]
  }
}
```

Windows (PowerShell)

```
{
  "hooks": {
    "Notification": [
      {
        "matcher": "",
        "hooks": [
          {
            "type": "command",
            "command": "powershell.exe -Command \"[System.Reflection.Assembly]::Lo
adWithPartialName('System.Windows.Forms');
[System.Windows.Forms.MessageBox]::Show('Claude Code needs your attention',
'Claude Code')\""
          }
        ]
      }
    ]
  }
}
```

Auto-format code after edits

Automatically run Prettier on every file Claude edits, so formatting stays consistent without manual intervention.

This hook uses the `PostToolUse` event with an `Edit|Write` matcher, so it runs only after file-editing tools. The command extracts the edited file path with `jq` and passes it to Prettier. Add this to `.claude/settings.json` in your project root:

```
{
  "hooks": {
    "PostToolUse": [
      {
        "matcher": "Edit|Write",
        "hooks": [
          {
            "type": "command",
            "command": "jq -r '.tool_input.file_path' | xargs npx prettier --
write"
          }
        ]
      }
    ]
  }
}
```

> **Note:**
>
> The Bash examples on this page use `jq` for JSON parsing. Install it with `brew install jq` (macOS), `apt-get install jq` (Debian/Ubuntu), or see `jq` downloads.

Block edits to protected files

Prevent Claude from modifying sensitive files like `.env`, `package-lock.json`, or anything in `.git/`. Claude receives feedback explaining why the edit was blocked, so it can adjust its approach.

This example uses a separate script file that the hook calls. The script checks the target file path against a list of protected patterns and exits with code 2 to block the edit.

Step 1: Create the hook script

Save this to `.claude/hooks/protect-files.sh`:

```
#!/bin/bash
## protect-files.sh

INPUT=$(cat)
FILE_PATH=$(echo "$INPUT" | jq -r '.tool_input.file_path // empty')

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

for pattern in "${PROTECTED_PATTERNS[@]}"; do
  if [[ "$FILE_PATH" == *"$pattern"* ]]; then
    echo "Blocked: $FILE_PATH matches protected pattern '$pattern'" >&2
    exit 2
  fi
done

exit 0
```

Step 2: Make the script executable (macOS/Linux)

Hook scripts must be executable for Claude Code to run them:

```
chmod +x .claude/hooks/protect-files.sh
```

Step 3: Register the hook

Add a `PreToolUse` hook to `.claude/settings.json` that runs the script before any `Edit` or `Write` tool call:

```
{
  "hooks": {
    "PreToolUse": [
      {
        "matcher": "Edit|Write",
        "hooks": [
          {
            "type": "command",
            "command": "\"$CLAUDE_PROJECT_DIR\"/.claude/hooks/protect-files.sh"
          }
        ]
      }
    ]
  }
}
```

Re-inject context after compaction

When Claude's context window fills up, compaction summarizes the conversation to free space. This can lose important details. Use a `SessionStart` hook with a `compact` matcher to re-inject critical context after every compaction.

Any text your command writes to stdout is added to Claude's context. This example reminds Claude of project conventions and recent work. Add this to `.claude/settings.json` in your project root:

```
{
  "hooks": {
    "SessionStart": [
      {
        "matcher": "compact",
        "hooks": [
          {
            "type": "command",
            "command": "echo 'Reminder: use Bun, not npm. Run bun test before
committing. Current sprint: auth refactor.'"
          }
        ]
      }
    ]
  }
}
```

You can replace the `echo` with any command that produces dynamic output, like `git log --oneline -5` to show recent commits. For injecting context on every session start, consider using CLAUDE.md instead. For environment variables, see `CLAUDE_ENV_FILE` in the reference.

Audit configuration changes

Track when settings or skills files change during a session. The `ConfigChange` event fires when an external process or editor modifies a configuration file, so you can log changes for compliance or block unauthorized modifications.

This example appends each change to an audit log. Add this to `~/.claude/settings.json`:

```
{
  "hooks": {
    "ConfigChange": [
      {
        "matcher": "",
        "hooks": [
          {
            "type": "command",
            "command": "jq -c '{timestamp: now | todate, source: .source,
file: .file_path}' >> ~/claude-config-audit.log"
          }
        ]
      }
    ]
  }
}
```

The matcher filters by configuration type: `user_settings`, `project_settings`, `local_settings`, `policy_settings`, or `skills`. To block a change from taking effect, exit with code 2 or return `{"decision": "block"}`. See the ConfigChange reference for the full input schema.

Auto-approve specific permission prompts

Skip the approval dialog for tool calls you always allow. This example auto-approves `ExitPlanMode`, the tool Claude calls when it finishes presenting a plan and asks to proceed, so you aren't prompted every time a plan is ready.

Unlike the exit-code examples above, auto-approval requires your hook to write a JSON decision to stdout. A `PermissionRequest` hook fires when Claude Code is about to show a permission dialog, and returning `"behavior": "allow"` answers it on your behalf.

The matcher scopes the hook to `ExitPlanMode` only, so no other prompts are affected. Add this to `~/.claude/settings.json`:

```
{
  "hooks": {
    "PermissionRequest": [
      {
        "matcher": "ExitPlanMode",
        "hooks": [
          {
            "type": "command",
            "command": "echo '{\"hookSpecificOutput\": {\"hookEventName\": \"Permi
ssionRequest\", \"decision\": {\"behavior\": \"allow\"}}}'"
          }
        ]
      }
    ]
  }
}
```

When the hook approves, Claude Code exits plan mode and restores whatever permission mode was active before you entered plan mode. The transcript shows “Allowed by PermissionRequest hook” where the dialog would have appeared. The hook path always keeps the current conversation: it cannot clear context and start a fresh implementation session the way the dialog can.

To set a specific permission mode instead, your hook’s output can include an `updatedPermissions` array with a `setMode` entry. The `mode` value is any permission mode like `default`, `acceptEdits`, or `bypassPermissions`, and `destination: "session"` applies it for the current session only.

To switch the session to `acceptEdits`, your hook writes this JSON to stdout:

```
{
  "hookSpecificOutput": {
    "hookEventName": "PermissionRequest",
    "decision": {
      "behavior": "allow",
      "updatedPermissions": [
        { "type": "setMode", "mode": "acceptEdits", "destination": "session" }
      ]
    }
  }
}
```

Keep the matcher as narrow as possible. Matching on `.*` or leaving the matcher empty would auto-approve every permission prompt, including file writes and shell commands. See the PermissionRequest reference for the full set of decision fields.

How hooks work

Hook events fire at specific lifecycle points in Claude Code. When an event fires, all matching hooks run in parallel, and identical hook commands are automatically deduplicated. The table below shows each event and when it triggers:

Event	When it fires
`SessionStart`	When a session begins or resumes
`UserPromptSubmit`	When you submit a prompt, before Claude processes it
`PreToolUse`	Before a tool call executes. Can block it
`PermissionRequest`	When a permission dialog appears
`PostToolUse`	After a tool call succeeds
`PostToolUseFailure`	After a tool call fails
`Notification`	When Claude Code sends a notification
`SubagentStart`	When a subagent is spawned
`SubagentStop`	When a subagent finishes
`Stop`	When Claude finishes responding

Event	When it fires
`TeammateIdle`	When an agent team teammate is about to go idle
`TaskCompleted`	When a task is being marked as completed
`InstructionsLoaded`	When a CLAUDE.md or `.claude/rules/*.md` file is loaded into context. Fires at session start and when files are lazily loaded during a session
`ConfigChange`	When a configuration file changes during a session
`WorktreeCreate`	When a worktree is being created via `--worktree` or `isolation: "worktree"`. Replaces default git behavior
`WorktreeRemove`	When a worktree is being removed, either at session exit or when a subagent finishes
`PreCompact`	Before context compaction
`PostCompact`	After context compaction completes
`Elicitation`	When an MCP server requests user input during a tool call
`ElicitationResult`	After a user responds to an MCP elicitation, before the response is sent back to the server
`SessionEnd`	When a session terminates

Each hook has a `type` that determines how it runs. Most hooks use `"type": "command"`, which runs a shell command. Three other types are available:

- `"type": "http"`: POST event data to a URL. See HTTP hooks.
- `"type": "prompt"`: single-turn LLM evaluation. See Prompt-based hooks.
- `"type": "agent"`: multi-turn verification with tool access. See Agent-based hooks.

Read input and return output

Hooks communicate with Claude Code through stdin, stdout, stderr, and exit codes. When an event fires, Claude Code passes event-specific data as JSON to your script's stdin. Your script reads that data, does its work, and tells Claude Code what to do next via the exit code.

Hook input

Every event includes common fields like `session_id` and `cwd`, but each event type adds different data. For example, when Claude runs a Bash command, a `PreToolUse` hook receives something like this on stdin:

```
{
  "session_id": "abc123",          // unique ID for this session
  "cwd": "/Users/sarah/myproject", // working directory when the event fired
  "hook_event_name": "PreToolUse", // which event triggered this hook
  "tool_name": "Bash",             // the tool Claude is about to use
  "tool_input": {                  // the arguments Claude passed to the tool
    "command": "npm test"          // for Bash, this is the shell command
  }
}
```

Your script can parse that JSON and act on any of those fields. `UserPromptSubmit` hooks get the `prompt` text instead, `SessionStart` hooks get the `source` (startup, resume, clear, compact), and so on. See Common input fields in the reference for shared fields, and each event's section for event-specific schemas.

Hook output

Your script tells Claude Code what to do next by writing to stdout or stderr and exiting with a specific code. For example, a `PreToolUse` hook that wants to block a command:

```
#!/bin/bash
INPUT=$(cat)
COMMAND=$(echo "$INPUT" | jq -r '.tool_input.command')

if echo "$COMMAND" | grep -q "drop table"; then
  echo "Blocked: dropping tables is not allowed" >&2  # stderr becomes Claude's
feedback
  exit 2                                              # exit 2 = block the action
fi

exit 0  # exit 0 = let it proceed
```

The exit code determines what happens next:

- **Exit 0**: the action proceeds. For `UserPromptSubmit` and `SessionStart` hooks, anything you write to stdout is added to Claude's context.
- **Exit 2**: the action is blocked. Write a reason to stderr, and Claude receives it as feedback so it can adjust.
- **Any other exit code**: the action proceeds. Stderr is logged but not shown to Claude. Toggle verbose mode with `Ctrl+O` to see these messages in the transcript.

Structured JSON output

Exit codes give you two options: allow or block. For more control, exit 0 and print a JSON object to stdout instead.

> **Note:**
>
> Use exit 2 to block with a stderr message, or exit 0 with JSON for structured control. Don't mix them: Claude Code ignores JSON when you exit 2.

For example, a `PreToolUse` hook can deny a tool call and tell Claude why, or escalate it to the user for approval:

```
{
  "hookSpecificOutput": {
    "hookEventName": "PreToolUse",
    "permissionDecision": "deny",
    "permissionDecisionReason": "Use rg instead of grep for better performance"
  }
}
```

Claude Code reads `permissionDecision` and cancels the tool call, then feeds `permissionDecisionReason` back to Claude as feedback. These three options are specific to `PreToolUse`:

- `"allow"`: proceed without showing a permission prompt
- `"deny"`: cancel the tool call and send the reason to Claude
- `"ask"`: show the permission prompt to the user as normal

Other events use different decision patterns. For example, `PostToolUse` and `Stop` hooks use a top-level `decision: "block"` field, while `PermissionRequest` uses `hookSpecificOutput.decision.behavior`. See the summary table in the reference for a full breakdown by event.

For `UserPromptSubmit` hooks, use `additionalContext` instead to inject text into Claude's context. Prompt-based hooks (`type: "prompt"`) handle output differently: see Prompt-based hooks.

Filter hooks with matchers

Without a matcher, a hook fires on every occurrence of its event. Matchers let you narrow that down. For example, if you want to run a formatter only after file edits (not after every tool call), add a matcher to your `PostToolUse` hook:

```
{
  "hooks": {
    "PostToolUse": [
      {
        "matcher": "Edit|Write",
        "hooks": [
          { "type": "command", "command": "prettier --write ..." }
        ]
      }
    ]
  }
}
```

The `"Edit|Write"` matcher is a regex pattern that matches the tool name. The hook only fires when Claude uses the `Edit` or `Write` tool, not when it uses `Bash` , `Read` , or any other tool.

Each event type matches on a specific field. Matchers support exact strings and regex patterns:

Event	What the matcher filters	Example matcher values
`PreToolUse`, `PostToolUse`, `PostToolUseFailure`, `PermissionRequest`	tool name	`Bash`, `Edit|Write`, `mcp__.*`
`SessionStart`	how the session started	`startup`, `resume`, `clear`, `compact`
`SessionEnd`	why the session ended	`clear`, `logout`, `prompt_input_exit`, `bypass_permissions_disabled`, `other`
`Notification`	notification type	`permission_prompt`, `idle_prompt`, `auth_success`, `elicitation_dialog`
`SubagentStart`	agent type	`Bash`, `Explore`, `Plan`, or custom agent names
`PreCompact`	what triggered compaction	`manual`, `auto`
`SubagentStop`	agent type	same values as `SubagentStart`
`ConfigChange`	configuration source	`user_settings`, `project_settings`, `local_settings`, `policy_settings`, `skills`
`UserPromptSubmit`, `Stop`, `TeammateIdle`, `TaskCompleted`, `WorktreeCreate`, `WorktreeRemove`	no matcher support	always fires on every occurrence

A few more examples showing matchers on different event types:

Log every Bash command

Match only `Bash` tool calls and log each command to a file. The `PostToolUse` event fires after the command completes, so `tool_input.command` contains what ran. The hook receives the event data as JSON on stdin, and `jq -r '.tool_input.command'` extracts just the command string, which `>>` appends to the log file:

```
{
  "hooks": {
    "PostToolUse": [
      {
        "matcher": "Bash",
        "hooks": [
          {
            "type": "command",
            "command": "jq -r '.tool_input.command' >> ~/.claude/command-log.txt"
          }
        ]
      }
    ]
  }
}
```

Match MCP tools

MCP tools use a different naming convention than built-in tools: `mcp__<server>__<tool>`, where `<server>` is the MCP server name and `<tool>` is the tool it provides. For example, `mcp__github__search_repositories` or `mcp__filesystem__read_file`. Use a regex matcher to target all tools from a specific server, or match across servers with a pattern like `mcp__.*__write.*`. See Match MCP tools in the reference for the full list of examples.

The command below extracts the tool name from the hook's JSON input with `jq` and writes it to stderr, where it shows up in verbose mode (`Ctrl+O`):

```
{
  "hooks": {
    "PreToolUse": [
      {
        "matcher": "mcp__github__.*",
        "hooks": [
          {
            "type": "command",
            "command": "echo \"GitHub tool called: $(jq -r '.tool_name')\" >&2"
          }
        ]
      }
    ]
  }
}
```

Clean up on session end

The `SessionEnd` event supports matchers on the reason the session ended. This hook only fires on `clear` (when you run `/clear`), not on normal exits:

```
{
  "hooks": {
    "SessionEnd": [
      {
        "matcher": "clear",
        "hooks": [
          {
            "type": "command",
            "command": "rm -f /tmp/claude-scratch-*.txt"
          }
        ]
      }
    ]
  }
}
```

For full matcher syntax, see the Hooks reference.

Configure hook location

Where you add a hook determines its scope:

Location	Scope	Shareable
`~/.claude/settings.json`	All your projects	No, local to your machine
`.claude/settings.json`	Single project	Yes, can be committed to the repo
`.claude/settings.local.json`	Single project	No, gitignored
Managed policy settings	Organization-wide	Yes, admin-controlled
Plugin `hooks/hooks.json`	When plugin is enabled	Yes, bundled with the plugin
Skill or agent frontmatter	While the skill or agent is active	Yes, defined in the component file

Run `/hooks` in Claude Code to browse all configured hooks grouped by event. To disable all hooks at once, set `"disableAllHooks": true` in your settings file.

If you edit settings files directly while Claude Code is running, the file watcher normally picks up hook changes automatically.

Prompt-based hooks

For decisions that require judgment rather than deterministic rules, use `type: "prompt"` hooks. Instead of running a shell command, Claude Code sends your prompt and the hook's input data to a Claude model (Haiku by default) to make the decision. You can specify a different model with the `model` field if you need more capability.

The model's only job is to return a yes/no decision as JSON:

- `"ok": true` : the action proceeds
- `"ok": false` : the action is blocked. The model's `"reason"` is fed back to Claude so it can adjust.

This example uses a `Stop` hook to ask the model whether all requested tasks are complete. If the model returns `"ok": false` , Claude keeps working and uses the `reason` as its next instruction:

```
{
  "hooks": {
    "Stop": [
      {
        "hooks": [
          {
            "type": "prompt",
            "prompt": "Check if all tasks are complete. If not, respond with
{\"ok\": false, \"reason\": \"what remains to be done\"}."
          }
        ]
      }
    ]
  }
}
```

For full configuration options, see Prompt-based hooks in the reference.

Agent-based hooks

When verification requires inspecting files or running commands, use `type: "agent"` hooks. Unlike prompt hooks which make a single LLM call, agent hooks spawn a subagent that can read files, search code, and use other tools to verify conditions before returning a decision.

Agent hooks use the same `"ok"` / `"reason"` response format as prompt hooks, but with a longer default timeout of 60 seconds and up to 50 tool-use turns.

This example verifies that tests pass before allowing Claude to stop:

```
{
  "hooks": {
    "Stop": [
      {
        "hooks": [
          {
            "type": "agent",
            "prompt": "Verify that all unit tests pass. Run the test suite and
check the results. $ARGUMENTS",
            "timeout": 120
          }
        ]
      }
    ]
  }
}
```

Use prompt hooks when the hook input data alone is enough to make a decision. Use agent hooks when you need to verify something against the actual state of the codebase.

For full configuration options, see Agent-based hooks in the reference.

HTTP hooks

Use `type: "http"` hooks to POST event data to an HTTP endpoint instead of running a shell command. The endpoint receives the same JSON that a command hook would receive on stdin, and returns results through the HTTP response body using the same JSON format.

HTTP hooks are useful when you want a web server, cloud function, or external service to handle hook logic: for example, a shared audit service that logs tool use events across a team.

This example posts every tool use to a local logging service:

```
{
  "hooks": {
    "PostToolUse": [
      {
        "hooks": [
          {
            "type": "http",
            "url": "http://localhost:8080/hooks/tool-use",
            "headers": {
              "Authorization": "Bearer $MY_TOKEN"
            },
            "allowedEnvVars": ["MY_TOKEN"]
          }
        ]
      }
    ]
  }
}
```

The endpoint should return a JSON response body using the same output format as command hooks. To block a tool call, return a 2xx response with the appropriate `hookSpecificOutput` fields. HTTP status codes alone cannot block actions.

Header values support environment variable interpolation using `$VAR_NAME` or `${VAR_NAME}` syntax. Only variables listed in the `allowedEnvVars` array are resolved; all other `$VAR` references remain empty.

For full configuration options and response handling, see HTTP hooks in the reference.

Limitations and troubleshooting

Limitations

- Command hooks communicate through stdout, stderr, and exit codes only. They cannot trigger commands or tool calls directly. HTTP hooks communicate through the response body instead.
- Hook timeout is 10 minutes by default, configurable per hook with the `timeout` field (in seconds).
- `PostToolUse` hooks cannot undo actions since the tool has already executed.

- `PermissionRequest` hooks do not fire in non-interactive mode (`-p`). Use `PreToolUse` hooks for automated permission decisions.
- `Stop` hooks fire whenever Claude finishes responding, not only at task completion. They do not fire on user interrupts.

Hook not firing

The hook is configured but never executes.

- Run `/hooks` and confirm the hook appears under the correct event
- Check that the matcher pattern matches the tool name exactly (matchers are case-sensitive)
- Verify you're triggering the right event type (e.g., `PreToolUse` fires before tool execution, `PostToolUse` fires after)
- If using `PermissionRequest` hooks in non-interactive mode (`-p`), switch to `PreToolUse` instead

Hook error in output

You see a message like "PreToolUse hook error: ..." in the transcript.

- Your script exited with a non-zero code unexpectedly. Test it manually by piping sample JSON:

```
echo '{"tool_name":"Bash","tool_input":{"command":"ls"}}' | ./my-hook.sh
echo $?  # Check the exit code
```

- If you see "command not found", use absolute paths or `$CLAUDE_PROJECT_DIR` to reference scripts
- If you see "jq: command not found", install `jq` or use Python/Node.js for JSON parsing
- If the script isn't running at all, make it executable: `chmod +x ./my-hook.sh`

`/hooks` shows no hooks configured

You edited a settings file but the hooks don't appear in the menu.

- File edits are normally picked up automatically. If they haven't appeared after a few seconds, the file watcher may have missed the change: restart your session to force a reload.
- Verify your JSON is valid (trailing commas and comments are not allowed)

- Confirm the settings file is in the correct location: `.claude/settings.json` for project hooks, `~/.claude/settings.json` for global hooks

Stop hook runs forever

Claude keeps working in an infinite loop instead of stopping.

Your Stop hook script needs to check whether it already triggered a continuation. Parse the `stop_hook_active` field from the JSON input and exit early if it's `true`:

```
#!/bin/bash
INPUT=$(cat)
if [ "$(echo "$INPUT" | jq -r '.stop_hook_active')" = "true" ]; then
  exit 0  # Allow Claude to stop
fi
## ... rest of your hook logic
```

JSON validation failed

Claude Code shows a JSON parsing error even though your hook script outputs valid JSON.

When Claude Code runs a hook, it spawns a shell that sources your profile (`~/.zshrc` or `~/.bashrc`). If your profile contains unconditional `echo` statements, that output gets prepended to your hook's JSON:

```
Shell ready on arm64
{"decision": "block", "reason": "Not allowed"}
```

Claude Code tries to parse this as JSON and fails. To fix this, wrap echo statements in your shell profile so they only run in interactive shells:

```
## In ~/.zshrc or ~/.bashrc
if [[ $- == *i* ]]; then
  echo "Shell ready"
fi
```

The `$-` variable contains shell flags, and `i` means interactive. Hooks run in non-interactive shells, so the echo is skipped.

Debug techniques

Toggle verbose mode with `Ctrl+O` to see hook output in the transcript, or run `claude --debug` for full execution details including which hooks matched and their exit codes.

Learn more

- Hooks reference: full event schemas, JSON output format, async hooks, and MCP tool hooks
- Security considerations: review before deploying hooks in shared or production environments
- Bash command validator example: complete reference implementation

Connect Claude Code to tools via MCP

> *Learn how to connect Claude Code to your tools with the Model Context Protocol.*

Claude Code can connect to hundreds of external tools and data sources through the Model Context Protocol (MCP), an open source standard for AI-tool integrations. MCP servers give Claude Code access to your tools, databases, and APIs.

What you can do with MCP

With MCP servers connected, you can ask Claude Code to:

- **Implement features from issue trackers**: "Add the feature described in JIRA issue ENG-4521 and create a PR on GitHub."
- **Analyze monitoring data**: "Check Sentry and Statsig to check the usage of the feature described in ENG-4521."
- **Query databases**: "Find emails of 10 random users who used feature ENG-4521, based on our PostgreSQL database."
- **Integrate designs**: "Update our standard email template based on the new Figma designs that were posted in Slack"
- **Automate workflows**: "Create Gmail drafts inviting these 10 users to a feedback session about the new feature."

Popular MCP servers

Here are some commonly used MCP servers you can connect to Claude Code:

> **Warning:**
>
> Use third party MCP servers at your own risk - Anthropic has not verified the correctness or security of all these servers. Make sure you trust MCP servers you are installing. Be especially careful when using MCP servers that could fetch untrusted content, as these can expose you to prompt injection risk.

Server	Description	Command
Notion	Connect your Notion workspace to search, update, and power workflows across tools	`claude mcp add --transport http notion https://mcp.notion.com/mcp`
Canva	Search, create, autofill, and export Canva designs	`claude mcp add --transport http canva https://mcp.canva.com/mcp`
Figma	Generate diagrams and better code from Figma context	`claude mcp add --transport http figma-remote-mcp https://mcp.figma.com/mcp`
Atlassian	Access Jira & Confluence from Claude	`claude mcp add --transport http atlassian https://mcp.atlassian.com/v1/mcp`
Linear	Manage issues, projects & team workflows in Linear	`claude mcp add --transport http linear https://mcp.linear.app/mcp`
monday.com	Manage projects, boards, and workflows in monday.com	`claude mcp add --transport http monday https://mcp.monday.com/mcp`
Intercom	Access to Intercom data for better customer insights	`claude mcp add --transport http intercom https://mcp.intercom.com/mcp`
Vercel	Analyze, debug, and manage projects and deployments	`claude mcp add --transport http vercel https://mcp.vercel.com`

Server	Description	Command
Granola	The AI notepad for meetings	`claude mcp add --transport http granola https://mcp.granola.ai/mcp`
Asana	Connect to Asana to coordinate tasks, projects, and goals	`claude mcp add --transport streamable-http asana https://mcp.asana.com/v2/mcp`
Miro	Access and create new content on Miro boards	`claude mcp add --transport http miro https://mcp.miro.com/`
Sentry	Search, query, and debug errors intelligently	`claude mcp add --transport http sentry https://mcp.sentry.dev/mcp`
Supabase	Manage databases, authentication, and storage	`claude mcp add --transport http supabase https://mcp.supabase.com/mcp`
Hugging Face	Access the Hugging Face Hub and thousands of Gradio Apps	`claude mcp add --transport http hugging-face https://huggingface.co/mcp`
Context7	Up-to-date docs for LLMs and AI code editors	`claude mcp add --transport http context7 https://mcp.context7.com/mcp`
Stripe	Payment processing and financial infrastructure tools	`claude mcp add --transport http stripe https://mcp.stripe.com`

Server	Description	Command
Microsoft Learn	Search trusted Microsoft docs to power your development	`claude mcp add --transport http microsoft-learn https://learn.microsoft.com/api/mcp`
Clay	Find prospects. Research accounts. Personalize outreach	`claude mcp add --transport http clay https://api.clay.com/v3/mcp`
Webflow	Manage Webflow CMS, pages, assets and sites	`claude mcp add --transport http webflow https://mcp.webflow.com/mcp`
Cloudflare	Build applications with compute, storage, and AI	`claude mcp add --transport http cloudflare https://bindings.mcp.cloudflare.com/mcp`
Ramp	Search, access, and analyze your Ramp financial data	`claude mcp add --transport http ramp https://ramp-mcp-remote.ramp.com/mcp`
ZoomInfo	Enrich contacts & accounts with GTM intelligence	`claude mcp add --transport http zoominfo https://mcp.zoominfo.com/mcp`
Netlify	Create, deploy, manage, and secure websites on Netlify	`claude mcp add --transport http netlify https://netlify-mcp.netlify.app/mcp`
Make	Run Make scenarios and manage your Make account	`claude mcp add --transport http make https://mcp.make.com`

Server	Description	Command
GoDaddy	Search domains and check availability	`claude mcp add --transport http godaddy https://api.godaddy.com/v1/domains/mcp`
Google Cloud BigQuery	BigQuery: Advanced analytical insights for agents	`claude mcp add --transport http bigquery https://bigquery.googleapis.com/mcp`
PayPal	Access PayPal payments platform	`claude mcp add --transport http paypal https://mcp.paypal.com/mcp`
PostHog	Query, analyze, and manage your PostHog insights	`claude mcp add --transport http posthog https://mcp.posthog.com/mcp`
Similarweb	Real time web, mobile app, and market data	`claude mcp add --transport http similarweb https://mcp.similarweb.com`
Crypto.com	Real time prices, orders, charts, and more for crypto	`claude mcp add --transport http crypto.com https://mcp.crypto.com/market-data/mcp`
Attio	Search, manage, and update your Attio CRM from Claude	`claude mcp add --transport http attio https://mcp.attio.com/mcp`
Trivago	Find your ideal hotel at the best price	`claude mcp add --transport http trivago https://mcp.trivago.com/mcp`

Server	Description	Command
Jam	Record screen and collect automatic context for issues	`claude mcp add --transport http jam https://mcp.jam.dev/mcp`
Consensus	Explore scientific research	`claude mcp add --transport http consensus https://mcp.consensus.app/mcp`
Clockwise	Advanced scheduling and time management for work	`claude mcp add --transport http clockwise https://mcp.getclockwise.com/mcp`
Square	Search and manage transaction, merchant, and payment data	`claude mcp add --transport sse square https://mcp.squareup.com/sse`
Egnyte	Securely access and analyze Egnyte content	`claude mcp add --transport http egnyte https://mcp-server.egnyte.com/mcp`
Pylon	Search and manage Pylon support issues	`claude mcp add --transport http pylon https://mcp.usepylon.com/`
Honeycomb	Query and explore observability data and SLOs	`claude mcp add --transport http honeycomb https://mcp.honeycomb.io/mcp`

Note:

Need a specific integration? Find hundreds more MCP servers on GitHub, or build your own using the MCP SDK.

Installing MCP servers

MCP servers can be configured in three different ways depending on your needs:

Option 1: Add a remote HTTP server

HTTP servers are the recommended option for connecting to remote MCP servers. This is the most widely supported transport for cloud-based services.

```
## Basic syntax
claude mcp add --transport http <name> <url>

## Real example: Connect to Notion
claude mcp add --transport http notion https://mcp.notion.com/mcp

## Example with Bearer token
claude mcp add --transport http secure-api https://api.example.com/mcp \
  --header "Authorization: Bearer your-token"
```

Option 2: Add a remote SSE server

> **Warning:**
>
> The SSE (Server-Sent Events) transport is deprecated. Use HTTP servers instead, where available.

```
## Basic syntax
claude mcp add --transport sse <name> <url>

## Real example: Connect to Asana
claude mcp add --transport sse asana https://mcp.asana.com/sse

## Example with authentication header
claude mcp add --transport sse private-api https://api.company.com/sse \
  --header "X-API-Key: your-key-here"
```

Option 3: Add a local stdio server

Stdio servers run as local processes on your machine. They're ideal for tools that need direct system access or custom scripts.

```
## Basic syntax
claude mcp add [options] <name> -- <command> [args...]

## Real example: Add Airtable server
claude mcp add --transport stdio --env AIRTABLE_API_KEY=YOUR_KEY airtable \
  -- npx -y airtable-mcp-server
```

Note:

Important: Option ordering

All options (`--transport` , `--env` , `--scope` , `--header`) must come **before** the server name. The `--` (double dash) then separates the server name from the command and arguments that get passed to the MCP server.

For example:

- `claude mcp add --transport stdio myserver -- npx server` → runs `npx server`
- `claude mcp add --transport stdio --env KEY=value myserver -- python server.py --port 8080` → runs `python server.py --port 8080` with `KEY=value` in environment

This prevents conflicts between Claude's flags and the server's flags.

Managing your servers

Once configured, you can manage your MCP servers with these commands:

```
## List all configured servers
claude mcp list

## Get details for a specific server
claude mcp get github

## Remove a server
claude mcp remove github

## (within Claude Code) Check server status
/mcp
```

Dynamic tool updates

Claude Code supports MCP `list_changed` notifications, allowing MCP servers to dynamically update their available tools, prompts, and resources without requiring you to disconnect and reconnect. When an MCP server sends a `list_changed` notification, Claude Code automatically refreshes the available capabilities from that server.

Tip:

Tips:

- Use the `--scope` flag to specify where the configuration is stored:
- `local` (default): Available only to you in the current project (was called `project` in older versions)
- `project` : Shared with everyone in the project via `.mcp.json` file
- `user` : Available to you across all projects (was called `global` in older versions)
- Set environment variables with `--env` flags (for example, `--env KEY=value`)
- Configure MCP server startup timeout using the MCP_TIMEOUT environment variable (for example, `MCP_TIMEOUT=10000 claude` sets a 10-second timeout)
- Claude Code will display a warning when MCP tool output exceeds 10,000 tokens. To increase this limit, set the `MAX_MCP_OUTPUT_TOKENS` environment variable (for example, `MAX_MCP_OUTPUT_TOKENS=50000`)
- Use `/mcp` to authenticate with remote servers that require OAuth 2.0 authentication

Warning:

Windows Users: On native Windows (not WSL), local MCP servers that use `npx` require the `cmd /c` wrapper to ensure proper execution.

```
## This creates command="cmd" which Windows can execute
claude mcp add --transport stdio my-server -- cmd /c npx -y @some/package
```

Without the `cmd /c` wrapper, you'll encounter "Connection closed" errors because Windows cannot directly execute `npx` . (See the note above for an explanation of the `--` parameter.)

Plugin-provided MCP servers

Plugins can bundle MCP servers, automatically providing tools and integrations when the plugin is enabled. Plugin MCP servers work identically to user-configured servers.

How plugin MCP servers work:

- Plugins define MCP servers in `.mcp.json` at the plugin root or inline in `plugin.json`

- When a plugin is enabled, its MCP servers start automatically
- Plugin MCP tools appear alongside manually configured MCP tools
- Plugin servers are managed through plugin installation (not `/mcp` commands)

Example plugin MCP configuration:

In `.mcp.json` at plugin root:

```
{
  "database-tools": {
    "command": "${CLAUDE_PLUGIN_ROOT}/servers/db-server",
    "args": ["--config", "${CLAUDE_PLUGIN_ROOT}/config.json"],
    "env": {
      "DB_URL": "${DB_URL}"
    }
  }
}
```

Or inline in `plugin.json`:

```
{
  "name": "my-plugin",
  "mcpServers": {
    "plugin-api": {
      "command": "${CLAUDE_PLUGIN_ROOT}/servers/api-server",
      "args": ["--port", "8080"]
    }
  }
}
```

Plugin MCP features:

- **Automatic lifecycle**: At session startup, servers for enabled plugins connect automatically. If you enable or disable a plugin during a session, run `/reload-plugins` to connect or disconnect its MCP servers
- **Environment variables**: Use `${CLAUDE_PLUGIN_ROOT}` for plugin-relative paths
- **User environment access**: Access to same environment variables as manually configured servers

- **Multiple transport types**: Support stdio, SSE, and HTTP transports (transport support may vary by server)

Viewing plugin MCP servers:

```
## Within Claude Code, see all MCP servers including plugin ones
/mcp
```

Plugin servers appear in the list with indicators showing they come from plugins.

Benefits of plugin MCP servers:

- **Bundled distribution**: Tools and servers packaged together
- **Automatic setup**: No manual MCP configuration needed
- **Team consistency**: Everyone gets the same tools when plugin is installed

See the plugin components reference for details on bundling MCP servers with plugins.

MCP installation scopes

MCP servers can be configured at three different scope levels, each serving distinct purposes for managing server accessibility and sharing. Understanding these scopes helps you determine the best way to configure servers for your specific needs.

Local scope

Local-scoped servers represent the default configuration level and are stored in `~/.claude.json` under your project's path. These servers remain private to you and are only accessible when working within the current project directory. This scope is ideal for personal development servers, experimental configurations, or servers containing sensitive credentials that shouldn't be shared.

> **Note:**
>
> The term "local scope" for MCP servers differs from general local settings. MCP local-scoped servers are stored in `~/.claude.json` (your home directory), while general local settings use `.claude/settings.local.json` (in the project directory). See Settings for details on settings file locations.

```
## Add a local-scoped server (default)
claude mcp add --transport http stripe https://mcp.stripe.com

## Explicitly specify local scope
claude mcp add --transport http stripe --scope local https://mcp.stripe.com
```

Project scope

Project-scoped servers enable team collaboration by storing configurations in a `.mcp.json` file at your project's root directory. This file is designed to be checked into version control, ensuring all team members have access to the same MCP tools and services. When you add a project-scoped server, Claude Code automatically creates or updates this file with the appropriate configuration structure.

```
## Add a project-scoped server
claude mcp add --transport http paypal --scope project https://mcp.paypal.com/mcp
```

The resulting `.mcp.json` file follows a standardized format:

```
{
  "mcpServers": {
    "shared-server": {
      "command": "/path/to/server",
      "args": [],
      "env": {}
    }
  }
}
```

For security reasons, Claude Code prompts for approval before using project-scoped servers from `.mcp.json` files. If you need to reset these approval choices, use the `claude mcp reset-project-choices` command.

User scope

User-scoped servers are stored in `~/.claude.json` and provide cross-project accessibility, making them available across all projects on your machine while remaining private to your user account. This scope works well for personal utility servers, development tools, or services you frequently use across different projects.

```
## Add a user server
claude mcp add --transport http hubspot --scope user https://mcp.hubspot.com/
anthropic
```

Choosing the right scope

Select your scope based on:

- **Local scope**: Personal servers, experimental configurations, or sensitive credentials specific to one project
- **Project scope**: Team-shared servers, project-specific tools, or services required for collaboration
- **User scope**: Personal utilities needed across multiple projects, development tools, or frequently used services

> **Note:**
>
> **Where are MCP servers stored?**
>
> - **User and local scope**: `~/.claude.json` (in the `mcpServers` field or under project paths)
> - **Project scope**: `.mcp.json` in your project root (checked into source control)
> - **Managed**: `managed-mcp.json` in system directories (see Managed MCP configuration)

Scope hierarchy and precedence

MCP server configurations follow a clear precedence hierarchy. When servers with the same name exist at multiple scopes, the system resolves conflicts by prioritizing local-scoped servers first, followed by project-scoped servers, and finally user-scoped servers. This design ensures that personal configurations can override shared ones when needed.

Environment variable expansion in `.mcp.json`

Claude Code supports environment variable expansion in `.mcp.json` files, allowing teams to share configurations while maintaining flexibility for machine-specific paths and sensitive values like API keys.

Supported syntax:

- `${VAR}` - Expands to the value of environment variable `VAR`
- `${VAR:-default}` - Expands to `VAR` if set, otherwise uses `default`

Expansion locations: Environment variables can be expanded in:

- `command` - The server executable path
- `args` - Command-line arguments
- `env` - Environment variables passed to the server
- `url` - For HTTP server types
- `headers` - For HTTP server authentication

Example with variable expansion:

```
{
  "mcpServers": {
    "api-server": {
      "type": "http",
      "url": "${API_BASE_URL:-https://api.example.com}/mcp",
      "headers": {
        "Authorization": "Bearer ${API_KEY}"
      }
    }
  }
}
```

If a required environment variable is not set and has no default value, Claude Code will fail to parse the config.

Practical examples

{/* ### Example: Automate browser testing with Playwright

```
claude mcp add --transport stdio playwright -- npx -y @playwright/mcp@latest
```

Then write and run browser tests:

```
Test if the login flow works with test@example.com
```

```
Take a screenshot of the checkout page on mobile
```

```
  Verify that the search feature returns results
  ``` */}

Example: Monitor errors with Sentry

```bash  theme={null}
claude mcp add --transport http sentry https://mcp.sentry.dev/mcp
```

Authenticate with your Sentry account:

```
/mcp
```

Then debug production issues:

```
What are the most common errors in the last 24 hours?
```

```
Show me the stack trace for error ID abc123
```

```
Which deployment introduced these new errors?
```

Example: Connect to GitHub for code reviews

```
claude mcp add --transport http github https://api.githubcopilot.com/mcp/
```

Authenticate if needed by selecting “Authenticate” for GitHub:

```
/mcp
```

Then work with GitHub:

```
Review PR #456 and suggest improvements
```

```
Create a new issue for the bug we just found
```

```
Show me all open PRs assigned to me
```

Example: Query your PostgreSQL database

```
claude mcp add --transport stdio db -- npx -y @bytebase/dbhub \
  --dsn "postgresql://readonly:pass@prod.db.com:5432/analytics"
```

Then query your database naturally:

```
What's our total revenue this month?
```

```
Show me the schema for the orders table
```

```
Find customers who haven't made a purchase in 90 days
```

Authenticate with remote MCP servers

Many cloud-based MCP servers require authentication. Claude Code supports OAuth 2.0 for secure connections.

Step 1: Add the server that requires authentication

For example:

```
claude mcp add --transport http sentry https://mcp.sentry.dev/mcp
```

Step 2: Use the /mcp command within Claude Code

In Claude code, use the command:

```
/mcp
```

Then follow the steps in your browser to login.

> **Tip:**
>
> Tips:
>
> - Authentication tokens are stored securely and refreshed automatically
> - Use "Clear authentication" in the `/mcp` menu to revoke access
> - If your browser doesn't open automatically, copy the provided URL and open it manually
> - If the browser redirect fails with a connection error after authenticating, paste the full callback URL from your browser's address bar into the URL prompt that appears in Claude Code
> - OAuth authentication works with HTTP servers

Use a fixed OAuth callback port

Some MCP servers require a specific redirect URI registered in advance. By default, Claude Code picks a random available port for the OAuth callback. Use `--callback-port` to fix the port so it matches a pre-registered redirect URI of the form `http://localhost:PORT/callback`.

You can use `--callback-port` on its own (with dynamic client registration) or together with `--client-id` (with pre-configured credentials).

```
## Fixed callback port with dynamic client registration
claude mcp add --transport http \
  --callback-port 8080 \
  my-server https://mcp.example.com/mcp
```

Use pre-configured OAuth credentials

Some MCP servers don't support automatic OAuth setup. If you see an error like "Incompatible auth server: does not support dynamic client registration," the server requires pre-configured credentials. Register an OAuth app through the server's developer portal first, then provide the credentials when adding the server.

Step 1: Register an OAuth app with the server

Create an app through the server's developer portal and note your client ID and client secret.

Many servers also require a redirect URI. If so, choose a port and register a redirect URI in the format `http://localhost:PORT/callback`. Use that same port with `--callback-port` in the next step.

Step 2: Add the server with your credentials

Choose one of the following methods. The port used for `--callback-port` can be any available port. It just needs to match the redirect URI you registered in the previous step.

claude mcp add

Use `--client-id` to pass your app's client ID. The `--client-secret` flag prompts for the secret with masked input:

```
claude mcp add --transport http \
  --client-id your-client-id --client-secret --callback-port 8080 \
  my-server https://mcp.example.com/mcp
```

claude mcp add-json

Include the `oauth` object in the JSON config and pass `--client-secret` as a separate flag:

```
claude mcp add-json my-server \
  '{"type":"http","url":"https://mcp.example.com/mcp","oauth":{"clientId":"your-client-id","callbackPort":8080}}' \
  --client-secret
```

claude mcp add-json (callback port only)

Use `--callback-port` without a client ID to fix the port while using dynamic client registration:

```
claude mcp add-json my-server \
  '{"type":"http","url":"https://mcp.example.com/mcp","oauth":
{"callbackPort":8080}}'
```

CI / env var

Set the secret via environment variable to skip the interactive prompt:

```
MCP_CLIENT_SECRET=your-secret claude mcp add --transport http \
  --client-id your-client-id --client-secret --callback-port 8080 \
  my-server https://mcp.example.com/mcp
```

Step 3: Authenticate in Claude Code

Run `/mcp` in Claude Code and follow the browser login flow.

> **Tip:**
>
> Tips:
>
> - The client secret is stored securely in your system keychain (macOS) or a credentials file, not in your config
> - If the server uses a public OAuth client with no secret, use only `--client-id` without `--client-secret`
> - `--callback-port` can be used with or without `--client-id`
> - These flags only apply to HTTP and SSE transports. They have no effect on stdio servers
> - Use `claude mcp get <name>` to verify that OAuth credentials are configured for a server

Override OAuth metadata discovery

If your MCP server returns errors on the standard OAuth metadata endpoint (`/.well-known/oauth-authorization-server`) but exposes a working OIDC endpoint, you can tell Claude Code to fetch OAuth metadata directly from a URL you specify, bypassing the standard discovery chain.

Set `authServerMetadataUrl` in the `oauth` object of your server's config in `.mcp.json` :

```
{
  "mcpServers": {
    "my-server": {
      "type": "http",
      "url": "https://mcp.example.com/mcp",
      "oauth": {
        "authServerMetadataUrl": "https://auth.example.com/.well-known/openid-
configuration"
      }
    }
  }
}
```

The URL must use `https://`. This option requires Claude Code v2.1.64 or later.

Add MCP servers from JSON configuration

If you have a JSON configuration for an MCP server, you can add it directly:

Step 1: Add an MCP server from JSON

```
## Basic syntax
claude mcp add-json <name> '<json>'

## Example: Adding an HTTP server with JSON configuration
claude mcp add-json weather-api '{"type":"http","url":"https://api.weather.com/
mcp","headers":{"Authorization":"Bearer token"}}'

## Example: Adding a stdio server with JSON configuration
claude mcp add-json local-weather '{"type":"stdio","command":"/path/to/weather-
cli","args":["--api-key","abc123"],"env":{"CACHE_DIR":"/tmp"}}'

## Example: Adding an HTTP server with pre-configured OAuth credentials
claude mcp add-json my-server '{"type":"http","url":"https://mcp.example.com/
mcp","oauth":{"clientId":"your-client-id","callbackPort":8080}}' --client-secret
```

Step 2: Verify the server was added

```
claude mcp get weather-api
```

Tip:

Tips:

- Make sure the JSON is properly escaped in your shell
- The JSON must conform to the MCP server configuration schema
- You can use `--scope user` to add the server to your user configuration instead of the project-specific one

Import MCP servers from Claude Desktop

If you've already configured MCP servers in Claude Desktop, you can import them:

Step 1: Import servers from Claude Desktop

```
## Basic syntax
claude mcp add-from-claude-desktop
```

Step 2: Select which servers to import

After running the command, you'll see an interactive dialog that allows you to select which servers you want to import.

Step 3: Verify the servers were imported

```
claude mcp list
```

Tip:

Tips:

- This feature only works on macOS and Windows Subsystem for Linux (WSL)
- It reads the Claude Desktop configuration file from its standard location on those platforms
- Use the `--scope user` flag to add servers to your user configuration
- Imported servers will have the same names as in Claude Desktop
- If servers with the same names already exist, they will get a numerical suffix (for example, `server_1`)

Use MCP servers from Claude.ai

If you've logged into Claude Code with a Claude.ai account, MCP servers you've added in Claude.ai are automatically available in Claude Code:

Step 1: Configure MCP servers in Claude.ai

Add servers at claude.ai/settings/connectors. On Team and Enterprise plans, only admins can add servers.

Step 2: Authenticate the MCP server

Complete any required authentication steps in Claude.ai.

Step 3: View and manage servers in Claude Code

In Claude Code, use the command:

```
/mcp
```

Claude.ai servers appear in the list with indicators showing they come from Claude.ai.

To disable claude.ai MCP servers in Claude Code, set the `ENABLE_CLAUDEAI_MCP_SERVERS` environment variable to `false`:

```
ENABLE_CLAUDEAI_MCP_SERVERS=false claude
```

Use Claude Code as an MCP server

You can use Claude Code itself as an MCP server that other applications can connect to:

```
## Start Claude as a stdio MCP server
claude mcp serve
```

You can use this in Claude Desktop by adding this configuration to claude_desktop_config.json:

```
{
  "mcpServers": {
    "claude-code": {
      "type": "stdio",
      "command": "claude",
      "args": ["mcp", "serve"],
      "env": {}
    }
  }
}
```

Warning:

Configuring the executable path: The `command` field must reference the Claude Code executable. If the `claude` command is not in your system's PATH, you'll need to specify the full path to the executable.

To find the full path:

```
which claude
```

Then use the full path in your configuration:

```
{
  "mcpServers": {
    "claude-code": {
      "type": "stdio",
      "command": "/full/path/to/claude",
      "args": ["mcp", "serve"],
      "env": {}
    }
  }
}
```

Without the correct executable path, you'll encounter errors like `spawn claude ENOENT`.

Tip:

Tips:

- The server provides access to Claude's tools like View, Edit, LS, etc.

- In Claude Desktop, try asking Claude to read files in a directory, make edits, and more.
- Note that this MCP server is only exposing Claude Code's tools to your MCP client, so your own client is responsible for implementing user confirmation for individual tool calls.

MCP output limits and warnings

When MCP tools produce large outputs, Claude Code helps manage the token usage to prevent overwhelming your conversation context:

- **Output warning threshold**: Claude Code displays a warning when any MCP tool output exceeds 10,000 tokens
- **Configurable limit**: You can adjust the maximum allowed MCP output tokens using the `MAX_MCP_OUTPUT_TOKENS` environment variable
- **Default limit**: The default maximum is 25,000 tokens

To increase the limit for tools that produce large outputs:

```
## Set a higher limit for MCP tool outputs
export MAX_MCP_OUTPUT_TOKENS=50000
claude
```

This is particularly useful when working with MCP servers that:

- Query large datasets or databases
- Generate detailed reports or documentation
- Process extensive log files or debugging information

> **Warning:**
>
> If you frequently encounter output warnings with specific MCP servers, consider increasing the limit or configuring the server to paginate or filter its responses.

Respond to MCP elicitation requests

MCP servers can request structured input from you mid-task using elicitation. When a server needs information it can't get on its own, Claude Code displays an interactive dialog and passes your response back to the server. No configuration is required on your side: elicitation dialogs appear automatically when a server requests them.

Servers can request input in two ways:

- **Form mode**: Claude Code shows a dialog with form fields defined by the server (for example, a username and password prompt). Fill in the fields and submit.
- **URL mode**: Claude Code opens a browser URL for authentication or approval. Complete the flow in the browser, then confirm in the CLI.

To auto-respond to elicitation requests without showing a dialog, use the `Elicitation` hook.

If you're building an MCP server that uses elicitation, see the MCP elicitation specification for protocol details and schema examples.

Use MCP resources

MCP servers can expose resources that you can reference using @ mentions, similar to how you reference files.

Reference MCP resources

Step 1: List available resources

Type `@` in your prompt to see available resources from all connected MCP servers. Resources appear alongside files in the autocomplete menu.

Step 2: Reference a specific resource

Use the format `@server:protocol://resource/path` to reference a resource:

```
Can you analyze @github:issue://123 and suggest a fix?
```

```
Please review the API documentation at @docs:file://api/authentication
```

Step 3: Multiple resource references

You can reference multiple resources in a single prompt:

```
Compare @postgres:schema://users with @docs:file://database/user-model
```

> **Tip:**

Tips:

- Resources are automatically fetched and included as attachments when referenced
- Resource paths are fuzzy-searchable in the @ mention autocomplete
- Claude Code automatically provides tools to list and read MCP resources when servers support them
- Resources can contain any type of content that the MCP server provides (text, JSON, structured data, etc.)

Scale with MCP Tool Search

When you have many MCP servers configured, tool definitions can consume a significant portion of your context window. MCP Tool Search solves this by dynamically loading tools on-demand instead of preloading all of them.

How it works

Claude Code automatically enables Tool Search when your MCP tool descriptions would consume more than 10% of the context window. You can adjust this threshold or disable tool search entirely. When triggered:

1. MCP tools are deferred rather than loaded into context upfront
2. Claude uses a search tool to discover relevant MCP tools when needed
3. Only the tools Claude actually needs are loaded into context
4. MCP tools continue to work exactly as before from your perspective

For MCP server authors

If you're building an MCP server, the server instructions field becomes more useful with Tool Search enabled. Server instructions help Claude understand when to search for your tools, similar to how skills work.

Add clear, descriptive server instructions that explain:

- What category of tasks your tools handle
- When Claude should search for your tools
- Key capabilities your server provides

Configure tool search

Tool search is enabled by default: MCP tools are deferred and discovered on demand. When `ANTHROPIC_BASE_URL` points to a non-first-party host, tool search is disabled by default because most proxies do not forward `tool_reference` blocks. Set

`ENABLE_TOOL_SEARCH` explicitly if your proxy does. This feature requires models that support `tool_reference` blocks: Sonnet 4 and later, or Opus 4 and later. Haiku models do not support tool search.

Control tool search behavior with the `ENABLE_TOOL_SEARCH` environment variable:

Value	Behavior
(unset)	Enabled by default. Disabled when `ANTHROPIC_BASE_URL` is a non-first-party host
`true`	Always enabled, including for non-first-party `ANTHROPIC_BASE_URL`
`auto`	Activates when MCP tools exceed 10% of context
`auto:<N>`	Activates at custom threshold, where `<N>` is a percentage (e.g., `auto:5` for 5%)
`false`	Disabled, all MCP tools loaded upfront

```
## Use a custom 5% threshold
ENABLE_TOOL_SEARCH=auto:5 claude

## Disable tool search entirely
ENABLE_TOOL_SEARCH=false claude
```

Or set the value in your settings.json `env` field.

You can also disable the MCPSearch tool specifically using the `disallowedTools` setting:

```
{
  "permissions": {
    "deny": ["MCPSearch"]
  }
}
```

Use MCP prompts as commands

MCP servers can expose prompts that become available as commands in Claude Code.

Execute MCP prompts

Step 1: Discover available prompts

Type `/` to see all available commands, including those from MCP servers. MCP prompts appear with the format `/mcp__servername__promptname`.

Step 2: Execute a prompt without arguments

```
/mcp__github__list_prs
```

Step 3: Execute a prompt with arguments

Many prompts accept arguments. Pass them space-separated after the command:

```
/mcp__github__pr_review 456
```

```
/mcp__jira__create_issue "Bug in login flow" high
```

> **Tip:**
>
> Tips:
>
> - MCP prompts are dynamically discovered from connected servers
> - Arguments are parsed based on the prompt's defined parameters
> - Prompt results are injected directly into the conversation
> - Server and prompt names are normalized (spaces become underscores)

Managed MCP configuration

For organizations that need centralized control over MCP servers, Claude Code supports two configuration options:

1. **Exclusive control with `managed-mcp.json`**: Deploy a fixed set of MCP servers that users cannot modify or extend
2. **Policy-based control with allowlists/denylists**: Allow users to add their own servers, but restrict which ones are permitted

These options allow IT administrators to:

- **Control which MCP servers employees can access**: Deploy a standardized set of approved MCP servers across the organization
- **Prevent unauthorized MCP servers**: Restrict users from adding unapproved MCP servers

- **Disable MCP entirely**: Remove MCP functionality completely if needed

Option 1: Exclusive control with managed-mcp.json

When you deploy a `managed-mcp.json` file, it takes **exclusive control** over all MCP servers. Users cannot add, modify, or use any MCP servers other than those defined in this file. This is the simplest approach for organizations that want complete control.

System administrators deploy the configuration file to a system-wide directory:

- macOS: `/Library/Application Support/ClaudeCode/managed-mcp.json`
- Linux and WSL: `/etc/claude-code/managed-mcp.json`
- Windows: `C:\Program Files\ClaudeCode\managed-mcp.json`

> **Note:**
>
> These are system-wide paths (not user home directories like `~/Library/...`) that require administrator privileges. They are designed to be deployed by IT administrators.

The `managed-mcp.json` file uses the same format as a standard `.mcp.json` file:

```
{
  "mcpServers": {
    "github": {
      "type": "http",
      "url": "https://api.githubcopilot.com/mcp/"
    },
    "sentry": {
      "type": "http",
      "url": "https://mcp.sentry.dev/mcp"
    },
    "company-internal": {
      "type": "stdio",
      "command": "/usr/local/bin/company-mcp-server",
      "args": ["--config", "/etc/company/mcp-config.json"],
      "env": {
        "COMPANY_API_URL": "https://internal.company.com"
      }
    }
  }
}
```

Option 2: Policy-based control with allowlists and denylists

Instead of taking exclusive control, administrators can allow users to configure their own MCP servers while enforcing restrictions on which servers are permitted. This approach uses `allowedMcpServers` and `deniedMcpServers` in the managed settings file.

> **Note:**
>
> **Choosing between options**: Use Option 1 (`managed-mcp.json`) when you want to deploy a fixed set of servers with no user customization. Use Option 2 (allowlists/denylists) when you want to allow users to add their own servers within policy constraints.

Restriction options

Each entry in the allowlist or denylist can restrict servers in three ways:

1. **By server name** (`serverName`): Matches the configured name of the server
2. **By command** (`serverCommand`): Matches the exact command and arguments used to start stdio servers

3. **By URL pattern** (`serverUrl`): Matches remote server URLs with wildcard support

Important: Each entry must have exactly one of `serverName`, `serverCommand`, or `serverUrl`.

Example configuration

```
{
  "allowedMcpServers": [
    // Allow by server name
    { "serverName": "github" },
    { "serverName": "sentry" },

    // Allow by exact command (for stdio servers)
    { "serverCommand": ["npx", "-y", "@modelcontextprotocol/server-filesystem"] },
    { "serverCommand": ["python", "/usr/local/bin/approved-server.py"] },

    // Allow by URL pattern (for remote servers)
    { "serverUrl": "https://mcp.company.com/*" },
    { "serverUrl": "https://*.internal.corp/*" }
  ],
  "deniedMcpServers": [
    // Block by server name
    { "serverName": "dangerous-server" },

    // Block by exact command (for stdio servers)
    { "serverCommand": ["npx", "-y", "unapproved-package"] },

    // Block by URL pattern (for remote servers)
    { "serverUrl": "https://*.untrusted.com/*" }
  ]
}
```

How command-based restrictions work

Exact matching:

- Command arrays must match **exactly** - both the command and all arguments in the correct order

- Example: `["npx", "-y", "server"]` will NOT match `["npx", "server"]` or `["npx", "-y", "server", "--flag"]`

Stdio server behavior:

- When the allowlist contains **any** `serverCommand` entries, stdio servers **must** match one of those commands
- Stdio servers cannot pass by name alone when command restrictions are present
- This ensures administrators can enforce which commands are allowed to run

Non-stdio server behavior:

- Remote servers (HTTP, SSE, WebSocket) use URL-based matching when `serverUrl` entries exist in the allowlist
- If no URL entries exist, remote servers fall back to name-based matching
- Command restrictions do not apply to remote servers

How URL-based restrictions work

URL patterns support wildcards using `*` to match any sequence of characters. This is useful for allowing entire domains or subdomains.

Wildcard examples:

- `https://mcp.company.com/*` - Allow all paths on a specific domain
- `https://*.example.com/*` - Allow any subdomain of example.com
- `http://localhost:*/*` - Allow any port on localhost

Remote server behavior:

- When the allowlist contains **any** `serverUrl` entries, remote servers **must** match one of those URL patterns
- Remote servers cannot pass by name alone when URL restrictions are present
- This ensures administrators can enforce which remote endpoints are allowed

```
{
  "allowedMcpServers": [
    { "serverUrl": "https://mcp.company.com/*" },
    { "serverUrl": "https://*.internal.corp/*" }
  ]
}
```

Result:

- HTTP server at `https://mcp.company.com/api`: ✅ Allowed (matches URL pattern)
- HTTP server at `https://api.internal.corp/mcp`: ✅ Allowed (matches wildcard subdomain)
- HTTP server at `https://external.com/mcp`: ❌ Blocked (doesn't match any URL pattern)
- Stdio server with any command: ❌ Blocked (no name or command entries to match)

```
{
  "allowedMcpServers": [
    { "serverCommand": ["npx", "-y", "approved-package"] }
  ]
}
```

Result:

- Stdio server with `["npx", "-y", "approved-package"]`: ✅ Allowed (matches command)
- Stdio server with `["node", "server.js"]`: ❌ Blocked (doesn't match command)
- HTTP server named "my-api": ❌ Blocked (no name entries to match)

```
{
  "allowedMcpServers": [
    { "serverName": "github" },
    { "serverCommand": ["npx", "-y", "approved-package"] }
  ]
}
```

Result:

- Stdio server named "local-tool" with `["npx", "-y", "approved-package"]`: ✅ Allowed (matches command)
- Stdio server named "local-tool" with `["node", "server.js"]`: ❌ Blocked (command entries exist but doesn't match)
- Stdio server named "github" with `["node", "server.js"]`: ❌ Blocked (stdio servers must match commands when command entries exist)
- HTTP server named "github": ✅ Allowed (matches name)

- HTTP server named “other-api”: ❌ Blocked (name doesn’t match)

```
{
  "allowedMcpServers": [
    { "serverName": "github" },
    { "serverName": "internal-tool" }
  ]
}
```

Result:

- Stdio server named “github” with any command: ✅ Allowed (no command restrictions)
- Stdio server named “internal-tool” with any command: ✅ Allowed (no command restrictions)
- HTTP server named “github”: ✅ Allowed (matches name)
- Any server named “other”: ❌ Blocked (name doesn’t match)

Allowlist behavior (`allowedMcpServers`)

- `undefined` (default): No restrictions - users can configure any MCP server
- Empty array `[]` : Complete lockdown - users cannot configure any MCP servers
- List of entries: Users can only configure servers that match by name, command, or URL pattern

Denylist behavior (`deniedMcpServers`)

- `undefined` (default): No servers are blocked
- Empty array `[]` : No servers are blocked
- List of entries: Specified servers are explicitly blocked across all scopes

Important notes

- **Option 1 and Option 2 can be combined**: If `managed-mcp.json` exists, it has exclusive control and users cannot add servers. Allowlists/denylists still apply to the managed servers themselves.
- **Denylist takes absolute precedence**: If a server matches a denylist entry (by name, command, or URL), it will be blocked even if it’s on the allowlist

- Name-based, command-based, and URL-based restrictions work together: a server passes if it matches **either** a name entry, a command entry, or a URL pattern (unless blocked by denylist)

> **Note:**
>
> **When using `managed-mcp.json`**: Users cannot add MCP servers through `claude mcp add` or configuration files. The `allowedMcpServers` and `deniedMcpServers` settings still apply to filter which managed servers are actually loaded.

Extend Claude with skills

> *Create, manage, and share skills to extend Claude's capabilities in Claude Code. Includes custom commands and bundled skills.*

Skills extend what Claude can do. Create a `SKILL.md` file with instructions, and Claude adds it to its toolkit. Claude uses skills when relevant, or you can invoke one directly with `/skill-name`.

> **Note:**
>
> For built-in commands like `/help` and `/compact`, see the built-in commands reference.
>
> **Custom commands have been merged into skills.** A file at `.claude/commands/deploy.md` and a skill at `.claude/skills/deploy/SKILL.md` both create `/deploy` and work the same way. Your existing `.claude/commands/` files keep working. Skills add optional features: a directory for supporting files, frontmatter to control whether you or Claude invokes them, and the ability for Claude to load them automatically when relevant.

Claude Code skills follow the Agent Skills open standard, which works across multiple AI tools. Claude Code extends the standard with additional features like invocation control, subagent execution, and dynamic context injection.

Bundled skills

Bundled skills ship with Claude Code and are available in every session. Unlike built-in commands, which execute fixed logic directly, bundled skills are prompt-based: they give Claude a detailed playbook and let it orchestrate the work using its tools. This means bundled skills can spawn parallel agents, read files, and adapt to your codebase.

You invoke bundled skills the same way as any other skill: type `/` followed by the skill name. In the table below, `<arg>` indicates a required argument and `[arg]` indicates an optional one.

Skill	Purpose
`/batch <instruction>`	Orchestrate large-scale changes across a codebase in parallel. Researches the codebase, decomposes the work into 5 to 30 independent units, and presents a plan. Once approved, spawns one background agent per unit in an isolated git worktree. Each agent implements its unit, runs tests, and opens a pull request. Requires a git repository. Example: `/batch migrate src/ from Solid to React`
`/claude-api`	Load Claude API reference material for your project's language (Python, TypeScript, Java, Go, Ruby, C#, PHP, or cURL) and Agent SDK reference for Python and TypeScript. Covers tool use, streaming, batches, structured outputs, and common pitfalls. Also activates automatically when your code imports `anthropic`, `@anthropic-ai/sdk`, or `claude_agent_sdk`
`/debug [description]`	Troubleshoot your current Claude Code session by reading the session debug log. Optionally describe the issue to focus the analysis
`/loop [interval] <prompt>`	Run a prompt repeatedly on an interval while the session stays open. Useful for polling a deployment, babysitting a PR, or periodically re-running another skill. Example: `/loop 5m check if the deploy finished`. See Run prompts on a schedule
`/simplify [focus]`	Review your recently changed files for code reuse, quality, and efficiency issues, then fix them. Spawns three review agents in parallel, aggregates their findings, and applies fixes. Pass text to focus on specific concerns: `/simplify focus on memory efficiency`

Getting started

Create your first skill

This example creates a skill that teaches Claude to explain code using visual diagrams and analogies. Since it uses default frontmatter, Claude can load it automatically when you ask how something works, or you can invoke it directly with `/explain-code`.

Step 1: Create the skill directory

Create a directory for the skill in your personal skills folder. Personal skills are available across all your projects.

```
mkdir -p ~/.claude/skills/explain-code
```

Step 2: Write SKILL.md

Every skill needs a `SKILL.md` file with two parts: YAML frontmatter (between `---` markers) that tells Claude when to use the skill, and markdown content with instructions Claude follows when the skill is invoked. The `name` field becomes the `/slash-command`, and the `description` helps Claude decide when to load it automatically.

Create `~/.claude/skills/explain-code/SKILL.md`:

```
---
name: explain-code
description: Explains code with visual diagrams and analogies. Use when explaining
how code works, teaching about a codebase, or when the user asks "how does this
work?"
---

When explaining code, always include:

1. **Start with an analogy**: Compare the code to something from everyday life
2. **Draw a diagram**: Use ASCII art to show the flow, structure, or relationships
3. **Walk through the code**: Explain step-by-step what happens
4. **Highlight a gotcha**: What's a common mistake or misconception?

Keep explanations conversational. For complex concepts, use multiple analogies.
```

Step 3: Test the skill

You can test it two ways:

Let Claude invoke it automatically by asking something that matches the description:

```
How does this code work?
```

Or invoke it directly with the skill name:

```
/explain-code src/auth/login.ts
```

Either way, Claude should include an analogy and ASCII diagram in its explanation.

Where skills live

Where you store a skill determines who can use it:

Location	Path	Applies to
Enterprise	See managed settings	All users in your organization
Personal	`~/.claude/skills/<skill-name>/SKILL.md`	All your projects
Project	`.claude/skills/<skill-name>/SKILL.md`	This project only
Plugin	`<plugin>/skills/<skill-name>/SKILL.md`	Where plugin is enabled

When skills share the same name across levels, higher-priority locations win: enterprise > personal > project. Plugin skills use a `plugin-name:skill-name` namespace, so they cannot conflict with other levels. If you have files in `.claude/commands/`, those work the same way, but if a skill and a command share the same name, the skill takes precedence.

Automatic discovery from nested directories

When you work with files in subdirectories, Claude Code automatically discovers skills from nested `.claude/skills/` directories. For example, if you're editing a file in `packages/frontend/`, Claude Code also looks for skills in `packages/frontend/.claude/skills/`. This supports monorepo setups where packages have their own skills.

Each skill is a directory with `SKILL.md` as the entrypoint:

```
my-skill/
├── SKILL.md           # Main instructions (required)
├── template.md        # Template for Claude to fill in
├── examples/
│   └── sample.md      # Example output showing expected format
└── scripts/
    └── validate.sh    # Script Claude can execute
```

The `SKILL.md` contains the main instructions and is required. Other files are optional and let you build more powerful skills: templates for Claude to fill in, example outputs showing the expected format, scripts Claude can execute, or detailed reference documentation. Reference these files from your `SKILL.md` so Claude knows what they contain and when to load them. See Add supporting files for more details.

> **Note:**
>
> Files in `.claude/commands/` still work and support the same frontmatter. Skills are recommended since they support additional features like supporting files.

Skills from additional directories

Skills defined in `.claude/skills/` within directories added via `--add-dir` are loaded automatically and picked up by live change detection, so you can edit them during a session without restarting.

> **Note:**
>
> CLAUDE.md files from `--add-dir` directories are not loaded by default. To load them, set `CLAUDE_CODE_ADDITIONAL_DIRECTORIES_CLAUDE_MD=1`. See Load from additional directories.

Configure skills

Skills are configured through YAML frontmatter at the top of `SKILL.md` and the markdown content that follows.

Types of skill content

Skill files can contain any instructions, but thinking about how you want to invoke them helps guide what to include:

Reference content adds knowledge Claude applies to your current work. Conventions, patterns, style guides, domain knowledge. This content runs inline so Claude can use it alongside your conversation context.

```
---
name: api-conventions
description: API design patterns for this codebase
---

When writing API endpoints:
- Use RESTful naming conventions
- Return consistent error formats
- Include request validation
```

Task content gives Claude step-by-step instructions for a specific action, like deployments, commits, or code generation. These are often actions you want to invoke directly with `/skill-name` rather than letting Claude decide when to run them. Add `disable-model-invocation: true` to prevent Claude from triggering it automatically.

```
---
name: deploy
description: Deploy the application to production
context: fork
disable-model-invocation: true
---

Deploy the application:
1. Run the test suite
2. Build the application
3. Push to the deployment target
```

Your `SKILL.md` can contain anything, but thinking through how you want the skill invoked (by you, by Claude, or both) and where you want it to run (inline or in a subagent) helps guide what to include. For complex skills, you can also add supporting files to keep the main skill focused.

Frontmatter reference

Beyond the markdown content, you can configure skill behavior using YAML frontmatter fields between `---` markers at the top of your `SKILL.md` file:

```
---
name: my-skill
description: What this skill does
disable-model-invocation: true
allowed-tools: Read, Grep
---

Your skill instructions here...
```

All fields are optional. Only `description` is recommended so Claude knows when to use the skill.

Field	Required	Description
`name`	No	Display name for the skill. If omitted, uses the directory name. Lowercase letters, numbers, and hyphens only (max 64 characters).
`description`	Recommended	What the skill does and when to use it. Claude uses this to decide when to apply the skill. If omitted, uses the first paragraph of markdown content.
`argument-hint`	No	Hint shown during autocomplete to indicate expected arguments. Example: `[issue-number]` or `[filename] [format]`.
`disable-model-invocation`	No	Set to `true` to prevent Claude from automatically loading this skill. Use for workflows you want to trigger manually with `/name`. Default: `false`.
`user-invocable`	No	Set to `false` to hide from the `/` menu. Use for background knowledge users shouldn't invoke directly. Default: `true`.
`allowed-tools`	No	Tools Claude can use without asking permission when this skill is active.
`model`	No	Model to use when this skill is active.
`context`	No	Set to `fork` to run in a forked subagent context.

Field	Required	Description
`agent`	No	Which subagent type to use when `context: fork` is set.
`hooks`	No	Hooks scoped to this skill's lifecycle. See Hooks in skills and agents for configuration format.

Available string substitutions

Skills support string substitution for dynamic values in the skill content:

Variable	Description
`$ARGUMENTS`	All arguments passed when invoking the skill. If `$ARGUMENTS` is not present in the content, arguments are appended as `ARGUMENTS: <value>`.
`$ARGUMENTS[N]`	Access a specific argument by 0-based index, such as `$ARGUMENTS[0]` for the first argument.
`$N`	Shorthand for `$ARGUMENTS[N]`, such as `$0` for the first argument or `$1` for the second.
`${CLAUDE_SESSION_ID}`	The current session ID. Useful for logging, creating session-specific files, or correlating skill output with sessions.
`${CLAUDE_SKILL_DIR}`	The directory containing the skill's `SKILL.md` file. For plugin skills, this is the skill's subdirectory within the plugin, not the plugin root. Use this in bash injection commands to reference scripts or files bundled with the skill, regardless of the current working directory.

Example using substitutions:

```
---
name: session-logger
description: Log activity for this session
---

Log the following to logs/${CLAUDE_SESSION_ID}.log:

$ARGUMENTS
```

Add supporting files

Skills can include multiple files in their directory. This keeps `SKILL.md` focused on the essentials while letting Claude access detailed reference material only when needed. Large reference docs, API specifications, or example collections don't need to load into context every time the skill runs.

```
my-skill/
├── SKILL.md (required - overview and navigation)
├── reference.md (detailed API docs - loaded when needed)
├── examples.md (usage examples - loaded when needed)
└── scripts/
    └── helper.py (utility script - executed, not loaded)
```

Reference supporting files from `SKILL.md` so Claude knows what each file contains and when to load it:

```
### Additional resources

- For complete API details, see [reference.md](reference.md)
- For usage examples, see [examples.md](examples.md)
```

> **Tip:**
>
> Keep `SKILL.md` under 500 lines. Move detailed reference material to separate files.

Control who invokes a skill

By default, both you and Claude can invoke any skill. You can type `/skill-name` to invoke it directly, and Claude can load it automatically when relevant to your conversation. Two frontmatter fields let you restrict this:

- `disable-model-invocation: true` : Only you can invoke the skill. Use this for workflows with side effects or that you want to control timing, like `/commit` , `/deploy` , or `/send-slack-message` . You don't want Claude deciding to deploy because your code looks ready.
- `user-invocable: false` : Only Claude can invoke the skill. Use this for background knowledge that isn't actionable as a command. A `legacy-system-context` skill explains how an old system works. Claude should know this when relevant, but `/legacy-system-context` isn't a meaningful action for users to take.

This example creates a deploy skill that only you can trigger. The `disable-model-invocation: true` field prevents Claude from running it automatically:

```
---
name: deploy
description: Deploy the application to production
disable-model-invocation: true
---

Deploy $ARGUMENTS to production:

1. Run the test suite
2. Build the application
3. Push to the deployment target
4. Verify the deployment succeeded
```

Here's how the two fields affect invocation and context loading:

Front-matter	You can invoke	Claude can invoke	When loaded into context
(default)	Yes	Yes	Description always in context, full skill loads when invoked

Front-matter	You can invoke	Claude can invoke	When loaded into context
`disable-model-invocation: true`	Yes	No	Description not in context, full skill loads when you invoke
`user-invocable: false`	No	Yes	Description always in context, full skill loads when invoked

> **Note:**
>
> In a regular session, skill descriptions are loaded into context so Claude knows what's available, but full skill content only loads when invoked. Subagents with preloaded skills work differently: the full skill content is injected at startup.

Restrict tool access

Use the `allowed-tools` field to limit which tools Claude can use when a skill is active. This skill creates a read-only mode where Claude can explore files but not modify them:

```
---
name: safe-reader
description: Read files without making changes
allowed-tools: Read, Grep, Glob
---
```

Pass arguments to skills

Both you and Claude can pass arguments when invoking a skill. Arguments are available via the `$ARGUMENTS` placeholder.

This skill fixes a GitHub issue by number. The `$ARGUMENTS` placeholder gets replaced with whatever follows the skill name:

```
---
name: fix-issue
description: Fix a GitHub issue
disable-model-invocation: true
---

Fix GitHub issue $ARGUMENTS following our coding standards.

1. Read the issue description
2. Understand the requirements
3. Implement the fix
4. Write tests
5. Create a commit
```

When you run `/fix-issue 123`, Claude receives "Fix GitHub issue 123 following our coding standards..."

If you invoke a skill with arguments but the skill doesn't include `$ARGUMENTS`, Claude Code appends `ARGUMENTS: <your input>` to the end of the skill content so Claude still sees what you typed.

To access individual arguments by position, use `$ARGUMENTS[N]` or the shorter `$N`:

```
---
name: migrate-component
description: Migrate a component from one framework to another
---

Migrate the $ARGUMENTS[0] component from $ARGUMENTS[1] to $ARGUMENTS[2].
Preserve all existing behavior and tests.
```

Running `/migrate-component SearchBar React Vue` replaces `$ARGUMENTS[0]` with `SearchBar`, `$ARGUMENTS[1]` with `React`, and `$ARGUMENTS[2]` with `Vue`. The same skill using the `$N` shorthand:

```
---
name: migrate-component
description: Migrate a component from one framework to another
---

Migrate the $0 component from $1 to $2.
Preserve all existing behavior and tests.
```

Advanced patterns

Inject dynamic context

The `!` command`` syntax runs shell commands before the skill content is sent to Claude. The command output replaces the placeholder, so Claude receives actual data, not the command itself.

This skill summarizes a pull request by fetching live PR data with the GitHub CLI. The `!` gh pr diff`` and other commands run first, and their output gets inserted into the prompt:

```
---
name: pr-summary
description: Summarize changes in a pull request
context: fork
agent: Explore
allowed-tools: Bash(gh *)
---

### Pull request context
- PR diff: !`gh pr diff`
- PR comments: !`gh pr view --comments`
- Changed files: !`gh pr diff --name-only`

### Your task
Summarize this pull request...
```

When this skill runs:

1. Each `!` command`` executes immediately (before Claude sees anything)
2. The output replaces the placeholder in the skill content

3. Claude receives the fully-rendered prompt with actual PR data

This is preprocessing, not something Claude executes. Claude only sees the final result.

> **Tip:**
>
> To enable extended thinking in a skill, include the word "ultrathink" anywhere in your skill content.

Run skills in a subagent

Add `context: fork` to your frontmatter when you want a skill to run in isolation. The skill content becomes the prompt that drives the subagent. It won't have access to your conversation history.

> **Warning:**
>
> `context: fork` only makes sense for skills with explicit instructions. If your skill contains guidelines like "use these API conventions" without a task, the subagent receives the guidelines but no actionable prompt, and returns without meaningful output.

Skills and subagents work together in two directions:

Approach	System prompt	Task	Also loads
Skill with `context: fork`	From agent type (`Explore` , `Plan` , etc.)	SKILL.md content	CLAUDE.md
Subagent with `skills` field	Subagent's markdown body	Claude's delegation message	Preloaded skills + CLAUDE.md

With `context: fork` , you write the task in your skill and pick an agent type to execute it. For the inverse (defining a custom subagent that uses skills as reference material), see Subagents.

Example: Research skill using Explore agent

This skill runs research in a forked Explore agent. The skill content becomes the task, and the agent provides read-only tools optimized for codebase exploration:

```
---
name: deep-research
description: Research a topic thoroughly
context: fork
agent: Explore
---

Research $ARGUMENTS thoroughly:

1. Find relevant files using Glob and Grep
2. Read and analyze the code
3. Summarize findings with specific file references
```

When this skill runs:

1. A new isolated context is created
2. The subagent receives the skill content as its prompt ("Research $ARGUMENTS thoroughly...")
3. The `agent` field determines the execution environment (model, tools, and permissions)
4. Results are summarized and returned to your main conversation

The `agent` field specifies which subagent configuration to use. Options include built-in agents (`Explore` , `Plan` , `general-purpose`) or any custom subagent from `.claude/agents/` . If omitted, uses `general-purpose` .

Restrict Claude's skill access

By default, Claude can invoke any skill that doesn't have `disable-model-invocation: true` set. Skills that define `allowed-tools` grant Claude access to those tools without per-use approval when the skill is active. Your permission settings still govern baseline approval behavior for all other tools. Built-in commands like `/compact` and `/init` are not available through the Skill tool.

Three ways to control which skills Claude can invoke:

Disable all skills by denying the Skill tool in `/permissions` :

```
## Add to deny rules:
Skill
```

Allow or deny specific skills using permission rules:

```
## Allow only specific skills
Skill(commit)
Skill(review-pr *)

## Deny specific skills
Skill(deploy *)
```

Permission syntax: `Skill(name)` for exact match, `Skill(name *)` for prefix match with any arguments.

Hide individual skills by adding `disable-model-invocation: true` to their frontmatter. This removes the skill from Claude's context entirely.

> **Note:**
>
> The `user-invocable` field only controls menu visibility, not Skill tool access. Use `disable-model-invocation: true` to block programmatic invocation.

Share skills

Skills can be distributed at different scopes depending on your audience:

- **Project skills**: Commit `.claude/skills/` to version control
- **Plugins**: Create a `skills/` directory in your plugin
- **Managed**: Deploy organization-wide through managed settings

Generate visual output

Skills can bundle and run scripts in any language, giving Claude capabilities beyond what's possible in a single prompt. One powerful pattern is generating visual output: interactive HTML files that open in your browser for exploring data, debugging, or creating reports.

This example creates a codebase explorer: an interactive tree view where you can expand and collapse directories, see file sizes at a glance, and identify file types by color.

Create the Skill directory:

```
mkdir -p ~/.claude/skills/codebase-visualizer/scripts
```

Create `~/.claude/skills/codebase-visualizer/SKILL.md`. The description tells Claude when to activate this Skill, and the instructions tell Claude to run the bundled script:

```
---
name: codebase-visualizer
description: Generate an interactive collapsible tree visualization of your
codebase. Use when exploring a new repo, understanding project structure, or
identifying large files.
allowed-tools: Bash(python *)
---

## Codebase Visualizer

Generate an interactive HTML tree view that shows your project's file structure
with collapsible directories.

### Usage

Run the visualization script from your project root:

```bash
python ~/.claude/skills/codebase-visualizer/scripts/visualize.py .
```text

This creates `codebase-map.html` in the current directory and opens it in your
default browser.

### What the visualization shows

- **Collapsible directories**: Click folders to expand/collapse
- **File sizes**: Displayed next to each file
- **Colors**: Different colors for different file types
- **Directory totals**: Shows aggregate size of each folder
```

Create `~/.claude/skills/codebase-visualizer/scripts/visualize.py`. This script scans a directory tree and generates a self-contained HTML file with:

- A **summary sidebar** showing file count, directory count, total size, and number of file types
- A **bar chart** breaking down the codebase by file type (top 8 by size)
- A **collapsible tree** where you can expand and collapse directories, with color-coded file type indicators

The script requires Python but uses only built-in libraries, so there are no packages to install:

```
#!/usr/bin/env python3
"""Generate an interactive collapsible tree visualization of a codebase."""

from pathlib import Path
from collections import Counter

IGNORE = {'.git', 'node_modules', '__pycache__', '.venv', 'venv', 'dist', 'build'}

def scan(path: Path, stats: dict) -> dict:
    result = {"name": path.name, "children": [], "size": 0}
    try:
        for item in sorted(path.iterdir()):
            if item.name in IGNORE or item.name.startswith('.'):
                continue
            if item.is_file():
                size = item.stat().st_size
                ext = item.suffix.lower() or '(no ext)'
                result["children"].append({"name": item.name, "size": size,
"ext": ext})
                result["size"] += size
                stats["files"] += 1
                stats["extensions"][ext] += 1
                stats["ext_sizes"][ext] += size
            elif item.is_dir():
                stats["dirs"] += 1
                child = scan(item, stats)
                if child["children"]:
                    result["children"].append(child)
                    result["size"] += child["size"]
    except PermissionError:
        pass
    return result

def generate_html(data: dict, stats: dict, output: Path) -> None:
    ext_sizes = stats["ext_sizes"]
    total_size = sum(ext_sizes.values()) or 1
    sorted_exts = sorted(ext_sizes.items(), key=lambda x: -x[1])[:8]
    colors = {
```

```
        '.js': '#f7df1e', '.ts': '#3178c6', '.py': '#3776ab', '.go': '#00add8',
        '.rs': '#dea584', '.rb': '#cc342d', '.css': '#264de4', '.html': '#e34c26',
        '.json': '#6b7280', '.md': '#083fa1', '.yaml': '#cb171e', '.yml': '#cb171e
',
        '.mdx': '#083fa1', '.tsx': '#3178c6', '.jsx': '#61dafb', '.sh': '#4eaa25',
    }
    lang_bars = "".join(
        f'<div class="bar-row"><span class="bar-label">{ext}</span>'
        f'<div class="bar" style="width:{(size/total_size)*100}%;background:{color
s.get(ext,"#6b7280")}"></div>'
        f'<span class="bar-pct">{(size/total_size)*100:.1f}%</span></div>'
        for ext, size in sorted_exts
    )
    def fmt(b):
        if b < 1024: return f"{b} B"
        if b < 1048576: return f"{b/1024:.1f} KB"
        return f"{b/1048576:.1f} MB"

    html = f'''<!DOCTYPE html>
<html><head>
  <meta charset="utf-8"><title>Codebase Explorer</title>
  <style>
    body {{ font: 14px/1.5 system-ui, sans-serif; margin: 0; background: #1a1a2e;
color: #eee; }}
    .container {{ display: flex; height: 100vh; }}
    .sidebar {{ width: 280px; background: #252542; padding: 20px; border-right:
1px solid #3d3d5c; overflow-y: auto; flex-shrink: 0; }}
    .main {{ flex: 1; padding: 20px; overflow-y: auto; }}
    h1 {{ margin: 0 0 10px 0; font-size: 18px; }}
    h2 {{ margin: 20px 0 10px 0; font-size: 14px; color: #888; text-transform:
uppercase; }}
    .stat {{ display: flex; justify-content: space-between; padding: 8px 0;
border-bottom: 1px solid #3d3d5c; }}
    .stat-value {{ font-weight: bold; }}
    .bar-row {{ display: flex; align-items: center; margin: 6px 0; }}
    .bar-label {{ width: 55px; font-size: 12px; color: #aaa; }}
    .bar {{ height: 18px; border-radius: 3px; }}
    .bar-pct {{ margin-left: 8px; font-size: 12px; color: #666; }}
    .tree {{ list-style: none; padding-left: 20px; }}
```

```
    details {{ cursor: pointer; }}
    summary {{ padding: 4px 8px; border-radius: 4px; }}
    summary:hover {{ background: #2d2d44; }}
    .folder {{ color: #ffd700; }}
    .file {{ display: flex; align-items: center; padding: 4px 8px; border-radius:
4px; }}
    .file:hover {{ background: #2d2d44; }}
    .size {{ color: #888; margin-left: auto; font-size: 12px; }}
    .dot {{ width: 8px; height: 8px; border-radius: 50%; margin-right: 8px; }}
  </style>
</head><body>
  <div class="container">
    <div class="sidebar">
      <h1>📊 Summary</h1>
      <div class="stat"><span>Files</span><span class="stat-
value">{stats["files"]:,}</span></div>
      <div class="stat"><span>Directories</span><span class="stat-value">{stats["d
irs"]:,}</span></div>
      <div class="stat"><span>Total size</span><span class="stat-
value">{fmt(data["size"])}</span></div>
      <div class="stat"><span>File types</span><span class="stat-value">{len(stats
["extensions"])}</span></div>
      <h2>By file type</h2>
      {lang_bars}
    </div>
    <div class="main">
      <h1>📁 {data["name"]}</h1>
      <ul class="tree" id="root"></ul>
    </div>
  </div>
  <script>
    const data = {json.dumps(data)};
    const colors = {json.dumps(colors)};
    function fmt(b) {{ if (b < 1024) return b + ' B'; if (b < 1048576) return (b/
1024).toFixed(1) + ' KB'; return (b/1048576).toFixed(1) + ' MB'; }}
    function render(node, parent) {{
      if (node.children) {{
        const det = document.createElement('details');
        det.open = parent === document.getElementById('root');
```

```
        det.innerHTML = `<summary><span class="folder">📁 ${{node.name}}</
span><span class="size">${{fmt(node.size)}}</span></summary>`;
        const ul = document.createElement('ul'); ul.className = 'tree';
        node.children.sort((a,b) => (b.children?1:0)-(a.children?1:0) ||
a.name.localeCompare(b.name));
        node.children.forEach(c => render(c, ul));
        det.appendChild(ul);
        const li = document.createElement('li'); li.appendChild(det);
parent.appendChild(li);
      }} else {{
        const li = document.createElement('li'); li.className = 'file';
        li.innerHTML = `<span class="dot" style="background:${{colors[node.ext]||'
#6b7280'}}"></span>${{node.name}}<span class="size">${{fmt(node.size)}}</span>`;
        parent.appendChild(li);
      }}
    }}
    data.children.forEach(c => render(c, document.getElementById('root')));
  </script>
</body></html>'''
    output.write_text(html)

if __name__ == '__main__':
    target = Path(sys.argv[1] if len(sys.argv) > 1 else '.').resolve()
    stats = {"files": 0, "dirs": 0, "extensions": Counter(), "ext_sizes":
Counter()}
    data = scan(target, stats)
    out = Path('codebase-map.html')
    generate_html(data, stats, out)
    print(f'Generated {out.absolute()}')
    webbrowser.open(f'file://{out.absolute()}')
```

To test, open Claude Code in any project and ask “Visualize this codebase.” Claude runs the script, generates `codebase-map.html`, and opens it in your browser.

This pattern works for any visual output: dependency graphs, test coverage reports, API documentation, or database schema visualizations. The bundled script does the heavy lifting while Claude handles orchestration.

Troubleshooting

Skill not triggering

If Claude doesn't use your skill when expected:

1. Check the description includes keywords users would naturally say
2. Verify the skill appears in `What skills are available?`
3. Try rephrasing your request to match the description more closely
4. Invoke it directly with `/skill-name` if the skill is user-invocable

Skill triggers too often

If Claude uses your skill when you don't want it:

1. Make the description more specific
2. Add `disable-model-invocation: true` if you only want manual invocation

Claude doesn't see all my skills

Skill descriptions are loaded into context so Claude knows what's available. If you have many skills, they may exceed the character budget. The budget scales dynamically at 2% of the context window, with a fallback of 16,000 characters. Run `/context` to check for a warning about excluded skills.

To override the limit, set the `SLASH_COMMAND_TOOL_CHAR_BUDGET` environment variable.

Related resources

- **Subagents**: delegate tasks to specialized agents
- **Plugins**: package and distribute skills with other extensions
- **Hooks**: automate workflows around tool events
- **Memory**: manage CLAUDE.md files for persistent context
- **Built-in commands**: reference for built-in `/` commands
- **Permissions**: control tool and skill access

Create plugins

> *Create custom plugins to extend Claude Code with skills, agents, hooks, and MCP servers.*

Plugins let you extend Claude Code with custom functionality that can be shared across projects and teams. This guide covers creating your own plugins with skills, agents, hooks, and MCP servers.

Looking to install existing plugins? See Discover and install plugins. For complete technical specifications, see Plugins reference.

When to use plugins vs standalone configuration

Claude Code supports two ways to add custom skills, agents, and hooks:

Approach	Skill names	Best for
Standalone (`.claude/` directory)	`/hello`	Personal workflows, project-specific customizations, quick experiments
Plugins (directories with `.claude-plugin/plugin.json`)	`/plugin-name:hello`	Sharing with teammates, distributing to community, versioned releases, reusable across projects

Use standalone configuration when:

- You're customizing Claude Code for a single project
- The configuration is personal and doesn't need to be shared
- You're experimenting with skills or hooks before packaging them
- You want short skill names like `/hello` or `/deploy`

Use plugins when:

- You want to share functionality with your team or community
- You need the same skills/agents across multiple projects
- You want version control and easy updates for your extensions

- You're distributing through a marketplace
- You're okay with namespaced skills like `/my-plugin:hello` (namespacing prevents conflicts between plugins)

> **Tip:**
>
> Start with standalone configuration in `.claude/` for quick iteration, then convert to a plugin when you're ready to share.

Quickstart

This quickstart walks you through creating a plugin with a custom skill. You'll create a manifest (the configuration file that defines your plugin), add a skill, and test it locally using the `--plugin-dir` flag.

Prerequisites

- Claude Code installed and authenticated
- Claude Code version 1.0.33 or later (run `claude --version` to check)

> **Note:**
>
> If you don't see the `/plugin` command, update Claude Code to the latest version. See Troubleshooting for upgrade instructions.

Create your first plugin

Step 1: Create the plugin directory

Every plugin lives in its own directory containing a manifest and your skills, agents, or hooks. Create one now:

```
mkdir my-first-plugin
```

Step 2: Create the plugin manifest

The manifest file at `.claude-plugin/plugin.json` defines your plugin's identity: its name, description, and version. Claude Code uses this metadata to display your plugin in the plugin manager.

Create the `.claude-plugin` directory inside your plugin folder:

```
mkdir my-first-plugin/.claude-plugin
```

Then create `my-first-plugin/.claude-plugin/plugin.json` with this content:

```
{
"name": "my-first-plugin",
"description": "A greeting plugin to learn the basics",
"version": "1.0.0",
"author": {
"name": "Your Name"
}
}
```

Field	Purpose
`name`	Unique identifier and skill namespace. Skills are prefixed with this (e.g., `/my-first-plugin:hello`).
`description`	Shown in the plugin manager when browsing or installing plugins.
`version`	Track releases using semantic versioning.
`author`	Optional. Helpful for attribution.

For additional fields like `homepage`, `repository`, and `license`, see the full manifest schema.

Step 3: Add a skill

Skills live in the `skills/` directory. Each skill is a folder containing a `SKILL.md` file. The folder name becomes the skill name, prefixed with the plugin's namespace (`hello/` in a plugin named `my-first-plugin` creates `/my-first-plugin:hello`).

Create a skill directory in your plugin folder:

```
mkdir -p my-first-plugin/skills/hello
```

Then create `my-first-plugin/skills/hello/SKILL.md` with this content:

```
---
description: Greet the user with a friendly message
disable-model-invocation: true
---

Greet the user warmly and ask how you can help them today.
```

Step 4: Test your plugin

Run Claude Code with the `--plugin-dir` flag to load your plugin:

```
claude --plugin-dir ./my-first-plugin
```

Once Claude Code starts, try your new skill:

```
/my-first-plugin:hello
```

You'll see Claude respond with a greeting. Run `/help` to see your skill listed under the plugin namespace.

> **Note:**
>
> **Why namespacing?** Plugin skills are always namespaced (like `/greet:hello`) to prevent conflicts when multiple plugins have skills with the same name.
>
> To change the namespace prefix, update the `name` field in `plugin.json`.

Step 5: Add skill arguments

Make your skill dynamic by accepting user input. The `$ARGUMENTS` placeholder captures any text the user provides after the skill name.

Update your `SKILL.md` file:

```
---
description: Greet the user with a personalized message
---

## Hello Skill

Greet the user named "$ARGUMENTS" warmly and ask how you can help them today. Make
the greeting personal and encouraging.
```

Run `/reload-plugins` to pick up the changes, then try the skill with your name:

```
/my-first-plugin:hello Alex
```

Claude will greet you by name. For more on passing arguments to skills, see Skills.

You've successfully created and tested a plugin with these key components:

- **Plugin manifest** (`.claude-plugin/plugin.json`): describes your plugin's metadata
- **Skills directory** (`skills/`): contains your custom skills
- **Skill arguments** (`$ARGUMENTS`): captures user input for dynamic behavior

> **Tip:**
>
> The `--plugin-dir` flag is useful for development and testing. When you're ready to share your plugin with others, see Create and distribute a plugin marketplace.

Plugin structure overview

You've created a plugin with a skill, but plugins can include much more: custom agents, hooks, MCP servers, and LSP servers.

> **Warning:**
>
> **Common mistake**: Don't put `commands/` , `agents/` , `skills/` , or `hooks/` inside the `.claude-plugin/` directory. Only `plugin.json` goes inside `.claude-plugin/` . All other directories must be at the plugin root level.

Directory	Location	Purpose
`.claude-plugin/`	Plugin root	Contains `plugin.json` manifest (optional if components use default locations)
`commands/`	Plugin root	Skills as Markdown files
`agents/`	Plugin root	Custom agent definitions
`skills/`	Plugin root	Agent Skills with `SKILL.md` files
`hooks/`	Plugin root	Event handlers in `hooks.json`
`.mcp.json`	Plugin root	MCP server configurations
`.lsp.json`	Plugin root	LSP server configurations for code intelligence
`settings.json`	Plugin root	Default settings applied when the plugin is enabled

Note:

Next steps: Ready to add more features? Jump to Develop more complex plugins to add agents, hooks, MCP servers, and LSP servers. For complete technical specifications of all plugin components, see Plugins reference.

Develop more complex plugins

Once you're comfortable with basic plugins, you can create more sophisticated extensions.

Add Skills to your plugin

Plugins can include Agent Skills to extend Claude's capabilities. Skills are model-invoked: Claude automatically uses them based on the task context.

Add a `skills/` directory at your plugin root with Skill folders containing `SKILL.md` files:

```
my-plugin/
├── .claude-plugin/
│   └── plugin.json
└── skills/
    └── code-review/
        └── SKILL.md
```

Each `SKILL.md` needs frontmatter with `name` and `description` fields, followed by instructions:

```
---
name: code-review
description: Reviews code for best practices and potential issues. Use when
reviewing code, checking PRs, or analyzing code quality.
---

When reviewing code, check for:
1. Code organization and structure
2. Error handling
3. Security concerns
4. Test coverage
```

After installing the plugin, run `/reload-plugins` to load the Skills. For complete Skill authoring guidance including progressive disclosure and tool restrictions, see Agent Skills.

Add LSP servers to your plugin

> **Tip:**
>
> For common languages like TypeScript, Python, and Rust, install the pre-built LSP plugins from the official marketplace. Create custom LSP plugins only when you need support for languages not already covered.

LSP (Language Server Protocol) plugins give Claude real-time code intelligence. If you need to support a language that doesn't have an official LSP plugin, you can create your own by adding an `.lsp.json` file to your plugin:

```
{
  "go": {
    "command": "gopls",
    "args": ["serve"],
    "extensionToLanguage": {
      ".go": "go"
    }
  }
}
```

Users installing your plugin must have the language server binary installed on their machine.

For complete LSP configuration options, see LSP servers.

Ship default settings with your plugin

Plugins can include a `settings.json` file at the plugin root to apply default configuration when the plugin is enabled. Currently, only the `agent` key is supported.

Setting `agent` activates one of the plugin's custom agents as the main thread, applying its system prompt, tool restrictions, and model. This lets a plugin change how Claude Code behaves by default when enabled.

```
{
  "agent": "security-reviewer"
}
```

This example activates the `security-reviewer` agent defined in the plugin's `agents/` directory. Settings from `settings.json` take priority over `settings` declared in `plugin.json`. Unknown keys are silently ignored.

Organize complex plugins

For plugins with many components, organize your directory structure by functionality. For complete directory layouts and organization patterns, see Plugin directory structure.

Test your plugins locally

Use the `--plugin-dir` flag to test plugins during development. This loads your plugin directly without requiring installation.

```
claude --plugin-dir ./my-plugin
```

When a `--plugin-dir` plugin has the same name as an installed marketplace plugin, the local copy takes precedence for that session. This lets you test changes to a plugin you already have installed without uninstalling it first. Marketplace plugins force-enabled by managed settings are the only exception and cannot be overridden.

As you make changes to your plugin, run `/reload-plugins` to pick up the updates without restarting. This reloads commands, skills, agents, hooks, plugin MCP servers, and plugin LSP servers. Test your plugin components:

- Try your skills with `/plugin-name:skill-name`
- Check that agents appear in `/agents`
- Verify hooks work as expected

> **Tip:**
>
> You can load multiple plugins at once by specifying the flag multiple times:
>
> ```
> claude --plugin-dir ./plugin-one --plugin-dir ./plugin-two
> ```

Debug plugin issues

If your plugin isn't working as expected:

1. **Check the structure**: Ensure your directories are at the plugin root, not inside `.claude-plugin/`
2. **Test components individually**: Check each command, agent, and hook separately
3. **Use validation and debugging tools**: See Debugging and development tools for CLI commands and troubleshooting techniques

Share your plugins

When your plugin is ready to share:

1. **Add documentation**: Include a `README.md` with installation and usage instructions
2. **Version your plugin**: Use semantic versioning in your `plugin.json`
3. **Create or use a marketplace**: Distribute through plugin marketplaces for installation
4. **Test with others**: Have team members test the plugin before wider distribution

Once your plugin is in a marketplace, others can install it using the instructions in Discover and install plugins.

Submit your plugin to the official marketplace

To submit a plugin to the official Anthropic marketplace, use one of the in-app submission forms:

- **Claude.ai**: claude.ai/settings/plugins/submit
- **Console**: platform.claude.com/plugins/submit

> **Note:**
>
> For complete technical specifications, debugging techniques, and distribution strategies, see Plugins reference.

Convert existing configurations to plugins

If you already have skills or hooks in your `.claude/` directory, you can convert them into a plugin for easier sharing and distribution.

Migration steps

Step 1: Create the plugin structure

Create a new plugin directory:

```
mkdir -p my-plugin/.claude-plugin
```

Create the manifest file at `my-plugin/.claude-plugin/plugin.json`:

```
{
  "name": "my-plugin",
  "description": "Migrated from standalone configuration",
  "version": "1.0.0"
}
```

Step 2: Copy your existing files

Copy your existing configurations to the plugin directory:

```
## Copy commands
cp -r .claude/commands my-plugin/

## Copy agents (if any)
cp -r .claude/agents my-plugin/

## Copy skills (if any)
cp -r .claude/skills my-plugin/
```

Step 3: Migrate hooks

If you have hooks in your settings, create a hooks directory:

```
mkdir my-plugin/hooks
```

Create `my-plugin/hooks/hooks.json` with your hooks configuration. Copy the `hooks` object from your `.claude/settings.json` or `settings.local.json`, since the format is the same. The command receives hook input as JSON on stdin, so use `jq` to extract the file path:

```
{
  "hooks": {
    "PostToolUse": [
      {
        "matcher": "Write|Edit",
        "hooks": [{ "type": "command", "command":
"jq -r '.tool_input.file_path' | xargs npm run lint:fix" }]
      }
    ]
  }
}
```

Step 4: Test your migrated plugin

Load your plugin to verify everything works:

```
claude --plugin-dir ./my-plugin
```

Test each component: run your commands, check agents appear in `/agents`, and verify hooks trigger correctly.

What changes when migrating

Standalone (`.claude/`)	Plugin
Only available in one project	Can be shared via marketplaces
Files in `.claude/commands/`	Files in `plugin-name/commands/`
Hooks in `settings.json`	Hooks in `hooks/hooks.json`
Must manually copy to share	Install with `/plugin install`

> **Note:**
>
> After migrating, you can remove the original files from `.claude/` to avoid duplicates. The plugin version will take precedence when loaded.

Next steps

Now that you understand Claude Code's plugin system, here are suggested paths for different goals:

For plugin users

- Discover and install plugins: browse marketplaces and install plugins
- Configure team marketplaces: set up repository-level plugins for your team

For plugin developers

- Create and distribute a marketplace: package and share your plugins
- Plugins reference: complete technical specifications
- Dive deeper into specific plugin components:
  - Skills: skill development details
  - Subagents: agent configuration and capabilities
  - Hooks: event handling and automation
  - MCP: external tool integration

Plugins reference

> *Complete technical reference for Claude Code plugin system, including schemas, CLI commands, and component specifications.*

> **Tip:**
>
> Looking to install plugins? See Discover and install plugins. For creating plugins, see Plugins. For distributing plugins, see Plugin marketplaces.

This reference provides complete technical specifications for the Claude Code plugin system, including component schemas, CLI commands, and development tools.

A **plugin** is a self-contained directory of components that extends Claude Code with custom functionality. Plugin components include skills, agents, hooks, MCP servers, and LSP servers.

Plugin components reference

Skills

Plugins add skills to Claude Code, creating `/name` shortcuts that you or Claude can invoke.

Location: `skills/` or `commands/` directory in plugin root

File format: Skills are directories with `SKILL.md`; commands are simple markdown files

Skill structure:

```
skills/
├── pdf-processor/
│   ├── SKILL.md
│   ├── reference.md (optional)
│   └── scripts/ (optional)
└── code-reviewer/
    └── SKILL.md
```

Integration behavior:

- Skills and commands are automatically discovered when the plugin is installed
- Claude can invoke them automatically based on task context
- Skills can include supporting files alongside SKILL.md

For complete details, see Skills.

Agents

Plugins can provide specialized subagents for specific tasks that Claude can invoke automatically when appropriate.

Location: `agents/` directory in plugin root

File format: Markdown files describing agent capabilities

Agent structure:

```
---
name: agent-name
description: What this agent specializes in and when Claude should invoke it
---

Detailed system prompt for the agent describing its role, expertise, and behavior.
```

Integration points:

- Agents appear in the `/agents` interface
- Claude can invoke agents automatically based on task context
- Agents can be invoked manually by users
- Plugin agents work alongside built-in Claude agents

For complete details, see Subagents.

Hooks

Plugins can provide event handlers that respond to Claude Code events automatically.

Location: `hooks/hooks.json` in plugin root, or inline in plugin.json

Format: JSON configuration with event matchers and actions

Hook configuration:

```
{
  "hooks": {
    "PostToolUse": [
      {
        "matcher": "Write|Edit",
        "hooks": [
          {
            "type": "command",
            "command": "${CLAUDE_PLUGIN_ROOT}/scripts/format-code.sh"
          }
        ]
      }
    ]
  }
}
```

Available events:

- `PreToolUse`: Before Claude uses any tool
- `PostToolUse`: After Claude successfully uses any tool
- `PostToolUseFailure`: After Claude tool execution fails
- `PermissionRequest`: When a permission dialog is shown
- `UserPromptSubmit`: When user submits a prompt
- `Notification`: When Claude Code sends notifications
- `Stop`: When Claude attempts to stop
- `SubagentStart`: When a subagent is started
- `SubagentStop`: When a subagent attempts to stop
- `SessionStart`: At the beginning of sessions
- `SessionEnd`: At the end of sessions
- `TeammateIdle`: When an agent team teammate is about to go idle
- `TaskCompleted`: When a task is being marked as completed
- `PreCompact`: Before conversation history is compacted

Hook types:

- `command`: Execute shell commands or scripts
- `prompt`: Evaluate a prompt with an LLM (uses `$ARGUMENTS` placeholder for context)

- `agent`: Run an agentic verifier with tools for complex verification tasks

MCP servers

Plugins can bundle Model Context Protocol (MCP) servers to connect Claude Code with external tools and services.

Location: `.mcp.json` in plugin root, or inline in plugin.json

Format: Standard MCP server configuration

MCP server configuration:

```
{
  "mcpServers": {
    "plugin-database": {
      "command": "${CLAUDE_PLUGIN_ROOT}/servers/db-server",
      "args": ["--config", "${CLAUDE_PLUGIN_ROOT}/config.json"],
      "env": {
        "DB_PATH": "${CLAUDE_PLUGIN_ROOT}/data"
      }
    },
    "plugin-api-client": {
      "command": "npx",
      "args": ["@company/mcp-server", "--plugin-mode"],
      "cwd": "${CLAUDE_PLUGIN_ROOT}"
    }
  }
}
```

Integration behavior:

- Plugin MCP servers start automatically when the plugin is enabled
- Servers appear as standard MCP tools in Claude's toolkit
- Server capabilities integrate seamlessly with Claude's existing tools
- Plugin servers can be configured independently of user MCP servers

LSP servers

> **Tip:**

> Looking to use LSP plugins? Install them from the official marketplace: search for "lsp" in the `/plugin` Discover tab. This section documents how to create LSP plugins for languages not covered by the official marketplace.

Plugins can provide Language Server Protocol (LSP) servers to give Claude real-time code intelligence while working on your codebase.

LSP integration provides:

- **Instant diagnostics**: Claude sees errors and warnings immediately after each edit
- **Code navigation**: go to definition, find references, and hover information
- **Language awareness**: type information and documentation for code symbols

Location: `.lsp.json` in plugin root, or inline in `plugin.json`

Format: JSON configuration mapping language server names to their configurations

`.lsp.json` file format:

```
{
  "go": {
    "command": "gopls",
    "args": ["serve"],
    "extensionToLanguage": {
      ".go": "go"
    }
  }
}
```

Inline in `plugin.json`:

```
{
  "name": "my-plugin",
  "lspServers": {
    "go": {
      "command": "gopls",
      "args": ["serve"],
      "extensionToLanguage": {
        ".go": "go"
      }
    }
  }
}
```

Required fields:

Field	Description
`command`	The LSP binary to execute (must be in PATH)
`extensionToLanguage`	Maps file extensions to language identifiers

Optional fields:

Field	Description
`args`	Command-line arguments for the LSP server
`transport`	Communication transport: `stdio` (default) or `socket`
`env`	Environment variables to set when starting the server
`initializationOptions`	Options passed to the server during initialization
`settings`	Settings passed via `workspace/didChangeConfiguration`
`workspaceFolder`	Workspace folder path for the server
`startupTimeout`	Max time to wait for server startup (milliseconds)
`shutdownTimeout`	Max time to wait for graceful shutdown (milliseconds)
`restartOnCrash`	Whether to automatically restart the server if it crashes

Field	Description
`maxRestarts`	Maximum number of restart attempts before giving up

> **Warning:**
>
> **You must install the language server binary separately.** LSP plugins configure how Claude Code connects to a language server, but they don't include the server itself. If you see `Executable not found in $PATH` in the `/plugin` Errors tab, install the required binary for your language.

Available LSP plugins:

Plugin	Language server	Install command
`pyright-lsp`	Pyright (Python)	`pip install pyright` or `npm install -g pyright`
`typescript-lsp`	TypeScript Language Server	`npm install -g typescript-language-server typescript`
`rust-lsp`	rust-analyzer	See rust-analyzer installation

Install the language server first, then install the plugin from the marketplace.

Plugin installation scopes

When you install a plugin, you choose a **scope** that determines where the plugin is available and who else can use it:

Scope	Settings file	Use case
`user`	`~/.claude/settings.json`	Personal plugins available across all projects (default)
`project`	`.claude/settings.json`	Team plugins shared via version control
`local`	`.claude/settings.local.json`	Project-specific plugins, gitignored

Scope	Settings file	Use case
`managed`	Managed settings	Managed plugins (read-only, update only)

Plugins use the same scope system as other Claude Code configurations. For installation instructions and scope flags, see Install plugins. For a complete explanation of scopes, see Configuration scopes.

Plugin manifest schema

The `.claude-plugin/plugin.json` file defines your plugin's metadata and configuration. This section documents all supported fields and options.

The manifest is optional. If omitted, Claude Code auto-discovers components in default locations and derives the plugin name from the directory name. Use a manifest when you need to provide metadata or custom component paths.

Complete schema

```
{
  "name": "plugin-name",
  "version": "1.2.0",
  "description": "Brief plugin description",
  "author": {
    "name": "Author Name",
    "email": "author@example.com",
    "url": "https://github.com/author"
  },
  "homepage": "https://docs.example.com/plugin",
  "repository": "https://github.com/author/plugin",
  "license": "MIT",
  "keywords": ["keyword1", "keyword2"],
  "commands": ["./custom/commands/special.md"],
  "agents": "./custom/agents/",
  "skills": "./custom/skills/",
  "hooks": "./config/hooks.json",
  "mcpServers": "./mcp-config.json",
  "outputStyles": "./styles/",
  "lspServers": "./.lsp.json"
}
```

Required fields

If you include a manifest, `name` is the only required field.

Field	Type	Description	Example
`name`	string	Unique identifier (kebab-case, no spaces)	`"deployment-tools"`

This name is used for namespacing components. For example, in the UI, the agent `agent-creator` for the plugin with name `plugin-dev` will appear as `plugin-dev:agent-creator`.

Metadata fields

Field	Type	Description	Example
`version`	string	Semantic version. If also set in the marketplace entry, `plugin.json` takes priority. You only need to set it in one place.	`"2.1.0"`
`description`	string	Brief explanation of plugin purpose	`"Deployment automation tools"`
`author`	object	Author information	`{"name": "Dev Team", "email": "dev@company.com"}`
`homepage`	string	Documentation URL	`"https://docs.example.com"`
`repository`	string	Source code URL	`"https://github.com/user/plugin"`
`license`	string	License identifier	`"MIT"`, `"Apache-2.0"`
`keywords`	array	Discovery tags	`["deployment", "ci-cd"]`

Component path fields

Field	Type	Description	Example
`commands`	string|array	Additional command files/directories	`"./custom/cmd.md"` or `["./cmd1.md"]`
`agents`	string|array	Additional agent files	`"./custom/agents/reviewer.md"`

Field	Type	Description	Example
`skills`	string|array	Additional skill directories	`"./custom/skills/"`
`hooks`	string|array|object	Hook config paths or inline config	`"./my-extra-hooks.json"`
`mcpServers`	string|array|object	MCP config paths or inline config	`"./my-extra-mcp-config.json"`
`outputStyles`	string|array	Additional output style files/directories	`"./styles/"`
`lspServers`	string|array|object	Language Server Protocol configs for code intelligence (go to definition, find references, etc.)	`"./.lsp.json"`

Path behavior rules

Important: Custom paths supplement default directories - they don't replace them.

- If `commands/` exists, it's loaded in addition to custom command paths
- All paths must be relative to plugin root and start with `./`
- Commands from custom paths use the same naming and namespacing rules
- Multiple paths can be specified as arrays for flexibility

Path examples:

```
{
  "commands": [
    "./specialized/deploy.md",
    "./utilities/batch-process.md"
  ],
  "agents": [
    "./custom-agents/reviewer.md",
    "./custom-agents/tester.md"
  ]
}
```

Environment variables

`${CLAUDE_PLUGIN_ROOT}` : Contains the absolute path to your plugin directory. Use this in hooks, MCP servers, and scripts to ensure correct paths regardless of installation location.

```
{
  "hooks": {
    "PostToolUse": [
      {
        "hooks": [
          {
            "type": "command",
            "command": "${CLAUDE_PLUGIN_ROOT}/scripts/process.sh"
          }
        ]
      }
    ]
  }
}
```

Plugin caching and file resolution

Plugins are specified in one of two ways:

- Through `claude --plugin-dir` , for the duration of a session.
- Through a marketplace, installed for future sessions.

For security and verification purposes, Claude Code copies *marketplace* plugins to the user's local **plugin cache** (`~/.claude/plugins/cache`) rather than using them in-place. Understanding this behavior is important when developing plugins that reference external files.

Path traversal limitations

Installed plugins cannot reference files outside their directory. Paths that traverse outside the plugin root (such as `../shared-utils`) will not work after installation because those external files are not copied to the cache.

Working with external dependencies

If your plugin needs to access files outside its directory, you can create symbolic links to external files within your plugin directory. Symlinks are honored during the copy process:

```
## Inside your plugin directory
ln -s /path/to/shared-utils ./shared-utils
```

The symlinked content will be copied into the plugin cache. This provides flexibility while maintaining the security benefits of the caching system.

Plugin directory structure

Standard plugin layout

A complete plugin follows this structure:

```
enterprise-plugin/
├── .claude-plugin/           # Metadata directory (optional)
│   └── plugin.json             # plugin manifest
├── commands/                 # Default command location
│   ├── status.md
│   └── logs.md
├── agents/                   # Default agent location
│   ├── security-reviewer.md
│   ├── performance-tester.md
│   └── compliance-checker.md
├── skills/                   # Agent Skills
│   ├── code-reviewer/
│   │   └── SKILL.md
│   └── pdf-processor/
│       ├── SKILL.md
│       └── scripts/
├── hooks/                    # Hook configurations
│   ├── hooks.json           # Main hook config
│   └── security-hooks.json  # Additional hooks
├── settings.json             # Default settings for the plugin
├── .mcp.json                 # MCP server definitions
├── .lsp.json                 # LSP server configurations
├── scripts/                  # Hook and utility scripts
│   ├── security-scan.sh
│   ├── format-code.py
│   └── deploy.js
├── LICENSE                   # License file
└── CHANGELOG.md              # Version history
```

Warning:

The `.claude-plugin/` directory contains the `plugin.json` file. All other directories (commands/, agents/, skills/, hooks/) must be at the plugin root, not inside `.claude-plugin/`.

File locations reference

Component	Default Location	Purpose
Manifest	`.claude-plugin/plugin.json`	Plugin metadata and configuration (optional)
Commands	`commands/`	Skill Markdown files (legacy; use `skills/` for new skills)
Agents	`agents/`	Subagent Markdown files
Skills	`skills/`	Skills with `<name>/SKILL.md` structure
Hooks	`hooks/hooks.json`	Hook configuration
MCP servers	`.mcp.json`	MCP server definitions
LSP servers	`.lsp.json`	Language server configurations
Settings	`settings.json`	Default configuration applied when the plugin is enabled. Only `agent` settings are currently supported

CLI commands reference

Claude Code provides CLI commands for non-interactive plugin management, useful for scripting and automation.

plugin install

Install a plugin from available marketplaces.

```
claude plugin install <plugin> [options]
```

Arguments:

- `<plugin>`: Plugin name or `plugin-name@marketplace-name` for a specific marketplace

Options:

Option	Description	Default
`-s, --scope <scope>`	Installation scope: `user`, `project`, or `local`	`user`
`-h, --help`	Display help for command	

Scope determines which settings file the installed plugin is added to. For example, –scope project writes to `enabledPlugins` in .claude/settings.json, making the plugin available to everyone who clones the project repository.

Examples:

```
## Install to user scope (default)
claude plugin install formatter@my-marketplace

## Install to project scope (shared with team)
claude plugin install formatter@my-marketplace --scope project

## Install to local scope (gitignored)
claude plugin install formatter@my-marketplace --scope local
```

plugin uninstall

Remove an installed plugin.

```
claude plugin uninstall <plugin> [options]
```

Arguments:

- `<plugin>`: Plugin name or `plugin-name@marketplace-name`

Options:

Option	Description	Default
`-s, --scope <scope>`	Uninstall from scope: `user`, `project`, or `local`	`user`
`-h, --help`	Display help for command	

Aliases: `remove`, `rm`

plugin enable

Enable a disabled plugin.

```
claude plugin enable <plugin> [options]
```

Arguments:

- `<plugin>`: Plugin name or `plugin-name@marketplace-name`

Options:

Option	Description	Default
`-s, --scope <scope>`	Scope to enable: `user`, `project`, or `local`	`user`
`-h, --help`	Display help for command	

plugin disable

Disable a plugin without uninstalling it.

```
claude plugin disable <plugin> [options]
```

Arguments:

- `<plugin>`: Plugin name or `plugin-name@marketplace-name`

Options:

Option	Description	Default
`-s, --scope <scope>`	Scope to disable: `user`, `project`, or `local`	`user`
`-h, --help`	Display help for command	

plugin update

Update a plugin to the latest version.

```
claude plugin update <plugin> [options]
```

Arguments:

- `<plugin>`: Plugin name or `plugin-name@marketplace-name`

Options:

Option	Description	Default
`-s, --scope <scope>`	Scope to update: `user`, `project`, `local`, or `managed`	`user`
`-h, --help`	Display help for command	

Debugging and development tools

Debugging commands

Use `claude --debug` (or `/debug` within the TUI) to see plugin loading details:

This shows:

- Which plugins are being loaded
- Any errors in plugin manifests
- Command, agent, and hook registration
- MCP server initialization

Common issues

Issue	Cause	Solution
Plugin not loading	Invalid `plugin.json`	Validate JSON syntax with `claude plugin validate` or `/plugin validate`
Commands not appearing	Wrong directory structure	Ensure `commands/` at root, not in `.claude-plugin/`
Hooks not firing	Script not executable	Run `chmod +x script.sh`

Issue	Cause	Solution
MCP server fails	Missing `${CLAUDE_PLUGIN_ROOT}`	Use variable for all plugin paths
Path errors	Absolute paths used	All paths must be relative and start with `./`
LSP `Executable not found in $PATH`	Language server not installed	Install the binary (e.g., `npm install -g typescript-language-server typescript`)

Example error messages

Manifest validation errors:

- `Invalid JSON syntax: Unexpected token } in JSON at position 142`: check for missing commas, extra commas, or unquoted strings
- `Plugin has an invalid manifest file at .claude-plugin/plugin.json. Validation errors: name: Required`: a required field is missing
- `Plugin has a corrupt manifest file at .claude-plugin/plugin.json. JSON parse error: ...`: JSON syntax error

Plugin loading errors:

- `Warning: No commands found in plugin my-plugin custom directory: ./cmds. Expected .md files or SKILL.md in subdirectories.`: command path exists but contains no valid command files
- `Plugin directory not found at path: ./plugins/my-plugin. Check that the marketplace entry has the correct path.`: the `source` path in marketplace.json points to a non-existent directory
- `Plugin my-plugin has conflicting manifests: both plugin.json and marketplace entry specify components.`: remove duplicate component definitions or remove `strict: false` in marketplace entry

Hook troubleshooting

Hook script not executing:

1. Check the script is executable: `chmod +x ./scripts/your-script.sh`
2. Verify the shebang line: First line should be `#!/bin/bash` or `#!/usr/bin/env bash`

3. Check the path uses `${CLAUDE_PLUGIN_ROOT}`: `"command": "${CLAUDE_PLUGIN_ROOT}/scripts/your-script.sh"`
4. Test the script manually: `./scripts/your-script.sh`

Hook not triggering on expected events:

1. Verify the event name is correct (case-sensitive): `PostToolUse`, not `postToolUse`
2. Check the matcher pattern matches your tools: `"matcher": "Write|Edit"` for file operations
3. Confirm the hook type is valid: `command`, `prompt`, or `agent`

MCP server troubleshooting

Server not starting:

1. Check the command exists and is executable
2. Verify all paths use `${CLAUDE_PLUGIN_ROOT}` variable
3. Check the MCP server logs: `claude --debug` shows initialization errors
4. Test the server manually outside of Claude Code

Server tools not appearing:

1. Ensure the server is properly configured in `.mcp.json` or `plugin.json`
2. Verify the server implements the MCP protocol correctly
3. Check for connection timeouts in debug output

Directory structure mistakes

Symptoms: Plugin loads but components (commands, agents, hooks) are missing.

Correct structure: Components must be at the plugin root, not inside `.claude-plugin/`. Only `plugin.json` belongs in `.claude-plugin/`.

```
my-plugin/
├── .claude-plugin/
│   └── plugin.json      ← Only manifest here
├── commands/            ← At root level
├── agents/              ← At root level
└── hooks/               ← At root level
```

If your components are inside `.claude-plugin/`, move them to the plugin root.

Debug checklist:

1. Run `claude --debug` and look for "loading plugin" messages
2. Check that each component directory is listed in the debug output
3. Verify file permissions allow reading the plugin files

Distribution and versioning reference

Version management

Follow semantic versioning for plugin releases:

```
{
  "name": "my-plugin",
  "version": "2.1.0"
}
```

Version format: `MAJOR.MINOR.PATCH`

- **MAJOR**: Breaking changes (incompatible API changes)
- **MINOR**: New features (backward-compatible additions)
- **PATCH**: Bug fixes (backward-compatible fixes)

Best practices:

- Start at `1.0.0` for your first stable release
- Update the version in `plugin.json` before distributing changes
- Document changes in a `CHANGELOG.md` file
- Use pre-release versions like `2.0.0-beta.1` for testing

> **Warning:**
>
> Claude Code uses the version to determine whether to update your plugin. If you change your plugin's code but don't bump the version in `plugin.json`, your plugin's existing users won't see your changes due to caching.
>
> If your plugin is within a marketplace directory, you can manage the version through `marketplace.json` instead and omit the `version` field from `plugin.json`.

See also

- Plugins - Tutorials and practical usage
- Plugin marketplaces - Creating and managing marketplaces
- Skills - Skill development details
- Subagents - Agent configuration and capabilities
- Hooks - Event handling and automation
- MCP - External tool integration
- Settings - Configuration options for plugins

Create and distribute a plugin marketplace

> *Build and host plugin marketplaces to distribute Claude Code extensions across teams and communities.*

A **plugin marketplace** is a catalog that lets you distribute plugins to others. Marketplaces provide centralized discovery, version tracking, automatic updates, and support for multiple source types (git repositories, local paths, and more). This guide shows you how to create your own marketplace to share plugins with your team or community.

Looking to install plugins from an existing marketplace? See Discover and install prebuilt plugins.

Overview

Creating and distributing a marketplace involves:

1. **Creating plugins**: build one or more plugins with commands, agents, hooks, MCP servers, or LSP servers. This guide assumes you already have plugins to distribute; see Create plugins for details on how to create them.
2. **Creating a marketplace file**: define a `marketplace.json` that lists your plugins and where to find them (see Create the marketplace file).
3. **Host the marketplace**: push to GitHub, GitLab, or another git host (see Host and distribute marketplaces).
4. **Share with users**: users add your marketplace with `/plugin marketplace add` and install individual plugins (see Discover and install plugins).

Once your marketplace is live, you can update it by pushing changes to your repository. Users refresh their local copy with `/plugin marketplace update`.

Walkthrough: create a local marketplace

This example creates a marketplace with one plugin: a `/quality-review` skill for code reviews. You'll create the directory structure, add a skill, create the plugin manifest and marketplace catalog, then install and test it.

Step 1: Create the directory structure

```
mkdir -p my-marketplace/.claude-plugin
mkdir -p my-marketplace/plugins/quality-review-plugin/.claude-plugin
mkdir -p my-marketplace/plugins/quality-review-plugin/skills/quality-review
```

Step 2: Create the skill

Create a `SKILL.md` file that defines what the `/quality-review` skill does.

```
---
description: Review code for bugs, security, and performance
disable-model-invocation: true
---

Review the code I've selected or the recent changes for:
- Potential bugs or edge cases
- Security concerns
- Performance issues
- Readability improvements

Be concise and actionable.
```

Step 3: Create the plugin manifest

Create a `plugin.json` file that describes the plugin. The manifest goes in the `.claude-plugin/` directory.

```
{
  "name": "quality-review-plugin",
  "description": "Adds a /quality-review skill for quick code reviews",
  "version": "1.0.0"
}
```

Step 4: Create the marketplace file

Create the marketplace catalog that lists your plugin.

```
{
  "name": "my-plugins",
  "owner": {
    "name": "Your Name"
  },
  "plugins": [
    {
      "name": "quality-review-plugin",
      "source": "./plugins/quality-review-plugin",
      "description": "Adds a /quality-review skill for quick code reviews"
    }
  ]
}
```

Step 5: Add and install

Add the marketplace and install the plugin.

```
/plugin marketplace add ./my-marketplace
/plugin install quality-review-plugin@my-plugins
```

Step 6: Try it out

Select some code in your editor and run your new command.

```
/review
```

To learn more about what plugins can do, including hooks, agents, MCP servers, and LSP servers, see Plugins.

> **Note:**
>
> **How plugins are installed**: When users install a plugin, Claude Code copies the plugin directory to a cache location. This means plugins can't reference files outside their directory using paths like `../shared-utils`, because those files won't be copied.
>
> If you need to share files across plugins, use symlinks (which are followed during copying). See Plugin caching and file resolution for details.

Create the marketplace file

Create `.claude-plugin/marketplace.json` in your repository root. This file defines your marketplace's name, owner information, and a list of plugins with their sources.

Each plugin entry needs at minimum a `name` and `source` (where to fetch it from). See the full schema below for all available fields.

```
{
  "name": "company-tools",
  "owner": {
    "name": "DevTools Team",
    "email": "devtools@example.com"
  },
  "plugins": [
    {
      "name": "code-formatter",
      "source": "./plugins/formatter",
      "description": "Automatic code formatting on save",
      "version": "2.1.0",
      "author": {
        "name": "DevTools Team"
      }
    },
    {
      "name": "deployment-tools",
      "source": {
        "source": "github",
        "repo": "company/deploy-plugin"
      },
      "description": "Deployment automation tools"
    }
  ]
}
```

Marketplace schema

Required fields

Field	Type	Description	Example
`name`	string	Marketplace identifier (kebab-case, no spaces). This is public-facing: users see it when installing plugins (for example, `/plugin install my-tool@your-marketplace`).	`"acme-tools"`
`owner`	object	Marketplace maintainer information (see fields below)	
`plugins`	array	List of available plugins	See below

Note:

Reserved names: The following marketplace names are reserved for official Anthropic use and cannot be used by third-party marketplaces: `claude-code-marketplace`, `claude-code-plugins`, `claude-plugins-official`, `anthropic-marketplace`, `anthropic-plugins`, `agent-skills`, `life-sciences`. Names that impersonate official marketplaces (like `official-claude-plugins` or `anthropic-tools-v2`) are also blocked.

Owner fields

Field	Type	Required	Description
`name`	string	Yes	Name of the maintainer or team
`email`	string	No	Contact email for the maintainer

Optional metadata

Field	Type	Description
`metadata.description`	string	Brief marketplace description
`metadata.version`	string	Marketplace version
`metadata.pluginRoot`	string	Base directory prepended to relative plugin source paths (for example, `"./plugins"` lets you write `"source": "formatter"` instead of `"source": "./plugins/formatter"`)

Plugin entries

Each plugin entry in the `plugins` array describes a plugin and where to find it. You can include any field from the plugin manifest schema (like `description`, `version`, `author`, `commands`, `hooks`, etc.), plus these marketplace-specific fields: `source`, `category`, `tags`, and `strict`.

Required fields

Field	Type	Description
`name`	string	Plugin identifier (kebab-case, no spaces). This is public-facing: users see it when installing (for example, `/plugin install my-plugin@marketplace`).
`source`	string|object	Where to fetch the plugin from (see Plugin sources below)

Optional plugin fields

Standard metadata fields:

Field	Type	Description
`description`	string	Brief plugin description
`version`	string	Plugin version
`author`	object	Plugin author information (`name` required, `email` optional)
`homepage`	string	Plugin homepage or documentation URL
`repository`	string	Source code repository URL
`license`	string	SPDX license identifier (for example, MIT, Apache-2.0)
`keywords`	array	Tags for plugin discovery and categorization
`category`	string	Plugin category for organization
`tags`	array	Tags for searchability
`strict`	boolean	Controls whether `plugin.json` is the authority for component definitions (default: true). See Strict mode below.

Component configuration fields:

Field	Type	Description
`commands`	string|array	Custom paths to command files or directories
`agents`	string|array	Custom paths to agent files
`hooks`	string|object	Custom hooks configuration or path to hooks file

Field	Type	Description
`mcpServers`	string|object	MCP server configurations or path to MCP config
`lspServers`	string|object	LSP server configurations or path to LSP config

Plugin sources

Plugin sources tell Claude Code where to fetch each individual plugin listed in your marketplace. These are set in the `source` field of each plugin entry in `marketplace.json`.

Once a plugin is cloned or copied into the local machine, it is copied into the local versioned plugin cache at `~/.claude/plugins/cache`.

Source	Type	Fields	Notes
Relative path	`string` (e.g. `"./my-plugin"`)	—	Local directory within the marketplace repo. Must start with `./`
`github`	object	`repo`, `ref?`, `sha?`	
`url`	object	`url`, `ref?`, `sha?`	Git URL source
`git-subdir`	object	`url`, `path`, `ref?`, `sha?`	Subdirectory within a git repo. Clones sparsely to minimize bandwidth for monorepos
`npm`	object	`package`, `version?`, `registry?`	Installed via `npm install`
`pip`	object	`package`, `version?`, `registry?`	Installed via pip

> **Note:**

Marketplace sources vs plugin sources: These are different concepts that control different things.

- **Marketplace source** — where to fetch the `marketplace.json` catalog itself. Set when users run `/plugin marketplace add` or in `extraKnownMarketplaces` settings. Supports `ref` (branch/tag) but not `sha`.
- **Plugin source** — where to fetch an individual plugin listed in the marketplace. Set in the `source` field of each plugin entry inside `marketplace.json`. Supports both `ref` (branch/tag) and `sha` (exact commit).

For example, a marketplace hosted at `acme-corp/plugin-catalog` (marketplace source) can list a plugin fetched from `acme-corp/code-formatter` (plugin source). The marketplace source and plugin source point to different repositories and are pinned independently.

Relative paths

For plugins in the same repository, use a path starting with `./`:

```
{
  "name": "my-plugin",
  "source": "./plugins/my-plugin"
}
```

Paths resolve relative to the marketplace root, which is the directory containing `.claude-plugin/`. In the example above, `./plugins/my-plugin` points to `<repo>/plugins/my-plugin`, even though `marketplace.json` lives at `<repo>/.claude-plugin/marketplace.json`. Do not use `../` to climb out of `.claude-plugin/`.

Note:

Relative paths only work when users add your marketplace via Git (GitHub, GitLab, or git URL). If users add your marketplace via a direct URL to the `marketplace.json` file, relative paths will not resolve correctly. For URL-based distribution, use GitHub, npm, or git URL sources instead. See Troubleshooting for details.

GitHub repositories

```
{
  "name": "github-plugin",
  "source": {
    "source": "github",
    "repo": "owner/plugin-repo"
  }
}
```

You can pin to a specific branch, tag, or commit:

```
{
  "name": "github-plugin",
  "source": {
    "source": "github",
    "repo": "owner/plugin-repo",
    "ref": "v2.0.0",
    "sha": "a1b2c3d4e5f6a7b8c9d0e1f2a3b4c5d6e7f8a9b0"
  }
}
```

Field	Type	Description
`repo`	string	Required. GitHub repository in `owner/repo` format
`ref`	string	Optional. Git branch or tag (defaults to repository default branch)
`sha`	string	Optional. Full 40-character git commit SHA to pin to an exact version

Git repositories

```
{
  "name": "git-plugin",
  "source": {
    "source": "url",
    "url": "https://gitlab.com/team/plugin.git"
  }
}
```

You can pin to a specific branch, tag, or commit:

```
{
  "name": "git-plugin",
  "source": {
    "source": "url",
    "url": "https://gitlab.com/team/plugin.git",
    "ref": "main",
    "sha": "a1b2c3d4e5f6a7b8c9d0e1f2a3b4c5d6e7f8a9b0"
  }
}
```

Field	Type	Description
`url`	string	Required. Full git repository URL (`https://` or `git@`). The `.git` suffix is optional, so Azure DevOps and AWS CodeCommit URLs without the suffix work
`ref`	string	Optional. Git branch or tag (defaults to repository default branch)
`sha`	string	Optional. Full 40-character git commit SHA to pin to an exact version

Git subdirectories

Use `git-subdir` to point to a plugin that lives inside a subdirectory of a git repository. Claude Code uses a sparse, partial clone to fetch only the subdirectory, minimizing bandwidth for large monorepos.

```
{
  "name": "my-plugin",
  "source": {
    "source": "git-subdir",
    "url": "https://github.com/acme-corp/monorepo.git",
    "path": "tools/claude-plugin"
  }
}
```

You can pin to a specific branch, tag, or commit:

```
{
  "name": "my-plugin",
  "source": {
    "source": "git-subdir",
    "url": "https://github.com/acme-corp/monorepo.git",
    "path": "tools/claude-plugin",
    "ref": "v2.0.0",
    "sha": "a1b2c3d4e5f6a7b8c9d0e1f2a3b4c5d6e7f8a9b0"
  }
}
```

The `url` field also accepts a GitHub shorthand (`owner/repo`) or SSH URLs (`git@github.com:owner/repo.git`).

Field	Type	Description
`url`	string	Required. Git repository URL, GitHub `owner/repo` shorthand, or SSH URL

Field	Type	Description
`path`	string	Required. Subdirectory path within the repo containing the plugin (for example, `"tools/claude-plugin"`)
`ref`	string	Optional. Git branch or tag (defaults to repository default branch)
`sha`	string	Optional. Full 40-character git commit SHA to pin to an exact version

npm packages

Plugins distributed as npm packages are installed using `npm install`. This works with any package on the public npm registry or a private registry your team hosts.

```
{
  "name": "my-npm-plugin",
  "source": {
    "source": "npm",
    "package": "@acme/claude-plugin"
  }
}
```

To pin to a specific version, add the `version` field:

```
{
  "name": "my-npm-plugin",
  "source": {
    "source": "npm",
    "package": "@acme/claude-plugin",
    "version": "2.1.0"
  }
}
```

To install from a private or internal registry, add the `registry` field:

```
{
  "name": "my-npm-plugin",
  "source": {
    "source": "npm",
    "package": "@acme/claude-plugin",
    "version": "^2.0.0",
    "registry": "https://npm.example.com"
  }
}
```

Field	Type	Description
`package`	string	Required. Package name or scoped package (for example, `@org/plugin`)
`version`	string	Optional. Version or version range (for example, `2.1.0`, `^2.0.0`, `~1.5.0`)
`registry`	string	Optional. Custom npm registry URL. Defaults to the system npm registry (typically npmjs.org)

Advanced plugin entries

This example shows a plugin entry using many of the optional fields, including custom paths for commands, agents, hooks, and MCP servers:

```
{
  "name": "enterprise-tools",
  "source": {
    "source": "github",
    "repo": "company/enterprise-plugin"
  },
  "description": "Enterprise workflow automation tools",
  "version": "2.1.0",
  "author": {
    "name": "Enterprise Team",
    "email": "enterprise@example.com"
  },
  "homepage": "https://docs.example.com/plugins/enterprise-tools",
  "repository": "https://github.com/company/enterprise-plugin",
  "license": "MIT",
  "keywords": ["enterprise", "workflow", "automation"],
  "category": "productivity",
  "commands": [
    "./commands/core/",
    "./commands/enterprise/",
    "./commands/experimental/preview.md"
  ],
  "agents": ["./agents/security-reviewer.md", "./agents/compliance-checker.md"],
  "hooks": {
    "PostToolUse": [
      {
        "matcher": "Write|Edit",
        "hooks": [
          {
            "type": "command",
            "command": "${CLAUDE_PLUGIN_ROOT}/scripts/validate.sh"
          }
        ]
      }
    ]
  },
  "mcpServers": {
    "enterprise-db": {
```

```
      "command": "${CLAUDE_PLUGIN_ROOT}/servers/db-server",
      "args": ["--config", "${CLAUDE_PLUGIN_ROOT}/config.json"]
    }
  },
  "strict": false
}
```

Key things to notice:

- `commands` and `agents` : You can specify multiple directories or individual files. Paths are relative to the plugin root.
- `${CLAUDE_PLUGIN_ROOT}` : Use this variable in hooks and MCP server configs to reference files within the plugin's installation directory. This is necessary because plugins are copied to a cache location when installed.
- `strict: false` : Since this is set to false, the plugin doesn't need its own `plugin.json` . The marketplace entry defines everything. See Strict mode below.

Strict mode

The `strict` field controls whether `plugin.json` is the authority for component definitions (commands, agents, hooks, skills, MCP servers, output styles).

Value	Behavior
`true` (default)	`plugin.json` is the authority. The marketplace entry can supplement it with additional components, and both sources are merged.
`false`	The marketplace entry is the entire definition. If the plugin also has a `plugin.json` that declares components, that's a conflict and the plugin fails to load.

When to use each mode:

- `strict: true` : the plugin has its own `plugin.json` and manages its own components. The marketplace entry can add extra commands or hooks on top. This is the default and works for most plugins.
- `strict: false` : the marketplace operator wants full control. The plugin repo provides raw files, and the marketplace entry defines which of those files are exposed as commands, agents, hooks, etc. Useful when the marketplace restructures or curates a plugin's components differently than the plugin author intended.

Host and distribute marketplaces

Host on GitHub (recommended)

GitHub provides the easiest distribution method:

1. **Create a repository**: Set up a new repository for your marketplace
2. **Add marketplace file**: Create `.claude-plugin/marketplace.json` with your plugin definitions
3. **Share with teams**: Users add your marketplace with `/plugin marketplace add owner/repo`

Benefits: Built-in version control, issue tracking, and team collaboration features.

Host on other git services

Any git hosting service works, such as GitLab, Bitbucket, and self-hosted servers. Users add with the full repository URL:

```
/plugin marketplace add https://gitlab.com/company/plugins.git
```

Private repositories

Claude Code supports installing plugins from private repositories. For manual installation and updates, Claude Code uses your existing git credential helpers. If `git clone` works for a private repository in your terminal, it works in Claude Code too. Common credential helpers include `gh auth login` for GitHub, macOS Keychain, and `git-credential-store`.

Background auto-updates run at startup without credential helpers, since interactive prompts would block Claude Code from starting. To enable auto-updates for private marketplaces, set the appropriate authentication token in your environment:

Provider	Environment variables	Notes
GitHub	`GITHUB_TOKEN` or `GH_TOKEN`	Personal access token or GitHub App token
GitLab	`GITLAB_TOKEN` or `GL_TOKEN`	Personal access token or project token
Bitbucket	`BITBUCKET_TOKEN`	App password or repository access token

Set the token in your shell configuration (for example, `.bashrc`, `.zshrc`) or pass it when running Claude Code:

```
export GITHUB_TOKEN=ghp_xxxxxxxxxxxxxxxxxxxx
```

> **Note:**
>
> For CI/CD environments, configure the token as a secret environment variable. GitHub Actions automatically provides `GITHUB_TOKEN` for repositories in the same organization.

Test locally before distribution

Test your marketplace locally before sharing:

```
/plugin marketplace add ./my-local-marketplace
/plugin install test-plugin@my-local-marketplace
```

For the full range of add commands (GitHub, Git URLs, local paths, remote URLs), see Add marketplaces.

Require marketplaces for your team

You can configure your repository so team members are automatically prompted to install your marketplace when they trust the project folder. Add your marketplace to `.claude/settings.json`:

```
{
  "extraKnownMarketplaces": {
    "company-tools": {
      "source": {
        "source": "github",
        "repo": "your-org/claude-plugins"
      }
    }
  }
}
```

You can also specify which plugins should be enabled by default:

```
{
  "enabledPlugins": {
    "code-formatter@company-tools": true,
    "deployment-tools@company-tools": true
  }
}
```

For full configuration options, see Plugin settings.

Managed marketplace restrictions

For organizations requiring strict control over plugin sources, administrators can restrict which plugin marketplaces users are allowed to add using the `strictKnownMarketplaces` setting in managed settings.

When `strictKnownMarketplaces` is configured in managed settings, the restriction behavior depends on the value:

Value	Behavior
Undefined (default)	No restrictions. Users can add any marketplace
Empty array `[]`	Complete lockdown. Users cannot add any new marketplaces
List of sources	Users can only add marketplaces that match the allowlist exactly

Common configurations

Disable all marketplace additions:

```
{
  "strictKnownMarketplaces": []
}
```

Allow specific marketplaces only:

```
{
  "strictKnownMarketplaces": [
    {
      "source": "github",
      "repo": "acme-corp/approved-plugins"
    },
    {
      "source": "github",
      "repo": "acme-corp/security-tools",
      "ref": "v2.0"
    },
    {
      "source": "url",
      "url": "https://plugins.example.com/marketplace.json"
    }
  ]
}
```

Allow all marketplaces from an internal git server using regex pattern matching on the host:

```
{
  "strictKnownMarketplaces": [
    {
      "source": "hostPattern",
      "hostPattern": "^github\\.example\\.com$"
    }
  ]
}
```

Allow filesystem-based marketplaces from a specific directory using regex pattern matching on the path:

```
{
  "strictKnownMarketplaces": [
    {
      "source": "pathPattern",
      "pathPattern": "^/opt/approved/"
    }
  ]
}
```

Use `".*"` as the `pathPattern` to allow any filesystem path while still controlling network sources with `hostPattern`.

> **Note:**
>
> `strictKnownMarketplaces` restricts what users can add, but does not register marketplaces on its own. To make allowed marketplaces available automatically without users running `/plugin marketplace add`, pair it with `extraKnownMarketplaces` in the same `managed-settings.json`. See Using both together.

How restrictions work

Restrictions are validated early in the plugin installation process, before any network requests or filesystem operations occur. This prevents unauthorized marketplace access attempts.

The allowlist uses exact matching for most source types. For a marketplace to be allowed, all specified fields must match exactly:

- For GitHub sources: `repo` is required, and `ref` or `path` must also match if specified in the allowlist
- For URL sources: the full URL must match exactly
- For `hostPattern` sources: the marketplace host is matched against the regex pattern
- For `pathPattern` sources: the marketplace's filesystem path is matched against the regex pattern

Because `strictKnownMarketplaces` is set in managed settings, individual users and project configurations cannot override these restrictions.

For complete configuration details including all supported source types and comparison with `extraKnownMarketplaces`, see the strictKnownMarketplaces reference.

Version resolution and release channels

Plugin versions determine cache paths and update detection. You can specify the version in the plugin manifest (`plugin.json`) or in the marketplace entry (`marketplace.json`).

> **Warning:**
>
> When possible, avoid setting the version in both places. The plugin manifest always wins silently, which can cause the marketplace version to be ignored. For relative-path plugins, set the version in the marketplace entry. For all other plugin sources, set it in the plugin manifest.

Set up release channels

To support “stable” and “latest” release channels for your plugins, you can set up two marketplaces that point to different refs or SHAs of the same repo. You can then assign the two marketplaces to different user groups through managed settings.

> **Warning:**
>
> The plugin’s `plugin.json` must declare a different `version` at each pinned ref or commit. If two refs or commits have the same manifest version, Claude Code treats them as identical and skips the update.

Example

```
{
  "name": "stable-tools",
  "plugins": [
    {
      "name": "code-formatter",
      "source": {
        "source": "github",
        "repo": "acme-corp/code-formatter",
        "ref": "stable"
      }
    }
  ]
}
```

```
{
  "name": "latest-tools",
  "plugins": [
    {
      "name": "code-formatter",
      "source": {
        "source": "github",
        "repo": "acme-corp/code-formatter",
        "ref": "latest"
      }
    }
  ]
}
```

Assign channels to user groups

Assign each marketplace to the appropriate user group through managed settings. For example, the stable group receives:

```
{
  "extraKnownMarketplaces": {
    "stable-tools": {
      "source": {
        "source": "github",
        "repo": "acme-corp/stable-tools"
      }
    }
  }
}
```

The early-access group receives `latest-tools` instead:

```
{
  "extraKnownMarketplaces": {
    "latest-tools": {
      "source": {
        "source": "github",
        "repo": "acme-corp/latest-tools"
      }
    }
  }
}
```

Validation and testing

Test your marketplace before sharing.

Validate your marketplace JSON syntax:

```
claude plugin validate .
```

Or from within Claude Code:

```
/plugin validate .
```

Add the marketplace for testing:

```
/plugin marketplace add ./path/to/marketplace
```

Install a test plugin to verify everything works:

```
/plugin install test-plugin@marketplace-name
```

For complete plugin testing workflows, see Test your plugins locally. For technical troubleshooting, see Plugins reference.

Troubleshooting

Marketplace not loading

Symptoms: Can't add marketplace or see plugins from it

Solutions:

- Verify the marketplace URL is accessible
- Check that `.claude-plugin/marketplace.json` exists at the specified path
- Ensure JSON syntax is valid using `claude plugin validate` or `/plugin validate`
- For private repositories, confirm you have access permissions

Marketplace validation errors

Run `claude plugin validate .` or `/plugin validate .` from your marketplace directory to check for issues. Common errors:

Error	Cause	Solution
`File not found: .claude-plugin/marketplace.json`	Missing manifest	Create `.claude-plugin/marketplace.json` with required fields
`Invalid JSON syntax: Unexpected token...`	JSON syntax error	Check for missing commas, extra commas, or unquoted strings
`Duplicate plugin name "x" found in marketplace`	Two plugins share the same name	Give each plugin a unique `name` value
`plugins[0].source: Path contains ".."`	Source path contains `..`	Use paths relative to the marketplace root without `..`. See Relative paths

Warnings (non-blocking):

- `Marketplace has no plugins defined`: add at least one plugin to the `plugins` array
- `No marketplace description provided`: add `metadata.description` to help users understand your marketplace

Plugin installation failures

Symptoms: Marketplace appears but plugin installation fails

Solutions:

- Verify plugin source URLs are accessible
- Check that plugin directories contain required files
- For GitHub sources, ensure repositories are public or you have access
- Test plugin sources manually by cloning/downloading

Private repository authentication fails

Symptoms: Authentication errors when installing plugins from private repositories

Solutions:

For manual installation and updates:

- Verify you're authenticated with your git provider (for example, run `gh auth status` for GitHub)
- Check that your credential helper is configured correctly: `git config --global credential.helper`
- Try cloning the repository manually to verify your credentials work

For background auto-updates:

- Set the appropriate token in your environment: `echo $GITHUB_TOKEN`
- Check that the token has the required permissions (read access to the repository)
- For GitHub, ensure the token has the `repo` scope for private repositories
- For GitLab, ensure the token has at least `read_repository` scope
- Verify the token hasn't expired

Git operations time out

Symptoms: Plugin installation or marketplace updates fail with a timeout error like "Git clone timed out after 120s" or "Git pull timed out after 120s".

Cause: Claude Code uses a 120-second timeout for all git operations, including cloning plugin repositories and pulling marketplace updates. Large repositories or slow network connections may exceed this limit.

Solution: Increase the timeout using the `CLAUDE_CODE_PLUGIN_GIT_TIMEOUT_MS` environment variable. The value is in milliseconds:

```
export CLAUDE_CODE_PLUGIN_GIT_TIMEOUT_MS=300000  # 5 minutes
```

Plugins with relative paths fail in URL-based marketplaces

Symptoms: Added a marketplace via URL (such as `https://example.com/marketplace.json`), but plugins with relative path sources like `"./plugins/my-plugin"` fail to install with "path not found" errors.

Cause: URL-based marketplaces only download the `marketplace.json` file itself. They do not download plugin files from the server. Relative paths in the marketplace entry reference files on the remote server that were not downloaded.

Solutions:

- **Use external sources**: Change plugin entries to use GitHub, npm, or git URL sources instead of relative paths:

```
{ "name": "my-plugin", "source": { "source": "github", "repo": "owner/repo" } }
```

- **Use a Git-based marketplace**: Host your marketplace in a Git repository and add it with the git URL. Git-based marketplaces clone the entire repository, making relative paths work correctly.

Files not found after installation

Symptoms: Plugin installs but references to files fail, especially files outside the plugin directory

Cause: Plugins are copied to a cache directory rather than used in-place. Paths that reference files outside the plugin's directory (such as `../shared-utils`) won't work because those files aren't copied.

Solutions: See Plugin caching and file resolution for workarounds including symlinks and directory restructuring.

For additional debugging tools and common issues, see Debugging and development tools.

See also

- Discover and install prebuilt plugins - Installing plugins from existing marketplaces
- Plugins - Creating your own plugins
- Plugins reference - Complete technical specifications and schemas
- Plugin settings - Plugin configuration options
- strictKnownMarketplaces reference - Managed marketplace restrictions

Discover and install prebuilt plugins through marketplaces

> *Find and install plugins from marketplaces to extend Claude Code with new commands, agents, and capabilities.*

Plugins extend Claude Code with skills, agents, hooks, and MCP servers. Plugin marketplaces are catalogs that help you discover and install these extensions without building them yourself.

Looking to create and distribute your own marketplace? See Create and distribute a plugin marketplace.

How marketplaces work

A marketplace is a catalog of plugins that someone else has created and shared. Using a marketplace is a two-step process:

Step 1: Add the marketplace

This registers the catalog with Claude Code so you can browse what's available. No plugins are installed yet.

Step 2: Install individual plugins

Browse the catalog and install the plugins you want.

Think of it like adding an app store: adding the store gives you access to browse its collection, but you still choose which apps to download individually.

Official Anthropic marketplace

The official Anthropic marketplace (`claude-plugins-official`) is automatically available when you start Claude Code. Run `/plugin` and go to the **Discover** tab to browse what's available.

To install a plugin from the official marketplace:

```
/plugin install plugin-name@claude-plugins-official
```

> **Note:**
>
> The official marketplace is maintained by Anthropic. To submit a plugin to the official marketplace, use one of the in-app submission forms:
>
> - **Claude.ai**: claude.ai/settings/plugins/submit
> - **Console**: platform.claude.com/plugins/submit
>
> To distribute plugins independently, create your own marketplace and share it with users.

The official marketplace includes several categories of plugins:

Code intelligence

Code intelligence plugins enable Claude Code's built-in LSP tool, giving Claude the ability to jump to definitions, find references, and see type errors immediately after edits. These plugins configure Language Server Protocol connections, the same technology that powers VS Code's code intelligence.

These plugins require the language server binary to be installed on your system. If you already have a language server installed, Claude may prompt you to install the corresponding plugin when you open a project.

Language	Plugin	Binary required
C/C++	`clangd-lsp`	`clangd`
C#	`csharp-lsp`	`csharp-ls`
Go	`gopls-lsp`	`gopls`
Java	`jdtls-lsp`	`jdtls`
Kotlin	`kotlin-lsp`	`kotlin-language-server`
Lua	`lua-lsp`	`lua-language-server`
PHP	`php-lsp`	`intelephense`
Python	`pyright-lsp`	`pyright-langserver`
Rust	`rust-analyzer-lsp`	`rust-analyzer`
Swift	`swift-lsp`	`sourcekit-lsp`
TypeScript	`typescript-lsp`	`typescript-language-server`

You can also create your own LSP plugin for other languages.

> **Note:**
>
> If you see `Executable not found in $PATH` in the `/plugin` Errors tab after installing a plugin, install the required binary from the table above.

What Claude gains from code intelligence plugins

Once a code intelligence plugin is installed and its language server binary is available, Claude gains two capabilities:

- **Automatic diagnostics**: after every file edit Claude makes, the language server analyzes the changes and reports errors and warnings back automatically. Claude sees type errors, missing imports, and syntax issues without needing to run a compiler or linter. If Claude introduces an error, it notices and fixes the issue in the same turn. This requires no configuration beyond installing the plugin. You can see diagnostics inline by pressing **Ctrl+O** when the "diagnostics found" indicator appears.
- **Code navigation**: Claude can use the language server to jump to definitions, find references, get type info on hover, list symbols, find implementations, and trace call hierarchies. These operations give Claude more precise navigation than grep-based search, though availability may vary by language and environment.

If you run into issues, see Code intelligence troubleshooting.

External integrations

These plugins bundle pre-configured MCP servers so you can connect Claude to external services without manual setup:

- **Source control**: `github`, `gitlab`
- **Project management**: `atlassian` (Jira/Confluence), `asana`, `linear`, `notion`
- **Design**: `figma`
- **Infrastructure**: `vercel`, `firebase`, `supabase`
- **Communication**: `slack`
- **Monitoring**: `sentry`

Development workflows

Plugins that add commands and agents for common development tasks:

- **commit-commands**: Git commit workflows including commit, push, and PR creation

- **pr-review-toolkit**: Specialized agents for reviewing pull requests
- **agent-sdk-dev**: Tools for building with the Claude Agent SDK
- **plugin-dev**: Toolkit for creating your own plugins

Output styles

Customize how Claude responds:

- **explanatory-output-style**: Educational insights about implementation choices
- **learning-output-style**: Interactive learning mode for skill building

Try it: add the demo marketplace

Anthropic also maintains a demo plugins marketplace (`claude-code-plugins`) with example plugins that show what's possible with the plugin system. Unlike the official marketplace, you need to add this one manually.

Step 1: Add the marketplace

From within Claude Code, run the `plugin marketplace add` command for the `anthropics/claude-code` marketplace:

```
/plugin marketplace add anthropics/claude-code
```

This downloads the marketplace catalog and makes its plugins available to you.

Step 2: Browse available plugins

Run `/plugin` to open the plugin manager. This opens a tabbed interface with four tabs you can cycle through using **Tab** (or **Shift+Tab** to go backward):

- **Discover**: browse available plugins from all your marketplaces
- **Installed**: view and manage your installed plugins
- **Marketplaces**: add, remove, or update your added marketplaces
- **Errors**: view any plugin loading errors

Go to the **Discover** tab to see plugins from the marketplace you just added.

Step 3: Install a plugin

Select a plugin to view its details, then choose an installation scope:

- **User scope**: install for yourself across all projects
- **Project scope**: install for all collaborators on this repository

- **Local scope**: install for yourself in this repository only

For example, select **commit-commands** (a plugin that adds git workflow commands) and install it to your user scope.

You can also install directly from the command line:

```
/plugin install commit-commands@anthropics-claude-code
```

See Configuration scopes to learn more about scopes.

Step 4: Use your new plugin

After installing, run `/reload-plugins` to activate the plugin. Plugin commands are namespaced by the plugin name, so **commit-commands** provides commands like `/commit-commands:commit`.

Try it out by making a change to a file and running:

```
/commit-commands:commit
```

This stages your changes, generates a commit message, and creates the commit.

Each plugin works differently. Check the plugin's description in the **Discover** tab or its homepage to learn what commands and capabilities it provides.

The rest of this guide covers all the ways you can add marketplaces, install plugins, and manage your configuration.

Add marketplaces

Use the `/plugin marketplace add` command to add marketplaces from different sources.

> **Tip:**
>
> **Shortcuts**: You can use `/plugin market` instead of `/plugin marketplace`, and `rm` instead of `remove`.

- **GitHub repositories**: `owner/repo` format (for example, `anthropics/claude-code`)
- **Git URLs**: any git repository URL (GitLab, Bitbucket, self-hosted)
- **Local paths**: directories or direct paths to `marketplace.json` files
- **Remote URLs**: direct URLs to hosted `marketplace.json` files

Add from GitHub

Add a GitHub repository that contains a `.claude-plugin/marketplace.json` file using the `owner/repo` format—where `owner` is the GitHub username or organization and `repo` is the repository name.

For example, `anthropics/claude-code` refers to the `claude-code` repository owned by `anthropics`:

```
/plugin marketplace add anthropics/claude-code
```

Add from other Git hosts

Add any git repository by providing the full URL. This works with any Git host, including GitLab, Bitbucket, and self-hosted servers:

Using HTTPS:

```
/plugin marketplace add https://gitlab.com/company/plugins.git
```

Using SSH:

```
/plugin marketplace add git@gitlab.com:company/plugins.git
```

To add a specific branch or tag, append `#` followed by the ref:

```
/plugin marketplace add https://gitlab.com/company/plugins.git#v1.0.0
```

Add from local paths

Add a local directory that contains a `.claude-plugin/marketplace.json` file:

```
/plugin marketplace add ./my-marketplace
```

You can also add a direct path to a `marketplace.json` file:

```
/plugin marketplace add ./path/to/marketplace.json
```

Add from remote URLs

Add a remote `marketplace.json` file via URL:

```
/plugin marketplace add https://example.com/marketplace.json
```

> **Note:**
>
> URL-based marketplaces have some limitations compared to Git-based marketplaces. If you encounter "path not found" errors when installing plugins, see Troubleshooting.

Install plugins

Once you've added marketplaces, you can install plugins directly (installs to user scope by default):

```
/plugin install plugin-name@marketplace-name
```

To choose a different installation scope, use the interactive UI: run `/plugin`, go to the **Discover** tab, and press **Enter** on a plugin. You'll see options for:

- **User scope** (default): install for yourself across all projects
- **Project scope**: install for all collaborators on this repository (adds to `.claude/settings.json`)
- **Local scope**: install for yourself in this repository only (not shared with collaborators)

You may also see plugins with **managed** scope—these are installed by administrators via managed settings and cannot be modified.

Run `/plugin` and go to the **Installed** tab to see your plugins grouped by scope.

> **Warning:**
>
> Make sure you trust a plugin before installing it. Anthropic does not control what MCP servers, files, or other software are included in plugins and cannot verify that they work as intended. Check each plugin's homepage for more information.

Manage installed plugins

Run `/plugin` and go to the **Installed** tab to view, enable, disable, or uninstall your plugins. Type to filter the list by plugin name or description.

You can also manage plugins with direct commands.

Disable a plugin without uninstalling:

```
/plugin disable plugin-name@marketplace-name
```

Re-enable a disabled plugin:

```
/plugin enable plugin-name@marketplace-name
```

Completely remove a plugin:

```
/plugin uninstall plugin-name@marketplace-name
```

The `--scope` option lets you target a specific scope with CLI commands:

```
claude plugin install formatter@your-org --scope project
claude plugin uninstall formatter@your-org --scope project
```

Apply plugin changes without restarting

When you install, enable, or disable plugins during a session, run `/reload-plugins` to pick up all changes without restarting:

```
/reload-plugins
```

Claude Code reloads all active plugins and shows counts for reloaded commands, skills, agents, hooks, plugin MCP servers, and plugin LSP servers.

Manage marketplaces

You can manage marketplaces through the interactive `/plugin` interface or with CLI commands.

Use the interactive interface

Run `/plugin` and go to the **Marketplaces** tab to:

- View all your added marketplaces with their sources and status
- Add new marketplaces
- Update marketplace listings to fetch the latest plugins
- Remove marketplaces you no longer need

Use CLI commands

You can also manage marketplaces with direct commands.

List all configured marketplaces:

```
/plugin marketplace list
```

Refresh plugin listings from a marketplace:

```
/plugin marketplace update marketplace-name
```

Remove a marketplace:

```
/plugin marketplace remove marketplace-name
```

> **Warning:**
>
> Removing a marketplace will uninstall any plugins you installed from it.

Configure auto-updates

Claude Code can automatically update marketplaces and their installed plugins at startup. When auto-update is enabled for a marketplace, Claude Code refreshes the marketplace data and updates installed plugins to their latest versions. If any plugins were updated, you'll see a notification prompting you to run `/reload-plugins`.

Toggle auto-update for individual marketplaces through the UI:

1. Run `/plugin` to open the plugin manager
2. Select **Marketplaces**
3. Choose a marketplace from the list

4. Select **Enable auto-update** or **Disable auto-update**

Official Anthropic marketplaces have auto-update enabled by default. Third-party and local development marketplaces have auto-update disabled by default.

To disable all automatic updates entirely for both Claude Code and all plugins, set the `DISABLE_AUTOUPDATER` environment variable. See Auto updates for details.

To keep plugin auto-updates enabled while disabling Claude Code auto-updates, set `FORCE_AUTOUPDATE_PLUGINS=true` along with `DISABLE_AUTOUPDATER`:

```
export DISABLE_AUTOUPDATER=true
export FORCE_AUTOUPDATE_PLUGINS=true
```

This is useful when you want to manage Claude Code updates manually but still receive automatic plugin updates.

Configure team marketplaces

Team admins can set up automatic marketplace installation for projects by adding marketplace configuration to `.claude/settings.json`. When team members trust the repository folder, Claude Code prompts them to install these marketplaces and plugins.

Add `extraKnownMarketplaces` to your project's `.claude/settings.json`:

```
{
  "extraKnownMarketplaces": {
    "my-team-tools": {
      "source": {
        "source": "github",
        "repo": "your-org/claude-plugins"
      }
    }
  }
}
```

For full configuration options including `extraKnownMarketplaces` and `enabledPlugins`, see Plugin settings.

Security

Plugins and marketplaces are highly trusted components that can execute arbitrary code on your machine with your user privileges. Only install plugins and add marketplaces from sources you trust. Organizations can restrict which marketplaces users are allowed to add using managed marketplace restrictions.

Troubleshooting

/plugin command not recognized

If you see "unknown command" or the `/plugin` command doesn't appear:

1. **Check your version**: Run `claude --version`. Plugins require version 1.0.33 or later.
2. **Update Claude Code**:
   - **Homebrew**: `brew upgrade claude-code`
   - **npm**: `npm update -g @anthropic-ai/claude-code`
   - **Native installer**: Re-run the install command from Setup
3. **Restart Claude Code**: After updating, restart your terminal and run `claude` again.

Common issues

- **Marketplace not loading**: Verify the URL is accessible and that `.claude-plugin/marketplace.json` exists at the path
- **Plugin installation failures**: Check that plugin source URLs are accessible and repositories are public (or you have access)
- **Files not found after installation**: Plugins are copied to a cache, so paths referencing files outside the plugin directory won't work
- **Plugin skills not appearing**: Clear the cache with `rm -rf ~/.claude/plugins/cache`, restart Claude Code, and reinstall the plugin.

For detailed troubleshooting with solutions, see Troubleshooting in the marketplace guide. For debugging tools, see Debugging and development tools.

Code intelligence issues

- **Language server not starting**: verify the binary is installed and available in your `$PATH`. Check the `/plugin` Errors tab for details.
- **High memory usage**: language servers like `rust-analyzer` and `pyright` can consume significant memory on large projects. If you experience memory issues, disable

the plugin with `/plugin disable <plugin-name>` and rely on Claude's built-in search tools instead.

- **False positive diagnostics in monorepos**: language servers may report unresolved import errors for internal packages if the workspace isn't configured correctly. These don't affect Claude's ability to edit code.

Next steps

- **Build your own plugins**: See Plugins to create skills, agents, and hooks
- **Create a marketplace**: See Create a plugin marketplace to distribute plugins to your team or community
- **Technical reference**: See Plugins reference for complete specifications

Part 6: Advanced & Automation

Create custom subagents

> *Create and use specialized AI subagents in Claude Code for task-specific workflows and improved context management.*

Subagents are specialized AI assistants that handle specific types of tasks. Each subagent runs in its own context window with a custom system prompt, specific tool access, and independent permissions. When Claude encounters a task that matches a subagent's description, it delegates to that subagent, which works independently and returns results.

> **Note:**
>
> If you need multiple agents working in parallel and communicating with each other, see agent teams instead. Subagents work within a single session; agent teams coordinate across separate sessions.

Subagents help you:

- **Preserve context** by keeping exploration and implementation out of your main conversation
- **Enforce constraints** by limiting which tools a subagent can use
- **Reuse configurations** across projects with user-level subagents
- **Specialize behavior** with focused system prompts for specific domains
- **Control costs** by routing tasks to faster, cheaper models like Haiku

Claude uses each subagent's description to decide when to delegate tasks. When you create a subagent, write a clear description so Claude knows when to use it.

Claude Code includes several built-in subagents like **Explore**, **Plan**, and **general-purpose**. You can also create custom subagents to handle specific tasks. This page covers the built-in subagents, how to create your own, full configuration options, patterns for working with subagents, and example subagents.

Built-in subagents

Claude Code includes built-in subagents that Claude automatically uses when appropriate. Each inherits the parent conversation's permissions with additional tool restrictions.

Explore

A fast, read-only agent optimized for searching and analyzing codebases.

- **Model**: Haiku (fast, low-latency)
- **Tools**: Read-only tools (denied access to Write and Edit tools)
- **Purpose**: File discovery, code search, codebase exploration

Claude delegates to Explore when it needs to search or understand a codebase without making changes. This keeps exploration results out of your main conversation context.

When invoking Explore, Claude specifies a thoroughness level: **quick** for targeted lookups, **medium** for balanced exploration, or **very thorough** for comprehensive analysis.

Plan

A research agent used during plan mode to gather context before presenting a plan.

- **Model**: Inherits from main conversation
- **Tools**: Read-only tools (denied access to Write and Edit tools)
- **Purpose**: Codebase research for planning

When you're in plan mode and Claude needs to understand your codebase, it delegates research to the Plan subagent. This prevents infinite nesting (subagents cannot spawn other subagents) while still gathering necessary context.

General-purpose

A capable agent for complex, multi-step tasks that require both exploration and action.

- **Model**: Inherits from main conversation
- **Tools**: All tools
- **Purpose**: Complex research, multi-step operations, code modifications

Claude delegates to general-purpose when the task requires both exploration and modification, complex reasoning to interpret results, or multiple dependent steps.

Other

Claude Code includes additional helper agents for specific tasks. These are typically invoked automatically, so you don't need to use them directly.

Agent	Model	When Claude uses it
Bash	Inherits	Running terminal commands in a separate context
statusline-setup	Sonnet	When you run `/statusline` to configure your status line
Claude Code Guide	Haiku	When you ask questions about Claude Code features

Beyond these built-in subagents, you can create your own with custom prompts, tool restrictions, permission modes, hooks, and skills. The following sections show how to get started and customize subagents.

Quickstart: create your first subagent

Subagents are defined in Markdown files with YAML frontmatter. You can create them manually or use the `/agents` command.

This walkthrough guides you through creating a user-level subagent with the `/agent` command. The subagent reviews code and suggests improvements for the codebase.

Step 1: Open the subagents interface

In Claude Code, run:

```
/agents
```

Step 2: Create a new user-level agent

Select **Create new agent**, then choose **User-level**. This saves the subagent to `~/.claude/agents/` so it's available in all your projects.

Step 3: Generate with Claude

Select **Generate with Claude**. When prompted, describe the subagent:

```
A code improvement agent that scans files and suggests improvements
for readability, performance, and best practices. It should explain
each issue, show the current code, and provide an improved version.
```

Claude generates the system prompt and configuration. Press `e` to open it in your editor if you want to customize it.

Step 4: Select tools

For a read-only reviewer, deselect everything except **Read-only tools**. If you keep all tools selected, the subagent inherits all tools available to the main conversation.

Step 5: Select model

Choose which model the subagent uses. For this example agent, select **Sonnet**, which balances capability and speed for analyzing code patterns.

Step 6: Choose a color

Pick a background color for the subagent. This helps you identify which subagent is running in the UI.

Step 7: Save and try it out

Save the subagent. It's available immediately (no restart needed). Try it:

```
Use the code-improver agent to suggest improvements in this project
```

Claude delegates to your new subagent, which scans the codebase and returns improvement suggestions.

You now have a subagent you can use in any project on your machine to analyze codebases and suggest improvements.

You can also create subagents manually as Markdown files, define them via CLI flags, or distribute them through plugins. The following sections cover all configuration options.

Configure subagents

Use the /agents command

The `/agents` command provides an interactive interface for managing subagents. Run `/agents` to:

- View all available subagents (built-in, user, project, and plugin)
- Create new subagents with guided setup or Claude generation
- Edit existing subagent configuration and tool access
- Delete custom subagents
- See which subagents are active when duplicates exist

This is the recommended way to create and manage subagents. For manual creation or automation, you can also add subagent files directly.

To list all configured subagents from the command line without starting an interactive session, run `claude agents`. This shows agents grouped by source and indicates which are overridden by higher-priority definitions.

Choose the subagent scope

Subagents are Markdown files with YAML frontmatter. Store them in different locations depending on scope. When multiple subagents share the same name, the higher-priority location wins.

Location	Scope	Priority	How to create
`--agents` CLI flag	Current session	1 (highest)	Pass JSON when launching Claude Code
`.claude/agents/`	Current project	2	Interactive or manual
`~/.claude/agents/`	All your projects	3	Interactive or manual
Plugin's `agents/` directory	Where plugin is enabled	4 (lowest)	Installed with plugins

Project subagents (`.claude/agents/`) are ideal for subagents specific to a codebase. Check them into version control so your team can use and improve them collaboratively.

User subagents (`~/.claude/agents/`) are personal subagents available in all your projects.

CLI-defined subagents are passed as JSON when launching Claude Code. They exist only for that session and aren't saved to disk, making them useful for quick testing or automation scripts. You can define multiple subagents in a single `--agents` call:

```
claude --agents '{
  "code-reviewer": {
    "description": "Expert code reviewer. Use proactively after code changes.",
    "prompt": "You are a senior code reviewer. Focus on code quality, security,
and best practices.",
    "tools": ["Read", "Grep", "Glob", "Bash"],
    "model": "sonnet"
  },
  "debugger": {
    "description": "Debugging specialist for errors and test failures.",
    "prompt": "You are an expert debugger. Analyze errors, identify root causes,
and provide fixes."
  }
}'
```

The `--agents` flag accepts JSON with the same frontmatter fields as file-based subagents: `description` , `prompt` , `tools` , `disallowedTools` , `model` , `permissionMode` , `mcpServers` , `hooks` , `maxTurns` , `skills` , and `memory` . Use `prompt` for the system prompt, equivalent to the markdown body in file-based subagents.

Plugin subagents come from plugins you've installed. They appear in `/agents` alongside your custom subagents. See the plugin components reference for details on creating plugin subagents.

> **Note:**
>
> For security reasons, plugin subagents do not support the `hooks` , `mcpServers` , or `permissionMode` frontmatter fields. These fields are ignored when loading agents from a plugin. If you need them, copy the agent file into `.claude/agents/` or `~/.claude/agents/` . You can also add rules to `permissions.allow` in `settings.json` or `settings.local.json` , but these rules apply to the entire session, not just the plugin subagent.

Write subagent files

Subagent files use YAML frontmatter for configuration, followed by the system prompt in Markdown:

> **Note:**
>
> Subagents are loaded at session start. If you create a subagent by manually adding a file, restart your session or use `/agents` to load it immediately.

```
---
name: code-reviewer
description: Reviews code for quality and best practices
tools: Read, Glob, Grep
model: sonnet
---

You are a code reviewer. When invoked, analyze the code and provide
specific, actionable feedback on quality, security, and best practices.
```

The frontmatter defines the subagent's metadata and configuration. The body becomes the system prompt that guides the subagent's behavior. Subagents receive only this system prompt (plus basic environment details like working directory), not the full Claude Code system prompt.

Supported frontmatter fields

The following fields can be used in the YAML frontmatter. Only `name` and `description` are required.

Field	Required	Description
`name`	Yes	Unique identifier using lowercase letters and hyphens
`description`	Yes	When Claude should delegate to this subagent
`tools`	No	Tools the subagent can use. Inherits all tools if omitted

Field	Required	Description
`disallowedTools`	No	Tools to deny, removed from inherited or specified list
`model`	No	Model to use: `sonnet`, `opus`, `haiku`, a full model ID (for example, `claude-opus-4-6`), or `inherit`. Defaults to `inherit`
`permissionMode`	No	Permission mode: `default`, `acceptEdits`, `dontAsk`, `bypassPermissions`, or `plan`
`maxTurns`	No	Maximum number of agentic turns before the subagent stops
`skills`	No	Skills to load into the subagent's context at startup. The full skill content is injected, not just made available for invocation. Subagents don't inherit skills from the parent conversation
`mcpServers`	No	MCP servers available to this subagent. Each entry is either a server name referencing an already-configured server (e.g., `"slack"`) or an inline definition with the server name as key and a full MCP server config as value
`hooks`	No	Lifecycle hooks scoped to this subagent

Field	Required	Description
`memory`	No	Persistent memory scope: `user`, `project`, or `local`. Enables cross-session learning
`background`	No	Set to `true` to always run this subagent as a background task. Default: `false`
`isolation`	No	Set to `worktree` to run the subagent in a temporary git worktree, giving it an isolated copy of the repository. The worktree is automatically cleaned up if the subagent makes no changes

Choose a model

The `model` field controls which AI model the subagent uses:

- **Model alias**: Use one of the available aliases: `sonnet`, `opus`, or `haiku`
- **Full model ID**: Use a full model ID such as `claude-opus-4-6` or `claude-sonnet-4-6`. Accepts the same values as the `--model` flag
- **inherit**: Use the same model as the main conversation
- **Omitted**: If not specified, defaults to `inherit` (uses the same model as the main conversation)

Control subagent capabilities

You can control what subagents can do through tool access, permission modes, and conditional rules.

Available tools

Subagents can use any of Claude Code's internal tools. By default, subagents inherit all tools from the main conversation, including MCP tools.

To restrict tools, use the `tools` field (allowlist) or `disallowedTools` field (denylist):

```
---
name: safe-researcher
description: Research agent with restricted capabilities
tools: Read, Grep, Glob, Bash
disallowedTools: Write, Edit
---
```

Restrict which subagents can be spawned

When an agent runs as the main thread with `claude --agent`, it can spawn subagents using the Agent tool. To restrict which subagent types it can spawn, use `Agent(agent_type)` syntax in the `tools` field.

> **Note:**
>
> In version 2.1.63, the Task tool was renamed to Agent. Existing `Task(...)` references in settings and agent definitions still work as aliases.

```
---
name: coordinator
description: Coordinates work across specialized agents
tools: Agent(worker, researcher), Read, Bash
---
```

This is an allowlist: only the `worker` and `researcher` subagents can be spawned. If the agent tries to spawn any other type, the request fails and the agent sees only the allowed types in its prompt. To block specific agents while allowing all others, use `permissions.deny` instead.

To allow spawning any subagent without restrictions, use `Agent` without parentheses:

```
tools: Agent, Read, Bash
```

If `Agent` is omitted from the `tools` list entirely, the agent cannot spawn any subagents. This restriction only applies to agents running as the main thread with `claude --agent`. Subagents cannot spawn other subagents, so `Agent(agent_type)` has no effect in subagent definitions.

Scope MCP servers to a subagent

Use the `mcpServers` field to give a subagent access to MCP servers that aren't available in the main conversation. Inline servers defined here are connected when the subagent starts and disconnected when it finishes. String references share the parent session's connection.

Each entry in the list is either an inline server definition or a string referencing an MCP server already configured in your session:

```
---
name: browser-tester
description: Tests features in a real browser using Playwright
mcpServers:
  # Inline definition: scoped to this subagent only
  - playwright:
      type: stdio
      command: npx
      args: ["-y", "@playwright/mcp@latest"]
  # Reference by name: reuses an already-configured server
  - github
---

Use the Playwright tools to navigate, screenshot, and interact with pages.
```

Inline definitions use the same schema as `.mcp.json` server entries (`stdio` , `http` , `sse` , `ws`), keyed by the server name.

To keep an MCP server out of the main conversation entirely and avoid its tool descriptions consuming context there, define it inline here rather than in `.mcp.json` . The subagent gets the tools; the parent conversation does not.

Permission modes

The `permissionMode` field controls how the subagent handles permission prompts. Subagents inherit the permission context from the main conversation but can override the mode.

Mode	Behavior
`default`	Standard permission checking with prompts
`acceptEdits`	Auto-accept file edits

Mode	Behavior
`dontAsk`	Auto-deny permission prompts (explicitly allowed tools still work)
`bypassPermissions`	Skip all permission checks
`plan`	Plan mode (read-only exploration)

> **Warning:**
>
> Use `bypassPermissions` with caution. It skips all permission checks, allowing the subagent to execute any operation without approval.

If the parent uses `bypassPermissions`, this takes precedence and cannot be overridden.

Preload skills into subagents

Use the `skills` field to inject skill content into a subagent's context at startup. This gives the subagent domain knowledge without requiring it to discover and load skills during execution.

```
---
name: api-developer
description: Implement API endpoints following team conventions
skills:
  - api-conventions
  - error-handling-patterns
---

Implement API endpoints. Follow the conventions and patterns from the preloaded
skills.
```

The full content of each skill is injected into the subagent's context, not just made available for invocation. Subagents don't inherit skills from the parent conversation; you must list them explicitly.

> **Note:**

> This is the inverse of running a skill in a subagent. With `skills` in a subagent, the subagent controls the system prompt and loads skill content. With `context: fork` in a skill, the skill content is injected into the agent you specify. Both use the same underlying system.

Enable persistent memory

The `memory` field gives the subagent a persistent directory that survives across conversations. The subagent uses this directory to build up knowledge over time, such as codebase patterns, debugging insights, and architectural decisions.

```
---
name: code-reviewer
description: Reviews code for quality and best practices
memory: user
---

You are a code reviewer. As you review code, update your agent memory with
patterns, conventions, and recurring issues you discover.
```

Choose a scope based on how broadly the memory should apply:

Scope	Location	Use when
`user`	`~/.claude/agent-memory/<name-of-agent>/`	the subagent should remember learnings across all projects
`project`	`.claude/agent-memory/<name-of-agent>/`	the subagent's knowledge is project-specific and shareable via version control
`local`	`.claude/agent-memory-local/<name-of-agent>/`	the subagent's knowledge is project-specific but should not be checked into version control

When memory is enabled:

- The subagent's system prompt includes instructions for reading and writing to the memory directory.

- The subagent's system prompt also includes the first 200 lines of `MEMORY.md` in the memory directory, with instructions to curate `MEMORY.md` if it exceeds 200 lines.
- Read, Write, and Edit tools are automatically enabled so the subagent can manage its memory files.

Persistent memory tips

- `user` is the recommended default scope. Use `project` or `local` when the subagent's knowledge is only relevant to a specific codebase.
- Ask the subagent to consult its memory before starting work: "Review this PR, and check your memory for patterns you've seen before."
- Ask the subagent to update its memory after completing a task: "Now that you're done, save what you learned to your memory." Over time, this builds a knowledge base that makes the subagent more effective.
- Include memory instructions directly in the subagent's markdown file so it proactively maintains its own knowledge base:

```
Update your agent memory as you discover codepaths, patterns, library
locations, and key architectural decisions. This builds up institutional
knowledge across conversations. Write concise notes about what you found
and where.
```

Conditional rules with hooks

For more dynamic control over tool usage, use `PreToolUse` hooks to validate operations before they execute. This is useful when you need to allow some operations of a tool while blocking others.

This example creates a subagent that only allows read-only database queries. The `PreToolUse` hook runs the script specified in `command` before each Bash command executes:

```
---
name: db-reader
description: Execute read-only database queries
tools: Bash
hooks:
  PreToolUse:
    - matcher: "Bash"
      hooks:
        - type: command
          command: "./scripts/validate-readonly-query.sh"
---
```

Claude Code passes hook input as JSON via stdin to hook commands. The validation script reads this JSON, extracts the Bash command, and exits with code 2 to block write operations:

```
#!/bin/bash
## ./scripts/validate-readonly-query.sh

INPUT=$(cat)
COMMAND=$(echo "$INPUT" | jq -r '.tool_input.command // empty')

## Block SQL write operations (case-insensitive)
if echo "$COMMAND" | grep -iE '\b(INSERT|UPDATE|DELETE|DROP|CREATE|ALTER|TRUNCATE)
\b' > /dev/null; then
  echo "Blocked: Only SELECT queries are allowed" >&2
  exit 2
fi

exit 0
```

See Hook input for the complete input schema and exit codes for how exit codes affect behavior.

Disable specific subagents

You can prevent Claude from using specific subagents by adding them to the `deny` array in your settings. Use the format `Agent(subagent-name)` where `subagent-name` matches the subagent's name field.

```
{
  "permissions": {
    "deny": ["Agent(Explore)", "Agent(my-custom-agent)"]
  }
}
```

This works for both built-in and custom subagents. You can also use the `--disallowedTools` CLI flag:

```
claude --disallowedTools "Agent(Explore)"
```

See Permissions documentation for more details on permission rules.

Define hooks for subagents

Subagents can define hooks that run during the subagent's lifecycle. There are two ways to configure hooks:

1. **In the subagent's frontmatter**: Define hooks that run only while that subagent is active
2. **In** `settings.json` : Define hooks that run in the main session when subagents start or stop

Hooks in subagent frontmatter

Define hooks directly in the subagent's markdown file. These hooks only run while that specific subagent is active and are cleaned up when it finishes.

All hook events are supported. The most common events for subagents are:

Event	Matcher input	When it fires
`PreToolUse`	Tool name	Before the subagent uses a tool
`PostToolUse`	Tool name	After the subagent uses a tool

Event	Matcher input	When it fires
`Stop`	(none)	When the subagent finishes (converted to `SubagentStop` at runtime)

This example validates Bash commands with the `PreToolUse` hook and runs a linter after file edits with `PostToolUse`:

```
---
name: code-reviewer
description: Review code changes with automatic linting
hooks:
  PreToolUse:
    - matcher: "Bash"
      hooks:
        - type: command
          command: "./scripts/validate-command.sh $TOOL_INPUT"
  PostToolUse:
    - matcher: "Edit|Write"
      hooks:
        - type: command
          command: "./scripts/run-linter.sh"
---
```

`Stop` hooks in frontmatter are automatically converted to `SubagentStop` events.

Project-level hooks for subagent events

Configure hooks in `settings.json` that respond to subagent lifecycle events in the main session.

Event	Matcher input	When it fires
`SubagentStart`	Agent type name	When a subagent begins execution
`SubagentStop`	Agent type name	When a subagent completes

Both events support matchers to target specific agent types by name. This example runs a setup script only when the `db-agent` subagent starts, and a cleanup script when any subagent stops:

```
{
  "hooks": {
    "SubagentStart": [
      {
        "matcher": "db-agent",
        "hooks": [
          { "type": "command", "command": "./scripts/setup-db-connection.sh" }
        ]
      }
    ],
    "SubagentStop": [
      {
        "hooks": [
          { "type": "command", "command": "./scripts/cleanup-db-connection.sh" }
        ]
      }
    ]
  }
}
```

See Hooks for the complete hook configuration format.

Work with subagents

Understand automatic delegation

Claude automatically delegates tasks based on the task description in your request, the `description` field in subagent configurations, and current context. To encourage proactive delegation, include phrases like “use proactively” in your subagent’s description field.

You can also request a specific subagent explicitly:

```
Use the test-runner subagent to fix failing tests
Have the code-reviewer subagent look at my recent changes
```

Run subagents in foreground or background

Subagents can run in the foreground (blocking) or background (concurrent):

- **Foreground subagents** block the main conversation until complete. Permission prompts and clarifying questions (like `AskUserQuestion`) are passed through to you.
- **Background subagents** run concurrently while you continue working. Before launching, Claude Code prompts for any tool permissions the subagent will need, ensuring it has the necessary approvals upfront. Once running, the subagent inherits these permissions and auto-denies anything not pre-approved. If a background subagent needs to ask clarifying questions, that tool call fails but the subagent continues.

If a background subagent fails due to missing permissions, you can resume it in the foreground to retry with interactive prompts.

Claude decides whether to run subagents in the foreground or background based on the task. You can also:

- Ask Claude to "run this in the background"
- Press **Ctrl+B** to background a running task

To disable all background task functionality, set the `CLAUDE_CODE_DISABLE_BACKGROUND_TASKS` environment variable to `1`. See Environment variables.

Common patterns

Isolate high-volume operations

One of the most effective uses for subagents is isolating operations that produce large amounts of output. Running tests, fetching documentation, or processing log files can consume significant context. By delegating these to a subagent, the verbose output stays in the subagent's context while only the relevant summary returns to your main conversation.

```
Use a subagent to run the test suite and report only the failing tests with their
error messages
```

Run parallel research

For independent investigations, spawn multiple subagents to work simultaneously:

```
Research the authentication, database, and API modules in parallel using separate
subagents
```

Each subagent explores its area independently, then Claude synthesizes the findings. This works best when the research paths don't depend on each other.

> **Warning:**
>
> When subagents complete, their results return to your main conversation. Running many subagents that each return detailed results can consume significant context.

For tasks that need sustained parallelism or exceed your context window, agent teams give each worker its own independent context.

Chain subagents

For multi-step workflows, ask Claude to use subagents in sequence. Each subagent completes its task and returns results to Claude, which then passes relevant context to the next subagent.

```
Use the code-reviewer subagent to find performance issues, then use the optimizer
subagent to fix them
```

Choose between subagents and main conversation

Use the **main conversation** when:

- The task needs frequent back-and-forth or iterative refinement
- Multiple phases share significant context (planning → implementation → testing)
- You're making a quick, targeted change
- Latency matters. Subagents start fresh and may need time to gather context

Use **subagents** when:

- The task produces verbose output you don't need in your main context
- You want to enforce specific tool restrictions or permissions
- The work is self-contained and can return a summary

Consider Skills instead when you want reusable prompts or workflows that run in the main conversation context rather than isolated subagent context.

For a quick question about something already in your conversation, use `/btw` instead of a subagent. It sees your full context but has no tool access, and the answer is discarded rather than added to history.

> **Note:**
>
> Subagents cannot spawn other subagents. If your workflow requires nested delegation, use Skills or chain subagents from the main conversation.

Manage subagent context

Resume subagents

Each subagent invocation creates a new instance with fresh context. To continue an existing subagent's work instead of starting over, ask Claude to resume it.

Resumed subagents retain their full conversation history, including all previous tool calls, results, and reasoning. The subagent picks up exactly where it stopped rather than starting fresh.

When a subagent completes, Claude receives its agent ID. To resume a subagent, ask Claude to continue the previous work:

```
Use the code-reviewer subagent to review the authentication module
[Agent completes]

Continue that code review and now analyze the authorization logic
[Claude resumes the subagent with full context from previous conversation]
```

You can also ask Claude for the agent ID if you want to reference it explicitly, or find IDs in the transcript files at `~/.claude/projects/{project}/{sessionId}/subagents/`. Each transcript is stored as `agent-{agentId}.jsonl`.

Subagent transcripts persist independently of the main conversation:

- **Main conversation compaction**: When the main conversation compacts, subagent transcripts are unaffected. They're stored in separate files.
- **Session persistence**: Subagent transcripts persist within their session. You can resume a subagent after restarting Claude Code by resuming the same session.
- **Automatic cleanup**: Transcripts are cleaned up based on the `cleanupPeriodDays` setting (default: 30 days).

Auto-compaction

Subagents support automatic compaction using the same logic as the main conversation. By default, auto-compaction triggers at approximately 95% capacity. To trigger compaction earlier, set `CLAUDE_AUTOCOMPACT_PCT_OVERRIDE` to a lower percentage (for example, `50`). See environment variables for details.

Compaction events are logged in subagent transcript files:

```
{
  "type": "system",
  "subtype": "compact_boundary",
  "compactMetadata": {
    "trigger": "auto",
    "preTokens": 167189
  }
}
```

The `preTokens` value shows how many tokens were used before compaction occurred.

Example subagents

These examples demonstrate effective patterns for building subagents. Use them as starting points, or generate a customized version with Claude.

> **Tip:**
>
> **Best practices:**
>
> - **Design focused subagents:** each subagent should excel at one specific task
> - **Write detailed descriptions:** Claude uses the description to decide when to delegate
> - **Limit tool access:** grant only necessary permissions for security and focus
> - **Check into version control:** share project subagents with your team

Code reviewer

A read-only subagent that reviews code without modifying it. This example shows how to design a focused subagent with limited tool access (no Edit or Write) and a detailed prompt that specifies exactly what to look for and how to format output.

```
---
name: code-reviewer
description: Expert code review specialist. Proactively reviews code for quality,
security, and maintainability. Use immediately after writing or modifying code.
tools: Read, Grep, Glob, Bash
model: inherit
---

You are a senior code reviewer ensuring high standards of code quality and
security.

When invoked:
1. Run git diff to see recent changes
2. Focus on modified files
3. Begin review immediately

Review checklist:
- Code is clear and readable
- Functions and variables are well-named
- No duplicated code
- Proper error handling
- No exposed secrets or API keys
- Input validation implemented
- Good test coverage
- Performance considerations addressed

Provide feedback organized by priority:
- Critical issues (must fix)
- Warnings (should fix)
- Suggestions (consider improving)

Include specific examples of how to fix issues.
```

Debugger

A subagent that can both analyze and fix issues. Unlike the code reviewer, this one includes Edit because fixing bugs requires modifying code. The prompt provides a clear workflow from diagnosis to verification.

```
---
name: debugger
description: Debugging specialist for errors, test failures, and unexpected
behavior. Use proactively when encountering any issues.
tools: Read, Edit, Bash, Grep, Glob
---

You are an expert debugger specializing in root cause analysis.

When invoked:
1. Capture error message and stack trace
2. Identify reproduction steps
3. Isolate the failure location
4. Implement minimal fix
5. Verify solution works

Debugging process:
- Analyze error messages and logs
- Check recent code changes
- Form and test hypotheses
- Add strategic debug logging
- Inspect variable states

For each issue, provide:
- Root cause explanation
- Evidence supporting the diagnosis
- Specific code fix
- Testing approach
- Prevention recommendations

Focus on fixing the underlying issue, not the symptoms.
```

Data scientist

A domain-specific subagent for data analysis work. This example shows how to create subagents for specialized workflows outside of typical coding tasks. It explicitly sets `model: sonnet` for more capable analysis.

```
---
name: data-scientist
description: Data analysis expert for SQL queries, BigQuery operations, and data
insights. Use proactively for data analysis tasks and queries.
tools: Bash, Read, Write
model: sonnet
---

You are a data scientist specializing in SQL and BigQuery analysis.

When invoked:
1. Understand the data analysis requirement
2. Write efficient SQL queries
3. Use BigQuery command line tools (bq) when appropriate
4. Analyze and summarize results
5. Present findings clearly

Key practices:
- Write optimized SQL queries with proper filters
- Use appropriate aggregations and joins
- Include comments explaining complex logic
- Format results for readability
- Provide data-driven recommendations

For each analysis:
- Explain the query approach
- Document any assumptions
- Highlight key findings
- Suggest next steps based on data

Always ensure queries are efficient and cost-effective.
```

Database query validator

A subagent that allows Bash access but validates commands to permit only read-only SQL queries. This example shows how to use `PreToolUse` hooks for conditional validation when you need finer control than the `tools` field provides.

```
---
name: db-reader
description: Execute read-only database queries. Use when analyzing data or
generating reports.
tools: Bash
hooks:
  PreToolUse:
    - matcher: "Bash"
      hooks:
        - type: command
          command: "./scripts/validate-readonly-query.sh"
---

You are a database analyst with read-only access. Execute SELECT queries to answer
questions about the data.

When asked to analyze data:
1. Identify which tables contain the relevant data
2. Write efficient SELECT queries with appropriate filters
3. Present results clearly with context

You cannot modify data. If asked to INSERT, UPDATE, DELETE, or modify schema,
explain that you only have read access.
```

Claude Code passes hook input as JSON via stdin to hook commands. The validation script reads this JSON, extracts the command being executed, and checks it against a list of SQL write operations. If a write operation is detected, the script exits with code 2 to block execution and returns an error message to Claude via stderr.

Create the validation script anywhere in your project. The path must match the `command` field in your hook configuration:

```
#!/bin/bash
## Blocks SQL write operations, allows SELECT queries

## Read JSON input from stdin
INPUT=$(cat)

## Extract the command field from tool_input using jq
COMMAND=$(echo "$INPUT" | jq -r '.tool_input.command // empty')

if [ -z "$COMMAND" ]; then
  exit 0
fi

## Block write operations (case-insensitive)
if echo "$COMMAND" | grep -iE '\b(INSERT|UPDATE|DELETE|DROP|CREATE|ALTER|TRUNCATE|
REPLACE|MERGE)\b' > /dev/null; then
  echo "Blocked: Write operations not allowed. Use SELECT queries only." >&2
  exit 2
fi

exit 0
```

Make the script executable:

```
chmod +x ./scripts/validate-readonly-query.sh
```

The hook receives JSON via stdin with the Bash command in `tool_input.command`. Exit code 2 blocks the operation and feeds the error message back to Claude. See Hooks for details on exit codes and Hook input for the complete input schema.

Next steps

Now that you understand subagents, explore these related features:

- Distribute subagents with plugins to share subagents across teams or projects
- Run Claude Code programmatically with the Agent SDK for CI/CD and automation
- Use MCP servers to give subagents access to external tools and data

Orchestrate teams of Claude Code sessions

> *Coordinate multiple Claude Code instances working together as a team, with shared tasks, inter-agent messaging, and centralized management.*

Warning:

Agent teams are experimental and disabled by default. Enable them by adding `CLAUDE_CODE_EXPERIMENTAL_AGENT_TEAMS` to your settings.json or environment. Agent teams have known limitations around session resumption, task coordination, and shutdown behavior.

Agent teams let you coordinate multiple Claude Code instances working together. One session acts as the team lead, coordinating work, assigning tasks, and synthesizing results. Teammates work independently, each in its own context window, and communicate directly with each other.

Unlike subagents, which run within a single session and can only report back to the main agent, you can also interact with individual teammates directly without going through the lead.

Note:

Agent teams require Claude Code v2.1.32 or later. Check your version with `claude --version`.

This page covers:

- When to use agent teams, including best use cases and how they compare with subagents
- Starting a team
- Controlling teammates, including display modes, task assignment, and delegation
- Best practices for parallel work

When to use agent teams

Agent teams are most effective for tasks where parallel exploration adds real value. See use case examples for full scenarios. The strongest use cases are:

- **Research and review**: multiple teammates can investigate different aspects of a problem simultaneously, then share and challenge each other's findings
- **New modules or features**: teammates can each own a separate piece without stepping on each other
- **Debugging with competing hypotheses**: teammates test different theories in parallel and converge on the answer faster
- **Cross-layer coordination**: changes that span frontend, backend, and tests, each owned by a different teammate

Agent teams add coordination overhead and use significantly more tokens than a single session. They work best when teammates can operate independently. For sequential tasks, same-file edits, or work with many dependencies, a single session or subagents are more effective.

Compare with subagents

Both agent teams and subagents let you parallelize work, but they operate differently. Choose based on whether your workers need to communicate with each other:

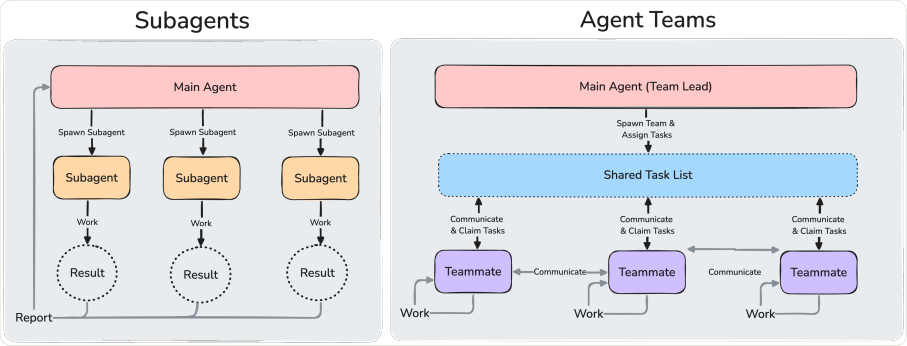


Diagram comparing subagent and agent team architectures. Subagents are spawned by the main agent, do work, and report results back. Agent teams coordinate through a shared task list, with teammates communicating directly with each other.

	Subagents	Agent teams
Context	Own context window; results return to the caller	Own context window; fully independent

	Subagents	Agent teams
Communication	Report results back to the main agent only	Teammates message each other directly
Coordination	Main agent manages all work	Shared task list with self-coordination
Best for	Focused tasks where only the result matters	Complex work requiring discussion and collaboration
Token cost	Lower: results summarized back to main context	Higher: each teammate is a separate Claude instance

Use subagents when you need quick, focused workers that report back. Use agent teams when teammates need to share findings, challenge each other, and coordinate on their own.

Enable agent teams

Agent teams are disabled by default. Enable them by setting the `CLAUDE_CODE_EXPERIMENTAL_AGENT_TEAMS` environment variable to `1`, either in your shell environment or through settings.json:

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

Start your first agent team

After enabling agent teams, tell Claude to create an agent team and describe the task and the team structure you want in natural language. Claude creates the team, spawns teammates, and coordinates work based on your prompt.

This example works well because the three roles are independent and can explore the problem without waiting on each other:

```
I'm designing a CLI tool that helps developers track TODO comments across
their codebase. Create an agent team to explore this from different angles: one
teammate on UX, one on technical architecture, one playing devil's advocate.
```

From there, Claude creates a team with a shared task list, spawns teammates for each perspective, has them explore the problem, synthesizes findings, and attempts to clean up the team when finished.

The lead's terminal lists all teammates and what they're working on. Use Shift+Down to cycle through teammates and message them directly. After the last teammate, Shift+Down wraps back to the lead.

If you want each teammate in its own split pane, see Choose a display mode.

Control your agent team

Tell the lead what you want in natural language. It handles team coordination, task assignment, and delegation based on your instructions.

Choose a display mode

Agent teams support two display modes:

- **In-process**: all teammates run inside your main terminal. Use Shift+Down to cycle through teammates and type to message them directly. Works in any terminal, no extra setup required.
- **Split panes**: each teammate gets its own pane. You can see everyone's output at once and click into a pane to interact directly. Requires tmux, or iTerm2.

> **Note:**
>
> `tmux` has known limitations on certain operating systems and traditionally works best on macOS. Using `tmux -CC` in iTerm2 is the suggested entrypoint into `tmux`.

The default is `"auto"`, which uses split panes if you're already running inside a tmux session, and in-process otherwise. The `"tmux"` setting enables split-pane mode and auto-detects whether to use tmux or iTerm2 based on your terminal. To override, set `teammateMode` in your settings.json:

```
{
  "teammateMode": "in-process"
}
```

To force in-process mode for a single session, pass it as a flag:

```
claude --teammate-mode in-process
```

Split-pane mode requires either tmux or iTerm2 with the `it2` CLI. To install manually:

- **tmux**: install through your system's package manager. See the tmux wiki for platform-specific instructions.
- **iTerm2**: install the `it2` CLI, then enable the Python API in **iTerm2 → Settings → General → Magic → Enable Python API**.

Specify teammates and models

Claude decides the number of teammates to spawn based on your task, or you can specify exactly what you want:

```
Create a team with 4 teammates to refactor these modules in parallel.
Use Sonnet for each teammate.
```

Require plan approval for teammates

For complex or risky tasks, you can require teammates to plan before implementing. The teammate works in read-only plan mode until the lead approves their approach:

```
Spawn an architect teammate to refactor the authentication module.
Require plan approval before they make any changes.
```

When a teammate finishes planning, it sends a plan approval request to the lead. The lead reviews the plan and either approves it or rejects it with feedback. If rejected, the teammate stays in plan mode, revises based on the feedback, and resubmits. Once approved, the teammate exits plan mode and begins implementation.

The lead makes approval decisions autonomously. To influence the lead's judgment, give it criteria in your prompt, such as "only approve plans that include test coverage" or "reject plans that modify the database schema."

Talk to teammates directly

Each teammate is a full, independent Claude Code session. You can message any teammate directly to give additional instructions, ask follow-up questions, or redirect their approach.

- **In-process mode**: use Shift+Down to cycle through teammates, then type to send them a message. Press Enter to view a teammate's session, then Escape to interrupt their current turn. Press Ctrl+T to toggle the task list.
- **Split-pane mode**: click into a teammate's pane to interact with their session directly. Each teammate has a full view of their own terminal.

Assign and claim tasks

The shared task list coordinates work across the team. The lead creates tasks and teammates work through them. Tasks have three states: pending, in progress, and completed. Tasks can also depend on other tasks: a pending task with unresolved dependencies cannot be claimed until those dependencies are completed.

The lead can assign tasks explicitly, or teammates can self-claim:

- **Lead assigns**: tell the lead which task to give to which teammate
- **Self-claim**: after finishing a task, a teammate picks up the next unassigned, unblocked task on its own

Task claiming uses file locking to prevent race conditions when multiple teammates try to claim the same task simultaneously.

Shut down teammates

To gracefully end a teammate's session:

```
Ask the researcher teammate to shut down
```

The lead sends a shutdown request. The teammate can approve, exiting gracefully, or reject with an explanation.

Clean up the team

When you're done, ask the lead to clean up:

```
Clean up the team
```

This removes the shared team resources. When the lead runs cleanup, it checks for active teammates and fails if any are still running, so shut them down first.

> **Warning:**
>
> Always use the lead to clean up. Teammates should not run cleanup because their team context may not resolve correctly, potentially leaving resources in an inconsistent state.

Enforce quality gates with hooks

Use hooks to enforce rules when teammates finish work or tasks complete:

- `TeammateIdle` : runs when a teammate is about to go idle. Exit with code 2 to send feedback and keep the teammate working.
- `TaskCompleted` : runs when a task is being marked complete. Exit with code 2 to prevent completion and send feedback.

How agent teams work

This section covers the architecture and mechanics behind agent teams. If you want to start using them, see Control your agent team above.

How Claude starts agent teams

There are two ways agent teams get started:

- **You request a team**: give Claude a task that benefits from parallel work and explicitly ask for an agent team. Claude creates one based on your instructions.
- **Claude proposes a team**: if Claude determines your task would benefit from parallel work, it may suggest creating a team. You confirm before it proceeds.

In both cases, you stay in control. Claude won't create a team without your approval.

Architecture

An agent team consists of:

Component	Role
Team lead	The main Claude Code session that creates the team, spawns teammates, and coordinates work

Component	Role
Teammates	Separate Claude Code instances that each work on assigned tasks
Task list	Shared list of work items that teammates claim and complete
Mailbox	Messaging system for communication between agents

See Choose a display mode for display configuration options. Teammate messages arrive at the lead automatically.

The system manages task dependencies automatically. When a teammate completes a task that other tasks depend on, blocked tasks unblock without manual intervention.

Teams and tasks are stored locally:

- **Team config**: `~/.claude/teams/{team-name}/config.json`
- **Task list**: `~/.claude/tasks/{team-name}/`

The team config contains a `members` array with each teammate's name, agent ID, and agent type. Teammates can read this file to discover other team members.

Permissions

Teammates start with the lead's permission settings. If the lead runs with `--dangerously-skip-permissions`, all teammates do too. After spawning, you can change individual teammate modes, but you can't set per-teammate modes at spawn time.

Context and communication

Each teammate has its own context window. When spawned, a teammate loads the same project context as a regular session: CLAUDE.md, MCP servers, and skills. It also receives the spawn prompt from the lead. The lead's conversation history does not carry over.

How teammates share information:

- **Automatic message delivery**: when teammates send messages, they're delivered automatically to recipients. The lead doesn't need to poll for updates.
- **Idle notifications**: when a teammate finishes and stops, they automatically notify the lead.

- **Shared task list**: all agents can see task status and claim available work.

Teammate messaging:

- **message**: send a message to one specific teammate
- **broadcast**: send to all teammates simultaneously. Use sparingly, as costs scale with team size.

Token usage

Agent teams use significantly more tokens than a single session. Each teammate has its own context window, and token usage scales with the number of active teammates. For research, review, and new feature work, the extra tokens are usually worthwhile. For routine tasks, a single session is more cost-effective. See agent team token costs for usage guidance.

Use case examples

These examples show how agent teams handle tasks where parallel exploration adds value.

Run a parallel code review

A single reviewer tends to gravitate toward one type of issue at a time. Splitting review criteria into independent domains means security, performance, and test coverage all get thorough attention simultaneously. The prompt assigns each teammate a distinct lens so they don't overlap:

```
Create an agent team to review PR #142. Spawn three reviewers:
- One focused on security implications
- One checking performance impact
- One validating test coverage
Have them each review and report findings.
```

Each reviewer works from the same PR but applies a different filter. The lead synthesizes findings across all three after they finish.

Investigate with competing hypotheses

When the root cause is unclear, a single agent tends to find one plausible explanation and stop looking. The prompt fights this by making teammates explicitly adversarial: each one's job is not only to investigate its own theory but to challenge the others'.

```
Users report the app exits after one message instead of staying connected.
Spawn 5 agent teammates to investigate different hypotheses. Have them talk to
each other to try to disprove each other's theories, like a scientific
debate. Update the findings doc with whatever consensus emerges.
```

The debate structure is the key mechanism here. Sequential investigation suffers from anchoring: once one theory is explored, subsequent investigation is biased toward it.

With multiple independent investigators actively trying to disprove each other, the theory that survives is much more likely to be the actual root cause.

Best practices

Give teammates enough context

Teammates load project context automatically, including CLAUDE.md, MCP servers, and skills, but they don't inherit the lead's conversation history. See Context and communication for details. Include task-specific details in the spawn prompt:

```
Spawn a security reviewer teammate with the prompt: "Review the authentication
module
at src/auth/ for security vulnerabilities. Focus on token handling, session
management, and input validation. The app uses JWT tokens stored in
httpOnly cookies. Report any issues with severity ratings."
```

Choose an appropriate team size

There's no hard limit on the number of teammates, but practical constraints apply:

- **Token costs scale linearly**: each teammate has its own context window and consumes tokens independently. See agent team token costs for details.
- **Coordination overhead increases**: more teammates means more communication, task coordination, and potential for conflicts
- **Diminishing returns**: beyond a certain point, additional teammates don't speed up work proportionally

Start with 3-5 teammates for most workflows. This balances parallel work with manageable coordination. The examples in this guide use 3-5 teammates because that range works well across different task types.

Having 5-6 tasks per teammate keeps everyone productive without excessive context switching. If you have 15 independent tasks, 3 teammates is a good starting point.

Scale up only when the work genuinely benefits from having teammates work simultaneously. Three focused teammates often outperform five scattered ones.

Size tasks appropriately

- **Too small**: coordination overhead exceeds the benefit
- **Too large**: teammates work too long without check-ins, increasing risk of wasted effort
- **Just right**: self-contained units that produce a clear deliverable, such as a function, a test file, or a review

> **Tip:**
>
> The lead breaks work into tasks and assigns them to teammates automatically. If it isn't creating enough tasks, ask it to split the work into smaller pieces. Having 5-6 tasks per teammate keeps everyone productive and lets the lead reassign work if someone gets stuck.

Wait for teammates to finish

Sometimes the lead starts implementing tasks itself instead of waiting for teammates. If you notice this:

```
Wait for your teammates to complete their tasks before proceeding
```

Start with research and review

If you're new to agent teams, start with tasks that have clear boundaries and don't require writing code: reviewing a PR, researching a library, or investigating a bug. These tasks show the value of parallel exploration without the coordination challenges that come with parallel implementation.

Avoid file conflicts

Two teammates editing the same file leads to overwrites. Break the work so each teammate owns a different set of files.

Monitor and steer

Check in on teammates' progress, redirect approaches that aren't working, and synthesize findings as they come in. Letting a team run unattended for too long increases the risk of wasted effort.

Troubleshooting

Teammates not appearing

If teammates aren't appearing after you ask Claude to create a team:

- In in-process mode, teammates may already be running but not visible. Press Shift+Down to cycle through active teammates.
- Check that the task you gave Claude was complex enough to warrant a team. Claude decides whether to spawn teammates based on the task.
- If you explicitly requested split panes, ensure tmux is installed and available in your PATH:

```
which tmux
```

- For iTerm2, verify the `it2` CLI is installed and the Python API is enabled in iTerm2 preferences.

Too many permission prompts

Teammate permission requests bubble up to the lead, which can create friction. Pre-approve common operations in your permission settings before spawning teammates to reduce interruptions.

Teammates stopping on errors

Teammates may stop after encountering errors instead of recovering. Check their output using Shift+Down in in-process mode or by clicking the pane in split mode, then either:

- Give them additional instructions directly
- Spawn a replacement teammate to continue the work

Lead shuts down before work is done

The lead may decide the team is finished before all tasks are actually complete. If this happens, tell it to keep going. You can also tell the lead to wait for teammates to finish before proceeding if it starts doing work instead of delegating.

Orphaned tmux sessions

If a tmux session persists after the team ends, it may not have been fully cleaned up. List sessions and kill the one created by the team:

```
tmux ls
tmux kill-session -t <session-name>
```

Limitations

Agent teams are experimental. Current limitations to be aware of:

- **No session resumption with in-process teammates**: `/resume` and `/rewind` do not restore in-process teammates. After resuming a session, the lead may attempt to message teammates that no longer exist. If this happens, tell the lead to spawn new teammates.
- **Task status can lag**: teammates sometimes fail to mark tasks as completed, which blocks dependent tasks. If a task appears stuck, check whether the work is actually done and update the task status manually or tell the lead to nudge the teammate.
- **Shutdown can be slow**: teammates finish their current request or tool call before shutting down, which can take time.
- **One team per session**: a lead can only manage one team at a time. Clean up the current team before starting a new one.
- **No nested teams**: teammates cannot spawn their own teams or teammates. Only the lead can manage the team.
- **Lead is fixed**: the session that creates the team is the lead for its lifetime. You can't promote a teammate to lead or transfer leadership.
- **Permissions set at spawn**: all teammates start with the lead's permission mode. You can change individual teammate modes after spawning, but you can't set per-teammate modes at spawn time.
- **Split panes require tmux or iTerm2**: the default in-process mode works in any terminal. Split-pane mode isn't supported in VS Code's integrated terminal, Windows Terminal, or Ghostty.

> **Tip:**
>
> `CLAUDE.md` **works normally**: teammates read `CLAUDE.md` files from their working directory. Use this to provide project-specific guidance to all teammates.

Next steps

Explore related approaches for parallel work and delegation:

- **Lightweight delegation**: subagents spawn helper agents for research or verification within your session, better for tasks that don't need inter-agent coordination
- **Manual parallel sessions**: Git worktrees let you run multiple Claude Code sessions yourself without automated team coordination
- **Compare approaches**: see the subagent vs agent team comparison for a side-by-side breakdown

Run Claude Code programmatically

> *Use the Agent SDK to run Claude Code programmatically from the CLI, Python, or TypeScript.*

The Agent SDK gives you the same tools, agent loop, and context management that power Claude Code. It's available as a CLI for scripts and CI/CD, or as Python and TypeScript packages for full programmatic control.

> **Note:**
>
> The CLI was previously called "headless mode." The `-p` flag and all CLI options work the same way.

To run Claude Code programmatically from the CLI, pass `-p` with your prompt and any CLI options:

```
claude -p "Find and fix the bug in auth.py" --allowedTools "Read,Edit,Bash"
```

This page covers using the Agent SDK via the CLI (`claude -p`). For the Python and TypeScript SDK packages with structured outputs, tool approval callbacks, and native message objects, see the full Agent SDK documentation.

Basic usage

Add the `-p` (or `--print`) flag to any `claude` command to run it non-interactively. All CLI options work with `-p` , including:

- `--continue` for continuing conversations
- `--allowedTools` for auto-approving tools
- `--output-format` for structured output

This example asks Claude a question about your codebase and prints the response:

```
claude -p "What does the auth module do?"
```

Examples

These examples highlight common CLI patterns.

Get structured output

Use `--output-format` to control how responses are returned:

- `text` (default): plain text output
- `json` : structured JSON with result, session ID, and metadata
- `stream-json` : newline-delimited JSON for real-time streaming

This example returns a project summary as JSON with session metadata, with the text result in the `result` field:

```
claude -p "Summarize this project" --output-format json
```

To get output conforming to a specific schema, use `--output-format json` with `--json-schema` and a JSON Schema definition. The response includes metadata about the request (session ID, usage, etc.) with the structured output in the `structured_output` field.

This example extracts function names and returns them as an array of strings:

```
claude -p "Extract the main function names from auth.py" \
  --output-format json \
  --json-schema '{"type":"object","properties":{"functions":
{"type":"array","items":{"type":"string"}}},"required":["functions"]}'
```

> **Tip:**
>
> Use a tool like jq to parse the response and extract specific fields:

```
## Extract the text result
claude -p "Summarize this project" --output-format json | jq -r '.result'

## Extract structured output
claude -p "Extract function names from auth.py" \
  --output-format json \
  --json-schema '{"type":"object","properties":{"functions":
{"type":"array","items":{"type":"string"}}},"required":["functions"]}' \
  | jq '.structured_output'
```

Stream responses

Use `--output-format stream-json` with `--verbose` and `--include-partial-messages` to receive tokens as they're generated. Each line is a JSON object representing an event:

```
claude -p "Explain recursion" --output-format stream-json --verbose --include-
partial-messages
```

The following example uses jq to filter for text deltas and display just the streaming text. The `-r` flag outputs raw strings (no quotes) and `-j` joins without newlines so tokens stream continuously:

```
claude -p "Write a poem" --output-format stream-json --verbose --include-partial-
messages | \
  jq -rj 'select(.type == "stream_event" and .event.delta.type? == "text_delta")
| .event.delta.text'
```

For programmatic streaming with callbacks and message objects, see Stream responses in real-time in the Agent SDK documentation.

Auto-approve tools

Use `--allowedTools` to let Claude use certain tools without prompting. This example runs a test suite and fixes failures, allowing Claude to execute Bash commands and read/edit files without asking for permission:

```
claude -p "Run the test suite and fix any failures" \
  --allowedTools "Bash,Read,Edit"
```

Create a commit

This example reviews staged changes and creates a commit with an appropriate message:

```
claude -p "Look at my staged changes and create an appropriate commit" \
  --allowedTools "Bash(git diff *),Bash(git log *),Bash(git status *),Bash(git
commit *)"
```

The `--allowedTools` flag uses permission rule syntax. The trailing `*` enables prefix matching, so `Bash(git diff *)` allows any command starting with `git diff`. The space before `*` is important: without it, `Bash(git diff*)` would also match `git diff-index`.

> **Note:**
>
> User-invoked skills like `/commit` and built-in commands are only available in interactive mode. In `-p` mode, describe the task you want to accomplish instead.

Customize the system prompt

Use `--append-system-prompt` to add instructions while keeping Claude Code's default behavior. This example pipes a PR diff to Claude and instructs it to review for security vulnerabilities:

```
gh pr diff "$1" | claude -p \
  --append-system-prompt "You are a security engineer. Review for
vulnerabilities." \
  --output-format json
```

See system prompt flags for more options including `--system-prompt` to fully replace the default prompt.

Continue conversations

Use `--continue` to continue the most recent conversation, or `--resume` with a session ID to continue a specific conversation. This example runs a review, then sends follow-up prompts:

```
## First request
claude -p "Review this codebase for performance issues"

## Continue the most recent conversation
claude -p "Now focus on the database queries" --continue
claude -p "Generate a summary of all issues found" --continue
```

If you're running multiple conversations, capture the session ID to resume a specific one:

```
session_id=$(claude -p "Start a review" --output-format json | jq -r
'.session_id')
claude -p "Continue that review" --resume "$session_id"
```

Next steps

- Agent SDK quickstart: build your first agent with Python or TypeScript
- CLI reference: all CLI flags and options
- GitHub Actions: use the Agent SDK in GitHub workflows
- GitLab CI/CD: use the Agent SDK in GitLab pipelines

Continue local sessions from any device with Remote Control

> *Continue a local Claude Code session from your phone, tablet, or any browser using Remote Control. Works with claude.ai/code and the Claude mobile app.*

Note:

Remote Control is available on all plans. Team and Enterprise admins must first enable Claude Code in admin settings.

Remote Control connects claude.ai/code or the Claude app for iOS and Android to a Claude Code session running on your machine. Start a task at your desk, then pick it up from your phone on the couch or a browser on another computer.

When you start a Remote Control session on your machine, Claude keeps running locally the entire time, so nothing moves to the cloud. With Remote Control you can:

- **Use your full local environment remotely**: your filesystem, MCP servers, tools, and project configuration all stay available
- **Work from both surfaces at once**: the conversation stays in sync across all connected devices, so you can send messages from your terminal, browser, and phone interchangeably
- **Survive interruptions**: if your laptop sleeps or your network drops, the session reconnects automatically when your machine comes back online

Unlike Claude Code on the web, which runs on cloud infrastructure, Remote Control sessions run directly on your machine and interact with your local filesystem. The web and mobile interfaces are just a window into that local session.

Note:

Remote Control requires Claude Code v2.1.51 or later. Check your version with `claude --version`.

This page covers setup, how to start and connect to sessions, and how Remote Control compares to Claude Code on the web.

Requirements

Before using Remote Control, confirm that your environment meets these conditions:

- **Subscription**: available on Pro, Max, Team, and Enterprise plans. Team and Enterprise admins must first enable Claude Code in admin settings. API keys are not supported.
- **Authentication**: run `claude` and use `/login` to sign in through claude.ai if you haven't already.
- **Workspace trust**: run `claude` in your project directory at least once to accept the workspace trust dialog.

Start a Remote Control session

You can start a dedicated Remote Control server, start an interactive session with Remote Control enabled, or connect a session that's already running.

Server mode

Navigate to your project directory and run:

```
claude remote-control
```

The process stays running in your terminal in server mode, waiting for remote connections. It displays a session URL you can use to connect from another device, and you can press spacebar to show a QR code for quick access from your phone. While a remote session is active, the terminal shows connection status and tool activity.

Available flags:

Flag	Description
`--name "My Project"`	Set a custom session title visible in the session list at claude.ai/code.
`--spawn <mode>`	How concurrent sessions are created. Press `w` at runtime to toggle. • `same-dir` (default): all sessions share the current working directory, so they can conflict if editing the same files. • `worktree`: each on-demand session gets its own git worktree. Requires a git repository.

Flag	Description
`--capacity <N>`	Maximum number of concurrent sessions. Default is 32.
`--verbose`	Show detailed connection and session logs.
`--sandbox` / `--no-sandbox`	Enable or disable sandboxing for filesystem and network isolation. Off by default.

Interactive session

To start a normal interactive Claude Code session with Remote Control enabled, use the `--remote-control` flag (or `--rc`):

```
claude --remote-control
```

Optionally pass a name for the session:

```
claude --remote-control "My Project"
```

This gives you a full interactive session in your terminal that you can also control from claude.ai or the Claude app. Unlike `claude remote-control` (server mode), you can type messages locally while the session is also available remotely.

From an existing session

If you're already in a Claude Code session and want to continue it remotely, use the `/remote-control` (or `/rc`) command:

```
/remote-control
```

Pass a name as an argument to set a custom session title:

```
/remote-control My Project
```

This starts a Remote Control session that carries over your current conversation history and displays a session URL and QR code you can use to connect from another device. The `--verbose`, `--sandbox`, and `--no-sandbox` flags are not available with this command.

Connect from another device

Once a Remote Control session is active, you have a few ways to connect from another device:

- **Open the session URL** in any browser to go directly to the session on claude.ai/code. Both `claude remote-control` and `/remote-control` display this URL in the terminal.
- **Scan the QR code** shown alongside the session URL to open it directly in the Claude app. With `claude remote-control`, press spacebar to toggle the QR code display.
- **Open claude.ai/code or the Claude app** and find the session by name in the session list. Remote Control sessions show a computer icon with a green status dot when online.

The remote session takes its name from the `--name` argument (or the name passed to `/remote-control`), your last message, your `/rename` value, or "Remote Control session" if there's no conversation history. If the environment already has an active session, you'll be asked whether to continue it or start a new one.

If you don't have the Claude app yet, use the `/mobile` command inside Claude Code to display a download QR code for iOS or Android.

Enable Remote Control for all sessions

By default, Remote Control only activates when you explicitly run `claude remote-control`, `claude --remote-control`, or `/remote-control`. To enable it automatically for every interactive session, run `/config` inside Claude Code and set **Enable Remote Control for all sessions** to `true`. Set it back to `false` to disable.

With this setting on, each interactive Claude Code process registers one remote session. If you run multiple instances, each one gets its own environment and session. To run multiple concurrent sessions from a single process, use server mode with `--spawn` instead.

Connection and security

Your local Claude Code session makes outbound HTTPS requests only and never opens inbound ports on your machine. When you start Remote Control, it registers with the Anthropic API and polls for work. When you connect from another device, the server routes messages between the web or mobile client and your local session over a streaming connection.

All traffic travels through the Anthropic API over TLS, the same transport security as any Claude Code session. The connection uses multiple short-lived credentials, each scoped to a single purpose and expiring independently.

Remote Control vs Claude Code on the web

Remote Control and Claude Code on the web both use the claude.ai/code interface. The key difference is where the session runs: Remote Control executes on your machine, so your local MCP servers, tools, and project configuration stay available. Claude Code on the web executes in Anthropic-managed cloud infrastructure.

Use Remote Control when you're in the middle of local work and want to keep going from another device. Use Claude Code on the web when you want to kick off a task without any local setup, work on a repo you don't have cloned, or run multiple tasks in parallel.

Limitations

- **One remote session per interactive process**: outside of server mode, each Claude Code instance supports one remote session at a time. Use server mode with `--spawn` to run multiple concurrent sessions from a single process.
- **Terminal must stay open**: Remote Control runs as a local process. If you close the terminal or stop the `claude` process, the session ends. Run `claude remote-control` again to start a new one.
- **Extended network outage**: if your machine is awake but unable to reach the network for more than roughly 10 minutes, the session times out and the process exits. Run `claude remote-control` again to start a new session.

Related resources

- Claude Code on the web: run sessions in Anthropic-managed cloud environments instead of on your machine
- Authentication: set up `/login` and manage credentials for claude.ai
- CLI reference: full list of flags and commands including `claude remote-control`

- Security: how Remote Control sessions fit into the Claude Code security model
- Data usage: what data flows through the Anthropic API during local and remote sessions

Run prompts on a schedule

> *Use /loop and the cron scheduling tools to run prompts repeatedly, poll for status, or set one-time reminders within a Claude Code session.*

> **Note:**
>
> Scheduled tasks require Claude Code v2.1.72 or later. Check your version with `claude --version`.

Scheduled tasks let Claude re-run a prompt automatically on an interval. Use them to poll a deployment, babysit a PR, check back on a long-running build, or remind yourself to do something later in the session.

Tasks are session-scoped: they live in the current Claude Code process and are gone when you exit. For durable scheduling that survives restarts and runs without an active terminal session, see Desktop scheduled tasks or GitHub Actions.

Schedule a recurring prompt with /loop

The `/loop` bundled skill is the quickest way to schedule a recurring prompt. Pass an optional interval and a prompt, and Claude sets up a cron job that fires in the background while the session stays open.

```
/loop 5m check if the deployment finished and tell me what happened
```

Claude parses the interval, converts it to a cron expression, schedules the job, and confirms the cadence and job ID.

Interval syntax

Intervals are optional. You can lead with them, trail with them, or leave them out entirely.

Form	Example	Parsed interval
Leading token	`/loop 30m check the build`	every 30 minutes
Trailing `every` clause	`/loop check the build every 2 hours`	every 2 hours

Form	Example	Parsed interval
No interval	`/loop check the build`	defaults to every 10 minutes

Supported units are `s` for seconds, `m` for minutes, `h` for hours, and `d` for days. Seconds are rounded up to the nearest minute since cron has one-minute granularity. Intervals that don't divide evenly into their unit, such as `7m` or `90m`, are rounded to the nearest clean interval and Claude tells you what it picked.

Loop over another command

The scheduled prompt can itself be a command or skill invocation. This is useful for re-running a workflow you've already packaged.

```
/loop 20m /review-pr 1234
```

Each time the job fires, Claude runs `/review-pr 1234` as if you had typed it.

Set a one-time reminder

For one-shot reminders, describe what you want in natural language instead of using `/loop`. Claude schedules a single-fire task that deletes itself after running.

```
remind me at 3pm to push the release branch
```

```
in 45 minutes, check whether the integration tests passed
```

Claude pins the fire time to a specific minute and hour using a cron expression and confirms when it will fire.

Manage scheduled tasks

Ask Claude in natural language to list or cancel tasks, or reference the underlying tools directly.

```
what scheduled tasks do I have?
```

```
cancel the deploy check job
```

Under the hood, Claude uses these tools:

Tool	Purpose
`CronCreate`	Schedule a new task. Accepts a 5-field cron expression, the prompt to run, and whether it recurs or fires once.
`CronList`	List all scheduled tasks with their IDs, schedules, and prompts.
`CronDelete`	Cancel a task by ID.

Each scheduled task has an 8-character ID you can pass to `CronDelete`. A session can hold up to 50 scheduled tasks at once.

How scheduled tasks run

The scheduler checks every second for due tasks and enqueues them at low priority. A scheduled prompt fires between your turns, not while Claude is mid-response. If Claude is busy when a task comes due, the prompt waits until the current turn ends.

All times are interpreted in your local timezone. A cron expression like `0 9 * * *` means 9am wherever you're running Claude Code, not UTC.

Jitter

To avoid every session hitting the API at the same wall-clock moment, the scheduler adds a small deterministic offset to fire times:

- Recurring tasks fire up to 10% of their period late, capped at 15 minutes. An hourly job might fire anywhere from `:00` to `:06`.
- One-shot tasks scheduled for the top or bottom of the hour fire up to 90 seconds early.

The offset is derived from the task ID, so the same task always gets the same offset. If exact timing matters, pick a minute that is not `:00` or `:30`, for example `3 9 * * *` instead of `0 9 * * *`, and the one-shot jitter will not apply.

Three-day expiry

Recurring tasks automatically expire 3 days after creation. The task fires one final time, then deletes itself. This bounds how long a forgotten loop can run. If you need a recurring task to last longer, cancel and recreate it before it expires, or use Desktop scheduled tasks for durable scheduling.

Cron expression reference

`CronCreate` accepts standard 5-field cron expressions: `minute hour day-of-month month day-of-week`. All fields support wildcards (`*`), single values (`5`), steps (`*/15`), ranges (`1-5`), and comma-separated lists (`1,15,30`).

Example	Meaning
`*/5 * * * *`	Every 5 minutes
`0 * * * *`	Every hour on the hour
`7 * * * *`	Every hour at 7 minutes past
`0 9 * * *`	Every day at 9am local
`0 9 * * 1-5`	Weekdays at 9am local
`30 14 15 3 *`	March 15 at 2:30pm local

Day-of-week uses `0` or `7` for Sunday through `6` for Saturday. Extended syntax like `L`, `W`, `?`, and name aliases such as `MON` or `JAN` is not supported.

When both day-of-month and day-of-week are constrained, a date matches if either field matches. This follows standard vixie-cron semantics.

Disable scheduled tasks

Set `CLAUDE_CODE_DISABLE_CRON=1` in your environment to disable the scheduler entirely. The cron tools and `/loop` become unavailable, and any already-scheduled tasks stop firing. See Environment variables for the full list of disable flags.

Limitations

Session-scoped scheduling has inherent constraints:

- Tasks only fire while Claude Code is running and idle. Closing the terminal or letting the session exit cancels everything.
- No catch-up for missed fires. If a task's scheduled time passes while Claude is busy on a long-running request, it fires once when Claude becomes idle, not once per missed interval.
- No persistence across restarts. Restarting Claude Code clears all session-scoped tasks.

For cron-driven automation that needs to run unattended, use a GitHub Actions workflow with a `schedule` trigger, or Desktop scheduled tasks if you want a graphical setup flow.

Code Review

> *Set up automated PR reviews that catch logic errors, security vulnerabilities, and regressions using multi-agent analysis of your full codebase*

Note:

Code Review is in research preview, available for Teams and Enterprise subscriptions. It is not available for organizations with Zero Data Retention enabled.

Code Review analyzes your GitHub pull requests and posts findings as inline comments on the lines of code where it found issues. A fleet of specialized agents examine the code changes in the context of your full codebase, looking for logic errors, security vulnerabilities, broken edge cases, and subtle regressions.

Findings are tagged by severity and don't approve or block your PR, so existing review workflows stay intact. You can tune what Claude flags by adding a `CLAUDE.md` or `REVIEW.md` file to your repository.

To run Claude in your own CI infrastructure instead of this managed service, see GitHub Actions or GitLab CI/CD.

This page covers:

- How reviews work
- Setup
- Customizing reviews with `CLAUDE.md` and `REVIEW.md`
- Pricing

How reviews work

Once an admin enables Code Review for your organization, reviews trigger when a PR opens, on every push, or when manually requested, depending on the repository's configured behavior. Commenting `@claude review` starts reviews on a PR in any mode.

When a review runs, multiple agents analyze the diff and surrounding code in parallel on Anthropic infrastructure. Each agent looks for a different class of issue, then a verification step checks candidates against actual code behavior to filter out false positives. The results

are deduplicated, ranked by severity, and posted as inline comments on the specific lines where issues were found. If no issues are found, Claude posts a short confirmation comment on the PR.

Reviews scale in cost with PR size and complexity, completing in 20 minutes on average. Admins can monitor review activity and spend via the analytics dashboard.

Severity levels

Each finding is tagged with a severity level:

Marker	Severity	Meaning
🔴	Normal	A bug that should be fixed before merging
🟡	Nit	A minor issue, worth fixing but not blocking
🟣	Pre-existing	A bug that exists in the codebase but was not introduced by this PR

Findings include a collapsible extended reasoning section you can expand to understand why Claude flagged the issue and how it verified the problem.

What Code Review checks

By default, Code Review focuses on correctness: bugs that would break production, not formatting preferences or missing test coverage. You can expand what it checks by adding guidance files to your repository.

Set up Code Review

An admin enables Code Review once for the organization and selects which repositories to include.

Step 1: Open Claude Code admin settings

Go to claude.ai/admin-settings/claude-code and find the Code Review section. You need admin access to your Claude organization and permission to install GitHub Apps in your GitHub organization.

Step 2: Start setup

Click **Setup**. This begins the GitHub App installation flow.

Step 3: Install the Claude GitHub App

Follow the prompts to install the Claude GitHub App to your GitHub organization. The app requests these repository permissions:

- **Contents**: read and write
- **Issues**: read and write
- **Pull requests**: read and write

Code Review uses read access to contents and write access to pull requests. The broader permission set also supports GitHub Actions if you enable that later.

Step 4: Select repositories

Choose which repositories to enable for Code Review. If you don't see a repository, make sure you gave the Claude GitHub App access to it during installation. You can add more repositories later.

Step 5: Set review triggers per repo

After setup completes, the Code Review section shows your repositories in a table. For each repository, use the **Review Behavior** dropdown to choose when reviews run:

- **Once after PR creation**: review runs once when a PR is opened or marked ready for review
- **After every push**: review runs on every push to the PR branch, catching new issues as the PR evolves and auto-resolving threads when you fix flagged issues
- **Manual**: reviews start only when someone comments `@claude review` on a PR; subsequent pushes to that PR are then reviewed automatically

Reviewing on every push runs the most reviews and costs the most. Manual mode is useful for high-traffic repos where you want to opt specific PRs into review, or to only start reviewing your PRs once they're ready.

The repositories table also shows the average cost per review for each repo based on recent activity. Use the row actions menu to turn Code Review on or off per repository, or to remove a repository entirely.

To verify setup, open a test PR. If you chose an automatic trigger, a check run named **Claude Code Review** appears within a few minutes. If you chose Manual, comment `@claude review` on the PR to start the first review. If no check run appears, confirm the repository is listed in your admin settings and the Claude GitHub App has access to it.

Manually trigger reviews

Comment `@claude review` on a pull request to start a review and opt that PR into push-triggered reviews going forward. This works regardless of the repository's configured trigger: use it to opt specific PRs into review in Manual mode, or to get an immediate re-review in other modes. Either way, pushes to that PR trigger reviews from then on.

For the comment to trigger a review:

- Post it as a top-level PR comment, not an inline comment on a diff line
- Put `@claude review` at the start of the comment
- You must have owner, member, or collaborator access to the repository
- The PR must be open and not a draft

If a review is already running on that PR, the request is queued until the in-progress review completes. You can monitor progress via the check run on the PR.

Customize reviews

Code Review reads two files from your repository to guide what it flags. Both are additive on top of the default correctness checks:

- `CLAUDE.md`: shared project instructions that Claude Code uses for all tasks, not just reviews. Use it when guidance also applies to interactive Claude Code sessions.
- `REVIEW.md`: review-only guidance, read exclusively during code reviews. Use it for rules that are strictly about what to flag or skip during review and would clutter your general `CLAUDE.md`.

CLAUDE.md

Code Review reads your repository's `CLAUDE.md` files and treats newly-introduced violations as nit-level findings. This works bidirectionally: if your PR changes code in a way that makes a `CLAUDE.md` statement outdated, Claude flags that the docs need updating too.

Claude reads `CLAUDE.md` files at every level of your directory hierarchy, so rules in a subdirectory's `CLAUDE.md` apply only to files under that path. See the memory documentation for more on how `CLAUDE.md` works.

For review-specific guidance that you don't want applied to general Claude Code sessions, use `REVIEW.md` instead.

REVIEW.md

Add a `REVIEW.md` file to your repository root for review-specific rules. Use it to encode:

- Company or team style guidelines: "prefer early returns over nested conditionals"
- Language- or framework-specific conventions not covered by linters
- Things Claude should always flag: "any new API route must have an integration test"
- Things Claude should skip: "don't comment on formatting in generated code under `/gen/`"

Example `REVIEW.md`:

```
## Code Review Guidelines

### Always check
- New API endpoints have corresponding integration tests
- Database migrations are backward-compatible
- Error messages don't leak internal details to users

### Style
- Prefer `match` statements over chained `isinstance` checks
- Use structured logging, not f-string interpolation in log calls

### Skip
- Generated files under `src/gen/`
- Formatting-only changes in `*.lock` files
```

Claude auto-discovers `REVIEW.md` at the repository root. No configuration needed.

View usage

Go to claude.ai/analytics/code-review to see Code Review activity across your organization. The dashboard shows:

Section	What it shows
PRs reviewed	Daily count of pull requests reviewed over the selected time range
Cost weekly	Weekly spend on Code Review

Section	What it shows
Feedback	Count of review comments that were auto-resolved because a developer addressed the issue
Repository breakdown	Per-repo counts of PRs reviewed and comments resolved

The repositories table in admin settings also shows average cost per review for each repo.

Pricing

Code Review is billed based on token usage. Reviews average $15-25, scaling with PR size, codebase complexity, and how many issues require verification. Code Review usage is billed separately through extra usage and does not count against your plan's included usage.

The review trigger you choose affects total cost:

- **Once after PR creation**: runs once per PR
- **After every push**: runs on each push, multiplying cost by the number of pushes
- **Manual**: no reviews until someone comments `@claude review` on a PR

In any mode, commenting `@claude review` opts the PR into push-triggered reviews, so additional cost accrues per push after that comment.

Costs appear on your Anthropic bill regardless of whether your organization uses AWS Bedrock or Google Vertex AI for other Claude Code features. To set a monthly spend cap for Code Review, go to claude.ai/admin-settings/usage and configure the limit for the Claude Code Review service.

Monitor spend via the weekly cost chart in analytics or the per-repo average cost column in admin settings.

Related resources

Code Review is designed to work alongside the rest of Claude Code. If you want to run reviews locally before opening a PR, need a self-hosted setup, or want to go deeper on how `CLAUDE.md` shapes Claude's behavior across tools, these pages are good next stops:

- Plugins: browse the plugin marketplace, including a `code-review` plugin for running on-demand reviews locally before pushing
- GitHub Actions: run Claude in your own GitHub Actions workflows for custom automation beyond code review

- GitLab CI/CD: self-hosted Claude integration for GitLab pipelines
- Memory: how `CLAUDE.md` files work across Claude Code
- Analytics: track Claude Code usage beyond code review

Part 7: CI/CD & Integrations

Claude Code GitHub Actions

> *Learn about integrating Claude Code into your development workflow with Claude Code GitHub Actions*

Claude Code GitHub Actions brings AI-powered automation to your GitHub workflow. With a simple `@claude` mention in any PR or issue, Claude can analyze your code, create pull requests, implement features, and fix bugs - all while following your project's standards. For automatic reviews posted on every PR without a trigger, see GitHub Code Review.

> **Note:**
>
> Claude Code GitHub Actions is built on top of the Claude Agent SDK, which enables programmatic integration of Claude Code into your applications. You can use the SDK to build custom automation workflows beyond GitHub Actions.

> **Info:**
>
> **Claude Opus 4.6 is now available.** Claude Code GitHub Actions default to Sonnet. To use Opus 4.6, configure the model parameter to use `claude-opus-4-6`.

Why use Claude Code GitHub Actions?

- **Instant PR creation**: Describe what you need, and Claude creates a complete PR with all necessary changes
- **Automated code implementation**: Turn issues into working code with a single command
- **Follows your standards**: Claude respects your `CLAUDE.md` guidelines and existing code patterns
- **Simple setup**: Get started in minutes with our installer and API key
- **Secure by default**: Your code stays on Github's runners

What can Claude do?

Claude Code provides a powerful GitHub Action that transforms how you work with code:

Claude Code Action

This GitHub Action allows you to run Claude Code within your GitHub Actions workflows. You can use this to build any custom workflow on top of Claude Code.

View repository →

Setup

Quick setup

The easiest way to set up this action is through Claude Code in the terminal. Just open claude and run `/install-github-app`.

This command will guide you through setting up the GitHub app and required secrets.

> **Note:**
>
> - You must be a repository admin to install the GitHub app and add secrets
> - The GitHub app will request read & write permissions for Contents, Issues, and Pull requests
> - This quickstart method is only available for direct Claude API users. If you're using AWS Bedrock or Google Vertex AI, please see the Using with AWS Bedrock & Google Vertex AI section.

Manual setup

If the `/install-github-app` command fails or you prefer manual setup, please follow these manual setup instructions:

1. **Install the Claude GitHub app** to your repository: https://github.com/apps/claude

   The Claude GitHub app requires the following repository permissions:

   - **Contents**: Read & write (to modify repository files)
   - **Issues**: Read & write (to respond to issues)
   - **Pull requests**: Read & write (to create PRs and push changes)

   For more details on security and permissions, see the security documentation.

2. **Add ANTHROPIC_API_KEY** to your repository secrets (Learn how to use secrets in GitHub Actions)
3. **Copy the workflow file** from examples/claude.yml into your repository's `.github/workflows/`

> **Tip:**
>
> After completing either the quickstart or manual setup, test the action by tagging `@claude` in an issue or PR comment.

Upgrading from Beta

> **Warning:**
>
> Claude Code GitHub Actions v1.0 introduces breaking changes that require updating your workflow files in order to upgrade to v1.0 from the beta version.

If you're currently using the beta version of Claude Code GitHub Actions, we recommend that you update your workflows to use the GA version. The new version simplifies configuration while adding powerful new features like automatic mode detection.

Essential changes

All beta users must make these changes to their workflow files in order to upgrade:

1. **Update the action version**: Change `@beta` to `@v1`
2. **Remove mode configuration**: Delete `mode: "tag"` or `mode: "agent"` (now auto-detected)
3. **Update prompt inputs**: Replace `direct_prompt` with `prompt`
4. **Move CLI options**: Convert `max_turns`, `model`, `custom_instructions`, etc. to `claude_args`

Breaking Changes Reference

Old Beta Input	New v1.0 Input
`mode`	*(Removed - auto-detected)*
`direct_prompt`	`prompt`
`override_prompt`	`prompt` with GitHub variables

Old Beta Input	New v1.0 Input
`custom_instructions`	`claude_args: --append-system-prompt`
`max_turns`	`claude_args: --max-turns`
`model`	`claude_args: --model`
`allowed_tools`	`claude_args: --allowedTools`
`disallowed_tools`	`claude_args: --disallowedTools`
`claude_env`	`settings` JSON format

Before and After Example

Beta version:

```yaml
- uses: anthropics/claude-code-action@beta
  with:
    mode: "tag"
    direct_prompt: "Review this PR for security issues"
    anthropic_api_key: ${{ secrets.ANTHROPIC_API_KEY }}
    custom_instructions: "Follow our coding standards"
    max_turns: "10"
    model: "claude-sonnet-4-6"
```

GA version (v1.0):

```yaml
- uses: anthropics/claude-code-action@v1
  with:
    prompt: "Review this PR for security issues"
    anthropic_api_key: ${{ secrets.ANTHROPIC_API_KEY }}
    claude_args: |
      --append-system-prompt "Follow our coding standards"
      --max-turns 10
      --model claude-sonnet-4-6
```

Tip:

> The action now automatically detects whether to run in interactive mode (responds to `@claude` mentions) or automation mode (runs immediately with a prompt) based on your configuration.

Example use cases

Claude Code GitHub Actions can help you with a variety of tasks. The examples directory contains ready-to-use workflows for different scenarios.

Basic workflow

```yaml
name: Claude Code
on:
  issue_comment:
    types: [created]
  pull_request_review_comment:
    types: [created]
jobs:
  claude:
    runs-on: ubuntu-latest
    steps:
      - uses: anthropics/claude-code-action@v1
        with:
          anthropic_api_key: ${{ secrets.ANTHROPIC_API_KEY }}
          # Responds to @claude mentions in comments
```

Using skills

```yaml
name: Code Review
on:
  pull_request:
    types: [opened, synchronize]
jobs:
  review:
    runs-on: ubuntu-latest
    steps:
      - uses: anthropics/claude-code-action@v1
        with:
          anthropic_api_key: ${{ secrets.ANTHROPIC_API_KEY }}
          prompt: "Review this pull request for code quality, correctness, and security. Analyze the diff, then post your findings as review comments."
          claude_args: "--max-turns 5"
```

Custom automation with prompts

```yaml
name: Daily Report
on:
  schedule:
    - cron: "0 9 * * *"
jobs:
  report:
    runs-on: ubuntu-latest
    steps:
      - uses: anthropics/claude-code-action@v1
        with:
          anthropic_api_key: ${{ secrets.ANTHROPIC_API_KEY }}
          prompt: "Generate a summary of yesterday's commits and open issues"
          claude_args: "--model opus"
```

Common use cases

In issue or PR comments:

```
@claude implement this feature based on the issue description
@claude how should I implement user authentication for this endpoint?
@claude fix the TypeError in the user dashboard component
```

Claude will automatically analyze the context and respond appropriately.

Best practices

CLAUDE.md configuration

Create a `CLAUDE.md` file in your repository root to define code style guidelines, review criteria, project-specific rules, and preferred patterns. This file guides Claude's understanding of your project standards.

Security considerations

> **Warning:**
>
> Never commit API keys directly to your repository.

For comprehensive security guidance including permissions, authentication, and best practices, see the Claude Code Action security documentation.

Always use GitHub Secrets for API keys:

- Add your API key as a repository secret named `ANTHROPIC_API_KEY`
- Reference it in workflows: `anthropic_api_key: ${{ secrets.ANTHROPIC_API_KEY }}`
- Limit action permissions to only what's necessary
- Review Claude's suggestions before merging

Always use GitHub Secrets (for example, `${{ secrets.ANTHROPIC_API_KEY }}`) rather than hardcoding API keys directly in your workflow files.

Optimizing performance

Use issue templates to provide context, keep your `CLAUDE.md` concise and focused, and configure appropriate timeouts for your workflows.

CI costs

When using Claude Code GitHub Actions, be aware of the associated costs:

GitHub Actions costs:

- Claude Code runs on GitHub-hosted runners, which consume your GitHub Actions minutes
- See GitHub's billing documentation for detailed pricing and minute limits

API costs:

- Each Claude interaction consumes API tokens based on the length of prompts and responses
- Token usage varies by task complexity and codebase size
- See Claude's pricing page for current token rates

Cost optimization tips:

- Use specific `@claude` commands to reduce unnecessary API calls
- Configure appropriate `--max-turns` in `claude_args` to prevent excessive iterations
- Set workflow-level timeouts to avoid runaway jobs
- Consider using GitHub's concurrency controls to limit parallel runs

Configuration examples

The Claude Code Action v1 simplifies configuration with unified parameters:

```
- uses: anthropics/claude-code-action@v1
  with:
    anthropic_api_key: ${{ secrets.ANTHROPIC_API_KEY }}
    prompt: "Your instructions here" # Optional
    claude_args: "--max-turns 5" # Optional CLI arguments
```

Key features:

- **Unified prompt interface** - Use `prompt` for all instructions
- **Skills** - Invoke installed skills directly from the prompt
- **CLI passthrough** - Any Claude Code CLI argument via `claude_args`
- **Flexible triggers** - Works with any GitHub event

Visit the examples directory for complete workflow files.

> **Tip:**

> When responding to issue or PR comments, Claude automatically responds to @claude mentions. For other events, use the `prompt` parameter to provide instructions.

Using with AWS Bedrock & Google Vertex AI

For enterprise environments, you can use Claude Code GitHub Actions with your own cloud infrastructure. This approach gives you control over data residency and billing while maintaining the same functionality.

Prerequisites

Before setting up Claude Code GitHub Actions with cloud providers, you need:

For Google Cloud Vertex AI:

1. A Google Cloud Project with Vertex AI enabled
2. Workload Identity Federation configured for GitHub Actions
3. A service account with the required permissions
4. A GitHub App (recommended) or use the default GITHUB_TOKEN

For AWS Bedrock:

1. An AWS account with Amazon Bedrock enabled
2. GitHub OIDC Identity Provider configured in AWS
3. An IAM role with Bedrock permissions
4. A GitHub App (recommended) or use the default GITHUB_TOKEN

Step 1: Create a custom GitHub App (Recommended for 3P Providers)

For best control and security when using 3P providers like Vertex AI or Bedrock, we recommend creating your own GitHub App:

1. Go to https://github.com/settings/apps/new
2. Fill in the basic information:

- **GitHub App name**: Choose a unique name (e.g., "YourOrg Claude Assistant")
- **Homepage URL**: Your organization's website or the repository URL

1. Configure the app settings:

- **Webhooks**: Uncheck "Active" (not needed for this integration)

1. Set the required permissions:

- **Repository permissions**:

- Contents: Read & Write
- Issues: Read & Write
- Pull requests: Read & Write

1. Click “Create GitHub App”
2. After creation, click “Generate a private key” and save the downloaded `.pem` file
3. Note your App ID from the app settings page
4. Install the app to your repository:

- From your app’s settings page, click “Install App” in the left sidebar
- Select your account or organization
- Choose “Only select repositories” and select the specific repository
- Click “Install”

1. Add the private key as a secret to your repository:

- Go to your repository’s Settings → Secrets and variables → Actions
- Create a new secret named `APP_PRIVATE_KEY` with the contents of the `.pem` file

1. Add the App ID as a secret:

- Create a new secret named `APP_ID` with your GitHub App’s ID

> **Note:**
>
> This app will be used with the actions/create-github-app-token action to generate authentication tokens in your workflows.

Alternative for Claude API or if you don’t want to setup your own Github app: Use the official Anthropic app:

1. Install from: https://github.com/apps/claude
2. No additional configuration needed for authentication

Step 2: Configure cloud provider authentication

Choose your cloud provider and set up secure authentication:

Configure AWS to allow GitHub Actions to authenticate securely without storing credentials.

> ***Security Note**: Use repository-specific configurations and grant only the minimum required permissions.*

Required Setup:

1. **Enable Amazon Bedrock**:

- Request access to Claude models in Amazon Bedrock
- For cross-region models, request access in all required regions

1. **Set up GitHub OIDC Identity Provider**:

- Provider URL: `https://token.actions.githubusercontent.com`
- Audience: `sts.amazonaws.com`

1. **Create IAM Role for GitHub Actions**:

- Trusted entity type: Web identity
- Identity provider: `token.actions.githubusercontent.com`
- Permissions: `AmazonBedrockFullAccess` policy
- Configure trust policy for your specific repository

Required Values:

After setup, you'll need:

- **AWS_ROLE_TO_ASSUME**: The ARN of the IAM role you created

> **Tip:**
>
> OIDC is more secure than using static AWS access keys because credentials are temporary and automatically rotated.

See AWS documentation for detailed OIDC setup instructions.

Configure Google Cloud to allow GitHub Actions to authenticate securely without storing credentials.

> ***Security Note**: Use repository-specific configurations and grant only the minimum required permissions.*

Required Setup:

1. **Enable APIs** in your Google Cloud project:

- IAM Credentials API
- Security Token Service (STS) API

- Vertex AI API

1. **Create Workload Identity Federation resources**:

- Create a Workload Identity Pool
- Add a GitHub OIDC provider with:
- Issuer: `https://token.actions.githubusercontent.com`
- Attribute mappings for repository and owner
- **Security recommendation**: Use repository-specific attribute conditions

1. **Create a Service Account**:

- Grant only `Vertex AI User` role
- **Security recommendation**: Create a dedicated service account per repository

1. **Configure IAM bindings**:

- Allow the Workload Identity Pool to impersonate the service account
- **Security recommendation**: Use repository-specific principal sets

Required Values:

After setup, you'll need:

- **GCP_WORKLOAD_IDENTITY_PROVIDER**: The full provider resource name
- **GCP_SERVICE_ACCOUNT**: The service account email address

> **Tip:**
>
> Workload Identity Federation eliminates the need for downloadable service account keys, improving security.

For detailed setup instructions, consult the Google Cloud Workload Identity Federation documentation.

Step 3: Add Required Secrets

Add the following secrets to your repository (Settings → Secrets and variables → Actions):

For Claude API (Direct):

1. **For API Authentication**:

- `ANTHROPIC_API_KEY` : Your Claude API key from console.anthropic.com

1. **For GitHub App (if using your own app)**:

- `APP_ID` : Your GitHub App's ID
- `APP_PRIVATE_KEY` : The private key (.pem) content

For Google Cloud Vertex AI

1. **For GCP Authentication**:

- `GCP_WORKLOAD_IDENTITY_PROVIDER`
- `GCP_SERVICE_ACCOUNT`

1. **For GitHub App (if using your own app)**:

- `APP_ID` : Your GitHub App's ID
- `APP_PRIVATE_KEY` : The private key (.pem) content

For AWS Bedrock

1. **For AWS Authentication**:

- `AWS_ROLE_TO_ASSUME`

1. **For GitHub App (if using your own app)**:

- `APP_ID` : Your GitHub App's ID
- `APP_PRIVATE_KEY` : The private key (.pem) content

Step 4: Create workflow files

Create GitHub Actions workflow files that integrate with your cloud provider. The examples below show complete configurations for both AWS Bedrock and Google Vertex AI:

Prerequisites:

- AWS Bedrock access enabled with Claude model permissions
- GitHub configured as an OIDC identity provider in AWS
- IAM role with Bedrock permissions that trusts GitHub Actions

Required GitHub secrets:

Secret Name	Description
`AWS_ROLE_TO_ASSUME`	ARN of the IAM role for Bedrock access
`APP_ID`	Your GitHub App ID (from app settings)
`APP_PRIVATE_KEY`	The private key you generated for your GitHub App

```yaml
name: Claude PR Action

permissions:
  contents: write
  pull-requests: write
  issues: write
  id-token: write

on:
  issue_comment:
    types: [created]
  pull_request_review_comment:
    types: [created]
  issues:
    types: [opened, assigned]

jobs:
  claude-pr:
    if: |
      (github.event_name == 'issue_comment' && contains(github.event.comment.body,
'@claude')) ||
      (github.event_name == 'pull_request_review_comment' &&
contains(github.event.comment.body, '@claude')) ||
      (github.event_name == 'issues' && contains(github.event.issue.body,
'@claude'))
    runs-on: ubuntu-latest
    env:
      AWS_REGION: us-west-2
    steps:
      - name: Checkout repository
        uses: actions/checkout@v4

      - name: Generate GitHub App token
        id: app-token
        uses: actions/create-github-app-token@v2
        with:
          app-id: ${{ secrets.APP_ID }}
          private-key: ${{ secrets.APP_PRIVATE_KEY }}
```

```
      - name: Configure AWS Credentials (OIDC)
        uses: aws-actions/configure-aws-credentials@v4
        with:
          role-to-assume: ${{ secrets.AWS_ROLE_TO_ASSUME }}
          aws-region: us-west-2

      - uses: anthropics/claude-code-action@v1
        with:
          github_token: ${{ steps.app-token.outputs.token }}
          use_bedrock: "true"
          claude_args: '--model us.anthropic.claude-sonnet-4-6 --max-turns 10'
```

> **Tip:**
>
> The model ID format for Bedrock includes a region prefix (for example, `us.anthropic.claude-sonnet-4-6`).

Prerequisites:

- Vertex AI API enabled in your GCP project
- Workload Identity Federation configured for GitHub
- Service account with Vertex AI permissions

Required GitHub secrets:

Secret Name	Description
`GCP_WORKLOAD_IDENTITY_PROVIDER`	Workload identity provider resource name
`GCP_SERVICE_ACCOUNT`	Service account email with Vertex AI access
`APP_ID`	Your GitHub App ID (from app settings)
`APP_PRIVATE_KEY`	The private key you generated for your GitHub App

```yaml
name: Claude PR Action

permissions:
  contents: write
  pull-requests: write
  issues: write
  id-token: write

on:
  issue_comment:
    types: [created]
  pull_request_review_comment:
    types: [created]
  issues:
    types: [opened, assigned]

jobs:
  claude-pr:
    if: |
      (github.event_name == 'issue_comment' && contains(github.event.comment.body,
'@claude')) ||
      (github.event_name == 'pull_request_review_comment' &&
contains(github.event.comment.body, '@claude')) ||
      (github.event_name == 'issues' && contains(github.event.issue.body,
'@claude'))
    runs-on: ubuntu-latest
    steps:
      - name: Checkout repository
        uses: actions/checkout@v4

      - name: Generate GitHub App token
        id: app-token
        uses: actions/create-github-app-token@v2
        with:
          app-id: ${{ secrets.APP_ID }}
          private-key: ${{ secrets.APP_PRIVATE_KEY }}

      - name: Authenticate to Google Cloud
```

```
        id: auth
        uses: google-github-actions/auth@v2
        with:
          workload_identity_provider: $
{{ secrets.GCP_WORKLOAD_IDENTITY_PROVIDER }}
          service_account: ${{ secrets.GCP_SERVICE_ACCOUNT }}

      - uses: anthropics/claude-code-action@v1
        with:
          github_token: ${{ steps.app-token.outputs.token }}
          trigger_phrase: "@claude"
          use_vertex: "true"
          claude_args: '--model claude-sonnet-4@20250514 --max-turns 10'
        env:
          ANTHROPIC_VERTEX_PROJECT_ID: ${{ steps.auth.outputs.project_id }}
          CLOUD_ML_REGION: us-east5
          VERTEX_REGION_CLAUDE_3_7_SONNET: us-east5
```

> **Tip:**
>
> The project ID is automatically retrieved from the Google Cloud authentication step, so you don't need to hardcode it.

Troubleshooting

Claude not responding to @claude commands

Verify the GitHub App is installed correctly, check that workflows are enabled, ensure API key is set in repository secrets, and confirm the comment contains `@claude` (not `/claude`).

CI not running on Claude's commits

Ensure you're using the GitHub App or custom app (not Actions user), check workflow triggers include the necessary events, and verify app permissions include CI triggers.

Authentication errors

Confirm API key is valid and has sufficient permissions. For Bedrock/Vertex, check credentials configuration and ensure secrets are named correctly in workflows.

Advanced configuration

Action parameters

The Claude Code Action v1 uses a simplified configuration:

Parameter	Description	Required
`prompt`	Instructions for Claude (plain text or a skill name)	No*
`claude_args`	CLI arguments passed to Claude Code	No
`anthropic_api_key`	Claude API key	Yes**
`github_token`	GitHub token for API access	No
`trigger_phrase`	Custom trigger phrase (default: “@claude”)	No
`use_bedrock`	Use AWS Bedrock instead of Claude API	No
`use_vertex`	Use Google Vertex AI instead of Claude API	No

*Prompt is optional - when omitted for issue/PR comments, Claude responds to trigger phrase
**Required for direct Claude API, not for Bedrock/Vertex

Pass CLI arguments

The `claude_args` parameter accepts any Claude Code CLI arguments:

```
claude_args: "--max-turns 5 --model claude-sonnet-4-6 --mcp-config /path/to/config.json"
```

Common arguments:

- `--max-turns` : Maximum conversation turns (default: 10)
- `--model` : Model to use (for example, `claude-sonnet-4-6`)
- `--mcp-config` : Path to MCP configuration
- `--allowed-tools` : Comma-separated list of allowed tools

- `--debug`: Enable debug output

Alternative integration methods

While the `/install-github-app` command is the recommended approach, you can also:

- **Custom GitHub App**: For organizations needing branded usernames or custom authentication flows. Create your own GitHub App with required permissions (contents, issues, pull requests) and use the actions/create-github-app-token action to generate tokens in your workflows.
- **Manual GitHub Actions**: Direct workflow configuration for maximum flexibility
- **MCP Configuration**: Dynamic loading of Model Context Protocol servers

See the Claude Code Action documentation for detailed guides on authentication, security, and advanced configuration.

Customizing Claude's behavior

You can configure Claude's behavior in two ways:

1. **CLAUDE.md**: Define coding standards, review criteria, and project-specific rules in a `CLAUDE.md` file at the root of your repository. Claude will follow these guidelines when creating PRs and responding to requests. Check out our Memory documentation for more details.
2. **Custom prompts**: Use the `prompt` parameter in the workflow file to provide workflow-specific instructions. This allows you to customize Claude's behavior for different workflows or tasks.

Claude will follow these guidelines when creating PRs and responding to requests.

Claude Code GitLab CI/CD

Learn about integrating Claude Code into your development workflow with GitLab CI/CD

Info:

Claude Code for GitLab CI/CD is currently in beta. Features and functionality may evolve as we refine the experience.

This integration is maintained by GitLab. For support, see the following GitLab issue.

Note:

This integration is built on top of the Claude Code CLI and Agent SDK, enabling programmatic use of Claude in your CI/CD jobs and custom automation workflows.

Why use Claude Code with GitLab?

- **Instant MR creation**: Describe what you need, and Claude proposes a complete MR with changes and explanation
- **Automated implementation**: Turn issues into working code with a single command or mention
- **Project-aware**: Claude follows your `CLAUDE.md` guidelines and existing code patterns
- **Simple setup**: Add one job to `.gitlab-ci.yml` and a masked CI/CD variable
- **Enterprise-ready**: Choose Claude API, AWS Bedrock, or Google Vertex AI to meet data residency and procurement needs
- **Secure by default**: Runs in your GitLab runners with your branch protection and approvals

How it works

Claude Code uses GitLab CI/CD to run AI tasks in isolated jobs and commit results back via MRs:

1. **Event-driven orchestration**: GitLab listens for your chosen triggers (for example, a comment that mentions `@claude` in an issue, MR, or review thread). The job collects context from the thread and repository, builds prompts from that input, and runs Claude Code.
2. **Provider abstraction**: Use the provider that fits your environment:
   - Claude API (SaaS)
   - AWS Bedrock (IAM-based access, cross-region options)
   - Google Vertex AI (GCP-native, Workload Identity Federation)
3. **Sandboxed execution**: Each interaction runs in a container with strict network and filesystem rules. Claude Code enforces workspace-scoped permissions to constrain writes. Every change flows through an MR so reviewers see the diff and approvals still apply.

Pick regional endpoints to reduce latency and meet data-sovereignty requirements while using existing cloud agreements.

What can Claude do?

Claude Code enables powerful CI/CD workflows that transform how you work with code:

- Create and update MRs from issue descriptions or comments
- Analyze performance regressions and propose optimizations
- Implement features directly in a branch, then open an MR
- Fix bugs and regressions identified by tests or comments
- Respond to follow-up comments to iterate on requested changes

Setup

Quick setup

The fastest way to get started is to add a minimal job to your `.gitlab-ci.yml` and set your API key as a masked variable.

1. **Add a masked CI/CD variable**
   - Go to **Settings → CI/CD → Variables**
   - Add `ANTHROPIC_API_KEY` (masked, protected as needed)

2. **Add a Claude job to `.gitlab-ci.yml`**

```
stages:
  - ai

claude:
  stage: ai
  image: node:24-alpine3.21
  # Adjust rules to fit how you want to trigger the job:
  # - manual runs
  # - merge request events
  # - web/API triggers when a comment contains '@claude'
  rules:
    - if: '$CI_PIPELINE_SOURCE == "web"'
    - if: '$CI_PIPELINE_SOURCE == "merge_request_event"'
  variables:
    GIT_STRATEGY: fetch
  before_script:
    - apk update
    - apk add --no-cache git curl bash
    - curl -fsSL https://claude.ai/install.sh | bash
  script:
    # Optional: start a GitLab MCP server if your setup provides one
    - /bin/gitlab-mcp-server || true
    # Use AI_FLOW_* variables when invoking via web/API triggers with context
payloads
    - echo "$AI_FLOW_INPUT for $AI_FLOW_CONTEXT on $AI_FLOW_EVENT"
    - >
      claude
      -p "${AI_FLOW_INPUT:-'Review this MR and implement the requested changes'}"
      --permission-mode acceptEdits
      --allowedTools "Bash Read Edit Write mcp__gitlab"
      --debug
```

After adding the job and your `ANTHROPIC_API_KEY` variable, test by running the job manually from **CI/CD → Pipelines**, or trigger it from an MR to let Claude propose updates in a branch and open an MR if needed.

> **Note:**
>
> To run on AWS Bedrock or Google Vertex AI instead of the Claude API, see the Using with AWS Bedrock & Google Vertex AI section below for authentication and environment setup.

Manual setup (recommended for production)

If you prefer a more controlled setup or need enterprise providers:

1. **Configure provider access**:
   - **Claude API**: Create and store `ANTHROPIC_API_KEY` as a masked CI/CD variable
   - **AWS Bedrock**: **Configure GitLab → AWS OIDC** and create an IAM role for Bedrock
   - **Google Vertex AI**: **Configure Workload Identity Federation for GitLab → GCP**
2. **Add project credentials for GitLab API operations**:
   - Use `CI_JOB_TOKEN` by default, or create a Project Access Token with `api` scope
   - Store as `GITLAB_ACCESS_TOKEN` (masked) if using a PAT
3. **Add the Claude job to `.gitlab-ci.yml`** (see examples below)
4. **(Optional) Enable mention-driven triggers**:
   - Add a project webhook for "Comments (notes)" to your event listener (if you use one)
   - Have the listener call the pipeline trigger API with variables like `AI_FLOW_INPUT` and `AI_FLOW_CONTEXT` when a comment contains `@claude`

Example use cases

Turn issues into MRs

In an issue comment:

```
@claude implement this feature based on the issue description
```

Claude analyzes the issue and codebase, writes changes in a branch, and opens an MR for review.

Get implementation help

In an MR discussion:

```
@claude suggest a concrete approach to cache the results of this API call
```

Claude proposes changes, adds code with appropriate caching, and updates the MR.

Fix bugs quickly

In an issue or MR comment:

```
@claude fix the TypeError in the user dashboard component
```

Claude locates the bug, implements a fix, and updates the branch or opens a new MR.

Using with AWS Bedrock & Google Vertex AI

For enterprise environments, you can run Claude Code entirely on your cloud infrastructure with the same developer experience.

AWS Bedrock

Prerequisites

Before setting up Claude Code with AWS Bedrock, you need:

1. An AWS account with Amazon Bedrock access to the desired Claude models
2. GitLab configured as an OIDC identity provider in AWS IAM
3. An IAM role with Bedrock permissions and a trust policy restricted to your GitLab project/refs
4. GitLab CI/CD variables for role assumption:

- `AWS_ROLE_TO_ASSUME` (role ARN)
- `AWS_REGION` (Bedrock region)

Setup instructions

Configure AWS to allow GitLab CI jobs to assume an IAM role via OIDC (no static keys).

Required setup:

1. Enable Amazon Bedrock and request access to your target Claude models
2. Create an IAM OIDC provider for GitLab if not already present
3. Create an IAM role trusted by the GitLab OIDC provider, restricted to your project and protected refs

4. Attach least-privilege permissions for Bedrock invoke APIs

Required values to store in CI/CD variables:

- `AWS_ROLE_TO_ASSUME`
- `AWS_REGION`

Add variables in Settings → CI/CD → Variables:

```
## For AWS Bedrock:
- AWS_ROLE_TO_ASSUME
- AWS_REGION
```

Use the AWS Bedrock job example above to exchange the GitLab job token for temporary AWS credentials at runtime.

Google Vertex AI

Prerequisites

Before setting up Claude Code with Google Vertex AI, you need:

1. A Google Cloud project with:

- Vertex AI API enabled
- Workload Identity Federation configured to trust GitLab OIDC

1. A dedicated service account with only the required Vertex AI roles
2. GitLab CI/CD variables for WIF:

- `GCP_WORKLOAD_IDENTITY_PROVIDER` (full resource name)
- `GCP_SERVICE_ACCOUNT` (service account email)

Setup instructions

Configure Google Cloud to allow GitLab CI jobs to impersonate a service account via Workload Identity Federation.

Required setup:

1. Enable IAM Credentials API, STS API, and Vertex AI API
2. Create a Workload Identity Pool and provider for GitLab OIDC
3. Create a dedicated service account with Vertex AI roles
4. Grant the WIF principal permission to impersonate the service account

Required values to store in CI/CD variables:

- `GCP_WORKLOAD_IDENTITY_PROVIDER`
- `GCP_SERVICE_ACCOUNT`

Add variables in Settings → CI/CD → Variables:

```
## For Google Vertex AI:
- GCP_WORKLOAD_IDENTITY_PROVIDER
- GCP_SERVICE_ACCOUNT
- CLOUD_ML_REGION (for example, us-east5)
```

Use the Google Vertex AI job example above to authenticate without storing keys.

Configuration examples

Below are ready-to-use snippets you can adapt to your pipeline.

Basic .gitlab-ci.yml (Claude API)

```
stages:
  - ai

claude:
  stage: ai
  image: node:24-alpine3.21
  rules:
    - if: '$CI_PIPELINE_SOURCE == "web"'
    - if: '$CI_PIPELINE_SOURCE == "merge_request_event"'
  variables:
    GIT_STRATEGY: fetch
  before_script:
    - apk update
    - apk add --no-cache git curl bash
    - curl -fsSL https://claude.ai/install.sh | bash
  script:
    - /bin/gitlab-mcp-server || true
    - >
      claude
      -p "${AI_FLOW_INPUT:-'Summarize recent changes and suggest improvements'}"
      --permission-mode acceptEdits
      --allowedTools "Bash Read Edit Write mcp__gitlab"
      --debug
  # Claude Code will use ANTHROPIC_API_KEY from CI/CD variables
```

AWS Bedrock job example (OIDC)

Prerequisites:

- Amazon Bedrock enabled with access to your chosen Claude model(s)
- GitLab OIDC configured in AWS with a role that trusts your GitLab project and refs
- IAM role with Bedrock permissions (least privilege recommended)

Required CI/CD variables:

- `AWS_ROLE_TO_ASSUME` : ARN of the IAM role for Bedrock access
- `AWS_REGION` : Bedrock region (for example, `us-west-2`)

```
claude-bedrock:
  stage: ai
  image: node:24-alpine3.21
  rules:
    - if: '$CI_PIPELINE_SOURCE == "web"'
  before_script:
    - apk add --no-cache bash curl jq git python3 py3-pip
    - pip install --no-cache-dir awscli
    - curl -fsSL https://claude.ai/install.sh | bash
    # Exchange GitLab OIDC token for AWS credentials
    - export AWS_WEB_IDENTITY_TOKEN_FILE="${CI_JOB_JWT_FILE:-/tmp/oidc_token}"
    - if [ -n "${CI_JOB_JWT_V2}" ]; then printf "%s" "$CI_JOB_JWT_V2" >
"$AWS_WEB_IDENTITY_TOKEN_FILE"; fi
    - >
      aws sts assume-role-with-web-identity
      --role-arn "$AWS_ROLE_TO_ASSUME"
      --role-session-name "gitlab-claude-$(date +%s)"
      --web-identity-token "file://$AWS_WEB_IDENTITY_TOKEN_FILE"
      --duration-seconds 3600 > /tmp/aws_creds.json
    - export AWS_ACCESS_KEY_ID="$(jq -r .Credentials.AccessKeyId /tmp/
aws_creds.json)"
    - export AWS_SECRET_ACCESS_KEY="$(jq -r .Credentials.SecretAccessKey /tmp/
aws_creds.json)"
    - export AWS_SESSION_TOKEN="$(jq -r .Credentials.SessionToken /tmp/
aws_creds.json)"
  script:
    - /bin/gitlab-mcp-server || true
    - >
      claude
      -p "${AI_FLOW_INPUT:-'Implement the requested changes and open an MR'}"
      --permission-mode acceptEdits
      --allowedTools "Bash Read Edit Write mcp__gitlab"
      --debug
  variables:
    AWS_REGION: "us-west-2"
```

Note:

> Model IDs for Bedrock include region-specific prefixes (for example, `us.anthropic.claude-sonnet-4-6`). Pass the desired model via your job configuration or prompt if your workflow supports it.

Google Vertex AI job example (Workload Identity Federation)

Prerequisites:

- Vertex AI API enabled in your GCP project
- Workload Identity Federation configured to trust GitLab OIDC
- A service account with Vertex AI permissions

Required CI/CD variables:

- `GCP_WORKLOAD_IDENTITY_PROVIDER`: Full provider resource name
- `GCP_SERVICE_ACCOUNT`: Service account email
- `CLOUD_ML_REGION`: Vertex region (for example, `us-east5`)

```
claude-vertex:
  stage: ai
  image: gcr.io/google.com/cloudsdktool/google-cloud-cli:slim
  rules:
    - if: '$CI_PIPELINE_SOURCE == "web"'
  before_script:
    - apt-get update && apt-get install -y git && apt-get clean
    - curl -fsSL https://claude.ai/install.sh | bash
    # Authenticate to Google Cloud via WIF (no downloaded keys)
    - >
      gcloud auth login --cred-file=<(cat <<EOF
      {
        "type": "external_account",
        "audience": "${GCP_WORKLOAD_IDENTITY_PROVIDER}",
        "subject_token_type": "urn:ietf:params:oauth:token-type:jwt",
        "service_account_impersonation_url": "https://
iamcredentials.googleapis.com/v1/projects/-/serviceAccounts/$
{GCP_SERVICE_ACCOUNT}:generateAccessToken",
        "token_url": "https://sts.googleapis.com/v1/token"
      }
      EOF
      )
    - gcloud config set project "$(gcloud projects list --
format='value(projectId)' --filter="name:${CI_PROJECT_NAMESPACE}" | head -n1)" ||
true
  script:
    - /bin/gitlab-mcp-server || true
    - >
      CLOUD_ML_REGION="${CLOUD_ML_REGION:-us-east5}"
      claude
      -p "${AI_FLOW_INPUT:-'Review and update code as requested'}"
      --permission-mode acceptEdits
      --allowedTools "Bash Read Edit Write mcp__gitlab"
      --debug
  variables:
    CLOUD_ML_REGION: "us-east5"
```

> **Note:**
>
> With Workload Identity Federation, you do not need to store service account keys. Use repository-specific trust conditions and least-privilege service accounts.

Best practices

CLAUDE.md configuration

Create a `CLAUDE.md` file at the repository root to define coding standards, review criteria, and project-specific rules. Claude reads this file during runs and follows your conventions when proposing changes.

Security considerations

Never commit API keys or cloud credentials to your repository. Always use GitLab CI/CD variables:

- Add `ANTHROPIC_API_KEY` as a masked variable (and protect it if needed)
- Use provider-specific OIDC where possible (no long-lived keys)
- Limit job permissions and network egress
- Review Claude's MRs like any other contributor

Optimizing performance

- Keep `CLAUDE.md` focused and concise
- Provide clear issue/MR descriptions to reduce iterations
- Configure sensible job timeouts to avoid runaway runs
- Cache npm and package installs in runners where possible

CI costs

When using Claude Code with GitLab CI/CD, be aware of associated costs:

- **GitLab Runner time**:
  - Claude runs on your GitLab runners and consumes compute minutes
  - See your GitLab plan's runner billing for details
- **API costs**:
  - Each Claude interaction consumes tokens based on prompt and response size
  - Token usage varies by task complexity and codebase size
  - See Anthropic pricing for details

- **Cost optimization tips**:
  - Use specific `@claude` commands to reduce unnecessary turns
  - Set appropriate `max_turns` and job timeout values
  - Limit concurrency to control parallel runs

Security and governance

- Each job runs in an isolated container with restricted network access
- Claude's changes flow through MRs so reviewers see every diff
- Branch protection and approval rules apply to AI-generated code
- Claude Code uses workspace-scoped permissions to constrain writes
- Costs remain under your control because you bring your own provider credentials

Troubleshooting

Claude not responding to @claude commands

- Verify your pipeline is being triggered (manually, MR event, or via a note event listener/webhook)
- Ensure CI/CD variables (`ANTHROPIC_API_KEY` or cloud provider settings) are present and unmasked
- Check that the comment contains `@claude` (not `/claude`) and that your mention trigger is configured

Job can't write comments or open MRs

- Ensure `CI_JOB_TOKEN` has sufficient permissions for the project, or use a Project Access Token with `api` scope
- Check the `mcp__gitlab` tool is enabled in `--allowedTools`
- Confirm the job runs in the context of the MR or has enough context via `AI_FLOW_*` variables

Authentication errors

- **For Claude API**: Confirm `ANTHROPIC_API_KEY` is valid and unexpired
- **For Bedrock/Vertex**: Verify OIDC/WIF configuration, role impersonation, and secret names; confirm region and model availability

Advanced configuration

Common parameters and variables

Claude Code supports these commonly used inputs:

- `prompt` / `prompt_file`: Provide instructions inline (`-p`) or via a file
- `max_turns`: Limit the number of back-and-forth iterations
- `timeout_minutes`: Limit total execution time
- `ANTHROPIC_API_KEY`: Required for the Claude API (not used for Bedrock/Vertex)
- Provider-specific environment: `AWS_REGION`, project/region vars for Vertex

> **Note:**
>
> Exact flags and parameters may vary by version of `@anthropic-ai/claude-code`. Run `claude --help` in your job to see supported options.

Customizing Claude's behavior

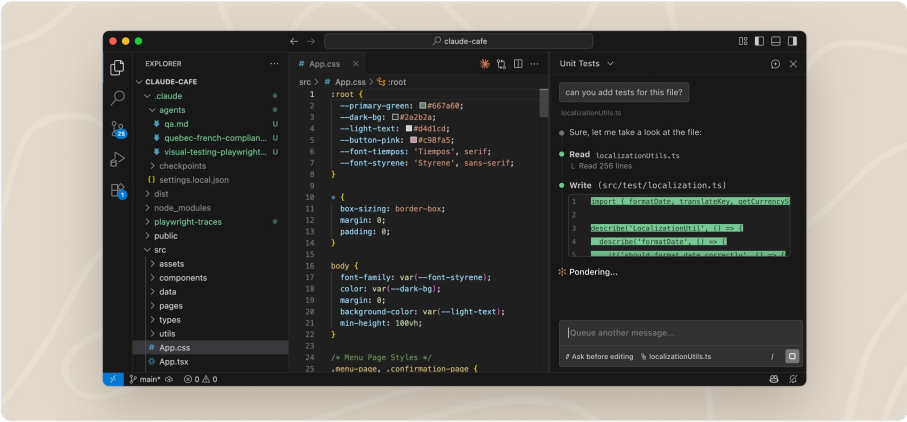
You can guide Claude in two primary ways:

1. **CLAUDE.md**: Define coding standards, security requirements, and project conventions. Claude reads this during runs and follows your rules.
2. **Custom prompts**: Pass task-specific instructions via `prompt` / `prompt_file` in the job. Use different prompts for different jobs (for example, review, implement, refactor).

Part 8: IDE & Platform Integration

Use Claude Code in VS Code

> *Install and configure the Claude Code extension for VS Code. Get AI coding assistance with inline diffs, @-mentions, plan review, and keyboard shortcuts.*



VS Code editor with the Claude Code extension panel open on the right side, showing a conversation with Claude

The VS Code extension provides a native graphical interface for Claude Code, integrated directly into your IDE. This is the recommended way to use Claude Code in VS Code.

With the extension, you can review and edit Claude's plans before accepting them, auto-accept edits as they're made, @-mention files with specific line ranges from your selection, access conversation history, and open multiple conversations in separate tabs or windows.

Prerequisites

Before installing, make sure you have:

- VS Code 1.98.0 or higher
- An Anthropic account (you'll sign in when you first open the extension). If you're using a third-party provider like Amazon Bedrock or Google Vertex AI, see Use third-party providers instead.

> **Tip:**
>
> The extension includes the CLI (command-line interface), which you can access from VS Code's integrated terminal for advanced features. See VS Code extension vs. Claude Code CLI for details.

Install the extension

Click the link for your IDE to install directly:

- Install for VS Code
- Install for Cursor

Or in VS Code, press `Cmd+Shift+X` (Mac) or `Ctrl+Shift+X` (Windows/Linux) to open the Extensions view, search for "Claude Code", and click **Install**.

> **Note:**
>
> If the extension doesn't appear after installation, restart VS Code or run "Developer: Reload Window" from the Command Palette.

Get started

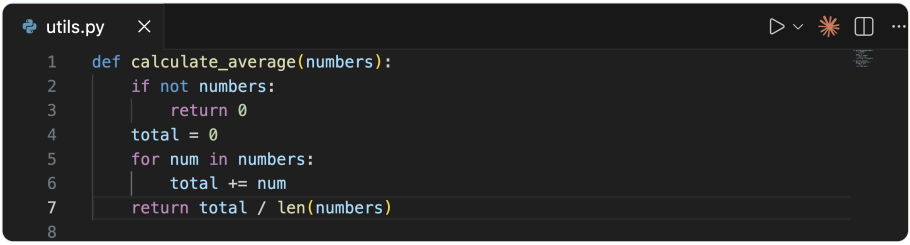
Once installed, you can start using Claude Code through the VS Code interface:

Step 1: Open the Claude Code panel

Throughout VS Code, the Spark icon indicates Claude Code:

Spark icon

The quickest way to open Claude is to click the Spark icon in the **Editor Toolbar** (top-right corner of the editor). The icon only appears when you have a file open.



VS Code editor showing the Spark icon in the Editor Toolbar

Other ways to open Claude Code:

- **Activity Bar**: click the Spark icon in the left sidebar to open the sessions list. Click any session to open it as a full editor tab, or start a new one. This icon is always visible in the Activity Bar.
- **Command Palette**: `Cmd+Shift+P` (Mac) or `Ctrl+Shift+P` (Windows/Linux), type "Claude Code", and select an option like "Open in New Tab"
- **Status Bar**: click **Claude Code** in the bottom-right corner of the window. This works even when no file is open.

When you first open the panel, a **Learn Claude Code** checklist appears. Work through each item by clicking **Show me**, or dismiss it with the X. To reopen it later, uncheck **Hide Onboarding** in VS Code settings under Extensions → Claude Code.

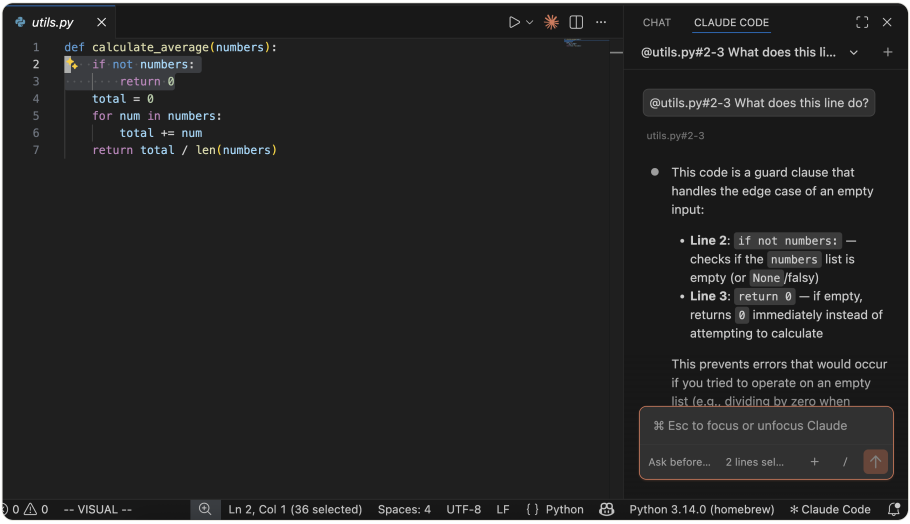
You can drag the Claude panel to reposition it anywhere in VS Code. See Customize your workflow for details.

Step 2: Send a prompt

Ask Claude to help with your code or files, whether that's explaining how something works, debugging an issue, or making changes.

> **Tip:**
>
> Claude automatically sees your selected text. Press `Option+K` (Mac) / `Alt+K` (Windows/Linux) to also insert an @-mention reference (like `@file.ts#5-10`) into your prompt.

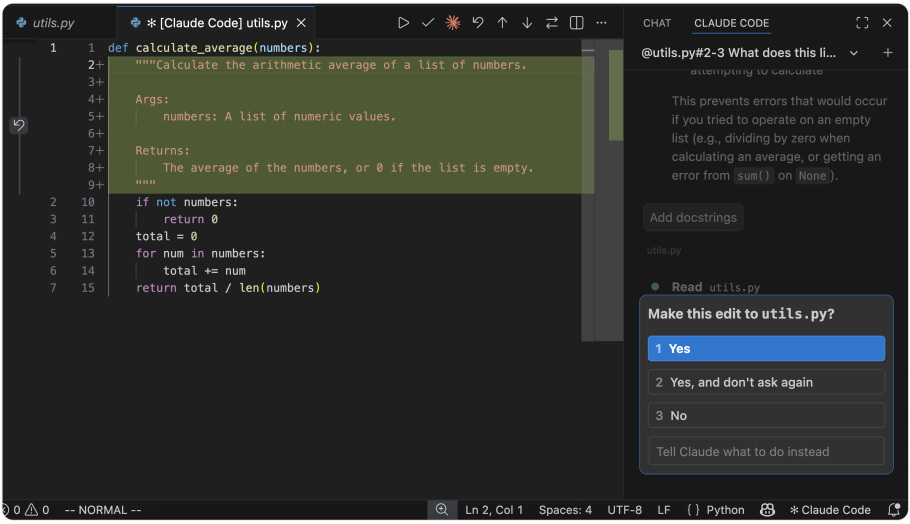
Here's an example of asking about a particular line in a file:



VS Code editor with lines 2-3 selected in a Python file, and the Claude Code panel showing a question about those lines with an @-mention reference

Step 3: Review changes

When Claude wants to edit a file, it shows a side-by-side comparison of the original and proposed changes, then asks for permission. You can accept, reject, or tell Claude what to do instead.



VS Code showing a diff of Claude's proposed changes with a permission prompt asking whether to make the edit

For more ideas on what you can do with Claude Code, see Common workflows.

> **Tip:**
>
> Run “Claude Code: Open Walkthrough” from the Command Palette for a guided tour of the basics.

Use the prompt box

The prompt box supports several features:

- **Permission modes**: click the mode indicator at the bottom of the prompt box to switch modes. In normal mode, Claude asks permission before each action. In Plan mode, Claude describes what it will do and waits for approval before making changes. VS Code automatically opens the plan as a full markdown document where you can add inline comments to give feedback before Claude begins. In auto-accept mode, Claude makes edits without asking. Set the default in VS Code settings under `claudeCode.initialPermissionMode`.
- **Command menu**: click `/` or type `/` to open the command menu. Options include attaching files, switching models, toggling extended thinking, and viewing plan usage (`/usage`). The Customize section provides access to MCP servers, hooks, memory, permissions, and plugins. Items with a terminal icon open in the integrated terminal.
- **Context indicator**: the prompt box shows how much of Claude’s context window you’re using. Claude automatically compacts when needed, or you can run `/compact` manually.
- **Extended thinking**: lets Claude spend more time reasoning through complex problems. Toggle it on via the command menu (`/`). See Extended thinking for details.
- **Multi-line input**: press `Shift+Enter` to add a new line without sending. This also works in the “Other” free-text input of question dialogs.

Reference files and folders

Use @-mentions to give Claude context about specific files or folders. When you type `@` followed by a file or folder name, Claude reads that content and can answer questions about it or make changes to it. Claude Code supports fuzzy matching, so you can type partial names to find what you need:

```
> Explain the logic in @auth (fuzzy matches auth.js, AuthService.ts, etc.)
> What's in @src/components/ (include a trailing slash for folders)
```

For large PDFs, you can ask Claude to read specific pages instead of the whole file: a single page, a range like pages 1-10, or an open-ended range like page 3 onward.

When you select text in the editor, Claude can see your highlighted code automatically. The prompt box footer shows how many lines are selected. Press `Option+K` (Mac) / `Alt+K` (Windows/Linux) to insert an @-mention with the file path and line numbers (e.g., `@app.ts#5-10`). Click the selection indicator to toggle whether Claude can see your highlighted text - the eye-slash icon means the selection is hidden from Claude.

You can also hold `Shift` while dragging files into the prompt box to add them as attachments. Click the X on any attachment to remove it from context.

Resume past conversations

Click the dropdown at the top of the Claude Code panel to access your conversation history. You can search by keyword or browse by time (Today, Yesterday, Last 7 days, etc.). Click any conversation to resume it with the full message history. Hover over a session to reveal rename and remove actions: rename to give it a descriptive title, or remove to delete it from the list. For more on resuming sessions, see Common workflows.

Resume remote sessions from Claude.ai

If you use Claude Code on the web, you can resume those remote sessions directly in VS Code. This requires signing in with **Claude.ai Subscription**, not Anthropic Console.

Step 1: Open Past Conversations

Click the **Past Conversations** dropdown at the top of the Claude Code panel.

Step 2: Select the Remote tab

The dialog shows two tabs: Local and Remote. Click **Remote** to see sessions from claude.ai.

Step 3: Select a session to resume

Browse or search your remote sessions. Click any session to download it and continue the conversation locally.

> **Note:**
>
> Only web sessions started with a GitHub repository appear in the Remote tab. Resuming loads the conversation history locally; changes are not synced back to claude.ai.

Customize your workflow

Once you're up and running, you can reposition the Claude panel, run multiple sessions, or switch to terminal mode.

Choose where Claude lives

You can drag the Claude panel to reposition it anywhere in VS Code. Grab the panel's tab or title bar and drag it to:

- **Secondary sidebar**: the right side of the window. Keeps Claude visible while you code.
- **Primary sidebar**: the left sidebar with icons for Explorer, Search, etc.
- **Editor area**: opens Claude as a tab alongside your files. Useful for side tasks.

> **Tip:**
>
> Use the sidebar for your main Claude session and open additional tabs for side tasks. Claude remembers your preferred location. The Activity Bar sessions list icon is separate from the Claude panel: the sessions list is always visible in the Activity Bar, while the Claude panel icon only appears there when the panel is docked to the left sidebar.

Run multiple conversations

Use **Open in New Tab** or **Open in New Window** from the Command Palette to start additional conversations. Each conversation maintains its own history and context, allowing you to work on different tasks in parallel.

When using tabs, a small colored dot on the spark icon indicates status: blue means a permission request is pending, orange means Claude finished while the tab was hidden.

Switch to terminal mode

By default, the extension opens a graphical chat panel. If you prefer the CLI-style interface, open the Use Terminal setting and check the box.

You can also open VS Code settings (`Cmd+,` on Mac or `Ctrl+,` on Windows/Linux), go to Extensions → Claude Code, and check **Use Terminal**.

Manage plugins

The VS Code extension includes a graphical interface for installing and managing plugins. Type `/plugins` in the prompt box to open the **Manage plugins** interface.

Install plugins

The plugin dialog shows two tabs: **Plugins** and **Marketplaces**.

In the Plugins tab:

- **Installed plugins** appear at the top with toggle switches to enable or disable them
- **Available plugins** from your configured marketplaces appear below
- Search to filter plugins by name or description
- Click **Install** on any available plugin

When you install a plugin, choose the installation scope:

- **Install for you**: available in all your projects (user scope)
- **Install for this project**: shared with project collaborators (project scope)
- **Install locally**: only for you, only in this repository (local scope)

Manage marketplaces

Switch to the **Marketplaces** tab to add or remove plugin sources:

- Enter a GitHub repo, URL, or local path to add a new marketplace
- Click the refresh icon to update a marketplace's plugin list
- Click the trash icon to remove a marketplace

After making changes, a banner prompts you to restart Claude Code to apply the updates.

> **Note:**
>
> Plugin management in VS Code uses the same CLI commands under the hood. Plugins and marketplaces you configure in the extension are also available in the CLI, and vice versa.

For more about the plugin system, see Plugins and Plugin marketplaces.

Automate browser tasks with Chrome

Connect Claude to your Chrome browser to test web apps, debug with console logs, and automate browser workflows without leaving VS Code. This requires the Claude in Chrome extension version 1.0.36 or higher.

Type `@browser` in the prompt box followed by what you want Claude to do:

```
@browser go to localhost:3000 and check the console for errors
```

You can also open the attachment menu to select specific browser tools like opening a new tab or reading page content.

Claude opens new tabs for browser tasks and shares your browser's login state, so it can access any site you're already signed into.

For setup instructions, the full list of capabilities, and troubleshooting, see Use Claude Code with Chrome.

VS Code commands and shortcuts

Open the Command Palette (`Cmd+Shift+P` on Mac or `Ctrl+Shift+P` on Windows/Linux) and type "Claude Code" to see all available VS Code commands for the Claude Code extension.

Some shortcuts depend on which panel is "focused" (receiving keyboard input). When your cursor is in a code file, the editor is focused. When your cursor is in Claude's prompt box, Claude is focused. Use `Cmd+Esc` / `Ctrl+Esc` to toggle between them.

> **Note:**
>
> These are VS Code commands for controlling the extension. Not all built-in Claude Code commands are available in the extension. See VS Code extension vs. Claude Code CLI for details.

Command	Shortcut	Description
Focus Input	`Cmd+Esc` (Mac) / `Ctrl+Esc` (Windows/Linux)	Toggle focus between editor and Claude
Open in Side Bar	-	Open Claude in the left sidebar
Open in Terminal	-	Open Claude in terminal mode
Open in New Tab	`Cmd+Shift+Esc` (Mac) / `Ctrl+Shift+Esc` (Windows/Linux)	Open a new conversation as an editor tab

Command	Shortcut	Description
Open in New Window	-	Open a new conversation in a separate window
New Conversation	`Cmd+N` (Mac) / `Ctrl+N` (Windows/Linux)	Start a new conversation (requires Claude to be focused)
Insert @-Mention Reference	`Option+K` (Mac) / `Alt+K` (Windows/Linux)	Insert a reference to the current file and selection (requires editor to be focused)
Show Logs	-	View extension debug logs
Logout	-	Sign out of your Anthropic account

Configure settings

The extension has two types of settings:

- **Extension settings** in VS Code: control the extension's behavior within VS Code. Open with `Cmd+,` (Mac) or `Ctrl+,` (Windows/Linux), then go to Extensions → Claude Code. You can also type `/` and select **General Config** to open settings.
- **Claude Code settings** in `~/.claude/settings.json`: shared between the extension and CLI. Use for allowed commands, environment variables, hooks, and MCP servers. See Settings for details.

> **Tip:**
>
> Add `"$schema": "https://json.schemastore.org/claude-code-settings.json"` to your `settings.json` to get autocomplete and inline validation for all available settings directly in VS Code.

Extension settings

Setting	Default	Description
`selectedModel`	`default`	Model for new conversations. Change per-session with `/model`.

Setting	Default	Description
`useTerminal`	`false`	Launch Claude in terminal mode instead of graphical panel
`initialPermissionMode`	`default`	Controls approval prompts: `default` (ask each time), `plan`, `acceptEdits`, or `bypassPermissions`
`preferredLocation`	`panel`	Where Claude opens: `sidebar` (right) or `panel` (new tab)
`autosave`	`true`	Auto-save files before Claude reads or writes them
`useCtrlEnterToSend`	`false`	Use Ctrl/Cmd+Enter instead of Enter to send prompts
`enableNewConversationShortcut`	`true`	Enable Cmd/Ctrl+N to start a new conversation
`hideOnboarding`	`false`	Hide the onboarding checklist (graduation cap icon)
`respectGitIgnore`	`true`	Exclude .gitignore patterns from file searches
`environmentVariables`	`[]`	Set environment variables for the Claude process. Use Claude Code settings instead for shared config.
`disableLoginPrompt`	`false`	Skip authentication prompts (for third-party provider setups)
`allowDangerouslySkipPermissions`	`false`	Bypass all permission prompts. **Use with extreme caution.**

Setting	Default	Description
`claudeProcessWrapper`	-	Executable path used to launch the Claude process

VS Code extension vs. Claude Code CLI

Claude Code is available as both a VS Code extension (graphical panel) and a CLI (command-line interface in the terminal). Some features are only available in the CLI. If you need a CLI-only feature, run `claude` in VS Code's integrated terminal.

Feature	CLI	VS Code Extension
Commands and skills	All	Subset (type `/` to see available)
MCP server config	Yes	Partial (add servers via CLI; manage existing servers with `/mcp` in the chat panel)
Checkpoints	Yes	Yes
`!` bash shortcut	Yes	No
Tab completion	Yes	No

Rewind with checkpoints

The VS Code extension supports checkpoints, which track Claude's file edits and let you rewind to a previous state. Hover over any message to reveal the rewind button, then choose from three options:

- **Fork conversation from here**: start a new conversation branch from this message while keeping all code changes intact
- **Rewind code to here**: revert file changes back to this point in the conversation while keeping the full conversation history
- **Fork conversation and rewind code**: start a new conversation branch and revert file changes to this point

For full details on how checkpoints work and their limitations, see Checkpointing.

Run CLI in VS Code

To use the CLI while staying in VS Code, open the integrated terminal (`Ctrl+`` ` on Windows/Linux or `Cmd+`` ` on Mac) and run `claude` . The CLI automatically integrates with your IDE for features like diff viewing and diagnostic sharing.

If using an external terminal, run `/ide` inside Claude Code to connect it to VS Code.

Switch between extension and CLI

The extension and CLI share the same conversation history. To continue an extension conversation in the CLI, run `claude --resume` in the terminal. This opens an interactive picker where you can search for and select your conversation.

Include terminal output in prompts

Reference terminal output in your prompts using `@terminal:name` where `name` is the terminal's title. This lets Claude see command output, error messages, or logs without copy-pasting.

Monitor background processes

When Claude runs long-running commands, the extension shows progress in the status bar. However, visibility for background tasks is limited compared to the CLI. For better visibility, have Claude output the command so you can run it in VS Code's integrated terminal.

Connect to external tools with MCP

MCP (Model Context Protocol) servers give Claude access to external tools, databases, and APIs.

To add an MCP server, open the integrated terminal (`Ctrl+`` ` or `Cmd+`` `) and run:

```
claude mcp add --transport http github https://api.githubcopilot.com/mcp/
```

Once configured, ask Claude to use the tools (e.g., "Review PR #456").

To manage MCP servers without leaving VS Code, type `/mcp` in the chat panel. The MCP management dialog lets you enable or disable servers, reconnect to a server, and manage OAuth authentication. See the MCP documentation for available servers.

Work with git

Claude Code integrates with git to help with version control workflows directly in VS Code. Ask Claude to commit changes, create pull requests, or work across branches.

Create commits and pull requests

Claude can stage changes, write commit messages, and create pull requests based on your work:

```
> commit my changes with a descriptive message
> create a pr for this feature
> summarize the changes I've made to the auth module
```

When creating pull requests, Claude generates descriptions based on the actual code changes and can add context about testing or implementation decisions.

Use git worktrees for parallel tasks

Use the `--worktree` (`-w`) flag to start Claude in an isolated worktree with its own files and branch:

```
claude --worktree feature-auth
```

Each worktree maintains independent file state while sharing git history. This prevents Claude instances from interfering with each other when working on different tasks. For more details, see Run parallel sessions with Git worktrees.

Use third-party providers

By default, Claude Code connects directly to Anthropic's API. If your organization uses Amazon Bedrock, Google Vertex AI, or Microsoft Foundry to access Claude, configure the extension to use your provider instead:

Step 1: Disable login prompt

Open the Disable Login Prompt setting and check the box.

You can also open VS Code settings (`Cmd+,` on Mac or `Ctrl+,` on Windows/Linux), search for "Claude Code login", and check **Disable Login Prompt**.

Step 2: Configure your provider

Follow the setup guide for your provider:

- Claude Code on Amazon Bedrock
- Claude Code on Google Vertex AI
- Claude Code on Microsoft Foundry

These guides cover configuring your provider in `~/.claude/settings.json`, which ensures your settings are shared between the VS Code extension and the CLI.

Security and privacy

Your code stays private. Claude Code processes your code to provide assistance but does not use it to train models. For details on data handling and how to opt out of logging, see Data and privacy.

With auto-edit permissions enabled, Claude Code can modify VS Code configuration files (like `settings.json` or `tasks.json`) that VS Code may execute automatically. To reduce risk when working with untrusted code:

- Enable VS Code Restricted Mode for untrusted workspaces
- Use manual approval mode instead of auto-accept for edits
- Review changes carefully before accepting them

Fix common issues

Extension won't install

- Ensure you have a compatible version of VS Code (1.98.0 or later)
- Check that VS Code has permission to install extensions
- Try installing directly from the VS Code Marketplace

Spark icon not visible

The Spark icon appears in the **Editor Toolbar** (top-right of editor) when you have a file open. If you don't see it:

1. **Open a file**: The icon requires a file to be open. Having just a folder open isn't enough.
2. **Check VS Code version**: Requires 1.98.0 or higher (Help → About)
3. **Restart VS Code**: Run "Developer: Reload Window" from the Command Palette
4. **Disable conflicting extensions**: Temporarily disable other AI extensions (Cline, Continue, etc.)

5. **Check workspace trust**: The extension doesn't work in Restricted Mode

Alternatively, click " Claude Code" in the **Status Bar** (bottom-right corner). This works even without a file open. You can also use the **Command Palette** (`Cmd+Shift+P` / `Ctrl+Shift+P`) and type "Claude Code".

Claude Code never responds

If Claude Code isn't responding to your prompts:

1. **Check your internet connection**: Ensure you have a stable internet connection
2. **Start a new conversation**: Try starting a fresh conversation to see if the issue persists
3. **Try the CLI**: Run `claude` from the terminal to see if you get more detailed error messages

If problems persist, file an issue on GitHub with details about the error.

Uninstall the extension

To uninstall the Claude Code extension:

1. Open the Extensions view (`Cmd+Shift+X` on Mac or `Ctrl+Shift+X` on Windows/Linux)
2. Search for "Claude Code"
3. Click **Uninstall**

To also remove extension data and reset all settings:

```
rm -rf ~/.vscode/globalStorage/anthropic.claude-code
```

For additional help, see the troubleshooting guide.

Next steps

Now that you have Claude Code set up in VS Code:

- Explore common workflows to get the most out of Claude Code
- Set up MCP servers to extend Claude's capabilities with external tools. Add servers using the CLI, then manage them with `/mcp` in the chat panel.
- Configure Claude Code settings to customize allowed commands, hooks, and more. These settings are shared between the extension and CLI.

JetBrains IDEs

> *Use Claude Code with JetBrains IDEs including IntelliJ, PyCharm, WebStorm, and more*

Claude Code integrates with JetBrains IDEs through a dedicated plugin, providing features like interactive diff viewing, selection context sharing, and more.

Supported IDEs

The Claude Code plugin works with most JetBrains IDEs, including:

- IntelliJ IDEA
- PyCharm
- Android Studio
- WebStorm
- PhpStorm
- GoLand

Features

- **Quick launch**: Use `Cmd+Esc` (Mac) or `Ctrl+Esc` (Windows/Linux) to open Claude Code directly from your editor, or click the Claude Code button in the UI
- **Diff viewing**: Code changes can be displayed directly in the IDE diff viewer instead of the terminal
- **Selection context**: The current selection/tab in the IDE is automatically shared with Claude Code
- **File reference shortcuts**: Use `Cmd+Option+K` (Mac) or `Alt+Ctrl+K` (Linux/Windows) to insert file references (for example, @File#L1-99)
- **Diagnostic sharing**: Diagnostic errors (lint, syntax, etc.) from the IDE are automatically shared with Claude as you work

Installation

Marketplace Installation

Find and install the Claude Code plugin from the JetBrains marketplace and restart your IDE.

If you haven't installed Claude Code yet, see our quickstart guide for installation instructions.

> **Note:**
>
> After installing the plugin, you may need to restart your IDE completely for it to take effect.

Usage

From Your IDE

Run `claude` from your IDE's integrated terminal, and all integration features will be active.

From External Terminals

Use the `/ide` command in any external terminal to connect Claude Code to your JetBrains IDE and activate all features:

```
claude
```

```
/ide
```

If you want Claude to have access to the same files as your IDE, start Claude Code from the same directory as your IDE project root.

Configuration

Claude Code Settings

Configure IDE integration through Claude Code's settings:

1. Run `claude`
2. Enter the `/config` command

3. Set the diff tool to `auto` for automatic IDE detection

Plugin Settings

Configure the Claude Code plugin by going to **Settings → Tools → Claude Code [Beta]**:

General Settings

- **Claude command**: Specify a custom command to run Claude (for example, `claude`, `/usr/local/bin/claude`, or `npx @anthropic/claude`)
- **Suppress notification for Claude command not found**: Skip notifications about not finding the Claude command
- **Enable using Option+Enter for multi-line prompts** (macOS only): When enabled, Option+Enter inserts new lines in Claude Code prompts. Disable if experiencing issues with the Option key being captured unexpectedly (requires terminal restart)
- **Enable automatic updates**: Automatically check for and install plugin updates (applied on restart)

> **Tip:**
>
> For WSL users: Set `wsl -d Ubuntu -- bash -lic "claude"` as your Claude command (replace `Ubuntu` with your WSL distribution name)

ESC Key Configuration

If the ESC key doesn't interrupt Claude Code operations in JetBrains terminals:

1. Go to **Settings → Tools → Terminal**
2. Either:
   - Uncheck "Move focus to the editor with Escape", or
   - Click "Configure terminal keybindings" and delete the "Switch focus to Editor" shortcut
3. Apply the changes

This allows the ESC key to properly interrupt Claude Code operations.

Special Configurations

Remote Development

> **Warning:**
>
> When using JetBrains Remote Development, you must install the plugin in the remote host via **Settings → Plugin (Host)**.

The plugin must be installed on the remote host, not on your local client machine.

WSL Configuration

> **Warning:**
>
> WSL users may need additional configuration for IDE detection to work properly. See our WSL troubleshooting guide for detailed setup instructions.

WSL configuration may require:

- Proper terminal configuration
- Networking mode adjustments
- Firewall settings updates

Troubleshooting

Plugin Not Working

- Ensure you're running Claude Code from the project root directory
- Check that the JetBrains plugin is enabled in the IDE settings
- Completely restart the IDE (you may need to do this multiple times)
- For Remote Development, ensure the plugin is installed in the remote host

IDE Not Detected

- Verify the plugin is installed and enabled
- Restart the IDE completely
- Check that you're running Claude Code from the integrated terminal
- For WSL users, see the WSL troubleshooting guide

Command Not Found

If clicking the Claude icon shows “command not found”:

1. Verify Claude Code is installed: `npm list -g @anthropic-ai/claude-code`
2. Configure the Claude command path in plugin settings
3. For WSL users, use the WSL command format mentioned in the configuration section

Security Considerations

When Claude Code runs in a JetBrains IDE with auto-edit permissions enabled, it may be able to modify IDE configuration files that can be automatically executed by your IDE. This may increase the risk of running Claude Code in auto-edit mode and allow bypassing Claude Code’s permission prompts for bash execution.

When running in JetBrains IDEs, consider:

- Using manual approval mode for edits
- Taking extra care to ensure Claude is only used with trusted prompts
- Being aware of which files Claude Code has access to modify

For additional help, see our troubleshooting guide.

Use Claude Code Desktop

> *Get more out of Claude Code Desktop: parallel sessions with Git isolation, visual diff review, app previews, PR monitoring, permission modes, connectors, and enterprise configuration.*

The Code tab within the Claude Desktop app lets you use Claude Code through a graphical interface instead of the terminal.

Desktop adds these capabilities on top of the standard Claude Code experience:

- Visual diff review with inline comments
- Live app preview with dev servers
- GitHub PR monitoring with auto-fix and auto-merge
- Parallel sessions with automatic Git worktree isolation
- Scheduled tasks that run Claude on a recurring schedule
- Connectors for GitHub, Slack, Linear, and more
- Local, SSH, and cloud environments

> **Tip:**
>
> New to Desktop? Start with Get started to install the app and make your first edit.

This page covers working with code, managing sessions, extending Claude Code, scheduled tasks, and configuration. It also includes a CLI comparison and troubleshooting.

Start a session

Before you send your first message, configure four things in the prompt area:

- **Environment**: choose where Claude runs. Select **Local** for your machine, **Remote** for Anthropic-hosted cloud sessions, or an **SSH connection** for a remote machine you manage. See environment configuration.
- **Project folder**: select the folder or repository Claude works in. For remote sessions, you can add multiple repositories.
- **Model**: pick a model from the dropdown next to the send button. The model is locked once the session starts.

- **Permission mode**: choose how much autonomy Claude has from the mode selector. You can change this during the session.

Type your task and press **Enter** to start. Each session tracks its own context and changes independently.

Work with code

Give Claude the right context, control how much it does on its own, and review what it changed.

Use the prompt box

Type what you want Claude to do and press **Enter** to send. Claude reads your project files, makes changes, and runs commands based on your permission mode. You can interrupt Claude at any point: click the stop button or type your correction and press **Enter**. Claude stops what it's doing and adjusts based on your input.

The + button next to the prompt box gives you access to file attachments, skills, connectors, and plugins.

Add files and context to prompts

The prompt box supports two ways to bring in external context:

- **@mention files**: type `@` followed by a filename to add a file to the conversation context. Claude can then read and reference that file.
- **Attach files**: attach images, PDFs, and other files to your prompt using the attachment button, or drag and drop files directly into the prompt. This is useful for sharing screenshots of bugs, design mockups, or reference documents.

Choose a permission mode

Permission modes control how much autonomy Claude has during a session: whether it asks before editing files, running commands, or both. You can switch modes at any time using the mode selector next to the send button. Start with Ask permissions to see exactly what Claude does, then move to Auto accept edits or Plan mode as you get comfortable.

Mode	Settings key	Behavior
Ask permissions	`default`	Claude asks before editing files or running commands. You see a diff and can accept or reject each change. Recommended for new users.
Auto accept edits	`acceptEdits`	Claude auto-accepts file edits but still asks before running terminal commands. Use this when you trust file changes and want faster iteration.
Plan mode	`plan`	Claude analyzes your code and creates a plan without modifying files or running commands. Good for complex tasks where you want to review the approach first.
Bypass permissions	`bypassPermissions`	Claude runs without any permission prompts, equivalent to `--dangerously-skip-permissions` in the CLI. Enable in your Settings → Claude Code under "Allow bypass permissions mode". Only use this in sandboxed containers or VMs. Enterprise admins can disable this option.

The `dontAsk` permission mode is available only in the CLI.

> **Tip:**
>
> Start complex tasks in Plan mode so Claude maps out an approach before making changes. Once you approve the plan, switch to Auto accept edits or Ask permissions to execute it. See explore first, then plan, then code for more on this workflow.

Remote sessions support Auto accept edits and Plan mode. Ask permissions is not available because remote sessions auto-accept file edits by default, and Bypass permissions is not available because the remote environment is already sandboxed.

Enterprise admins can restrict which permission modes are available. See enterprise configuration for details.

Preview your app

Claude can start a dev server and open an embedded browser to verify its changes. This works for frontend web apps as well as backend servers: Claude can test API endpoints, view server logs, and iterate on issues it finds. In most cases, Claude starts the server automatically after editing project files. You can also ask Claude to preview at any time. By default, Claude auto-verifies changes after every edit.

From the preview panel, you can:

- Interact with your running app directly in the embedded browser
- Watch Claude verify its own changes automatically: it takes screenshots, inspects the DOM, clicks elements, fills forms, and fixes issues it finds
- Start or stop servers from the **Preview** dropdown in the session toolbar
- Persist cookies and local storage across server restarts by selecting **Persist sessions** in the dropdown, so you don't have to re-login during development
- Edit the server configuration or stop all servers at once

Claude creates the initial server configuration based on your project. If your app uses a custom dev command, edit `.claude/launch.json` to match your setup. See Configure preview servers for the full reference.

To clear saved session data, toggle **Persist preview sessions** off in Settings → Claude Code. To disable preview entirely, toggle off **Preview** in Settings → Claude Code.

Review changes with diff view

After Claude makes changes to your code, the diff view lets you review modifications file by file before creating a pull request.

When Claude changes files, a diff stats indicator appears showing the number of lines added and removed, such as `+12 -1`. Click this indicator to open the diff viewer, which displays a file list on the left and the changes for each file on the right.

To comment on specific lines, click any line in the diff to open a comment box. Type your feedback and press **Enter** to add the comment. After adding comments to multiple lines, submit all comments at once:

- **macOS**: press **Cmd+Enter**
- **Windows**: press **Ctrl+Enter**

Claude reads your comments and makes the requested changes, which appear as a new diff you can review.

Review your code

In the diff view, click **Review code** in the top-right toolbar to ask Claude to evaluate the changes before you commit. Claude examines the current diffs and leaves comments directly in the diff view. You can respond to any comment or ask Claude to revise.

The review focuses on high-signal issues: compile errors, definite logic errors, security vulnerabilities, and obvious bugs. It does not flag style, formatting, pre-existing issues, or anything a linter would catch.

Monitor pull request status

After you open a pull request, a CI status bar appears in the session. Claude Code uses the GitHub CLI to poll check results and surface failures.

- **Auto-fix**: when enabled, Claude automatically attempts to fix failing CI checks by reading the failure output and iterating.
- **Auto-merge**: when enabled, Claude merges the PR once all checks pass. The merge method is squash. Auto-merge must be enabled in your GitHub repository settings for this to work.

Use the **Auto-fix** and **Auto-merge** toggles in the CI status bar to enable either option. Claude Code also sends a desktop notification when CI finishes.

> **Note:**
>
> PR monitoring requires the GitHub CLI (`gh`) to be installed and authenticated on your machine. If `gh` is not installed, Desktop prompts you to install it the first time you try to create a PR.

Manage sessions

Each session is an independent conversation with its own context and changes. You can run multiple sessions in parallel or send work to the cloud.

Work in parallel with sessions

Click + **New session** in the sidebar to work on multiple tasks in parallel. For Git repositories, each session gets its own isolated copy of your project using Git worktrees, so changes in one session don't affect other sessions until you commit them.

Worktrees are stored in `<project-root>/.claude/worktrees/` by default. You can change this to a custom directory in Settings → Claude Code under "Worktree location". You can also set a branch prefix that gets prepended to every worktree branch name, which is useful for keeping Claude-created branches organized. To remove a worktree when you're done, hover over the session in the sidebar and click the archive icon.

> **Note:**
>
> Session isolation requires Git. Most Macs include Git by default. Run `git --version` in Terminal to check. On Windows, Git is required for the Code tab to work: download Git for Windows, install it, and restart the app. If you run into Git errors, try a Cowork session to help troubleshoot your setup.

Use the filter icon at the top of the sidebar to filter sessions by status (Active, Archived) and environment (Local, Cloud). To rename a session or check context usage, click the session title in the toolbar at the top of the active session. When context fills up, Claude automatically summarizes the conversation and continues working. You can also type `/compact` to trigger summarization earlier and free up context space. See the context window for details on how compaction works.

Run long-running tasks remotely

For large refactors, test suites, migrations, or other long-running tasks, select **Remote** instead of **Local** when starting a session. Remote sessions run on Anthropic's cloud infrastructure and continue even if you close the app or shut down your computer. Check back anytime to see progress or steer Claude in a different direction. You can also monitor remote sessions from claude.ai/code or the Claude iOS app.

Remote sessions also support multiple repositories. After selecting a cloud environment, click the + button next to the repo pill to add additional repositories to the session. Each repo gets its own branch selector. This is useful for tasks that span multiple codebases, such as updating a shared library and its consumers.

See Claude Code on the web for more on how remote sessions work.

Continue in another surface

The **Continue in** menu, accessible from the VS Code icon in the bottom right of the session toolbar, lets you move your session to another surface:

- **Claude Code on the Web**: sends your local session to continue running remotely. Desktop pushes your branch, generates a summary of the conversation, and creates a new remote session with the full context. You can then choose to archive the local session or keep it. This requires a clean working tree, and is not available for SSH sessions.
- **Your IDE**: opens your project in a supported IDE at the current working directory.

Extend Claude Code

Connect external services, add reusable workflows, customize Claude's behavior, and configure preview servers.

Connect external tools

For local and SSH sessions, click the + button next to the prompt box and select **Connectors** to add integrations like Google Calendar, Slack, GitHub, Linear, Notion, and more. You can add connectors before or during a session. Connectors are not available for remote sessions.

To manage or disconnect connectors, go to Settings → Connectors in the desktop app, or select **Manage connectors** from the Connectors menu in the prompt box.

Once connected, Claude can read your calendar, send messages, create issues, and interact with your tools directly. You can ask Claude what connectors are configured in your session.

Connectors are MCP servers with a graphical setup flow. Use them for quick integration with supported services. For integrations not listed in Connectors, add MCP servers manually via settings files. You can also create custom connectors.

Use skills

Skills extend what Claude can do. Claude loads them automatically when relevant, or you can invoke one directly: type `/` in the prompt box or click the + button and select **Slash commands** to browse what's available. This includes built-in commands, your custom skills, project skills from your codebase, and skills from any installed plugins. Select one and it appears highlighted in the input field. Type your task after it and send as usual.

Install plugins

Plugins are reusable packages that add skills, agents, hooks, MCP servers, and LSP configurations to Claude Code. You can install plugins from the desktop app without using the terminal.

For local and SSH sessions, click the + button next to the prompt box and select **Plugins** to see your installed plugins and their commands. To add a plugin, select **Add plugin** from the submenu to open the plugin browser, which shows available plugins from your configured marketplaces including the official Anthropic marketplace. Select **Manage plugins** to enable, disable, or uninstall plugins.

Plugins can be scoped to your user account, a specific project, or local-only. Plugins are not available for remote sessions. For the full plugin reference including creating your own plugins, see plugins.

Configure preview servers

Claude automatically detects your dev server setup and stores the configuration in `.claude/launch.json` at the root of the folder you selected when starting the session. Preview uses this folder as its working directory, so if you selected a parent folder, subfolders with their own dev servers won't be detected automatically. To work with a subfolder's server, either start a session in that folder directly or add a configuration manually.

To customize how your server starts, for example to use `yarn dev` instead of `npm run dev` or to change the port, edit the file manually or click **Edit configuration** in the Preview dropdown to open it in your code editor. The file supports JSON with comments.

```
{
  "version": "0.0.1",
  "configurations": [
    {
      "name": "my-app",
      "runtimeExecutable": "npm",
      "runtimeArgs": ["run", "dev"],
      "port": 3000
    }
  ]
}
```

You can define multiple configurations to run different servers from the same project, such as a frontend and an API. See the examples below.

Auto-verify changes

When `autoVerify` is enabled, Claude automatically verifies code changes after editing files. It takes screenshots, checks for errors, and confirms changes work before completing its response.

Auto-verify is on by default. Disable it per-project by adding `"autoVerify": false` to `.claude/launch.json`, or toggle it from the **Preview** dropdown menu.

```
{
  "version": "0.0.1",
  "autoVerify": false,
  "configurations": [...]
}
```

When disabled, preview tools are still available and you can ask Claude to verify at any time. Auto-verify makes it automatic after every edit.

Configuration fields

Each entry in the `configurations` array accepts the following fields:

Field	Type	Description
`name`	string	A unique identifier for this server
`runtimeExecutable`	string	The command to run, such as `npm`, `yarn`, or `node`
`runtimeArgs`	string[]	Arguments passed to `runtimeExecutable`, such as `["run", "dev"]`
`port`	number	The port your server listens on. Defaults to 3000

Field	Type	Description
`cwd`	string	Working directory relative to your project root. Defaults to the project root. Use `${workspaceFolder}` to reference the project root explicitly
`env`	object	Additional environment variables as key-value pairs, such as `{ "NODE_ENV": "development" }`. Don't put secrets here since this file is committed to your repo. Secrets set in your shell profile are inherited automatically.
`autoPort`	boolean	How to handle port conflicts. See below
`program`	string	A script to run with `node`. See when to use `program` vs `runtimeExecutable`
`args`	string[]	Arguments passed to `program`. Only used when `program` is set

When to use `program` vs `runtimeExecutable`

Use `runtimeExecutable` with `runtimeArgs` to start a dev server through a package manager. For example, `"runtimeExecutable": "npm"` with `"runtimeArgs": ["run", "dev"]` runs `npm run dev`.

Use `program` when you have a standalone script you want to run with `node` directly. For example, `"program": "server.js"` runs `node server.js`. Pass additional flags with `args`.

Port conflicts

The `autoPort` field controls what happens when your preferred port is already in use:

- `true` : Claude finds and uses a free port automatically. Suitable for most dev servers.
- `false` : Claude fails with an error. Use this when your server must use a specific port, such as for OAuth callbacks or CORS allowlists.
- **Not set (default)**: Claude asks whether the server needs that exact port, then saves your answer.

When Claude picks a different port, it passes the assigned port to your server via the `PORT` environment variable.

Examples

These configurations show common setups for different project types:

Next.js

This configuration runs a Next.js app using Yarn on port 3000:

```
{
  "version": "0.0.1",
  "configurations": [
    {
      "name": "web",
      "runtimeExecutable": "yarn",
      "runtimeArgs": ["dev"],
      "port": 3000
    }
  ]
}
```

Multiple servers

For a monorepo with a frontend and an API server, define multiple configurations. The frontend uses `autoPort: true` so it picks a free port if 3000 is taken, while the API server requires port 8080 exactly:

```
{
  "version": "0.0.1",
  "configurations": [
    {
      "name": "frontend",
      "runtimeExecutable": "npm",
      "runtimeArgs": ["run", "dev"],
      "cwd": "apps/web",
      "port": 3000,
      "autoPort": true
    },
    {
      "name": "api",
      "runtimeExecutable": "npm",
      "runtimeArgs": ["run", "start"],
      "cwd": "server",
      "port": 8080,
      "env": { "NODE_ENV": "development" },
      "autoPort": false
    }
  ]
}
```

Node.js script

To run a Node.js script directly instead of using a package manager command, use the `program` field:

```
{
  "version": "0.0.1",
  "configurations": [
    {
      "name": "server",
      "program": "server.js",
      "args": ["--verbose"],
      "port": 4000
    }
  ]
}
```

Schedule recurring tasks

Scheduled tasks start a new local session automatically at a time and frequency you choose. Use them for recurring work like daily code reviews, dependency update checks, or morning briefings that pull from your calendar and inbox.

Tasks run on your machine, so the desktop app must be open and your computer awake for them to fire. See How scheduled tasks run for details on missed runs and catch-up behavior.

> **Note:**
>
> By default, scheduled tasks run against whatever state your working directory is in, including uncommitted changes. Enable the worktree toggle in the prompt input to give each run its own isolated Git worktree, the same way parallel sessions work.

To create a scheduled task, click **Schedule** in the sidebar, then **+ New task**. Configure these fields:

Field	Description
Name	Identifier for the task. Converted to lowercase kebab-case and used as the folder name on disk. Must be unique across your tasks.
Description	Short summary shown in the task list.

Field	Description
Prompt	The instructions sent to Claude when the task runs. Write this the same way you'd write any message in the prompt box. The prompt input also includes controls for model, permission mode, working folder, and worktree.
Frequency	How often the task runs. See frequency options below.

You can also create a task by describing what you want in any session. For example, "set up a daily code review that runs every morning at 9am."

Frequency options

- **Manual**: no schedule, only runs when you click **Run now**. Useful for saving a prompt you trigger on demand
- **Hourly**: runs every hour. Each task gets a fixed offset of up to 10 minutes from the top of the hour to stagger API traffic
- **Daily**: shows a time picker, defaults to 9:00 AM local time
- **Weekdays**: same as Daily but skips Saturday and Sunday
- **Weekly**: shows a time picker and a day picker

For intervals the picker doesn't offer (every 15 minutes, first of each month, etc.), ask Claude in any Desktop session to set the schedule. Use plain language; for example, "schedule a task to run all the tests every 6 hours."

How scheduled tasks run

Scheduled tasks run locally on your machine. Desktop checks the schedule every minute while the app is open and starts a fresh session when a task is due, independent of any manual sessions you have open. Each task gets a fixed delay of up to 10 minutes after the scheduled time to stagger API traffic. The delay is deterministic: the same task always starts at the same offset.

When a task fires, you get a desktop notification and a new session appears under a **Scheduled** section in the sidebar. Open it to see what Claude did, review changes, or respond to permission prompts. The session works like any other: Claude can edit files, run commands, create commits, and open pull requests.

Tasks only run while the desktop app is running and your computer is awake. If your computer sleeps through a scheduled time, the run is skipped. To prevent idle-sleep, enable **Keep computer awake** in Settings under **Desktop app → General**. Closing the laptop lid still puts it to sleep.

Missed runs

When the app starts or your computer wakes, Desktop checks whether each task missed any runs in the last seven days. If it did, Desktop starts exactly one catch-up run for the most recently missed time and discards anything older. A daily task that missed six days runs once on wake. Desktop shows a notification when a catch-up run starts.

Keep this in mind when writing prompts. A task scheduled for 9am might run at 11pm if your computer was asleep all day. If timing matters, add guardrails to the prompt itself, for example: "Only review today's commits. If it's after 5pm, skip the review and just post a summary of what was missed."

Permissions for scheduled tasks

Each task has its own permission mode, which you set when creating or editing the task. Allow rules from `~/.claude/settings.json` also apply to scheduled task sessions. If a task runs in Ask mode and needs to run a tool it doesn't have permission for, the run stalls until you approve it. The session stays open in the sidebar so you can answer later.

To avoid stalls, click **Run now** after creating a task, watch for permission prompts, and select "always allow" for each one. Future runs of that task auto-approve the same tools without prompting. You can review and revoke these approvals from the task's detail page.

Manage scheduled tasks

Click a task in the **Schedule** list to open its detail page. From here you can:

- **Run now**: start the task immediately without waiting for the next scheduled time
- **Toggle repeats**: pause or resume scheduled runs without deleting the task
- **Edit**: change the prompt, frequency, folder, or other settings
- **Review history**: see every past run, including ones that were skipped because your computer was asleep
- **Review allowed permissions**: see and revoke saved tool approvals for this task from the **Always allowed** panel
- **Delete**: remove the task and archive all sessions it created

You can also manage tasks by asking Claude in any Desktop session. For example, "pause my dependency-audit task", "delete the standup-prep task", or "show me my scheduled tasks."

To edit a task's prompt on disk, open `~/.claude/scheduled-tasks/<task-name>/SKILL.md` (or under `CLAUDE_CONFIG_DIR` if set). The file uses YAML frontmatter for `name` and `description`, with the prompt as the body. Changes take effect on the next run. Schedule, folder, model, and enabled state are not in this file: change them through the Edit form or ask Claude.

Environment configuration

The environment you pick when starting a session determines where Claude executes and how you connect:

- **Local**: runs on your machine with direct access to your files
- **Remote**: runs on Anthropic's cloud infrastructure. Sessions continue even if you close the app.
- **SSH**: runs on a remote machine you connect to over SSH, such as your own servers, cloud VMs, or dev containers

Local sessions

Local sessions inherit environment variables from your shell. If you need additional variables, set them in your shell profile, such as `~/.zshrc` or `~/.bashrc`, and restart the desktop app. See environment variables for the full list of supported variables.

Extended thinking is enabled by default, which improves performance on complex reasoning tasks but uses additional tokens. To disable thinking entirely, set `MAX_THINKING_TOKENS=0` in your shell profile. On Opus, `MAX_THINKING_TOKENS` is ignored except for `0` because adaptive reasoning controls thinking depth instead.

Remote sessions

Remote sessions continue in the background even if you close the app. Usage counts toward your subscription plan limits with no separate compute charges.

You can create custom cloud environments with different network access levels and environment variables. Select the environment dropdown when starting a remote session and choose **Add environment**. See cloud environments for details on configuring network access and environment variables.

SSH sessions

SSH sessions let you run Claude Code on a remote machine while using the desktop app as your interface. This is useful for working with codebases that live on cloud VMs, dev containers, or servers with specific hardware or dependencies.

To add an SSH connection, click the environment dropdown before starting a session and select **+ Add SSH connection**. The dialog asks for:

- **Name**: a friendly label for this connection
- **SSH Host**: `user@hostname` or a host defined in `~/.ssh/config`
- **SSH Port**: defaults to 22 if left empty, or uses the port from your SSH config
- **Identity File**: path to your private key, such as `~/.ssh/id_rsa`. Leave empty to use the default key or your SSH config.

Once added, the connection appears in the environment dropdown. Select it to start a session on that machine. Claude runs on the remote machine with access to its files and tools.

Claude Code must be installed on the remote machine. Once connected, SSH sessions support permission modes, connectors, plugins, and MCP servers.

Enterprise configuration

Organizations on Teams or Enterprise plans can manage desktop app behavior through admin console controls, managed settings files, and device management policies.

Admin console controls

These settings are configured through the admin settings console:

- **Enable or disable the Code tab**: control whether users in your organization can access Claude Code in the desktop app
- **Disable Bypass permissions mode**: prevent users in your organization from enabling bypass permissions mode
- **Disable Claude Code on the web**: enable or disable remote sessions for your organization

Managed settings

Managed settings override project and user settings and apply when Desktop spawns CLI sessions. You can set these keys in your organization's managed settings file or push them remotely through the admin console.

Key	Description
`disableBypassPermissionsMode`	set to `"disable"` to prevent users from enabling Bypass permissions mode. See managed settings.

For the complete list of managed-only settings including `allowManagedPermissionRulesOnly` and `allowManagedHooksOnly`, see managed-only settings.

Remote managed settings uploaded through the admin console currently apply to CLI and IDE sessions only. For Desktop-specific restrictions, use the admin console controls above.

Device management policies

IT teams can manage the desktop app through MDM on macOS or group policy on Windows. Available policies include enabling or disabling the Claude Code feature, controlling auto-updates, and setting a custom deployment URL.

- **macOS**: configure via `com.anthropic.Claude` preference domain using tools like Jamf or Kandji
- **Windows**: configure via registry at `SOFTWARE\Policies\Claude`

Authentication and SSO

Enterprise organizations can require SSO for all users. See authentication for plan-level details and Setting up SSO for SAML and OIDC configuration.

Data handling

Claude Code processes your code locally in local sessions or on Anthropic's cloud infrastructure in remote sessions. Conversations and code context are sent to Anthropic's API for processing. See data handling for details on data retention, privacy, and compliance.

Deployment

Desktop can be distributed through enterprise deployment tools:

- **macOS**: distribute via MDM such as Jamf or Kandji using the `.dmg` installer
- **Windows**: deploy via MSIX package or `.exe` installer. See Deploy Claude Desktop for Windows for enterprise deployment options including silent installation

For network configuration such as proxy settings, firewall allowlisting, and LLM gateways, see network configuration.

For the full enterprise configuration reference, see the enterprise configuration guide.

Coming from the CLI?

If you already use the Claude Code CLI, Desktop runs the same underlying engine with a graphical interface. You can run both simultaneously on the same machine, even on the same project. Each maintains separate session history, but they share configuration and project memory via CLAUDE.md files.

To move a CLI session into Desktop, run `/desktop` in the terminal. Claude saves your session and opens it in the desktop app, then exits the CLI. This command is available on macOS and Windows only.

> **Tip:**
>
> When to use Desktop vs CLI: use Desktop when you want visual diff review, file attachments, or session management in a sidebar. Use the CLI when you need scripting, automation, third-party providers, or prefer a terminal workflow.

CLI flag equivalents

This table shows the desktop app equivalent for common CLI flags. Flags not listed have no desktop equivalent because they are designed for scripting or automation.

CLI	Desktop equivalent
`--model sonnet`	model dropdown next to the send button, before starting a session
`--resume`, `--continue`	click a session in the sidebar
`--permission-mode`	mode selector next to the send button
`--dangerously-skip-permissions`	Bypass permissions mode. Enable in Settings → Claude Code → "Allow bypass permissions mode". Enterprise admins can disable this setting.
`--add-dir`	add multiple repos with the + button in remote sessions
`--allowedTools`, `--disallowedTools`	not available in Desktop
`--verbose`	not available. Check system logs: Console.app on macOS, Event Viewer → Windows Logs → Application on Windows
`--print`, `--output-format`	not available. Desktop is interactive only.

CLI	Desktop equivalent
`ANTHROPIC_MODEL` env var	model dropdown next to the send button
`MAX_THINKING_TOKENS` env var	set in shell profile; applies to local sessions. See environment configuration.

Shared configuration

Desktop and CLI read the same configuration files, so your setup carries over:

- **CLAUDE.md** files in your project are used by both
- **MCP servers** configured in `~/.claude.json` or `.mcp.json` work in both
- **Hooks** and **skills** defined in settings apply to both
- **Settings** in `~/.claude.json` and `~/.claude/settings.json` are shared. Permission rules, allowed tools, and other settings in `settings.json` apply to Desktop sessions.
- **Models**: Sonnet, Opus, and Haiku are available in both. In Desktop, select the model from the dropdown next to the send button before starting a session. You cannot change the model during an active session.

> **Note:**
>
> **MCP servers: desktop chat app vs Claude Code**: MCP servers configured for the Claude Desktop chat app in `claude_desktop_config.json` are separate from Claude Code and will not appear in the Code tab. To use MCP servers in Claude Code, configure them in `~/.claude.json` or your project's `.mcp.json` file. See MCP configuration for details.

Feature comparison

This table compares core capabilities between the CLI and Desktop. For a full list of CLI flags, see the CLI reference.

Feature	CLI	Desktop
Permission modes	all modes including `dontAsk`	Ask permissions, Auto accept edits, Plan mode, and Bypass permissions via Settings

Feature	CLI	Desktop
`--dangerously-skip-permissions`	CLI flag	Bypass permissions mode. Enable in Settings → Claude Code → "Allow bypass permissions mode"
Third-party providers	Bedrock, Vertex, Foundry	not available. Desktop connects to Anthropic's API directly.
MCP servers	configure in settings files	Connectors UI for local and SSH sessions, or settings files
Plugins	`/plugin` command	plugin manager UI
@mention files	text-based	with autocomplete
File attachments	not available	images, PDFs
Session isolation	`--worktree` flag	automatic worktrees
Multiple sessions	separate terminals	sidebar tabs
Recurring tasks	cron jobs, CI pipelines	scheduled tasks
Scripting and automation	`--print`, Agent SDK	not available

What's not available in Desktop

The following features are only available in the CLI or VS Code extension:

- **Third-party providers**: Desktop connects to Anthropic's API directly. Use the CLI with Bedrock, Vertex, or Foundry instead.
- **Linux**: the desktop app is available on macOS and Windows only.
- **Inline code suggestions**: Desktop does not provide autocomplete-style suggestions. It works through conversational prompts and explicit code changes.
- **Agent teams**: multi-agent orchestration is available via the CLI and Agent SDK, not in Desktop.

Troubleshooting

Check your version

To see which version of the desktop app you're running:

- **macOS**: click **Claude** in the menu bar, then **About Claude**
- **Windows**: click **Help**, then **About**

Click the version number to copy it to your clipboard.

403 or authentication errors in the Code tab

If you see `Error 403: Forbidden` or other authentication failures when using the Code tab:

1. Sign out and back in from the app menu. This is the most common fix.
2. Verify you have an active paid subscription: Pro, Max, Teams, or Enterprise.
3. If the CLI works but Desktop does not, quit the desktop app completely, not just close the window, then reopen and sign in again.
4. Check your internet connection and proxy settings.

Blank or stuck screen on launch

If the app opens but shows a blank or unresponsive screen:

1. Restart the app.
2. Check for pending updates. The app auto-updates on launch.
3. On Windows, check Event Viewer for crash logs under **Windows Logs → Application**.

"Failed to load session"

If you see `Failed to load session`, the selected folder may no longer exist, a Git repository may require Git LFS that isn't installed, or file permissions may prevent access. Try selecting a different folder or restarting the app.

Session not finding installed tools

If Claude can't find tools like `npm`, `node`, or other CLI commands, verify the tools work in your regular terminal, check that your shell profile properly sets up PATH, and restart the desktop app to reload environment variables.

Git and Git LFS errors

On Windows, Git is required for the Code tab to start local sessions. If you see “Git is required,” install Git for Windows and restart the app.

If you see “Git LFS is required by this repository but is not installed,” install Git LFS from git-lfs.com, run `git lfs install`, and restart the app.

MCP servers not working on Windows

If MCP server toggles don’t respond or servers fail to connect on Windows, check that the server is properly configured in your settings, restart the app, verify the server process is running in Task Manager, and review server logs for connection errors.

App won’t quit

- **macOS**: press Cmd+Q. If the app doesn’t respond, use Force Quit with Cmd+Option+Esc, select Claude, and click Force Quit.
- **Windows**: use Task Manager with Ctrl+Shift+Esc to end the Claude process.

Windows-specific issues

- **PATH not updated after install**: open a new terminal window. PATH updates only apply to new terminal sessions.
- **Concurrent installation error**: if you see an error about another installation in progress but there isn’t one, try running the installer as Administrator.
- **ARM64**: Windows ARM64 devices are fully supported.

Cowork tab unavailable on Intel Macs

The Cowork tab requires Apple Silicon (M1 or later) on macOS. On Windows, Cowork is available on all supported hardware. The Chat and Code tabs work normally on Intel Macs.

“Branch doesn’t exist yet” when opening in CLI

Remote sessions can create branches that don’t exist on your local machine. Click the branch name in the session toolbar to copy it, then fetch it locally:

```
git fetch origin <branch-name>
git checkout <branch-name>
```

Still stuck?

- Search or file a bug on GitHub Issues

- Visit the Claude support center

When filing a bug, include your desktop app version, your operating system, the exact error message, and relevant logs. On macOS, check Console.app. On Windows, check Event Viewer → Windows Logs → Application.

Use Claude Code with Chrome (beta)

> *Connect Claude Code to your Chrome browser to test web apps, debug with console logs, automate form filling, and extract data from web pages.*

Claude Code integrates with the Claude in Chrome browser extension to give you browser automation capabilities from the CLI or the VS Code extension. Build your code, then test and debug in the browser without switching contexts.

Claude opens new tabs for browser tasks and shares your browser's login state, so it can access any site you're already signed into. Browser actions run in a visible Chrome window in real time. When Claude encounters a login page or CAPTCHA, it pauses and asks you to handle it manually.

> **Note:**
>
> Chrome integration is in beta and currently works with Google Chrome and Microsoft Edge. It is not yet supported on Brave, Arc, or other Chromium-based browsers. WSL (Windows Subsystem for Linux) is also not supported.

Capabilities

With Chrome connected, you can chain browser actions with coding tasks in a single workflow:

- **Live debugging**: read console errors and DOM state directly, then fix the code that caused them
- **Design verification**: build a UI from a Figma mock, then open it in the browser to verify it matches
- **Web app testing**: test form validation, check for visual regressions, or verify user flows
- **Authenticated web apps**: interact with Google Docs, Gmail, Notion, or any app you're logged into without API connectors
- **Data extraction**: pull structured information from web pages and save it locally
- **Task automation**: automate repetitive browser tasks like data entry, form filling, or multi-site workflows

- **Session recording**: record browser interactions as GIFs to document or share what happened

Prerequisites

Before using Claude Code with Chrome, you need:

- Google Chrome or Microsoft Edge browser
- Claude in Chrome extension version 1.0.36 or higher, available in the Chrome Web Store for both browsers
- Claude Code version 2.0.73 or higher
- A direct Anthropic plan (Pro, Max, Teams, or Enterprise)

> **Note:**
>
> Chrome integration is not available through third-party providers like Amazon Bedrock, Google Cloud Vertex AI, or Microsoft Foundry. If you access Claude exclusively through a third-party provider, you need a separate claude.ai account to use this feature.

Get started in the CLI

Step 1: Launch Claude Code with Chrome

Start Claude Code with the `--chrome` flag:

```
claude --chrome
```

You can also enable Chrome from within an existing session by running `/chrome`.

Step 2: Ask Claude to use the browser

This example navigates to a page, interacts with it, and reports what it finds, all from your terminal or editor:

```
Go to code.claude.com/docs, click on the search box,
type "hooks", and tell me what results appear
```

Run `/chrome` at any time to check the connection status, manage permissions, or reconnect the extension.

For VS Code, see browser automation in VS Code.

Enable Chrome by default

To avoid passing `--chrome` each session, run `/chrome` and select “Enabled by default”.

In the VS Code extension, Chrome is available whenever the Chrome extension is installed. No additional flag is needed.

> **Note:**
>
> Enabling Chrome by default in the CLI increases context usage since browser tools are always loaded. If you notice increased context consumption, disable this setting and use `--chrome` only when needed.

Manage site permissions

Site-level permissions are inherited from the Chrome extension. Manage permissions in the Chrome extension settings to control which sites Claude can browse, click, and type on.

Example workflows

These examples show common ways to combine browser actions with coding tasks. Run `/mcp` and select `claude-in-chrome` to see the full list of available browser tools.

Test a local web application

When developing a web app, ask Claude to verify your changes work correctly:

```
I just updated the login form validation. Can you open localhost:3000,
try submitting the form with invalid data, and check if the error
messages appear correctly?
```

Claude navigates to your local server, interacts with the form, and reports what it observes.

Debug with console logs

Claude can read console output to help diagnose problems. Tell Claude what patterns to look for rather than asking for all console output, since logs can be verbose:

```
Open the dashboard page and check the console for any errors when
the page loads.
```

Claude reads the console messages and can filter for specific patterns or error types.

Automate form filling

Speed up repetitive data entry tasks:

```
I have a spreadsheet of customer contacts in contacts.csv. For each row,
go to the CRM at crm.example.com, click "Add Contact", and fill in the
name, email, and phone fields.
```

Claude reads your local file, navigates the web interface, and enters the data for each record.

Draft content in Google Docs

Use Claude to write directly in your documents without API setup:

```
Draft a project update based on the recent commits and add it to my
Google Doc at docs.google.com/document/d/abc123
```

Claude opens the document, clicks into the editor, and types the content. This works with any web app you're logged into: Gmail, Notion, Sheets, and more.

Extract data from web pages

Pull structured information from websites:

```
Go to the product listings page and extract the name, price, and
availability for each item. Save the results as a CSV file.
```

Claude navigates to the page, reads the content, and compiles the data into a structured format.

Run multi-site workflows

Coordinate tasks across multiple websites:

```
Check my calendar for meetings tomorrow, then for each meeting with
an external attendee, look up their company website and add a note
about what they do.
```

Claude works across tabs to gather information and complete the workflow.

Record a demo GIF

Create shareable recordings of browser interactions:

```
Record a GIF showing how to complete the checkout flow, from adding
an item to the cart through to the confirmation page.
```

Claude records the interaction sequence and saves it as a GIF file.

Troubleshooting

Extension not detected

If Claude Code shows “Chrome extension not detected”:

1. Verify the Chrome extension is installed and enabled in `chrome://extensions`
2. Verify Claude Code is up to date by running `claude --version`
3. Check that Chrome is running
4. Run `/chrome` and select “Reconnect extension” to re-establish the connection
5. If the issue persists, restart both Claude Code and Chrome

The first time you enable Chrome integration, Claude Code installs a native messaging host configuration file. Chrome reads this file on startup, so if the extension isn’t detected on your first attempt, restart Chrome to pick up the new configuration.

If the connection still fails, verify the host configuration file exists at:

For Chrome:

- **macOS**: `~/Library/Application Support/Google/Chrome/NativeMessagingHosts/com.anthropic.claude_code_browser_extension.json`
- **Linux**: `~/.config/google-chrome/NativeMessagingHosts/com.anthropic.claude_code_browser_extension.json`
- **Windows**: check `HKCU\Software\Google\Chrome\NativeMessagingHosts\` in the Windows Registry

For Edge:

- **macOS**: `~/Library/Application Support/Microsoft Edge/NativeMessagingHosts/com.anthropic.claude_code_browser_extension.json`

- **Linux**: `~/.config/microsoft-edge/NativeMessagingHosts/com.anthropic.claude_code_browser_extension.json`
- **Windows**: check `HKCU\Software\Microsoft\Edge\NativeMessagingHosts\` in the Windows Registry

Browser not responding

If Claude's browser commands stop working:

1. Check if a modal dialog (alert, confirm, prompt) is blocking the page. JavaScript dialogs block browser events and prevent Claude from receiving commands. Dismiss the dialog manually, then tell Claude to continue.
2. Ask Claude to create a new tab and try again
3. Restart the Chrome extension by disabling and re-enabling it in `chrome://extensions`

Connection drops during long sessions

The Chrome extension's service worker can go idle during extended sessions, which breaks the connection. If browser tools stop working after a period of inactivity, run `/chrome` and select "Reconnect extension".

Windows-specific issues

On Windows, you may encounter:

- **Named pipe conflicts (EADDRINUSE)**: if another process is using the same named pipe, restart Claude Code. Close any other Claude Code sessions that might be using Chrome.
- **Native messaging host errors**: if the native messaging host crashes on startup, try reinstalling Claude Code to regenerate the host configuration.

Common error messages

These are the most frequently encountered errors and how to resolve them:

Error	Cause	Fix
"Browser extension is not connected"	Native messaging host cannot reach the extension	Restart Chrome and Claude Code, then run `/chrome` to reconnect
"Extension not detected"	Chrome extension is not installed or is disabled	Install or enable the extension in `chrome://extensions`

Error	Cause	Fix
“No tab available”	Claude tried to act before a tab was ready	Ask Claude to create a new tab and retry
“Receiving end does not exist”	Extension service worker went idle	Run `/chrome` and select “Reconnect extension”

See also

- Use Claude Code in VS Code: browser automation in the VS Code extension
- CLI reference: command-line flags including `--chrome`
- Common workflows: more ways to use Claude Code
- Data and privacy: how Claude Code handles your data
- Getting started with Claude in Chrome: full documentation for the Chrome extension, including shortcuts, scheduling, and permissions

Claude Code in Slack

> *Delegate coding tasks directly from your Slack workspace*

Claude Code in Slack brings the power of Claude Code directly into your Slack workspace. When you mention `@Claude` with a coding task, Claude automatically detects the intent and creates a Claude Code session on the web, allowing you to delegate development work without leaving your team conversations.

This integration is built on the existing Claude for Slack app but adds intelligent routing to Claude Code on the web for coding-related requests.

Use cases

- **Bug investigation and fixes**: Ask Claude to investigate and fix bugs as soon as they're reported in Slack channels.
- **Quick code reviews and modifications**: Have Claude implement small features or refactor code based on team feedback.
- **Collaborative debugging**: When team discussions provide crucial context (e.g., error reproductions or user reports), Claude can use that information to inform its debugging approach.
- **Parallel task execution**: Kick off coding tasks in Slack while you continue other work, receiving notifications when complete.

Prerequisites

Before using Claude Code in Slack, ensure you have the following:

Requirement	Details
Claude Plan	Pro, Max, Teams, or Enterprise with Claude Code access (premium seats)
Claude Code on the web	Access to Claude Code on the web must be enabled
GitHub Account	Connected to Claude Code on the web with at least one repository authenticated

Requirement	Details
Slack Authentication	Your Slack account linked to your Claude account via the Claude app

Setting up Claude Code in Slack

Step 1: Install the Claude App in Slack

A workspace administrator must install the Claude app from the Slack App Marketplace. Visit the Slack App Marketplace and click “Add to Slack” to begin the installation process.

Step 2: Connect your Claude account

After the app is installed, authenticate your individual Claude account:

1. Open the Claude app in Slack by clicking on “Claude” in your Apps section
2. Navigate to the App Home tab
3. Click “Connect” to link your Slack account with your Claude account
4. Complete the authentication flow in your browser

Step 3: Configure Claude Code on the web

Ensure your Claude Code on the web is properly configured:

- Visit claude.ai/code and sign in with the same account you connected to Slack
- Connect your GitHub account if not already connected
- Authenticate at least one repository that you want Claude to work with

Step 4: Choose your routing mode

After connecting your accounts, configure how Claude handles your messages in Slack. Navigate to the Claude App Home in Slack to find the **Routing Mode** setting.

Mode	Behavior
Code only	Claude routes all @mentions to Claude Code sessions. Best for teams using Claude in Slack exclusively for development tasks.
Code + Chat	Claude analyzes each message and intelligently routes between Claude Code (for coding tasks) and Claude Chat (for writing, analysis, and general questions). Best for teams who want a single @Claude entry point for all types of work.

> **Note:**

> In Code + Chat mode, if Claude routes a message to Chat but you wanted a coding session, you can click "Retry as Code" to create a Claude Code session instead. Similarly, if it's routed to Code but you wanted a Chat session, you can choose that option in that thread.

Step 5: Add Claude to channels

Claude is not automatically added to any channels after installation. To use Claude in a channel, invite it by typing `/invite @Claude` in that channel. Claude can only respond to @mentions in channels where it has been added.

How it works

Automatic detection

When you mention @Claude in a Slack channel or thread, Claude automatically analyzes your message to determine if it's a coding task. If Claude detects coding intent, it will route your request to Claude Code on the web instead of responding as a regular chat assistant.

You can also explicitly tell Claude to handle a request as a coding task, even if it doesn't automatically detect it.

> **Note:**
>
> Claude Code in Slack only works in channels (public or private). It does not work in direct messages (DMs).

Context gathering

From threads: When you @mention Claude in a thread, it gathers context from all messages in that thread to understand the full conversation.

From channels: When mentioned directly in a channel, Claude looks at recent channel messages for relevant context.

This context helps Claude understand the problem, select the appropriate repository, and inform its approach to the task.

> **Warning:**
>
> When @Claude is invoked in Slack, Claude is given access to the conversation context to better understand your request. Claude may follow directions from other messages in the context, so users should make sure to only use Claude in trusted Slack conversations.

Session flow

1. **Initiation**: You @mention Claude with a coding request
2. **Detection**: Claude analyzes your message and detects coding intent
3. **Session creation**: A new Claude Code session is created on claude.ai/code
4. **Progress updates**: Claude posts status updates to your Slack thread as work progresses
5. **Completion**: When finished, Claude @mentions you with a summary and action buttons
6. **Review**: Click "View Session" to see the full transcript, or "Create PR" to open a pull request

User interface elements

App Home

The App Home tab shows your connection status and allows you to connect or disconnect your Claude account from Slack.

Message actions

- **View Session**: Opens the full Claude Code session in your browser where you can see all work performed, continue the session, or make additional requests.
- **Create PR**: Creates a pull request directly from the session's changes.
- **Retry as Code**: If Claude initially responds as a chat assistant but you wanted a coding session, click this button to retry the request as a Claude Code task.
- **Change Repo**: Allows you to select a different repository if Claude chose incorrectly.

Repository selection

Claude automatically selects a repository based on context from your Slack conversation. If multiple repositories could apply, Claude may display a dropdown allowing you to choose the correct one.

Access and permissions

User-level access

Access Type	Requirement
Claude Code Sessions	Each user runs sessions under their own Claude account

Access Type	Requirement
Usage & Rate Limits	Sessions count against the individual user's plan limits
Repository Access	Users can only access repositories they've personally connected
Session History	Sessions appear in your Claude Code history on claude.ai/code

Workspace-level access

Slack workspace administrators control whether the Claude app is available in their workspace:

Control	Description
App installation	Workspace admins decide whether to install the Claude app from the Slack App Marketplace
Enterprise Grid distribution	For Enterprise Grid organizations, organization admins can control which workspaces have access to the Claude app
App removal	Removing the app from a workspace immediately revokes access for all users in that workspace

Channel-based access control

Claude is not automatically added to any channels after installation. Users must explicitly invite Claude to channels where they want to use it:

- **Invite required**: Type `/invite @Claude` in any channel to add Claude to that channel
- **Channel membership controls access**: Claude can only respond to @mentions in channels where it has been added
- **Access gating through channels**: Admins can control who uses Claude Code by managing which channels Claude is invited to and who has access to those channels
- **Private channel support**: Claude works in both public and private channels, giving teams flexibility in controlling visibility

This channel-based model allows teams to restrict Claude Code usage to specific channels, providing an additional layer of access control beyond workspace-level permissions.

What's accessible where

In Slack: You'll see status updates, completion summaries, and action buttons. The full transcript is preserved and always accessible.

On the web: The complete Claude Code session with full conversation history, all code changes, file operations, and the ability to continue the session or create pull requests.

For Enterprise and Teams accounts, sessions created from Claude in Slack are automatically visible to the organization. See Claude Code on the Web sharing for more details.

Best practices

Writing effective requests

- **Be specific**: Include file names, function names, or error messages when relevant.
- **Provide context**: Mention the repository or project if it's not clear from the conversation.
- **Define success**: Explain what "done" looks like—should Claude write tests? Update documentation? Create a PR?
- **Use threads**: Reply in threads when discussing bugs or features so Claude can gather the full context.

When to use Slack vs. web

Use Slack when: Context already exists in a Slack discussion, you want to kick off a task asynchronously, or you're collaborating with teammates who need visibility.

Use the web directly when: You need to upload files, want real-time interaction during development, or are working on longer, more complex tasks.

Troubleshooting

Sessions not starting

1. Verify your Claude account is connected in the Claude App Home
2. Check that you have Claude Code on the web access enabled
3. Ensure you have at least one GitHub repository connected to Claude Code

Repository not showing

1. Connect the repository in Claude Code on the web at claude.ai/code
2. Verify your GitHub permissions for that repository

3. Try disconnecting and reconnecting your GitHub account

Wrong repository selected

1. Click the “Change Repo” button to select a different repository
2. Include the repository name in your request for more accurate selection

Authentication errors

1. Disconnect and reconnect your Claude account in the App Home
2. Ensure you’re signed into the correct Claude account in your browser
3. Check that your Claude plan includes Claude Code access

Session expiration

1. Sessions remain accessible in your Claude Code history on the web
2. You can continue or reference past sessions from claude.ai/code

Current limitations

- **GitHub only**: Currently supports repositories on GitHub.
- **One PR at a time**: Each session can create one pull request.
- **Rate limits apply**: Sessions use your individual Claude plan’s rate limits.
- **Web access required**: Users must have Claude Code on the web access; those without it will only get standard Claude chat responses.

Related resources

- Claude Code on the web Learn more about Claude Code on the web
- Claude for Slack General Claude for Slack documentation
- Slack App Marketplace Install the Claude app from the Slack Marketplace
- Claude Help Center Get additional support

Claude Code on the web

> *Run Claude Code tasks asynchronously on secure cloud infrastructure*

Note:

Claude Code on the web is currently in research preview.

What is Claude Code on the web?

Claude Code on the web lets developers kick off Claude Code from the Claude app. This is perfect for:

- **Answering questions**: Ask about code architecture and how features are implemented
- **Bug fixes and routine tasks**: Well-defined tasks that don't require frequent steering
- **Parallel work**: Tackle multiple bug fixes in parallel
- **Repositories not on your local machine**: Work on code you don't have checked out locally
- **Backend changes**: Where Claude Code can write tests and then write code to pass those tests

Claude Code is also available on the Claude app for iOS and Android for kicking off tasks on the go and monitoring work in progress.

You can kick off new tasks on the web from your terminal with `--remote`, or teleport web sessions back to your terminal to continue locally. To use the web interface while running Claude Code on your own machine instead of cloud infrastructure, see Remote Control.

Who can use Claude Code on the web?

Claude Code on the web is available in research preview to:

- **Pro users**
- **Max users**
- **Team users**
- **Enterprise users** with premium seats or Chat + Claude Code seats

Getting started

1. Visit claude.ai/code
2. Connect your GitHub account
3. Install the Claude GitHub app in your repositories
4. Select your default environment
5. Submit your coding task
6. Review changes in diff view, iterate with comments, then create a pull request

How it works

When you start a task on Claude Code on the web:

1. **Repository cloning**: Your repository is cloned to an Anthropic-managed virtual machine
2. **Environment setup**: Claude prepares a secure cloud environment with your code, then runs your setup script if configured
3. **Network configuration**: Internet access is configured based on your settings
4. **Task execution**: Claude analyzes code, makes changes, runs tests, and checks its work
5. **Completion**: You're notified when finished and can create a PR with the changes
6. **Results**: Changes are pushed to a branch, ready for pull request creation

Review changes with diff view

Diff view lets you see exactly what Claude changed before creating a pull request. Instead of clicking "Create PR" to review changes in GitHub, view the diff directly in the app and iterate with Claude until the changes are ready.

When Claude makes changes to files, a diff stats indicator appears showing the number of lines added and removed (for example, `+12 -1`). Select this indicator to open the diff viewer, which displays a file list on the left and the changes for each file on the right.

From the diff view, you can:

- Review changes file by file
- Comment on specific changes to request modifications
- Continue iterating with Claude based on what you see

This lets you refine changes through multiple rounds of feedback without creating draft PRs or switching to GitHub.

Moving tasks between web and terminal

You can start new tasks on the web from your terminal, or pull web sessions into your terminal to continue locally. Web sessions persist even if you close your laptop, and you can monitor them from anywhere including the Claude mobile app.

> **Note:**
>
> Session handoff is one-way: you can pull web sessions into your terminal, but you can't push an existing terminal session to the web. The `--remote` flag creates a *new* web session for your current repository.

From terminal to web

Start a web session from the command line with the `--remote` flag:

```
claude --remote "Fix the authentication bug in src/auth/login.ts"
```

This creates a new web session on claude.ai. The task runs in the cloud while you continue working locally. Use `/tasks` to check progress, or open the session on claude.ai or the Claude mobile app to interact directly. From there you can steer Claude, provide feedback, or answer questions just like any other conversation.

Tips for remote tasks

Plan locally, execute remotely: For complex tasks, start Claude in plan mode to collaborate on the approach, then send work to the web:

```
claude --permission-mode plan
```

In plan mode, Claude can only read files and explore the codebase. Once you're satisfied with the plan, start a remote session for autonomous execution:

```
claude --remote "Execute the migration plan in docs/migration-plan.md"
```

This pattern gives you control over the strategy while letting Claude execute autonomously in the cloud.

Run tasks in parallel: Each `--remote` command creates its own web session that runs independently. You can kick off multiple tasks and they'll all run simultaneously in separate sessions:

```
claude --remote "Fix the flaky test in auth.spec.ts"
claude --remote "Update the API documentation"
claude --remote "Refactor the logger to use structured output"
```

Monitor all sessions with `/tasks`. When a session completes, you can create a PR from the web interface or teleport the session to your terminal to continue working.

From web to terminal

There are several ways to pull a web session into your terminal:

- **Using `/teleport`**: From within Claude Code, run `/teleport` (or `/tp`) to see an interactive picker of your web sessions. If you have uncommitted changes, you'll be prompted to stash them first.
- **Using `--teleport`**: From the command line, run `claude --teleport` for an interactive session picker, or `claude --teleport <session-id>` to resume a specific session directly.
- **From `/tasks`**: Run `/tasks` to see your background sessions, then press `t` to teleport into one
- **From the web interface**: Click "Open in CLI" to copy a command you can paste into your terminal

When you teleport a session, Claude verifies you're in the correct repository, fetches and checks out the branch from the remote session, and loads the full conversation history into your terminal.

Requirements for teleporting

Teleport checks these requirements before resuming a session. If any requirement isn't met, you'll see an error or be prompted to resolve the issue.

Requirement	Details
Clean git state	Your working directory must have no uncommitted changes. Teleport prompts you to stash changes if needed.

Requirement	Details
Correct repository	You must run `--teleport` from a checkout of the same repository, not a fork.
Branch available	The branch from the web session must have been pushed to the remote. Teleport automatically fetches and checks it out.
Same account	You must be authenticated to the same Claude.ai account used in the web session.

Sharing sessions

To share a session, toggle its visibility according to the account types below. After that, share the session link as-is. Recipients who open your shared session will see the latest state of the session upon load, but the recipient's page will not update in real time.

Sharing from an Enterprise or Teams account

For Enterprise and Teams accounts, the two visibility options are **Private** and **Team**. Team visibility makes the session visible to other members of your Claude.ai organization. Repository access verification is enabled by default, based on the GitHub account connected to the recipient's account. Your account's display name is visible to all recipients with access. Claude in Slack sessions are automatically shared with Team visibility.

Sharing from a Max or Pro account

For Max and Pro accounts, the two visibility options are **Private** and **Public**. Public visibility makes the session visible to any user logged into claude.ai.

Check your session for sensitive content before sharing. Sessions may contain code and credentials from private GitHub repositories. Repository access verification is not enabled by default.

Enable repository access verification and/or withhold your name from your shared sessions by going to Settings > Claude Code > Sharing settings.

Managing sessions

Archiving sessions

You can archive sessions to keep your session list organized. Archived sessions are hidden from the default session list but can be viewed by filtering for archived sessions.

To archive a session, hover over the session in the sidebar and click the archive icon.

Deleting sessions

Deleting a session permanently removes the session and its data. This action cannot be undone. You can delete a session in two ways:

- **From the sidebar**: Filter for archived sessions, then hover over the session you want to delete and click the delete icon
- **From the session menu**: Open a session, click the dropdown next to the session title, and select **Delete**

You will be asked to confirm before a session is deleted.

Cloud environment

Default image

We build and maintain a universal image with common toolchains and language ecosystems pre-installed. This image includes:

- Popular programming languages and runtimes
- Common build tools and package managers
- Testing frameworks and linters

Checking available tools

To see what's pre-installed in your environment, ask Claude Code to run:

```
check-tools
```

This command displays:

- Programming languages and their versions
- Available package managers
- Installed development tools

Language-specific setups

The universal image includes pre-configured environments for:

- **Python**: Python 3.x with pip, poetry, and common scientific libraries
- **Node.js**: Latest LTS versions with npm, yarn, pnpm, and bun
- **Ruby**: Versions 3.1.6, 3.2.6, 3.3.6 (default: 3.3.6) with gem, bundler, and rbenv for version management
- **PHP**: Version 8.4.14
- **Java**: OpenJDK with Maven and Gradle
- **Go**: Latest stable version with module support
- **Rust**: Rust toolchain with cargo
- **C++**: GCC and Clang compilers

Databases

The universal image includes the following databases:

- **PostgreSQL**: Version 16
- **Redis**: Version 7.0

Environment configuration

When you start a session in Claude Code on the web, here's what happens under the hood:

1. **Environment preparation**: We clone your repository and run any configured setup script. The repo will be cloned with the default branch on your GitHub repo. If you would like to check out a specific branch, you can specify that in the prompt.
2. **Network configuration**: We configure internet access for the agent. Internet access is limited by default, but you can configure the environment to have no internet or full internet access based on your needs.
3. **Claude Code execution**: Claude Code runs to complete your task, writing code, running tests, and checking its work. You can guide and steer Claude throughout the session via the web interface. Claude respects context you've defined in your `CLAUDE.md`.
4. **Outcome**: When Claude completes its work, it will push the branch to remote. You will be able to create a PR for the branch.

> **Note:**

> Claude operates entirely through the terminal and CLI tools available in the environment. It uses the pre-installed tools in the universal image and any additional tools you install through hooks or dependency management.

To add a new environment: Select the current environment to open the environment selector, and then select "Add environment". This will open a dialog where you can specify the environment name, network access level, environment variables, and a setup script.

To update an existing environment: Select the current environment, to the right of the environment name, and select the settings button. This will open a dialog where you can update the environment name, network access, environment variables, and setup script.

To select your default environment from the terminal: If you have multiple environments configured, run `/remote-env` to choose which one to use when starting web sessions from your terminal with `--remote`. With a single environment, this command shows your current configuration.

> **Note:**
>
> Environment variables must be specified as key-value pairs, in `.env` format. For example:
>
> ```
> API_KEY=your_api_key
> DEBUG=true
> ```

Setup scripts

A setup script is a Bash script that runs when a new cloud session starts, before Claude Code launches. Use setup scripts to install dependencies, configure tools, or prepare anything the cloud environment needs that isn't in the default image.

Scripts run as root on Ubuntu 24.04, so `apt install` and most language package managers work.

> **Tip:**
>
> To check what's already installed before adding it to your script, ask Claude to run `check-tools` in a cloud session.

To add a setup script, open the environment settings dialog and enter your script in the **Setup script** field.

This example installs the `gh` CLI, which isn't in the default image:

```
#!/bin/bash
apt update && apt install -y gh
```

Setup scripts run only when creating a new session. They are skipped when resuming an existing session.

If the script exits non-zero, the session fails to start. Append `|| true` to non-critical commands to avoid blocking the session on a flaky install.

> **Note:**
>
> Setup scripts that install packages need network access to reach registries. The default network access allows connections to common package registries including npm, PyPI, RubyGems, and crates.io. Scripts will fail to install packages if your environment has network access disabled.

Setup scripts vs. SessionStart hooks

Use a setup script to install things the cloud needs but your laptop already has, like a language runtime or CLI tool. Use a SessionStart hook for project setup that should run everywhere, cloud and local, like `npm install`.

Both run at the start of a session, but they belong to different places:

	Setup scripts	SessionStart hooks
Attached to	The cloud environment	Your repository
Configured in	Cloud environment UI	`.claude/settings.json` in your repo
Runs	Before Claude Code launches, on new sessions only	After Claude Code launches, on every session including resumed
Scope	Cloud environments only	Both local and cloud

SessionStart hooks can also be defined in your user-level `~/.claude/settings.json` locally, but user-level settings don't carry over to cloud sessions. In the cloud, only hooks committed to the repo run.

Dependency management

Custom environment images and snapshots are not yet supported. Use setup scripts to install packages when a session starts, or SessionStart hooks for dependency installation that should also run in local environments. SessionStart hooks have known limitations.

To configure automatic dependency installation with a setup script, open your environment settings and add a script:

```
#!/bin/bash
npm install
pip install -r requirements.txt
```

Alternatively, you can use SessionStart hooks in your repository's `.claude/settings.json` file for dependency installation that should also run in local environments:

```
{
  "hooks": {
    "SessionStart": [
      {
        "matcher": "startup",
        "hooks": [
          {
            "type": "command",
            "command": "\"$CLAUDE_PROJECT_DIR\"/scripts/install_pkgs.sh"
          }
        ]
      }
    ]
  }
}
```

Create the corresponding script at `scripts/install_pkgs.sh`:

```
#!/bin/bash

## Only run in remote environments
if [ "$CLAUDE_CODE_REMOTE" != "true" ]; then
  exit 0
fi

npm install
pip install -r requirements.txt
exit 0
```

Make it executable: `chmod +x scripts/install_pkgs.sh`

Persist environment variables

SessionStart hooks can persist environment variables for subsequent Bash commands by writing to the file specified in the `CLAUDE_ENV_FILE` environment variable. For details, see SessionStart hooks in the hooks reference.

Dependency management limitations

- **Hooks fire for all sessions**: SessionStart hooks run in both local and remote environments. There is no hook configuration to scope a hook to remote sessions only. To skip local execution, check the `CLAUDE_CODE_REMOTE` environment variable in your script as shown above.
- **Requires network access**: Install commands need network access to reach package registries. If your environment is configured with "No internet" access, these hooks will fail. Use "Limited" (the default) or "Full" network access. The default allowlist includes common registries like npm, PyPI, RubyGems, and crates.io.
- **Proxy compatibility**: All outbound traffic in remote environments passes through a security proxy. Some package managers do not work correctly with this proxy. Bun is a known example.
- **Runs on every session start**: Hooks run each time a session starts or resumes, adding startup latency. Keep install scripts fast by checking whether dependencies are already present before reinstalling.

Network access and security

Network policy

GitHub proxy

For security, all GitHub operations go through a dedicated proxy service that transparently handles all git interactions. Inside the sandbox, the git client authenticates using a custom-built scoped credential. This proxy:

- Manages GitHub authentication securely - the git client uses a scoped credential inside the sandbox, which the proxy verifies and translates to your actual GitHub authentication token
- Restricts git push operations to the current working branch for safety
- Enables seamless cloning, fetching, and PR operations while maintaining security boundaries

Security proxy

Environments run behind an HTTP/HTTPS network proxy for security and abuse prevention purposes. All outbound internet traffic passes through this proxy, which provides:

- Protection against malicious requests
- Rate limiting and abuse prevention
- Content filtering for enhanced security

Access levels

By default, network access is limited to allowlisted domains.

You can configure custom network access, including disabling network access.

Default allowed domains

When using “Limited” network access, the following domains are allowed by default:

Anthropic Services

- api.anthropic.com
- statsig.anthropic.com
- platform.claude.com
- code.claude.com
- claude.ai

Version Control

- github.com
- www.github.com
- api.github.com
- npm.pkg.github.com
- raw.githubusercontent.com
- pkg-npm.githubusercontent.com
- objects.githubusercontent.com
- codeload.github.com
- avatars.githubusercontent.com
- camo.githubusercontent.com
- gist.github.com
- gitlab.com
- www.gitlab.com
- registry.gitlab.com
- bitbucket.org
- www.bitbucket.org
- api.bitbucket.org

Container Registries

- registry-1.docker.io
- auth.docker.io
- index.docker.io
- hub.docker.com
- www.docker.com
- production.cloudflare.docker.com
- download.docker.com
- gcr.io
- *.gcr.io
- ghcr.io
- mcr.microsoft.com
- *.data.mcr.microsoft.com
- public.ecr.aws

Cloud Platforms

- cloud.google.com
- accounts.google.com
- gcloud.google.com
- *.googleapis.com
- storage.googleapis.com
- compute.googleapis.com
- container.googleapis.com
- azure.com
- portal.azure.com
- microsoft.com
- www.microsoft.com
- *.microsoftonline.com
- packages.microsoft.com
- dotnet.microsoft.com
- dot.net
- visualstudio.com
- dev.azure.com
- *.amazonaws.com
- *.api.aws
- oracle.com
- www.oracle.com
- java.com
- www.java.com
- java.net
- www.java.net
- download.oracle.com
- yum.oracle.com

Package Managers - JavaScript/Node

- registry.npmjs.org
- www.npmjs.com
- www.npmjs.org

- npmjs.com
- npmjs.org
- yarnpkg.com
- registry.yarnpkg.com

Package Managers - Python

- pypi.org
- www.pypi.org
- files.pythonhosted.org
- pythonhosted.org
- test.pypi.org
- pypi.python.org
- pypa.io
- www.pypa.io

Package Managers - Ruby

- rubygems.org
- www.rubygems.org
- api.rubygems.org
- index.rubygems.org
- ruby-lang.org
- www.ruby-lang.org
- rubyforge.org
- www.rubyforge.org
- rubyonrails.org
- www.rubyonrails.org
- rvm.io
- get.rvm.io

Package Managers - Rust

- crates.io
- www.crates.io
- index.crates.io
- static.crates.io

- rustup.rs
- static.rust-lang.org
- www.rust-lang.org

Package Managers - Go

- proxy.golang.org
- sum.golang.org
- index.golang.org
- golang.org
- www.golang.org
- goproxy.io
- pkg.go.dev

Package Managers - JVM

- maven.org
- repo.maven.org
- central.maven.org
- repo1.maven.org
- jcenter.bintray.com
- gradle.org
- www.gradle.org
- services.gradle.org
- plugins.gradle.org
- kotlin.org
- www.kotlin.org
- spring.io
- repo.spring.io

Package Managers - Other Languages

- packagist.org (PHP Composer)
- www.packagist.org
- repo.packagist.org
- nuget.org (.NET NuGet)
- www.nuget.org

- api.nuget.org
- pub.dev (Dart/Flutter)
- api.pub.dev
- hex.pm (Elixir/Erlang)
- www.hex.pm
- cpan.org (Perl CPAN)
- www.cpan.org
- metacpan.org
- www.metacpan.org
- api.metacpan.org
- cocoapods.org (iOS/macOS)
- www.cocoapods.org
- cdn.cocoapods.org
- haskell.org
- www.haskell.org
- hackage.haskell.org
- swift.org
- www.swift.org

Linux Distributions

- archive.ubuntu.com
- security.ubuntu.com
- ubuntu.com
- www.ubuntu.com
- *.ubuntu.com
- ppa.launchpad.net
- launchpad.net
- www.launchpad.net

Development Tools & Platforms

- dl.k8s.io (Kubernetes)
- pkgs.k8s.io
- k8s.io
- www.k8s.io

- releases.hashicorp.com (HashiCorp)
- apt.releases.hashicorp.com
- rpm.releases.hashicorp.com
- archive.releases.hashicorp.com
- hashicorp.com
- www.hashicorp.com
- repo.anaconda.com (Anaconda/Conda)
- conda.anaconda.org
- anaconda.org
- www.anaconda.com
- anaconda.com
- continuum.io
- apache.org (Apache)
- www.apache.org
- archive.apache.org
- downloads.apache.org
- eclipse.org (Eclipse)
- www.eclipse.org
- download.eclipse.org
- nodejs.org (Node.js)
- www.nodejs.org

Cloud Services & Monitoring

- statsig.com
- www.statsig.com
- api.statsig.com
- sentry.io
- *.sentry.io
- http-intake.logs.datadoghq.com
- *.datadoghq.com
- *.datadoghq.eu

Content Delivery & Mirrors

- sourceforge.net

- *.sourceforge.net
- packagecloud.io
- *.packagecloud.io

Schema & Configuration

- json-schema.org
- www.json-schema.org
- json.schemastore.org
- www.schemastore.org

Model Context Protocol

- *.modelcontextprotocol.io

> **Note:**
>
> Domains marked with `*` indicate wildcard subdomain matching. For example, `*.gcr.io` allows access to any subdomain of `gcr.io`.

Security best practices for customized network access

1. **Principle of least privilege**: Only enable the minimum network access required
2. **Audit regularly**: Review allowed domains periodically
3. **Use HTTPS**: Always prefer HTTPS endpoints over HTTP

Security and isolation

Claude Code on the web provides strong security guarantees:

- **Isolated virtual machines**: Each session runs in an isolated, Anthropic-managed VM
- **Network access controls**: Network access is limited by default, and can be disabled

> **Note:**
>
> When running with network access disabled, Claude Code is allowed to communicate with the Anthropic API which may still allow data to exit the isolated Claude Code VM.

- **Credential protection**: Sensitive credentials (such as git credentials or signing keys) are never inside the sandbox with Claude Code. Authentication is handled through a secure proxy using scoped credentials

- **Secure analysis**: Code is analyzed and modified within isolated VMs before creating PRs

Pricing and rate limits

Claude Code on the web shares rate limits with all other Claude and Claude Code usage within your account. Running multiple tasks in parallel will consume more rate limits proportionately.

Limitations

- **Repository authentication**: You can only move sessions from web to local when you are authenticated to the same account
- **Platform restrictions**: Claude Code on the web only works with code hosted in GitHub. GitLab and other non-GitHub repositories cannot be used with cloud sessions

Best practices

1. **Automate environment setup**: Use setup scripts to install dependencies and configure tools before Claude Code launches. For more advanced scenarios, configure SessionStart hooks.
2. **Document requirements**: Clearly specify dependencies and commands in your `CLAUDE.md` file. If you have an `AGENTS.md` file, you can source it in your `CLAUDE.md` using `@AGENTS.md` to maintain a single source of truth.

Related resources

- Hooks configuration
- Settings reference
- Security
- Data usage

Part 9: Security & Privacy

Authentication

> *Log in to Claude Code and configure authentication for individuals, teams, and organizations.*

Claude Code supports multiple authentication methods depending on your setup. Individual users can log in with a Claude.ai account, while teams can use Claude for Teams or Enterprise, the Claude Console, or a cloud provider like Amazon Bedrock, Google Vertex AI, or Microsoft Foundry.

Log in to Claude Code

After installing Claude Code, run `claude` in your terminal. On first launch, Claude Code opens a browser window for you to log in.

If the browser doesn't open automatically, press `c` to copy the login URL to your clipboard, then paste it into your browser.

You can authenticate with any of these account types:

- **Claude Pro or Max subscription**: log in with your Claude.ai account. Subscribe at claude.com/pricing.
- **Claude for Teams or Enterprise**: log in with the Claude.ai account your team admin invited you to.
- **Claude Console**: log in with your Console credentials. Your admin must have invited you first.
- **Cloud providers**: if your organization uses Amazon Bedrock, Google Vertex AI, or Microsoft Foundry, set the required environment variables before running `claude`. No browser login is needed.

To log out and re-authenticate, type `/logout` at the Claude Code prompt.

If you're having trouble logging in, see authentication troubleshooting.

Set up team authentication

For teams and organizations, you can configure Claude Code access in one of these ways:

- Claude for Teams or Enterprise, recommended for most teams
- Claude Console
- Amazon Bedrock
- Google Vertex AI
- Microsoft Foundry

Claude for Teams or Enterprise

Claude for Teams and Claude for Enterprise provide the best experience for organizations using Claude Code. Team members get access to both Claude Code and Claude on the web with centralized billing and team management.

- **Claude for Teams**: self-service plan with collaboration features, admin tools, and billing management. Best for smaller teams.
- **Claude for Enterprise**: adds SSO, domain capture, role-based permissions, compliance API, and managed policy settings for organization-wide Claude Code configurations. Best for larger organizations with security and compliance requirements.

Step 1: Subscribe

Subscribe to Claude for Teams or contact sales for Claude for Enterprise.

Step 2: Invite team members

Invite team members from the admin dashboard.

Step 3: Install and log in

Team members install Claude Code and log in with their Claude.ai accounts.

Claude Console authentication

For organizations that prefer API-based billing, you can set up access through the Claude Console.

Step 1: Create or use a Console account

Use your existing Claude Console account or create a new one.

Step 2: Add users

You can add users through either method:

- Bulk invite users from within the Console: Settings -> Members -> Invite

- Set up SSO

Step 3: Assign roles

When inviting users, assign one of:

- **Claude Code** role: users can only create Claude Code API keys
- **Developer** role: users can create any kind of API key

Step 4: Users complete setup

Each invited user needs to:

- Accept the Console invite
- Check system requirements
- Install Claude Code
- Log in with Console account credentials

Cloud provider authentication

For teams using Amazon Bedrock, Google Vertex AI, or Microsoft Foundry:

Step 1: Follow provider setup

Follow the Bedrock docs, Vertex docs, or Microsoft Foundry docs.

Step 2: Distribute configuration

Distribute the environment variables and instructions for generating cloud credentials to your users. Read more about how to manage configuration here.

Step 3: Install Claude Code

Users can install Claude Code.

Credential management

Claude Code securely manages your authentication credentials:

- **Storage location**: on macOS, credentials are stored in the encrypted macOS Keychain.
- **Supported authentication types**: Claude.ai credentials, Claude API credentials, Azure Auth, Bedrock Auth, and Vertex Auth.
- **Custom credential scripts**: the `apiKeyHelper` setting can be configured to run a shell script that returns an API key.

- **Refresh intervals**: by default, `apiKeyHelper` is called after 5 minutes or on HTTP 401 response. Set `CLAUDE_CODE_API_KEY_HELPER_TTL_MS` environment variable for custom refresh intervals.

Security

> *Learn about Claude Code's security safeguards and best practices for safe usage.*

How we approach security

Security foundation

Your code's security is paramount. Claude Code is built with security at its core, developed according to Anthropic's comprehensive security program. Learn more and access resources (SOC 2 Type 2 report, ISO 27001 certificate, etc.) at Anthropic Trust Center.

Permission-based architecture

Claude Code uses strict read-only permissions by default. When additional actions are needed (editing files, running tests, executing commands), Claude Code requests explicit permission. Users control whether to approve actions once or allow them automatically.

We designed Claude Code to be transparent and secure. For example, we require approval for bash commands before executing them, giving you direct control. This approach enables users and organizations to configure permissions directly.

For detailed permission configuration, see Permissions.

Built-in protections

To mitigate risks in agentic systems:

- **Sandboxed bash tool**: Sandbox bash commands with filesystem and network isolation, reducing permission prompts while maintaining security. Enable with `/sandbox` to define boundaries where Claude Code can work autonomously
- **Write access restriction**: Claude Code can only write to the folder where it was started and its subfolders—it cannot modify files in parent directories without explicit permission. While Claude Code can read files outside the working directory (useful for accessing system libraries and dependencies), write operations are strictly confined to the project scope, creating a clear security boundary
- **Prompt fatigue mitigation**: Support for allowlisting frequently used safe commands per-user, per-codebase, or per-organization
- **Accept Edits mode**: Batch accept multiple edits while maintaining permission prompts for commands with side effects

User responsibility

Claude Code only has the permissions you grant it. You're responsible for reviewing proposed code and commands for safety before approval.

Protect against prompt injection

Prompt injection is a technique where an attacker attempts to override or manipulate an AI assistant's instructions by inserting malicious text. Claude Code includes several safeguards against these attacks:

Core protections

- **Permission system**: Sensitive operations require explicit approval
- **Context-aware analysis**: Detects potentially harmful instructions by analyzing the full request
- **Input sanitization**: Prevents command injection by processing user inputs
- **Command blocklist**: Blocks risky commands that fetch arbitrary content from the web like `curl` and `wget` by default. When explicitly allowed, be aware of permission pattern limitations

Privacy safeguards

We have implemented several safeguards to protect your data, including:

- Limited retention periods for sensitive information (see the Privacy Center to learn more)
- Restricted access to user session data
- User control over data training preferences. Consumer users can change their privacy settings at any time.

For full details, please review our Commercial Terms of Service (for Team, Enterprise, and API users) or Consumer Terms (for Free, Pro, and Max users) and Privacy Policy.

Additional safeguards

- **Network request approval**: Tools that make network requests require user approval by default
- **Isolated context windows**: Web fetch uses a separate context window to avoid injecting potentially malicious prompts

- **Trust verification**: First-time codebase runs and new MCP servers require trust verification
  - Note: Trust verification is disabled when running non-interactively with the `-p` flag
- **Command injection detection**: Suspicious bash commands require manual approval even if previously allowlisted
- **Fail-closed matching**: Unmatched commands default to requiring manual approval
- **Natural language descriptions**: Complex bash commands include explanations for user understanding
- **Secure credential storage**: API keys and tokens are encrypted. See Credential Management

> **Warning:**
>
> **Windows WebDAV security risk**: When running Claude Code on Windows, we recommend against enabling WebDAV or allowing Claude Code to access paths such as `\\*` that may contain WebDAV subdirectories. WebDAV has been deprecated by Microsoft due to security risks. Enabling WebDAV may allow Claude Code to trigger network requests to remote hosts, bypassing the permission system.

Best practices for working with untrusted content:

1. Review suggested commands before approval
2. Avoid piping untrusted content directly to Claude
3. Verify proposed changes to critical files
4. Use virtual machines (VMs) to run scripts and make tool calls, especially when interacting with external web services
5. Report suspicious behavior with `/bug`

> **Warning:**
>
> While these protections significantly reduce risk, no system is completely immune to all attacks. Always maintain good security practices when working with any AI tool.

MCP security

Claude Code allows users to configure Model Context Protocol (MCP) servers. The list of allowed MCP servers is configured in your source code, as part of Claude Code settings engineers check into source control.

We encourage either writing your own MCP servers or using MCP servers from providers that you trust. You are able to configure Claude Code permissions for MCP servers. Anthropic does not manage or audit any MCP servers.

IDE security

See VS Code security and privacy for more information on running Claude Code in an IDE.

Cloud execution security

When using Claude Code on the web, additional security controls are in place:

- **Isolated virtual machines**: Each cloud session runs in an isolated, Anthropic-managed VM
- **Network access controls**: Network access is limited by default and can be configured to be disabled or allow only specific domains
- **Credential protection**: Authentication is handled through a secure proxy that uses a scoped credential inside the sandbox, which is then translated to your actual GitHub authentication token
- **Branch restrictions**: Git push operations are restricted to the current working branch
- **Audit logging**: All operations in cloud environments are logged for compliance and audit purposes
- **Automatic cleanup**: Cloud environments are automatically terminated after session completion

For more details on cloud execution, see Claude Code on the web.

Remote Control sessions work differently: the web interface connects to a Claude Code process running on your local machine. All code execution and file access stays local, and the same data that flows during any local Claude Code session travels through the Anthropic API over TLS. No cloud VMs or sandboxing are involved. The connection uses multiple short-lived, narrowly scoped credentials, each limited to a specific purpose and expiring independently, to limit the blast radius of any single compromised credential.

Security best practices

Working with sensitive code

- Review all suggested changes before approval
- Use project-specific permission settings for sensitive repositories

- Consider using devcontainers for additional isolation
- Regularly audit your permission settings with `/permissions`

Team security

- Use managed settings to enforce organizational standards
- Share approved permission configurations through version control
- Train team members on security best practices
- Monitor Claude Code usage through OpenTelemetry metrics
- Audit or block settings changes during sessions with `ConfigChange` hooks

Reporting security issues

If you discover a security vulnerability in Claude Code:

1. Do not disclose it publicly
2. Report it through our HackerOne program
3. Include detailed reproduction steps
4. Allow time for us to address the issue before public disclosure

Related resources

- Sandboxing - Filesystem and network isolation for bash commands
- Permissions - Configure permissions and access controls
- Monitoring usage - Track and audit Claude Code activity
- Development containers - Secure, isolated environments
- Anthropic Trust Center - Security certifications and compliance

Sandboxing

> *Learn how Claude Code's sandboxed bash tool provides filesystem and network isolation for safer, more autonomous agent execution.*

Overview

Claude Code features native sandboxing to provide a more secure environment for agent execution while reducing the need for constant permission prompts. Instead of asking permission for each bash command, sandboxing creates defined boundaries upfront where Claude Code can work more freely with reduced risk.

The sandboxed bash tool uses OS-level primitives to enforce both filesystem and network isolation.

Why sandboxing matters

Traditional permission-based security requires constant user approval for bash commands. While this provides control, it can lead to:

- **Approval fatigue**: Repeatedly clicking "approve" can cause users to pay less attention to what they're approving
- **Reduced productivity**: Constant interruptions slow down development workflows
- **Limited autonomy**: Claude Code cannot work as efficiently when waiting for approvals

Sandboxing addresses these challenges by:

1. **Defining clear boundaries**: Specify exactly which directories and network hosts Claude Code can access
2. **Reducing permission prompts**: Safe commands within the sandbox don't require approval
3. **Maintaining security**: Attempts to access resources outside the sandbox trigger immediate notifications
4. **Enabling autonomy**: Claude Code can run more independently within defined limits

> **Warning:**

> Effective sandboxing requires **both** filesystem and network isolation. Without network isolation, a compromised agent could exfiltrate sensitive files like SSH keys. Without filesystem isolation, a compromised agent could backdoor system resources to gain network access. When configuring sandboxing it is important to ensure that your configured settings do not create bypasses in these systems.

How it works

Filesystem isolation

The sandboxed bash tool restricts file system access to specific directories:

- **Default writes behavior**: Read and write access to the current working directory and its subdirectories
- **Default read behavior**: Read access to the entire computer, except certain denied directories
- **Blocked access**: Cannot modify files outside the current working directory without explicit permission
- **Configurable**: Define custom allowed and denied paths through settings

You can grant write access to additional paths using `sandbox.filesystem.allowWrite` in your settings. These restrictions are enforced at the OS level (Seatbelt on macOS, bubblewrap on Linux), so they apply to all subprocess commands, including tools like `kubectl`, `terraform`, and `npm`, not just Claude's file tools.

Network isolation

Network access is controlled through a proxy server running outside the sandbox:

- **Domain restrictions**: Only approved domains can be accessed
- **User confirmation**: New domain requests trigger permission prompts (unless `allowManagedDomainsOnly` is enabled, which blocks non-allowed domains automatically)
- **Custom proxy support**: Advanced users can implement custom rules on outgoing traffic
- **Comprehensive coverage**: Restrictions apply to all scripts, programs, and subprocesses spawned by commands

OS-level enforcement

The sandboxed bash tool leverages operating system security primitives:

- **macOS**: Uses Seatbelt for sandbox enforcement

- **Linux**: Uses bubblewrap for isolation
- **WSL2**: Uses bubblewrap, same as Linux

WSL1 is not supported because bubblewrap requires kernel features only available in WSL2.

These OS-level restrictions ensure that all child processes spawned by Claude Code's commands inherit the same security boundaries.

Getting started

Prerequisites

On **macOS**, sandboxing works out of the box using the built-in Seatbelt framework.

On **Linux and WSL2**, install the required packages first:

Ubuntu/Debian

```
sudo apt-get install bubblewrap socat
```

Fedora

```
sudo dnf install bubblewrap socat
```

Enable sandboxing

You can enable sandboxing by running the `/sandbox` command:

```
/sandbox
```

This opens a menu where you can choose between sandbox modes. If required dependencies are missing (such as `bubblewrap` or `socat` on Linux), the menu displays installation instructions for your platform.

Sandbox modes

Claude Code offers two sandbox modes:

Auto-allow mode: Bash commands will attempt to run inside the sandbox and are automatically allowed without requiring permission. Commands that cannot be sandboxed (such as those needing network access to non-allowed hosts) fall back to the regular permission flow. Explicit ask/deny rules you've configured are always respected.

Regular permissions mode: All bash commands go through the standard permission flow, even when sandboxed. This provides more control but requires more approvals.

In both modes, the sandbox enforces the same filesystem and network restrictions. The difference is only in whether sandboxed commands are auto-approved or require explicit permission.

> **Info:**
>
> Auto-allow mode works independently of your permission mode setting. Even if you're not in "accept edits" mode, sandboxed bash commands will run automatically when auto-allow is enabled. This means bash commands that modify files within the sandbox boundaries will execute without prompting, even when file edit tools would normally require approval.

Configure sandboxing

Customize sandbox behavior through your `settings.json` file. See Settings for complete configuration reference.

Granting subprocess write access to specific paths

By default, sandboxed commands can only write to the current working directory. If subprocess commands like `kubectl`, `terraform`, or `npm` need to write outside the project directory, use `sandbox.filesystem.allowWrite` to grant access to specific paths:

```
{
  "sandbox": {
    "enabled": true,
    "filesystem": {
      "allowWrite": ["~/.kube", "//tmp/build"]
    }
  }
}
```

These paths are enforced at the OS level, so all commands running inside the sandbox, including their child processes, respect them. This is the recommended approach when a tool needs write access to a specific location, rather than excluding the tool from the sandbox entirely with `excludedCommands`.

When `allowWrite` (or `denyWrite` / `denyRead`) is defined in multiple settings scopes, the arrays are **merged**, meaning paths from every scope are combined, not replaced. For example, if managed settings allow writes to `//opt/company-tools` and a user adds

`~/.kube` in their personal settings, both paths are included in the final sandbox configuration. This means users and projects can extend the list without duplicating or overriding paths set by higher-priority scopes.

Path prefixes control how paths are resolved:

Prefix	Meaning	Example
`//`	Absolute path from filesystem root	`//tmp/build` becomes `/tmp/build`
`~/`	Relative to home directory	`~/.kube` becomes `$HOME/.kube`
`/`	Relative to the settings file's directory	`/build` becomes `$SETTINGS_DIR/build`
`./` or no prefix	Relative path (resolved by sandbox runtime)	`./output`

You can also deny write or read access using `sandbox.filesystem.denyWrite` and `sandbox.filesystem.denyRead`. These are merged with any paths from `Edit(...)` and `Read(...)` permission rules.

Tip:

Not all commands are compatible with sandboxing out of the box. Some notes that may help you make the most out of the sandbox:

- Many CLI tools require accessing certain hosts. As you use these tools, they will request permission to access certain hosts. Granting permission will allow them to access these hosts now and in the future, enabling them to safely execute inside the sandbox.
- `watchman` is incompatible with running in the sandbox. If you're running `jest`, consider using `jest --no-watchman`
- `docker` is incompatible with running in the sandbox. Consider specifying `docker` in `excludedCommands` to force it to run outside of the sandbox.

Note:

Claude Code includes an intentional escape hatch mechanism that allows commands to run outside the sandbox when necessary. When a command fails due to sandbox restrictions (such as network connectivity issues or incompatible tools), Claude is prompted to analyze the failure and may retry the command with the `dangerouslyDisableSandbox` parameter. Commands

> that use this parameter go through the normal Claude Code permissions flow requiring user permission to execute. This allows Claude Code to handle edge cases where certain tools or network operations cannot function within sandbox constraints.
>
> You can disable this escape hatch by setting `"allowUnsandboxedCommands": false` in your sandbox settings. When disabled, the `dangerouslyDisableSandbox` parameter is completely ignored and all commands must run sandboxed or be explicitly listed in `excludedCommands`.

Security benefits

Protection against prompt injection

Even if an attacker successfully manipulates Claude Code's behavior through prompt injection, the sandbox ensures your system remains secure:

Filesystem protection:

- Cannot modify critical config files such as `~/.bashrc`
- Cannot modify system-level files in `/bin/`
- Cannot read files that are denied in your Claude permission settings

Network protection:

- Cannot exfiltrate data to attacker-controlled servers
- Cannot download malicious scripts from unauthorized domains
- Cannot make unexpected API calls to unapproved services
- Cannot contact any domains not explicitly allowed

Monitoring and control:

- All access attempts outside the sandbox are blocked at the OS level
- You receive immediate notifications when boundaries are tested
- You can choose to deny, allow once, or permanently update your configuration

Reduced attack surface

Sandboxing limits the potential damage from:

- **Malicious dependencies**: NPM packages or other dependencies with harmful code
- **Compromised scripts**: Build scripts or tools with security vulnerabilities
- **Social engineering**: Attacks that trick users into running dangerous commands
- **Prompt injection**: Attacks that trick Claude into running dangerous commands

Transparent operation

When Claude Code attempts to access network resources outside the sandbox:

1. The operation is blocked at the OS level
2. You receive an immediate notification
3. You can choose to:
    - Deny the request
    - Allow it once
    - Update your sandbox configuration to permanently allow it

Security Limitations

- Network Sandboxing Limitations: The network filtering system operates by restricting the domains that processes are allowed to connect to. It does not otherwise inspect the traffic passing through the proxy and users are responsible for ensuring they only allow trusted domains in their policy.

> **Warning:**
>
> Users should be aware of potential risks that come from allowing broad domains like `github.com` that may allow for data exfiltration. Also, in some cases it may be possible to bypass the network filtering through domain fronting.

- Privilege Escalation via Unix Sockets: The `allowUnixSockets` configuration can inadvertently grant access to powerful system services that could lead to sandbox bypasses. For example, if it is used to allow access to `/var/run/docker.sock` this would effectively grant access to the host system through exploiting the docker socket. Users are encouraged to carefully consider any unix sockets that they allow through the sandbox.
- Filesystem Permission Escalation: Overly broad filesystem write permissions can enable privilege escalation attacks. Allowing writes to directories containing executables in `$PATH`, system configuration directories, or user shell configuration files (`.bashrc`, `.zshrc`) can lead to code execution in different security contexts when other users or system processes access these files.
- Linux Sandbox Strength: The Linux implementation provides strong filesystem and network isolation but includes an `enableWeakerNestedSandbox` mode that enables it to work inside of Docker environments without privileged namespaces. This option considerably weakens security and should only be used in cases where additional isolation is otherwise enforced.

How sandboxing relates to permissions

Sandboxing and permissions are complementary security layers that work together:

- **Permissions** control which tools Claude Code can use and are evaluated before any tool runs. They apply to all tools: Bash, Read, Edit, WebFetch, MCP, and others.
- **Sandboxing** provides OS-level enforcement that restricts what Bash commands can access at the filesystem and network level. It applies only to Bash commands and their child processes.

Filesystem and network restrictions are configured through both sandbox settings and permission rules:

- Use `sandbox.filesystem.allowWrite` to grant subprocess write access to paths outside the working directory
- Use `sandbox.filesystem.denyWrite` and `sandbox.filesystem.denyRead` to block subprocess access to specific paths
- Use `Read` and `Edit` deny rules to block access to specific files or directories
- Use `WebFetch` allow/deny rules to control domain access
- Use sandbox `allowedDomains` to control which domains Bash commands can reach

Paths from both `sandbox.filesystem` settings and permission rules are merged together into the final sandbox configuration.

This repository includes starter settings configurations for common deployment scenarios, including sandbox-specific examples. Use these as starting points and adjust them to fit your needs.

Advanced usage

Custom proxy configuration

For organizations requiring advanced network security, you can implement a custom proxy to:

- Decrypt and inspect HTTPS traffic
- Apply custom filtering rules
- Log all network requests
- Integrate with existing security infrastructure

```
{
  "sandbox": {
    "network": {
      "httpProxyPort": 8080,
      "socksProxyPort": 8081
    }
  }
}
```

Integration with existing security tools

The sandboxed bash tool works alongside:

- **Permission rules**: Combine with permission settings for defense-in-depth
- **Development containers**: Use with devcontainers for additional isolation
- **Enterprise policies**: Enforce sandbox configurations through managed settings

Best practices

1. **Start restrictive**: Begin with minimal permissions and expand as needed
2. **Monitor logs**: Review sandbox violation attempts to understand Claude Code's needs
3. **Use environment-specific configs**: Different sandbox rules for development vs. production contexts
4. **Combine with permissions**: Use sandboxing alongside IAM policies for comprehensive security
5. **Test configurations**: Verify your sandbox settings don't block legitimate workflows

Open source

The sandbox runtime is available as an open source npm package for use in your own agent projects. This enables the broader AI agent community to build safer, more secure autonomous systems. This can also be used to sandbox other programs you may wish to run. For example, to sandbox an MCP server you could run:

```
npx @anthropic-ai/sandbox-runtime <command-to-sandbox>
```

For implementation details and source code, visit the GitHub repository.

Limitations

- **Performance overhead**: Minimal, but some filesystem operations may be slightly slower
- **Compatibility**: Some tools that require specific system access patterns may need configuration adjustments, or may even need to be run outside of the sandbox
- **Platform support**: Supports macOS, Linux, and WSL2. WSL1 is not supported. Native Windows support is planned.

See also

- Security - Comprehensive security features and best practices
- Permissions - Permission configuration and access control
- Settings - Complete configuration reference
- CLI reference - Command-line options

Checkpointing

> *Track, rewind, and summarize Claude's edits and conversation to manage session state.*

Claude Code automatically tracks Claude's file edits as you work, allowing you to quickly undo changes and rewind to previous states if anything gets off track.

How checkpoints work

As you work with Claude, checkpointing automatically captures the state of your code before each edit. This safety net lets you pursue ambitious, wide-scale tasks knowing you can always return to a prior code state.

Automatic tracking

Claude Code tracks all changes made by its file editing tools:

- Every user prompt creates a new checkpoint
- Checkpoints persist across sessions, so you can access them in resumed conversations
- Automatically cleaned up along with sessions after 30 days (configurable)

Rewind and summarize

Press `Esc` twice (`Esc` + `Esc`) or use the `/rewind` command to open the rewind menu. A scrollable list shows each of your prompts from the session. Select the point you want to act on, then choose an action:

- **Restore code and conversation**: revert both code and conversation to that point
- **Restore conversation**: rewind to that message while keeping current code
- **Restore code**: revert file changes while keeping the conversation
- **Summarize from here**: compress the conversation from this point forward into a summary, freeing context window space
- **Never mind**: return to the message list without making changes

After restoring the conversation or summarizing, the original prompt from the selected message is restored into the input field so you can re-send or edit it.

Restore vs. summarize

The three restore options revert state: they undo code changes, conversation history, or both. “Summarize from here” works differently:

- Messages before the selected message stay intact
- The selected message and all subsequent messages get replaced with a compact AI-generated summary
- No files on disk are changed
- The original messages are preserved in the session transcript, so Claude can reference the details if needed

This is similar to `/compact`, but targeted: instead of summarizing the entire conversation, you keep early context in full detail and only compress the parts that are using up space. You can type optional instructions to guide what the summary focuses on.

> **Note:**
>
> Summarize keeps you in the same session and compresses context. If you want to branch off and try a different approach while preserving the original session intact, use fork instead (`claude --continue --fork-session`).

Common use cases

Checkpoints are particularly useful when:

- **Exploring alternatives**: try different implementation approaches without losing your starting point
- **Recovering from mistakes**: quickly undo changes that introduced bugs or broke functionality
- **Iterating on features**: experiment with variations knowing you can revert to working states
- **Freeing context space**: summarize a verbose debugging session from the midpoint forward, keeping your initial instructions intact

Limitations

Bash command changes not tracked

Checkpointing does not track files modified by bash commands. For example, if Claude Code runs:

```
rm file.txt
mv old.txt new.txt
cp source.txt dest.txt
```

These file modifications cannot be undone through rewind. Only direct file edits made through Claude's file editing tools are tracked.

External changes not tracked

Checkpointing only tracks files that have been edited within the current session. Manual changes you make to files outside of Claude Code and edits from other concurrent sessions are normally not captured, unless they happen to modify the same files as the current session.

Not a replacement for version control

Checkpoints are designed for quick, session-level recovery. For permanent version history and collaboration:

- Continue using version control (ex. Git) for commits, branches, and long-term history
- Checkpoints complement but don't replace proper version control
- Think of checkpoints as "local undo" and Git as "permanent history"

See also

- Interactive mode - Keyboard shortcuts and session controls
- Built-in commands - Accessing checkpoints using `/rewind`
- CLI reference - Command-line options

Data usage

> *Learn about Anthropic's data usage policies for Claude*

Data policies

Data training policy

Consumer users (Free, Pro, and Max plans): We give you the choice to allow your data to be used to improve future Claude models. We will train new models using data from Free, Pro, and Max accounts when this setting is on (including when you use Claude Code from these accounts).

Commercial users: (Team and Enterprise plans, API, 3rd-party platforms, and Claude Gov) maintain existing policies: Anthropic does not train generative models using code or prompts sent to Claude Code under commercial terms, unless the customer has chosen to provide their data to us for model improvement (for example, the Developer Partner Program).

Development Partner Program

If you explicitly opt in to methods to provide us with materials to train on, such as via the Development Partner Program, we may use those materials provided to train our models. An organization admin can expressly opt-in to the Development Partner Program for their organization. Note that this program is available only for Anthropic first-party API, and not for Bedrock or Vertex users.

Feedback using the `/bug` command

If you choose to send us feedback about Claude Code using the `/bug` command, we may use your feedback to improve our products and services. Transcripts shared via `/bug` are retained for 5 years.

Session quality surveys

When you see the "How is Claude doing this session?" prompt in Claude Code, responding to this survey (including selecting "Dismiss"), only your numeric rating (1, 2, 3, or dismiss) is recorded. We do not collect or store any conversation transcripts, inputs, outputs, or other session data as part of this survey. Unlike thumbs up/down feedback or `/bug` re-

ports, this session quality survey is a simple product satisfaction metric. Your responses to this survey do not impact your data training preferences and cannot be used to train our AI models.

To disable these surveys, set `CLAUDE_CODE_DISABLE_FEEDBACK_SURVEY=1`. The survey is also disabled when `DISABLE_TELEMETRY` or `CLAUDE_CODE_DISABLE_NONESSENTIAL_TRAFFIC` is set. To control frequency instead of disabling, set `feedbackSurveyRate` in your settings file to a probability between `0` and `1`.

Data retention

Anthropic retains Claude Code data based on your account type and preferences.

Consumer users (Free, Pro, and Max plans):

- Users who allow data use for model improvement: 5-year retention period to support model development and safety improvements
- Users who don't allow data use for model improvement: 30-day retention period
- Privacy settings can be changed at any time at claude.ai/settings/data-privacy-controls.

Commercial users (Team, Enterprise, and API):

- Standard: 30-day retention period
- Zero data retention: available for Claude Code on Claude for Enterprise. ZDR is enabled on a per-organization basis; each new organization must have ZDR enabled separately by your account team
- Local caching: Claude Code clients may store sessions locally for up to 30 days to enable session resumption (configurable)

You can delete individual Claude Code on the web sessions at any time. Deleting a session permanently removes the session's event data. For instructions on how to delete sessions, see Managing sessions.

Learn more about data retention practices in our Privacy Center.

For full details, please review our Commercial Terms of Service (for Team, Enterprise, and API users) or Consumer Terms (for Free, Pro, and Max users) and Privacy Policy.

Data access

For all first party users, you can learn more about what data is logged for local Claude Code and remote Claude Code. Remote Control sessions follow the local data flow since all execution happens on your machine. Note for remote Claude Code, Claude accesses the repository where you initiate your Claude Code session. Claude does not access repositories that you have connected but have not started a session in.

Local Claude Code: Data flow and dependencies

The diagram below shows how Claude Code connects to external services during installation and normal operation. Solid lines indicate required connections, while dashed lines represent optional or user-initiated data flows.

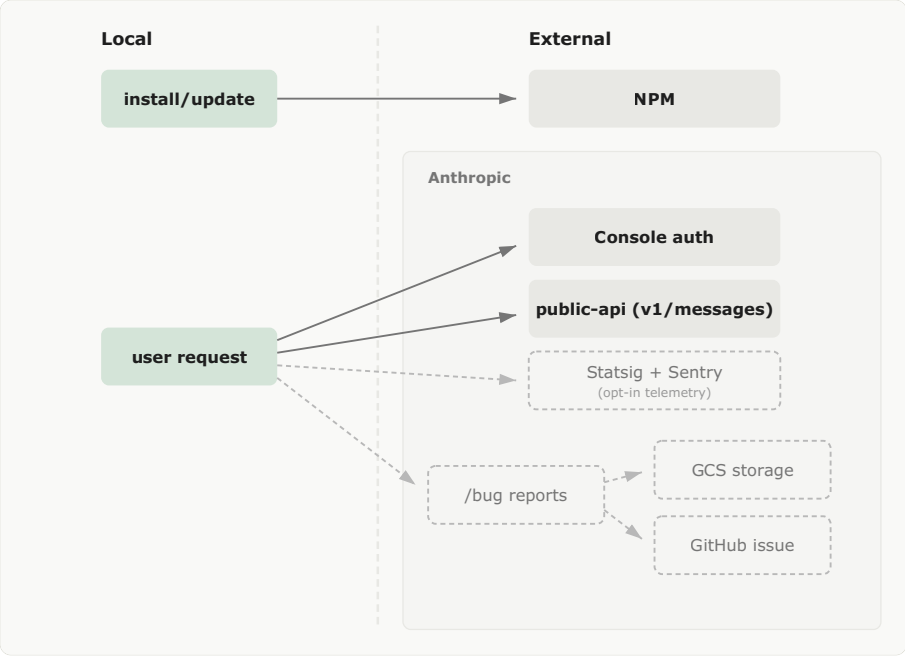


Diagram showing Claude Code's external connections: install/update connects to NPM, and user requests connect to Anthropic services including Console auth, public-api, and optionally Statsig, Sentry, and bug reporting

Claude Code is installed from NPM. Claude Code runs locally. In order to interact with the LLM, Claude Code sends data over the network. This data includes all user prompts and model outputs. The data is encrypted in transit via TLS and is not encrypted at rest. Claude Code is compatible with most popular VPNs and LLM proxies.

Claude Code is built on Anthropic's APIs. For details regarding our API's security controls, including our API logging procedures, please refer to compliance artifacts offered in the Anthropic Trust Center.

Cloud execution: Data flow and dependencies

When using Claude Code on the web, sessions run in Anthropic-managed virtual machines instead of locally. In cloud environments:

- **Code and data storage:** Your repository is cloned to an isolated VM. Code and session data are subject to the retention and usage policies for your account type (see Data retention section above)
- **Credentials:** GitHub authentication is handled through a secure proxy; your GitHub credentials never enter the sandbox
- **Network traffic:** All outbound traffic goes through a security proxy for audit logging and abuse prevention
- **Session data:** Prompts, code changes, and outputs follow the same data policies as local Claude Code usage

For security details about cloud execution, see Security.

Telemetry services

Claude Code connects from users' machines to the Statsig service to log operational metrics such as latency, reliability, and usage patterns. This logging does not include any code or file paths. Data is encrypted in transit using TLS and at rest using 256-bit AES encryption. Read more in the Statsig security documentation. To opt out of Statsig telemetry, set the `DISABLE_TELEMETRY` environment variable.

Claude Code connects from users' machines to Sentry for operational error logging. The data is encrypted in transit using TLS and at rest using 256-bit AES encryption. Read more in the Sentry security documentation. To opt out of error logging, set the `DISABLE_ERROR_REPORTING` environment variable.

When users run the `/bug` command, a copy of their full conversation history including code is sent to Anthropic. The data is encrypted in transit and at rest. Optionally, a Github issue is created in our public repository. To opt out of bug reporting, set the `DISABLE_BUG_COMMAND` environment variable.

Default behaviors by API provider

By default, error reporting, telemetry, and bug reporting are disabled when using Bedrock, Vertex, or Foundry. Session quality surveys are the exception and appear regardless of provider. You can opt out of all non-essential traffic, including surveys, at once by setting `CLAUDE_CODE_DISABLE_NONESSENTIAL_TRAFFIC`. Here are the full default behaviors:

Service	Claude API	Vertex API	Bedrock API	Foundry API
Statsig (Metrics)	Default on. `DISABLE_TELEMETRY=1` to disable.	Default off. `CLAUDE_CODE_USE_VERTEX` must be 1.	Default off. `CLAUDE_CODE_USE_BEDROCK` must be 1.	Default off. `CLAUDE_CODE_USE_FOUNDRY` must be 1.
Sentry (Errors)	Default on. `DISABLE_ERROR_REPORTING=1` to disable.	Default off. `CLAUDE_CODE_USE_VERTEX` must be 1.	Default off. `CLAUDE_CODE_USE_BEDROCK` must be 1.	Default off. `CLAUDE_CODE_USE_FOUNDRY` must be 1.
Claude API (`/bug` reports)	Default on. `DISABLE_BUG_COMMAND=1` to disable.	Default off. `CLAUDE_CODE_USE_VERTEX` must be 1.	Default off. `CLAUDE_CODE_USE_BEDROCK` must be 1.	Default off. `CLAUDE_CODE_USE_FOUNDRY` must be 1.
Session quality surveys	Default on. `CLAUDE_CODE_DISABLE_FEEDBACK_SURVEY=1` to disable.	Default on. `CLAUDE_CODE_DISABLE_FEEDBACK_SURVEY=1` to disable.	Default on. `CLAUDE_CODE_DISABLE_FEEDBACK_SURVEY=1` to disable.	Default on. `CLAUDE_CODE_DISABLE_FEEDBACK_SURVEY=1` to disable.

All environment variables can be checked into `settings.json` (read more).

Legal and compliance

> *Legal agreements, compliance certifications, and security information for Claude Code.*

Legal agreements

License

Your use of Claude Code is subject to:

- Commercial Terms - for Team, Enterprise, and Claude API users
- Consumer Terms of Service - for Free, Pro, and Max users

Commercial agreements

Whether you're using the Claude API directly (1P) or accessing it through AWS Bedrock or Google Vertex (3P), your existing commercial agreement will apply to Claude Code usage, unless we've mutually agreed otherwise.

Compliance

Healthcare compliance (BAA)

If a customer has a Business Associate Agreement (BAA) with us, and wants to use Claude Code, the BAA will automatically extend to cover Claude Code if the customer has executed a BAA and has Zero Data Retention (ZDR) activated. The BAA will be applicable to that customer's API traffic flowing through Claude Code. ZDR is enabled on a per-organization basis, so each organization must have ZDR enabled separately to be covered under the BAA.

Usage policy

Acceptable use

Claude Code usage is subject to the Anthropic Usage Policy. Advertised usage limits for Pro and Max plans assume ordinary, individual usage of Claude Code and the Agent SDK.

Authentication and credential use

Claude Code authenticates with Anthropic's servers using OAuth tokens or API keys. These authentication methods serve different purposes:

- **OAuth authentication** (used with Free, Pro, and Max plans) is intended exclusively for Claude Code and Claude.ai. Using OAuth tokens obtained through Claude Free, Pro, or Max accounts in any other product, tool, or service — including the Agent SDK — is not permitted and constitutes a violation of the Consumer Terms of Service.
- **Developers** building products or services that interact with Claude's capabilities, including those using the Agent SDK, should use API key authentication through Claude Console or a supported cloud provider. Anthropic does not permit third-party developers to offer Claude.ai login or to route requests through Free, Pro, or Max plan credentials on behalf of their users.

Anthropic reserves the right to take measures to enforce these restrictions and may do so without prior notice.

For questions about permitted authentication methods for your use case, please contact sales.

Security and trust

Trust and safety

You can find more information in the Anthropic Trust Center and Transparency Hub.

Security vulnerability reporting

Anthropic manages our security program through HackerOne. Use this form to report vulnerabilities.

Zero data retention

> *Learn about Zero Data Retention (ZDR) for Claude Code on Claude for Enterprise, including scope, disabled features, and how to request enablement.*

Zero Data Retention (ZDR) is available for Claude Code when used through Claude for Enterprise. When ZDR is enabled, prompts and model responses generated during Claude Code sessions are processed in real time and not stored by Anthropic after the response is returned, except where needed to comply with law or combat misuse.

ZDR on Claude for Enterprise gives enterprise customers the ability to use Claude Code with zero data retention and access administrative capabilities:

- Cost controls per user
- Analytics dashboard
- Server-managed settings
- Audit logs

ZDR for Claude Code on Claude for Enterprise applies only to Anthropic's direct platform. For Claude deployments on AWS Bedrock, Google Vertex AI, or Microsoft Foundry, refer to those platforms' data retention policies.

ZDR scope

ZDR covers Claude Code inference on Claude for Enterprise.

> **Warning:**
>
> ZDR is enabled on a per-organization basis. Each new organization requires ZDR to be enabled separately by your Anthropic account team. ZDR does not automatically apply to new organizations created under the same account. Contact your account team to enable ZDR for any new organizations.

What ZDR covers

ZDR covers model inference calls made through Claude Code on Claude for Enterprise. When you use Claude Code in your terminal, the prompts you send and the responses Claude generates are not retained by Anthropic. This applies regardless of which Claude model is used.

What ZDR does not cover

ZDR does not extend to the following, even for organizations with ZDR enabled. These features follow standard data retention policies:

Feature	Details
Chat on claude.ai	Chat conversations through the Claude for Enterprise web interface are not covered by ZDR.
Cowork	Cowork sessions are not covered by ZDR.
Claude Code Analytics	Does not store prompts or model responses, but collects productivity metadata such as account emails and usage statistics. Contribution metrics are not available for ZDR organizations; the analytics dashboard shows usage metrics only.
User and seat management	Administrative data such as account emails and seat assignments is retained under standard policies.
Third-party integrations	Data processed by third-party tools, MCP servers, or other external integrations is not covered by ZDR. Review those services' data handling practices independently.

Features disabled under ZDR

When ZDR is enabled for a Claude Code organization on Claude for Enterprise, certain features that require storing prompts or completions are automatically disabled at the backend level:

Feature	Reason
Claude Code on the Web	Requires server-side storage of conversation history.
Remote sessions from the Desktop app	Requires persistent session data that includes prompts and completions.
Feedback submission (`/feedback`)	Submitting feedback sends conversation data to Anthropic.

These features are blocked in the backend regardless of client-side display. If you see a disabled feature in the Claude Code terminal during startup, attempting to use it returns an error indicating the organization's policies do not allow that action.

Future features may also be disabled if they require storing prompts or completions.

Data retention for policy violations

Even with ZDR enabled, Anthropic may retain data where required by law or to address Usage Policy violations. If a session is flagged for a policy violation, Anthropic may retain the associated inputs and outputs for up to 2 years, consistent with Anthropic's standard ZDR policy.

Request ZDR

To request ZDR for Claude Code on Claude for Enterprise, contact your Anthropic account team. Your account team will submit the request internally, and Anthropic will review and enable ZDR on your organization after confirming eligibility. All enablement actions are audit-logged.

If you are currently using ZDR for Claude Code via pay-as-you-go API keys, you can transition to Claude for Enterprise to gain access to administrative features while maintaining ZDR for Claude Code. Contact your account team to coordinate the migration.

Part 10: Enterprise & Monitoring

Track team usage with analytics

> *View Claude Code usage metrics, track adoption, and measure engineering velocity in the analytics dashboard.*

Claude Code provides analytics dashboards to help organizations understand developer usage patterns, track contribution metrics, and measure how Claude Code impacts engineering velocity. Access the dashboard for your plan:

Plan	Dashboard URL	Includes	Read more
Claude for Teams / Enterprise	claude.ai/analytics/claude-code	Usage metrics, contribution metrics with GitHub integration, leaderboard, data export	Details
API (Claude Console)	platform.claude.com/claude-code	Usage metrics, spend tracking, team insights	Details

Access analytics for Teams and Enterprise

Navigate to claude.ai/analytics/claude-code. Admins and Owners can view the dashboard.

The Teams and Enterprise dashboard includes:

- **Usage metrics**: lines of code accepted, suggestion accept rate, daily active users and sessions
- **Contribution metrics**: PRs and lines of code shipped with Claude Code assistance, with GitHub integration
- **Leaderboard**: top contributors ranked by Claude Code usage
- **Data export**: download contribution data as CSV for custom reporting

Enable contribution metrics

> **Note:**
>
> Contribution metrics are in public beta and available on Claude for Teams and Claude for Enterprise plans. These metrics only cover users within your claude.ai organization. Usage through the Claude Console API or third-party integrations is not included.

Usage and adoption data is available for all Claude for Teams and Claude for Enterprise accounts. Contribution metrics require additional setup to connect your GitHub organization.

You need the Owner role to configure analytics settings. A GitHub admin must install the GitHub app.

> **Warning:**
>
> Contribution metrics are not available for organizations with Zero Data Retention enabled. The analytics dashboard will show usage metrics only.

Step 1: Install the GitHub app

A GitHub admin installs the Claude GitHub app on your organization's GitHub account at github.com/apps/claude.

Step 2: Enable Claude Code analytics

A Claude Owner navigates to claude.ai/admin-settings/claude-code and enables the Claude Code analytics feature.

Step 3: Enable GitHub analytics

On the same page, enable the "GitHub analytics" toggle.

Step 4: Authenticate with GitHub

Complete the GitHub authentication flow and select which GitHub organizations to include in the analysis.

Data typically appears within 24 hours after enabling, with daily updates. If no data appears, you may see one of these messages:

- **"GitHub app required"**: install the GitHub app to view contribution metrics
- **"Data processing in progress"**: check back in a few days and confirm the GitHub app is installed if data doesn't appear

Contribution metrics support GitHub Cloud and GitHub Enterprise Server.

Review summary metrics

> **Note:**
>
> These metrics are deliberately conservative and represent an underestimate of Claude Code's actual impact. Only lines and PRs where there is high confidence in Claude Code's involvement are counted.

The dashboard displays these summary metrics at the top:

- **PRs with CC**: total count of merged pull requests that contain at least one line of code written with Claude Code
- **Lines of code with CC**: total lines of code across all merged PRs that were written with Claude Code assistance. Only "effective lines" are counted: lines with more than 3 characters after normalization, excluding empty lines and lines with only brackets or trivial punctuation.
- **PRs with Claude Code (%)**: percentage of all merged PRs that contain Claude Code-assisted code
- **Suggestion accept rate**: percentage of times users accept Claude Code's code editing suggestions, including Edit, Write, and NotebookEdit tool usage
- **Lines of code accepted**: total lines of code written by Claude Code that users have accepted in their sessions. This excludes rejected suggestions and does not track subsequent deletions.

Explore the charts

The dashboard includes several charts to visualize trends over time.

Track adoption

The Adoption chart shows daily usage trends:

- **users**: daily active users
- **sessions**: number of active Claude Code sessions per day

Measure PRs per user

This chart displays individual developer activity over time:

- **PRs per user**: total number of PRs merged per day divided by daily active users
- **users**: daily active users

Use this to understand how individual productivity changes as Claude Code adoption increases.

View pull requests breakdown

The Pull requests chart shows a daily breakdown of merged PRs:

- **PRs with CC**: pull requests containing Claude Code-assisted code
- **PRs without CC**: pull requests without Claude Code-assisted code

Toggle to **Lines of code** view to see the same breakdown by lines of code rather than PR count.

Find top contributors

The Leaderboard shows the top 10 users ranked by contribution volume. Toggle between:

- **Pull requests**: shows PRs with Claude Code vs All PRs for each user
- **Lines of code**: shows lines with Claude Code vs All lines for each user

Click **Export all users** to download complete contribution data for all users as a CSV file. The export includes all users, not just the top 10 displayed.

PR attribution

When contribution metrics are enabled, Claude Code analyzes merged pull requests to determine which code was written with Claude Code assistance. This is done by matching Claude Code session activity against the code in each PR.

Tagging criteria

PRs are tagged as “with Claude Code” if they contain at least one line of code written during a Claude Code session. The system uses conservative matching: only code where there is high confidence in Claude Code’s involvement is counted as assisted.

Attribution process

When a pull request is merged:

1. Added lines are extracted from the PR diff
2. Claude Code sessions that edited matching files within a time window are identified
3. PR lines are matched against Claude Code output using multiple strategies
4. Metrics are calculated for AI-assisted lines and total lines

Before comparison, lines are normalized: whitespace is trimmed, multiple spaces are collapsed, quotes are standardized, and text is converted to lowercase.

Merged pull requests containing Claude Code-assisted lines are labeled as `claude-code-assisted` in GitHub.

Time window

Sessions from 21 days before to 2 days after the PR merge date are considered for attribution matching.

Excluded files

Certain files are automatically excluded from analysis because they are auto-generated:

- Lock files: package-lock.json, yarn.lock, Cargo.lock, and similar
- Generated code: Protobuf outputs, build artifacts, minified files
- Build directories: dist/, build/, node_modules/, target/
- Test fixtures: snapshots, cassettes, mock data
- Lines over 1,000 characters, which are likely minified or generated

Attribution notes

Keep these additional details in mind when interpreting attribution data:

- Code substantially rewritten by developers, with more than 20% difference, is not attributed to Claude Code
- Sessions outside the 21-day window are not considered
- The algorithm does not consider the PR source or destination branch when performing attribution

Get the most from analytics

Use contribution metrics to demonstrate ROI, identify adoption patterns, and find team members who can help others get started.

Monitor adoption

Track the Adoption chart and user counts to identify:

- Active users who can share best practices
- Overall adoption trends across your organization
- Dips in usage that may indicate friction or issues

Measure ROI

Contribution metrics help answer "Is this tool worth the investment?" with data from your own codebase:

- Track changes in PRs per user over time as adoption increases
- Compare PRs and lines of code shipped with vs. without Claude Code
- Use alongside DORA metrics, sprint velocity, or other engineering KPIs to understand changes from adopting Claude Code

Identify power users

The Leaderboard helps you find team members with high Claude Code adoption who can:

- Share prompting techniques and workflows with the team
- Provide feedback on what's working well
- Help onboard new users

Access data programmatically

To query this data through GitHub, search for PRs labeled with `claude-code-assisted`.

Access analytics for API customers

API customers using the Claude Console can access analytics at platform.claude.com/claude-code. You need the UsageView permission to access the dashboard, which is granted to Developer, Billing, Admin, Owner, and Primary Owner roles.

> **Note:**
>
> Contribution metrics with GitHub integration are not currently available for API customers. The Console dashboard shows usage and spend metrics only.

The Console dashboard displays:

- **Lines of code accepted**: total lines of code written by Claude Code that users have accepted in their sessions. This excludes rejected suggestions and does not track subsequent deletions.
- **Suggestion accept rate**: percentage of times users accept code editing tool usage, including Edit, Write, and NotebookEdit tools.
- **Activity**: daily active users and sessions shown on a chart.
- **Spend**: daily API costs in dollars alongside user count.

View team insights

The team insights table shows per-user metrics:

- **Members**: all users who have authenticated to Claude Code. API key users display by key identifier, OAuth users display by email address.
- **Spend this month**: per-user total API costs for the current month.
- **Lines this month**: per-user total of accepted code lines for the current month.

> **Note:**
>
> Spend figures in the Console dashboard are estimates for analytics purposes. For actual costs, refer to your billing page.

Related resources

- Monitoring with OpenTelemetry: export real-time metrics and events to your observability stack
- Manage costs effectively: set spend limits and optimize token usage
- Permissions: configure roles and permissions

Monitoring

> *Learn how to enable and configure OpenTelemetry for Claude Code.*

Track Claude Code usage, costs, and tool activity across your organization by exporting telemetry data through OpenTelemetry (OTel). Claude Code exports metrics as time series data via the standard metrics protocol, and events via the logs/events protocol. Configure your metrics and logs backends to match your monitoring requirements.

Quick start

Configure OpenTelemetry using environment variables:

```
## 1. Enable telemetry
export CLAUDE_CODE_ENABLE_TELEMETRY=1

## 2. Choose exporters (both are optional - configure only what you need)
export OTEL_METRICS_EXPORTER=otlp       # Options: otlp, prometheus, console
export OTEL_LOGS_EXPORTER=otlp          # Options: otlp, console

## 3. Configure OTLP endpoint (for OTLP exporter)
export OTEL_EXPORTER_OTLP_PROTOCOL=grpc
export OTEL_EXPORTER_OTLP_ENDPOINT=http://localhost:4317

## 4. Set authentication (if required)
export OTEL_EXPORTER_OTLP_HEADERS="Authorization=Bearer your-token"

## 5. For debugging: reduce export intervals
export OTEL_METRIC_EXPORT_INTERVAL=10000  # 10 seconds (default: 60000ms)
export OTEL_LOGS_EXPORT_INTERVAL=5000     # 5 seconds (default: 5000ms)

## 6. Run Claude Code
claude
```

Note:

> The default export intervals are 60 seconds for metrics and 5 seconds for logs. During setup, you may want to use shorter intervals for debugging purposes. Remember to reset these for production use.

For full configuration options, see the OpenTelemetry specification.

Administrator configuration

Administrators can configure OpenTelemetry settings for all users through the managed settings file. This allows for centralized control of telemetry settings across an organization. See the settings precedence for more information about how settings are applied.

Example managed settings configuration:

```
{
  "env": {
    "CLAUDE_CODE_ENABLE_TELEMETRY": "1",
    "OTEL_METRICS_EXPORTER": "otlp",
    "OTEL_LOGS_EXPORTER": "otlp",
    "OTEL_EXPORTER_OTLP_PROTOCOL": "grpc",
    "OTEL_EXPORTER_OTLP_ENDPOINT": "http://collector.example.com:4317",
    "OTEL_EXPORTER_OTLP_HEADERS": "Authorization=Bearer example-token"
  }
}
```

> **Note:**
>
> Managed settings can be distributed via MDM (Mobile Device Management) or other device management solutions. Environment variables defined in the managed settings file have high precedence and cannot be overridden by users.

Configuration details

Common configuration variables

Environment Variable	Description	Example Values
`CLAUDE_CODE_ENABLE_TELEMETRY`	Enables telemetry collection (required)	`1`
`OTEL_METRICS_EXPORTER`	Metrics exporter types, comma-separated	`console`, `otlp`, `prometheus`
`OTEL_LOGS_EXPORTER`	Logs/events exporter types, comma-separated	`console`, `otlp`
`OTEL_EXPORTER_OTLP_PROTOCOL`	Protocol for OTLP exporter, applies to all signals	`grpc`, `http/json`, `http/protobuf`
`OTEL_EXPORTER_OTLP_ENDPOINT`	OTLP collector endpoint for all signals	`http://localhost:4317`
`OTEL_EXPORTER_OTLP_METRICS_PROTOCOL`	Protocol for metrics, overrides general setting	`grpc`, `http/json`, `http/protobuf`
`OTEL_EXPORTER_OTLP_METRICS_ENDPOINT`	OTLP metrics endpoint, overrides general setting	`http://localhost:4318/v1/metrics`
`OTEL_EXPORTER_OTLP_LOGS_PROTOCOL`	Protocol for logs, overrides general setting	`grpc`, `http/json`, `http/protobuf`
`OTEL_EXPORTER_OTLP_LOGS_ENDPOINT`	OTLP logs endpoint, overrides general setting	`http://localhost:4318/v1/logs`
`OTEL_EXPORTER_OTLP_HEADERS`	Authentication headers for OTLP	`Authorization=Bearer token`
`OTEL_EXPORTER_OTLP_METRICS_CLIENT_KEY`	Client key for mTLS authentication	Path to client key file
`OTEL_EXPORTER_OTLP_METRICS_CLIENT_CERTIFICATE`	Client certificate for mTLS authentication	Path to client cert file

Environment Variable	Description	Example Values
`OTEL_METRIC_EXPORT_INTERVAL`	Export interval in milliseconds (default: 60000)	`5000`, `60000`
`OTEL_LOGS_EXPORT_INTERVAL`	Logs export interval in milliseconds (default: 5000)	`1000`, `10000`
`OTEL_LOG_USER_PROMPTS`	Enable logging of user prompt content (default: disabled)	`1` to enable
`OTEL_LOG_TOOL_DETAILS`	Enable logging of MCP server/tool names and skill names in tool events (default: disabled)	`1` to enable
`OTEL_EXPORTER_OTLP_METRICS_TEMPORALITY_PREFERENCE`	Metrics temporality preference (default: `delta`). Set to `cumulative` if your backend expects cumulative temporality	`delta`, `cumulative`
`CLAUDE_CODE_OTEL_HEADERS_HELPER_DEBOUNCE_MS`	Interval for refreshing dynamic headers (default: 1740000ms / 29 minutes)	`900000`

Metrics cardinality control

The following environment variables control which attributes are included in metrics to manage cardinality:

Environment Variable	Description	Default Value	Example to Disable
`OTEL_METRICS_INCLUDE_SESSION_ID`	Include session.id attribute in metrics	`true`	`false`
`OTEL_METRICS_INCLUDE_VERSION`	Include app.version attribute in metrics	`false`	`true`

Environment Variable	Description	Default Value	Example to Disable
`OTEL_METRICS_INCLUDE_ACCOUNT_UUID`	Include user.account_uuid attribute in metrics	`true`	`false`

These variables help control the cardinality of metrics, which affects storage requirements and query performance in your metrics backend. Lower cardinality generally means better performance and lower storage costs but less granular data for analysis.

Dynamic headers

For enterprise environments that require dynamic authentication, you can configure a script to generate headers dynamically:

Settings configuration

Add to your `.claude/settings.json`:

```
{
  "otelHeadersHelper": "/bin/generate_opentelemetry_headers.sh"
}
```

Script requirements

The script must output valid JSON with string key-value pairs representing HTTP headers:

```
#!/bin/bash
## Example: Multiple headers
echo "{\"Authorization\": \"Bearer $(get-token.sh)\", \"X-API-Key\": \"$(get-api-key.sh)\"}"
```

Refresh behavior

The headers helper script runs at startup and periodically thereafter to support token refresh. By default, the script runs every 29 minutes. Customize the interval with the `CLAUDE_CODE_OTEL_HEADERS_HELPER_DEBOUNCE_MS` environment variable.

Multi-team organization support

Organizations with multiple teams or departments can add custom attributes to distinguish between different groups using the `OTEL_RESOURCE_ATTRIBUTES` environment variable:

```
## Add custom attributes for team identification
export OTEL_RESOURCE_ATTRIBUTES="department=engineering,team.id=platform,cost_cent
er=eng-123"
```

These custom attributes will be included in all metrics and events, allowing you to:

- Filter metrics by team or department
- Track costs per cost center
- Create team-specific dashboards
- Set up alerts for specific teams

Warning:

Important formatting requirements for OTEL_RESOURCE_ATTRIBUTES:

The `OTEL_RESOURCE_ATTRIBUTES` environment variable uses comma-separated key=value pairs with strict formatting requirements:

- **No spaces allowed**: Values cannot contain spaces. For example, `user.organizationName=My Company` is invalid
- **Format**: Must be comma-separated key=value pairs: `key1=value1,key2=value2`
- **Allowed characters**: Only US-ASCII characters excluding control characters, whitespace, double quotes, commas, semicolons, and backslashes
- **Special characters**: Characters outside the allowed range must be percent-encoded

Examples:

```
## ❌ Invalid - contains spaces
export OTEL_RESOURCE_ATTRIBUTES="org.name=John's Organization"

## ✅ Valid - use underscores or camelCase instead
export OTEL_RESOURCE_ATTRIBUTES="org.name=Johns_Organization"
export OTEL_RESOURCE_ATTRIBUTES="org.name=JohnsOrganization"

## ✅ Valid - percent-encode special characters if needed
export OTEL_RESOURCE_ATTRIBUTES="org.name=John%27s%20Organization"
```

> Note: wrapping values in quotes doesn't escape spaces. For example, `org.name="My Company"` results in the literal value `"My Company"` (with quotes included), not `My Company`.

Example configurations

Set these environment variables before running `claude`. Each block shows a complete configuration for a different exporter or deployment scenario:

```
## Console debugging (1-second intervals)
export CLAUDE_CODE_ENABLE_TELEMETRY=1
export OTEL_METRICS_EXPORTER=console
export OTEL_METRIC_EXPORT_INTERVAL=1000

## OTLP/gRPC
export CLAUDE_CODE_ENABLE_TELEMETRY=1
export OTEL_METRICS_EXPORTER=otlp
export OTEL_EXPORTER_OTLP_PROTOCOL=grpc
export OTEL_EXPORTER_OTLP_ENDPOINT=http://localhost:4317

## Prometheus
export CLAUDE_CODE_ENABLE_TELEMETRY=1
export OTEL_METRICS_EXPORTER=prometheus

## Multiple exporters
export CLAUDE_CODE_ENABLE_TELEMETRY=1
export OTEL_METRICS_EXPORTER=console,otlp
export OTEL_EXPORTER_OTLP_PROTOCOL=http/json

## Different endpoints/backends for metrics and logs
export CLAUDE_CODE_ENABLE_TELEMETRY=1
export OTEL_METRICS_EXPORTER=otlp
export OTEL_LOGS_EXPORTER=otlp
export OTEL_EXPORTER_OTLP_METRICS_PROTOCOL=http/protobuf
export OTEL_EXPORTER_OTLP_METRICS_ENDPOINT=http://metrics.example.com:4318
export OTEL_EXPORTER_OTLP_LOGS_PROTOCOL=grpc
export OTEL_EXPORTER_OTLP_LOGS_ENDPOINT=http://logs.example.com:4317

## Metrics only (no events/logs)
export CLAUDE_CODE_ENABLE_TELEMETRY=1
export OTEL_METRICS_EXPORTER=otlp
export OTEL_EXPORTER_OTLP_PROTOCOL=grpc
export OTEL_EXPORTER_OTLP_ENDPOINT=http://localhost:4317

## Events/logs only (no metrics)
export CLAUDE_CODE_ENABLE_TELEMETRY=1
```

```
export OTEL_LOGS_EXPORTER=otlp
export OTEL_EXPORTER_OTLP_PROTOCOL=grpc
export OTEL_EXPORTER_OTLP_ENDPOINT=http://localhost:4317
```

Available metrics and events

Standard attributes

All metrics and events share these standard attributes:

Attribute	Description	Controlled By
`session.id`	Unique session identifier	`OTEL_METRICS_INCLUDE_SESSION_ID` (default: true)
`app.version`	Current Claude Code version	`OTEL_METRICS_INCLUDE_VERSION` (default: false)
`organization.id`	Organization UUID (when authenticated)	Always included when available
`user.account_uuid`	Account UUID (when authenticated)	`OTEL_METRICS_INCLUDE_ACCOUNT_UUID` (default: true)
`user.id`	Anonymous device/installation identifier, generated per Claude Code installation	Always included
`user.email`	User email address (when authenticated via OAuth)	Always included when available
`terminal.type`	Terminal type, such as `iTerm.app`, `vscode`, `cursor`, or `tmux`	Always included when detected

Metrics

Claude Code exports the following metrics:

Metric Name	Description	Unit
`claude_code.session.count`	Count of CLI sessions started	count

Metric Name	Description	Unit
`claude_code.lines_of_code.count`	Count of lines of code modified	count
`claude_code.pull_request.count`	Number of pull requests created	count
`claude_code.commit.count`	Number of git commits created	count
`claude_code.cost.usage`	Cost of the Claude Code session	USD
`claude_code.token.usage`	Number of tokens used	tokens
`claude_code.code_edit_tool.decision`	Count of code editing tool permission decisions	count
`claude_code.active_time.total`	Total active time in seconds	s

Metric details

Each metric includes the standard attributes listed above. Metrics with additional context-specific attributes are noted below.

Session counter

Incremented at the start of each session.

Attributes:

- All standard attributes

Lines of code counter

Incremented when code is added or removed.

Attributes:

- All standard attributes
- `type` : (`"added"` , `"removed"`)

Pull request counter

Incremented when creating pull requests via Claude Code.

Attributes:

- All standard attributes

Commit counter

Incremented when creating git commits via Claude Code.

Attributes:

- All standard attributes

Cost counter

Incremented after each API request.

Attributes:

- All standard attributes
- `model`: Model identifier (for example, "claude-sonnet-4-6")

Token counter

Incremented after each API request.

Attributes:

- All standard attributes
- `type`: (`"input"`, `"output"`, `"cacheRead"`, `"cacheCreation"`)
- `model`: Model identifier (for example, "claude-sonnet-4-6")

Code edit tool decision counter

Incremented when user accepts or rejects Edit, Write, or NotebookEdit tool usage.

Attributes:

- All standard attributes
- `tool_name`: Tool name (`"Edit"`, `"Write"`, `"NotebookEdit"`)
- `decision`: User decision (`"accept"`, `"reject"`)
- `source`: Decision source - `"config"`, `"hook"`, `"user_permanent"`, `"user_temporary"`, `"user_abort"`, or `"user_reject"`
- `language`: Programming language of the edited file, such as `"TypeScript"`, `"Python"`, `"JavaScript"`, or `"Markdown"`. Returns `"unknown"` for unrecognized file extensions.

Active time counter

Tracks actual time spent actively using Claude Code, excluding idle time. This metric is incremented during user interactions (typing, reading responses) and during CLI processing (tool execution, AI response generation).

Attributes:

- All standard attributes
- `type`: `"user"` for keyboard interactions, `"cli"` for tool execution and AI responses

Events

Claude Code exports the following events via OpenTelemetry logs/events (when `OTEL_LOGS_EXPORTER` is configured):

Event correlation attributes

When a user submits a prompt, Claude Code may make multiple API calls and run several tools. The `prompt.id` attribute lets you tie all of those events back to the single prompt that triggered them.

Attribute	Description
`prompt.id`	UUID v4 identifier linking all events produced while processing a single user prompt

To trace all activity triggered by a single prompt, filter your events by a specific `prompt.id` value. This returns the user_prompt event, any api_request events, and any tool_result events that occurred while processing that prompt.

> **Note:**
>
> `prompt.id` is intentionally excluded from metrics because each prompt generates a unique ID, which would create an ever-growing number of time series. Use it for event-level analysis and audit trails only.

User prompt event

Logged when a user submits a prompt.

Event Name: `claude_code.user_prompt`

Attributes:

- All standard attributes

- `event.name`: `"user_prompt"`
- `event.timestamp`: ISO 8601 timestamp
- `event.sequence`: monotonically increasing counter for ordering events within a session
- `prompt_length`: Length of the prompt
- `prompt`: Prompt content (redacted by default, enable with `OTEL_LOG_USER_PROMPTS=1`)

Tool result event

Logged when a tool completes execution.

Event Name: `claude_code.tool_result`

Attributes:

- All standard attributes
- `event.name`: `"tool_result"`
- `event.timestamp`: ISO 8601 timestamp
- `event.sequence`: monotonically increasing counter for ordering events within a session
- `tool_name`: Name of the tool
- `success`: `"true"` or `"false"`
- `duration_ms`: Execution time in milliseconds
- `error`: Error message (if failed)
- `decision_type`: Either `"accept"` or `"reject"`
- `decision_source`: Decision source - `"config"`, `"hook"`, `"user_permanent"`, `"user_temporary"`, `"user_abort"`, or `"user_reject"`
- `tool_result_size_bytes`: Size of the tool result in bytes
- `mcp_server_scope`: MCP server scope identifier (for MCP tools)
- `tool_parameters`: JSON string containing tool-specific parameters (when available)
  - For Bash tool: includes `bash_command`, `full_command`, `timeout`, `description`, `dangerouslyDisableSandbox`, and `git_commit_id` (the commit SHA, when a `git commit` command succeeds)
  - For MCP tools (when `OTEL_LOG_TOOL_DETAILS=1`): includes `mcp_server_name`, `mcp_tool_name`
  - For Skill tool (when `OTEL_LOG_TOOL_DETAILS=1`): includes `skill_name`

API request event

Logged for each API request to Claude.

Event Name: `claude_code.api_request`

Attributes:

- All standard attributes
- `event.name`: `"api_request"`
- `event.timestamp`: ISO 8601 timestamp
- `event.sequence`: monotonically increasing counter for ordering events within a session
- `model`: Model used (for example, "claude-sonnet-4-6")
- `cost_usd`: Estimated cost in USD
- `duration_ms`: Request duration in milliseconds
- `input_tokens`: Number of input tokens
- `output_tokens`: Number of output tokens
- `cache_read_tokens`: Number of tokens read from cache
- `cache_creation_tokens`: Number of tokens used for cache creation
- `speed`: `"fast"` or `"normal"`, indicating whether fast mode was active

API error event

Logged when an API request to Claude fails.

Event Name: `claude_code.api_error`

Attributes:

- All standard attributes
- `event.name`: `"api_error"`
- `event.timestamp`: ISO 8601 timestamp
- `event.sequence`: monotonically increasing counter for ordering events within a session
- `model`: Model used (for example, "claude-sonnet-4-6")
- `error`: Error message
- `status_code`: HTTP status code as a string, or `"undefined"` for non-HTTP errors
- `duration_ms`: Request duration in milliseconds
- `attempt`: Attempt number (for retried requests)

- `speed`: `"fast"` or `"normal"`, indicating whether fast mode was active

Tool decision event

Logged when a tool permission decision is made (accept/reject).

Event Name: `claude_code.tool_decision`

Attributes:

- All standard attributes
- `event.name`: `"tool_decision"`
- `event.timestamp`: ISO 8601 timestamp
- `event.sequence`: monotonically increasing counter for ordering events within a session
- `tool_name`: Name of the tool (for example, “Read”, “Edit”, “Write”, “NotebookEdit”)
- `decision`: Either `"accept"` or `"reject"`
- `source`: Decision source - `"config"`, `"hook"`, `"user_permanent"`, `"user_temporary"`, `"user_abort"`, or `"user_reject"`

Interpret metrics and events data

The exported metrics and events support a range of analyses:

Usage monitoring

Metric	Analysis Opportunity
`claude_code.token.usage`	Break down by `type` (input/output), user, team, or model
`claude_code.session.count`	Track adoption and engagement over time
`claude_code.lines_of_code.count`	Measure productivity by tracking code additions/removals
`claude_code.commit.count` & `claude_code.pull_request.count`	Understand impact on development workflows

Cost monitoring

The `claude_code.cost.usage` metric helps with:

- Tracking usage trends across teams or individuals
- Identifying high-usage sessions for optimization

> **Note:**
>
> Cost metrics are approximations. For official billing data, refer to your API provider (Claude Console, AWS Bedrock, or Google Cloud Vertex).

Alerting and segmentation

Common alerts to consider:

- Cost spikes
- Unusual token consumption
- High session volume from specific users

All metrics can be segmented by `user.account_uuid`, `organization.id`, `session.id`, `model`, and `app.version`.

Event analysis

The event data provides detailed insights into Claude Code interactions:

Tool Usage Patterns: analyze tool result events to identify:

- Most frequently used tools
- Tool success rates
- Average tool execution times
- Error patterns by tool type

Performance Monitoring: track API request durations and tool execution times to identify performance bottlenecks.

Backend considerations

Your choice of metrics and logs backends determines the types of analyses you can perform:

For metrics

- **Time series databases (for example, Prometheus)**: Rate calculations, aggregated metrics
- **Columnar stores (for example, ClickHouse)**: Complex queries, unique user analysis
- **Full-featured observability platforms (for example, Honeycomb, Datadog)**: Advanced querying, visualization, alerting

For events/logs

- **Log aggregation systems (for example, Elasticsearch, Loki)**: Full-text search, log analysis
- **Columnar stores (for example, ClickHouse)**: Structured event analysis
- **Full-featured observability platforms (for example, Honeycomb, Datadog)**: Correlation between metrics and events

For organizations requiring Daily/Weekly/Monthly Active User (DAU/WAU/MAU) metrics, consider backends that support efficient unique value queries.

Service information

All metrics and events are exported with the following resource attributes:

- `service.name`: `claude-code`
- `service.version`: Current Claude Code version
- `os.type`: Operating system type (for example, `linux`, `darwin`, `windows`)
- `os.version`: Operating system version string
- `host.arch`: Host architecture (for example, `amd64`, `arm64`)
- `wsl.version`: WSL version number (only present when running on Windows Subsystem for Linux)
- Meter Name: `com.anthropic.claude_code`

ROI measurement resources

For a comprehensive guide on measuring return on investment for Claude Code, including telemetry setup, cost analysis, productivity metrics, and automated reporting, see the Claude Code ROI Measurement Guide. This repository provides ready-to-use Docker Compose configurations, Prometheus and OpenTelemetry setups, and templates for generating productivity reports integrated with tools like Linear.

Security and privacy

- Telemetry is opt-in and requires explicit configuration
- Raw file contents and code snippets are not included in metrics or events. Tool execution events include bash commands and file paths in the `tool_parameters` field, which may contain sensitive values. If your commands may include secrets, configure your telemetry backend to filter or redact `tool_parameters`

- When authenticated via OAuth, `user.email` is included in telemetry attributes. If this is a concern for your organization, work with your telemetry backend to filter or redact this field
- User prompt content is not collected by default. Only prompt length is recorded. To include prompt content, set `OTEL_LOG_USER_PROMPTS=1`
- MCP server/tool names and skill names are not logged by default because they can reveal user-specific configurations. To include them, set `OTEL_LOG_TOOL_DETAILS=1`

Monitor Claude Code on Amazon Bedrock

For detailed Claude Code usage monitoring guidance for Amazon Bedrock, see Claude Code Monitoring Implementation (Bedrock).

Enterprise deployment overview

> *Learn how Claude Code can integrate with various third-party services and infrastructure to meet enterprise deployment requirements.*

Organizations can deploy Claude Code through Anthropic directly or through a cloud provider. This page helps you choose the right configuration.

Compare deployment options

For most organizations, Claude for Teams or Claude for Enterprise provides the best experience. Team members get access to both Claude Code and Claude on the web with a single subscription, centralized billing, and no infrastructure setup required.

Claude for Teams is self-service and includes collaboration features, admin tools, and billing management. Best for smaller teams that need to get started quickly.

Claude for Enterprise adds SSO and domain capture, role-based permissions, compliance API access, and managed policy settings for deploying organization-wide Claude Code configurations. Best for larger organizations with security and compliance requirements.

Learn more about Team plans and Enterprise plans.

If your organization has specific infrastructure requirements, compare the options below:

Feature Claude for Teams/Enterprise Anthropic Console Amazon Bedrock Google Vertex AI Microsoft Foundry

Best for Most organizations (recommended) Individual developers AWS-native deployments GCP-native deployments Azure-native deployments

Billing Teams: $150/seat (Premium) with PAYG available Enterprise: Contact Sales PAYG PAYG through AWS PAYG through GCP PAYG through Azure

Regions Supported countries Supported countries Multiple AWS regions Multiple GCP regions Multiple Azure regions

Prompt caching Enabled by default Enabled by default Enabled by default Enabled by default Enabled by default

Authentication Claude.ai SSO or email API key API key or AWS credentials GCP credentials API key or Microsoft Entra ID

Cost tracking Usage dashboard Usage dashboard AWS Cost Explorer GCP Billing Azure Cost Management

Includes Claude on web Yes No No No No

Enterprise features Team management, SSO, usage monitoring None IAM policies, CloudTrail IAM roles, Cloud Audit Logs RBAC policies, Azure Monitor

Select a deployment option to view setup instructions:

- Claude for Teams or Enterprise
- Anthropic Console
- Amazon Bedrock
- Google Vertex AI
- Microsoft Foundry

Configure proxies and gateways

Most organizations can use a cloud provider directly without additional configuration. However, you may need to configure a corporate proxy or LLM gateway if your organization has specific network or management requirements. These are different configurations that can be used together:

- **Corporate proxy**: Routes traffic through an HTTP/HTTPS proxy. Use this if your organization requires all outbound traffic to pass through a proxy server for security monitoring, compliance, or network policy enforcement. Configure with the `HTTPS_PROXY` or `HTTP_PROXY` environment variables. Learn more in Enterprise network configuration.
- **LLM Gateway**: A service that sits between Claude Code and the cloud provider to handle authentication and routing. Use this if you need centralized usage tracking across teams, custom rate limiting or budgets, or centralized authentication management. Configure with the `ANTHROPIC_BASE_URL`, `ANTHROPIC_BEDROCK_BASE_URL`, or `ANTHROPIC_VERTEX_BASE_URL` environment variables. Learn more in LLM gateway configuration.

The following examples show the environment variables to set in your shell or shell profile (`.bashrc`, `.zshrc`). See Settings for other configuration methods.

Amazon Bedrock

Corporate proxy

Route Bedrock traffic through your corporate proxy by setting the following environment variables:

```
## Enable Bedrock
export CLAUDE_CODE_USE_BEDROCK=1
export AWS_REGION=us-east-1

## Configure corporate proxy
export HTTPS_PROXY='https://proxy.example.com:8080'
```

LLM Gateway

Route Bedrock traffic through your LLM gateway by setting the following environment variables:

```
## Enable Bedrock
export CLAUDE_CODE_USE_BEDROCK=1

## Configure LLM gateway
export ANTHROPIC_BEDROCK_BASE_URL='https://your-llm-gateway.com/bedrock'
export CLAUDE_CODE_SKIP_BEDROCK_AUTH=1  # If gateway handles AWS auth
```

Microsoft Foundry

Corporate proxy

Route Foundry traffic through your corporate proxy by setting the following environment variables:

```
## Enable Microsoft Foundry
export CLAUDE_CODE_USE_FOUNDRY=1
export ANTHROPIC_FOUNDRY_RESOURCE=your-resource
export ANTHROPIC_FOUNDRY_API_KEY=your-api-key  # Or omit for Entra ID auth

## Configure corporate proxy
export HTTPS_PROXY='https://proxy.example.com:8080'
```

LLM Gateway

Route Foundry traffic through your LLM gateway by setting the following environment variables:

```
## Enable Microsoft Foundry
export CLAUDE_CODE_USE_FOUNDRY=1

## Configure LLM gateway
export ANTHROPIC_FOUNDRY_BASE_URL='https://your-llm-gateway.com'
export CLAUDE_CODE_SKIP_FOUNDRY_AUTH=1  # If gateway handles Azure auth
```

Google Vertex AI

Corporate proxy

Route Vertex AI traffic through your corporate proxy by setting the following environment variables:

```
## Enable Vertex
export CLAUDE_CODE_USE_VERTEX=1
export CLOUD_ML_REGION=us-east5
export ANTHROPIC_VERTEX_PROJECT_ID=your-project-id

## Configure corporate proxy
export HTTPS_PROXY='https://proxy.example.com:8080'
```

LLM Gateway

Route Vertex AI traffic through your LLM gateway by setting the following environment variables:

```
## Enable Vertex
export CLAUDE_CODE_USE_VERTEX=1

## Configure LLM gateway
export ANTHROPIC_VERTEX_BASE_URL='https://your-llm-gateway.com/vertex'
export CLAUDE_CODE_SKIP_VERTEX_AUTH=1  # If gateway handles GCP auth
```

> **Tip:**
>
> Use `/status` in Claude Code to verify your proxy and gateway configuration is applied correctly.

Best practices for organizations

Invest in documentation and memory

We strongly recommend investing in documentation so that Claude Code understands your codebase. Organizations can deploy CLAUDE.md files at multiple levels:

- **Organization-wide**: Deploy to system directories like `/Library/Application Support/ClaudeCode/CLAUDE.md` (macOS) for company-wide standards
- **Repository-level**: Create `CLAUDE.md` files in repository roots containing project architecture, build commands, and contribution guidelines. Check these into source control so all users benefit

Learn more in Memory and CLAUDE.md files.

Simplify deployment

If you have a custom development environment, we find that creating a “one click” way to install Claude Code is key to growing adoption across an organization.

Start with guided usage

Encourage new users to try Claude Code for codebase Q&A, or on smaller bug fixes or feature requests. Ask Claude Code to make a plan. Check Claude’s suggestions and give feedback if it’s off-track. Over time, as users understand this new paradigm better, then they’ll be more effective at letting Claude Code run more agentically.

Pin model versions for cloud providers

If you deploy through Bedrock, Vertex AI, or Foundry, pin specific model versions using `ANTHROPIC_DEFAULT_OPUS_MODEL`, `ANTHROPIC_DEFAULT_SONNET_MODEL`, and `ANTHROPIC_DEFAULT_HAIKU_MODEL`. Without pinning, Claude Code aliases resolve to the latest version, which can break users when Anthropic releases a new model that isn't yet enabled in your account. See Model configuration for details.

Configure security policies

Security teams can configure managed permissions for what Claude Code is and is not allowed to do, which cannot be overwritten by local configuration. Learn more.

Leverage MCP for integrations

MCP is a great way to give Claude Code more information, such as connecting to ticket management systems or error logs. We recommend that one central team configures MCP servers and checks a `.mcp.json` configuration into the codebase so that all users benefit. Learn more.

At Anthropic, we trust Claude Code to power development across every Anthropic codebase. We hope you enjoy using Claude Code as much as we do.

Next steps

Once you've chosen a deployment option and configured access for your team:

1. **Roll out to your team**: Share installation instructions and have team members install Claude Code and authenticate with their credentials.
2. **Set up shared configuration**: Create a CLAUDE.md file in your repositories to help Claude Code understand your codebase and coding standards.
3. **Configure permissions**: Review security settings to define what Claude Code can and cannot do in your environment.

Configure server-managed settings (public beta)

> *Centrally configure Claude Code for your organization through server-delivered settings, without requiring device management infrastructure.*

Server-managed settings allow administrators to centrally configure Claude Code through a web-based interface on Claude.ai. Claude Code clients automatically receive these settings when users authenticate with their organization credentials.

This approach is designed for organizations that do not have device management infrastructure in place, or need to manage settings for users on unmanaged devices.

> **Note:**
>
> Server-managed settings are in public beta and available for Claude for Teams and Claude for Enterprise customers. Features may evolve before general availability.

Requirements

To use server-managed settings, you need:

- Claude for Teams or Claude for Enterprise plan
- Claude Code version 2.1.38 or later for Claude for Teams, or version 2.1.30 or later for Claude for Enterprise
- Network access to `api.anthropic.com`

Choose between server-managed and endpoint-managed settings

Claude Code supports two approaches for centralized configuration. Server-managed settings deliver configuration from Anthropic's servers. Endpoint-managed settings are deployed directly to devices through native OS policies (macOS managed preferences, Windows registry) or managed settings files.

Approach	Best for	Security model
Server-managed settings	Organizations without MDM, or users on unmanaged devices	Settings delivered from Anthropic's servers at authentication time
Endpoint-managed settings	Organizations with MDM or endpoint management	Settings deployed to devices via MDM configuration profiles, registry policies, or managed settings files

If your devices are enrolled in an MDM or endpoint management solution, endpoint-managed settings provide stronger security guarantees because the settings file can be protected from user modification at the OS level.

Configure server-managed settings

Step 1: Open the admin console

In Claude.ai, navigate to **Admin Settings > Claude Code > Managed settings**.

Step 2: Define your settings

Add your configuration as JSON. All settings available in `settings.json` are supported, including managed-only settings like `disableBypassPermissionsMode`.

This example enforces a permission deny list and prevents users from bypassing permissions:

```
{
  "permissions": {
    "deny": [
      "Bash(curl *)",
      "Read(./.env)",
      "Read(./.env.*)",
      "Read(./secrets/**)"
    ],
    "disableBypassPermissionsMode": "disable"
  }
}
```

Step 3: Save and deploy

Save your changes. Claude Code clients receive the updated settings on their next startup or hourly polling cycle.

Verify settings delivery

To confirm that settings are being applied, ask a user to restart Claude Code. If the configuration includes settings that trigger the security approval dialog, the user sees a prompt describing the managed settings on startup. You can also verify that managed permission rules are active by having a user run `/permissions` to view their effective permission rules.

Access control

The following roles can manage server-managed settings:

- **Primary Owner**
- **Owner**

Restrict access to trusted personnel, as settings changes apply to all users in the organization.

Current limitations

Server-managed settings have the following limitations during the beta period:

- Settings apply uniformly to all users in the organization. Per-group configurations are not yet supported.
- MCP server configurations cannot be distributed through server-managed settings.

Settings delivery

Settings precedence

Server-managed settings and endpoint-managed settings both occupy the highest tier in the Claude Code settings hierarchy. No other settings level can override them, including command line arguments. When both are present, server-managed settings take precedence and endpoint-managed settings are not used.

Fetch and caching behavior

Claude Code fetches settings from Anthropic's servers at startup and polls for updates hourly during active sessions.

First launch without cached settings:

- Claude Code fetches settings asynchronously

- If the fetch fails, Claude Code continues without managed settings
- There is a brief window before settings load where restrictions are not yet enforced

Subsequent launches with cached settings:

- Cached settings apply immediately at startup
- Claude Code fetches fresh settings in the background
- Cached settings persist through network failures

Claude Code applies settings updates automatically without a restart, except for advanced settings like OpenTelemetry configuration, which require a full restart to take effect.

Security approval dialogs

Certain settings that could pose security risks require explicit user approval before being applied:

- **Shell command settings**: settings that execute shell commands
- **Custom environment variables**: variables not in the known safe allowlist
- **Hook configurations**: any hook definition

When these settings are present, users see a security dialog explaining what is being configured. Users must approve to proceed. If a user rejects the settings, Claude Code exits.

> **Note:**
>
> In non-interactive mode with the `-p` flag, Claude Code skips security dialogs and applies settings without user approval.

Platform availability

Server-managed settings require a direct connection to `api.anthropic.com` and are not available when using third-party model providers:

- Amazon Bedrock
- Google Vertex AI
- Microsoft Foundry
- Custom API endpoints via `ANTHROPIC_BASE_URL` or LLM gateways

Audit logging

Audit log events for settings changes are available through the compliance API or audit log export. Contact your Anthropic account team for access.

Audit events include the type of action performed, the account and device that performed the action, and references to the previous and new values.

Security considerations

Server-managed settings provide centralized policy enforcement, but they operate as a client-side control. On unmanaged devices, users with admin or sudo access can modify the Claude Code binary, filesystem, or network configuration.

Scenario	Behavior
User edits the cached settings file	Tampered file applies at startup, but correct settings restore on the next server fetch
User deletes the cached settings file	First-launch behavior occurs: settings fetch asynchronously with a brief unenforced window
API is unavailable	Cached settings apply if available, otherwise managed settings are not enforced until the next successful fetch
User authenticates with a different organization	Settings are not delivered for accounts outside the managed organization
User sets a non-default `ANTHROPIC_BASE_URL`	Server-managed settings are bypassed when using third-party API providers

To detect runtime configuration changes, use `ConfigChange` hooks to log modifications or block unauthorized changes before they take effect.

For stronger enforcement guarantees, use endpoint-managed settings on devices enrolled in an MDM solution.

See also

Related pages for managing Claude Code configuration:

- Settings: complete configuration reference including all available settings
- Endpoint-managed settings: managed settings deployed to devices by IT
- Authentication: set up user access to Claude Code
- Security: security safeguards and best practices

Part 11: Cloud Providers

Claude Code on Amazon Bedrock

> *Learn about configuring Claude Code through Amazon Bedrock, including setup, IAM configuration, and troubleshooting.*

Prerequisites

Before configuring Claude Code with Bedrock, ensure you have:

- An AWS account with Bedrock access enabled
- Access to desired Claude models (for example, Claude Sonnet 4.6) in Bedrock
- AWS CLI installed and configured (optional - only needed if you don't have another mechanism for getting credentials)
- Appropriate IAM permissions

> **Note:**
>
> If you are deploying Claude Code to multiple users, pin your model versions to prevent breakage when Anthropic releases new models.

Setup

1. Submit use case details

First-time users of Anthropic models are required to submit use case details before invoking a model. This is done once per account.

1. Ensure you have the right IAM permissions (see more on that below)
2. Navigate to the Amazon Bedrock console
3. Select **Chat/Text playground**
4. Choose any Anthropic model and you will be prompted to fill out the use case form

2. Configure AWS credentials

Claude Code uses the default AWS SDK credential chain. Set up your credentials using one of these methods:

Option A: AWS CLI configuration

```
aws configure
```

Option B: Environment variables (access key)

```
export AWS_ACCESS_KEY_ID=your-access-key-id
export AWS_SECRET_ACCESS_KEY=your-secret-access-key
export AWS_SESSION_TOKEN=your-session-token
```

Option C: Environment variables (SSO profile)

```
aws sso login --profile=<your-profile-name>

export AWS_PROFILE=your-profile-name
```

Option D: AWS Management Console credentials

```
aws login
```

Learn more about `aws login`.

Option E: Bedrock API keys

```
export AWS_BEARER_TOKEN_BEDROCK=your-bedrock-api-key
```

Bedrock API keys provide a simpler authentication method without needing full AWS credentials. Learn more about Bedrock API keys.

Advanced credential configuration

Claude Code supports automatic credential refresh for AWS SSO and corporate identity providers. Add these settings to your Claude Code settings file (see Settings for file locations).

When Claude Code detects that your AWS credentials are expired (either locally based on their timestamp or when Bedrock returns a credential error), it will automatically run your configured `awsAuthRefresh` and/or `awsCredentialExport` commands to obtain new credentials before retrying the request.

Example configuration

```
{
  "awsAuthRefresh": "aws sso login --profile myprofile",
  "env": {
    "AWS_PROFILE": "myprofile"
  }
}
```

Configuration settings explained

`awsAuthRefresh` : Use this for commands that modify the `.aws` directory, such as updating credentials, SSO cache, or config files. The command's output is displayed to the user, but interactive input isn't supported. This works well for browser-based SSO flows where the CLI displays a URL or code and you complete authentication in the browser.

`awsCredentialExport` : Only use this if you can't modify `.aws` and must directly return credentials. Output is captured silently and not shown to the user. The command must output JSON in this format:

```
{
  "Credentials": {
    "AccessKeyId": "value",
    "SecretAccessKey": "value",
    "SessionToken": "value"
  }
}
```

3. Configure Claude Code

Set the following environment variables to enable Bedrock:

```
## Enable Bedrock integration
export CLAUDE_CODE_USE_BEDROCK=1
export AWS_REGION=us-east-1  # or your preferred region

## Optional: Override the region for the small/fast model (Haiku)
export ANTHROPIC_SMALL_FAST_MODEL_AWS_REGION=us-west-2
```

When enabling Bedrock for Claude Code, keep the following in mind:

- `AWS_REGION` is a required environment variable. Claude Code does not read from the `.aws` config file for this setting.
- When using Bedrock, the `/login` and `/logout` commands are disabled since authentication is handled through AWS credentials.
- You can use settings files for environment variables like `AWS_PROFILE` that you don't want to leak to other processes. See Settings for more information.

4. Pin model versions

> **Warning:**
>
> Pin specific model versions for every deployment. If you use model aliases (`sonnet`, `opus`, `haiku`) without pinning, Claude Code may attempt to use a newer model version that isn't available in your Bedrock account, breaking existing users when Anthropic releases updates.

Set these environment variables to specific Bedrock model IDs:

```
export ANTHROPIC_DEFAULT_OPUS_MODEL='us.anthropic.claude-opus-4-6-v1'
export ANTHROPIC_DEFAULT_SONNET_MODEL='us.anthropic.claude-sonnet-4-6'
export ANTHROPIC_DEFAULT_HAIKU_MODEL='us.anthropic.claude-haiku-4-5-20251001-v1:0'
```

These variables use cross-region inference profile IDs (with the `us.` prefix). If you use a different region prefix or application inference profiles, adjust accordingly. For current and legacy model IDs, see Models overview. See Model configuration for the full list of environment variables.

Claude Code uses these default models when no pinning variables are set:

Model type	Default value
Primary model	`global.anthropic.claude-sonnet-4-6`
Small/fast model	`us.anthropic.claude-haiku-4-5-20251001-v1:0`

To customize models further, use one of these methods:

```
## Using inference profile ID
export ANTHROPIC_MODEL='global.anthropic.claude-sonnet-4-6'
export ANTHROPIC_SMALL_FAST_MODEL='us.anthropic.claude-haiku-4-5-20251001-v1:0'

## Using application inference profile ARN
export ANTHROPIC_MODEL='arn:aws:bedrock:us-east-2:your-account-id:application-inference-profile/your-model-id'

## Optional: Disable prompt caching if needed
export DISABLE_PROMPT_CACHING=1
```

> **Note:**
>
> Prompt caching may not be available in all regions.

Map each model version to an inference profile

The `ANTHROPIC_DEFAULT_*_MODEL` environment variables configure one inference profile per model family. If your organization needs to expose several versions of the same family in the `/model` picker, each routed to its own application inference profile ARN, use the `modelOverrides` setting in your settings file instead.

This example maps three Opus versions to distinct ARNs so users can switch between them without bypassing your organization's inference profiles:

```
{
  "modelOverrides": {
    "claude-opus-4-6": "arn:aws:bedrock:us-east-2:123456789012:application-inference-profile/opus-46-prod",
    "claude-opus-4-5-20251101": "arn:aws:bedrock:us-east-2:123456789012:application-inference-profile/opus-45-prod",
    "claude-opus-4-1-20250805": "arn:aws:bedrock:us-east-2:123456789012:application-inference-profile/opus-41-prod"
  }
}
```

When a user selects one of these versions in `/model`, Claude Code calls Bedrock with the mapped ARN. Versions without an override fall back to the built-in Bedrock model ID or any matching inference profile discovered at startup. See Override model IDs per version for details on how overrides interact with `availableModels` and other model settings.

IAM configuration

Create an IAM policy with the required permissions for Claude Code:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "AllowModelAndInferenceProfileAccess",
      "Effect": "Allow",
      "Action": [
        "bedrock:InvokeModel",
        "bedrock:InvokeModelWithResponseStream",
        "bedrock:ListInferenceProfiles"
      ],
      "Resource": [
        "arn:aws:bedrock:*:*:inference-profile/*",
        "arn:aws:bedrock:*:*:application-inference-profile/*",
        "arn:aws:bedrock:*:*:foundation-model/*"
      ]
    },
    {
      "Sid": "AllowMarketplaceSubscription",
      "Effect": "Allow",
      "Action": [
        "aws-marketplace:ViewSubscriptions",
        "aws-marketplace:Subscribe"
      ],
      "Resource": "*",
      "Condition": {
        "StringEquals": {
          "aws:CalledViaLast": "bedrock.amazonaws.com"
        }
      }
    }
  ]
}
```

For more restrictive permissions, you can limit the Resource to specific inference profile ARNs.

For details, see Bedrock IAM documentation.

> **Note:**
>
> Create a dedicated AWS account for Claude Code to simplify cost tracking and access control.

AWS Guardrails

Amazon Bedrock Guardrails let you implement content filtering for Claude Code. Create a Guardrail in the Amazon Bedrock console, publish a version, then add the Guardrail headers to your settings file. Enable Cross-Region inference on your Guardrail if you're using cross-region inference profiles.

Example configuration:

```
{
  "env": {
    "ANTHROPIC_CUSTOM_HEADERS": "X-Amzn-Bedrock-GuardrailIdentifier: your-guardrail-id\nX-Amzn-Bedrock-GuardrailVersion: 1"
  }
}
```

Troubleshooting

If you encounter region issues:

- Check model availability: `aws bedrock list-inference-profiles --region your-region`
- Switch to a supported region: `export AWS_REGION=us-east-1`
- Consider using inference profiles for cross-region access

If you receive an error "on-demand throughput isn't supported":

- Specify the model as an inference profile ID

Claude Code uses the Bedrock Invoke API and does not support the Converse API.

Additional resources

- Bedrock documentation
- Bedrock pricing
- Bedrock inference profiles

- Claude Code on Amazon Bedrock: Quick Setup Guide- Claude Code Monitoring Implementation (Bedrock)

Claude Code on Google Vertex AI

> *Learn about configuring Claude Code through Google Vertex AI, including setup, IAM configuration, and troubleshooting.*

Prerequisites

Before configuring Claude Code with Vertex AI, ensure you have:

- A Google Cloud Platform (GCP) account with billing enabled
- A GCP project with Vertex AI API enabled
- Access to desired Claude models (for example, Claude Sonnet 4.6)
- Google Cloud SDK (`gcloud`) installed and configured
- Quota allocated in desired GCP region

> **Note:**
>
> If you are deploying Claude Code to multiple users, pin your model versions to prevent breakage when Anthropic releases new models.

Region Configuration

Claude Code can be used with both Vertex AI global and regional endpoints.

> **Note:**
>
> Vertex AI may not support the Claude Code default models in all regions or on global endpoints. You may need to switch to a supported region, use a regional endpoint, or specify a supported model.

Setup

1. Enable Vertex AI API

Enable the Vertex AI API in your GCP project:

```
## Set your project ID
gcloud config set project YOUR-PROJECT-ID

## Enable Vertex AI API
gcloud services enable aiplatform.googleapis.com
```

2. Request model access

Request access to Claude models in Vertex AI:

1. Navigate to the Vertex AI Model Garden
2. Search for “Claude” models
3. Request access to desired Claude models (for example, Claude Sonnet 4.6)
4. Wait for approval (may take 24-48 hours)

3. Configure GCP credentials

Claude Code uses standard Google Cloud authentication.

For more information, see Google Cloud authentication documentation.

> **Note:**
>
> When authenticating, Claude Code will automatically use the project ID from the `ANTHROPIC_VERTEX_PROJECT_ID` environment variable. To override this, set one of these environment variables: `GCLOUD_PROJECT`, `GOOGLE_CLOUD_PROJECT`, or `GOOGLE_APPLICATION_CREDENTIALS`.

4. Configure Claude Code

Set the following environment variables:

```
## Enable Vertex AI integration
export CLAUDE_CODE_USE_VERTEX=1
export CLOUD_ML_REGION=global
export ANTHROPIC_VERTEX_PROJECT_ID=YOUR-PROJECT-ID

## Optional: Disable prompt caching if needed
export DISABLE_PROMPT_CACHING=1

## When CLOUD_ML_REGION=global, override region for unsupported models
export VERTEX_REGION_CLAUDE_3_5_HAIKU=us-east5

## Optional: Override regions for other specific models
export VERTEX_REGION_CLAUDE_3_5_SONNET=us-east5
export VERTEX_REGION_CLAUDE_3_7_SONNET=us-east5
export VERTEX_REGION_CLAUDE_4_0_OPUS=europe-west1
export VERTEX_REGION_CLAUDE_4_0_SONNET=us-east5
export VERTEX_REGION_CLAUDE_4_1_OPUS=europe-west1
```

Prompt caching is automatically supported when you specify the `cache_control` ephemeral flag. To disable it, set `DISABLE_PROMPT_CACHING=1`. For heightened rate limits, contact Google Cloud support. When using Vertex AI, the `/login` and `/logout` commands are disabled since authentication is handled through Google Cloud credentials.

5. Pin model versions

> **Warning:**
>
> Pin specific model versions for every deployment. If you use model aliases (`sonnet`, `opus`, `haiku`) without pinning, Claude Code may attempt to use a newer model version that isn't enabled in your Vertex AI project, breaking existing users when Anthropic releases updates.

Set these environment variables to specific Vertex AI model IDs:

```
export ANTHROPIC_DEFAULT_OPUS_MODEL='claude-opus-4-6'
export ANTHROPIC_DEFAULT_SONNET_MODEL='claude-sonnet-4-6'
export ANTHROPIC_DEFAULT_HAIKU_MODEL='claude-haiku-4-5@20251001'
```

For current and legacy model IDs, see Models overview. See Model configuration for the full list of environment variables.

Claude Code uses these default models when no pinning variables are set:

Model type	Default value
Primary model	`claude-sonnet-4-6`
Small/fast model	`claude-haiku-4-5@20251001`

To customize models further:

```
export ANTHROPIC_MODEL='claude-opus-4-6'
export ANTHROPIC_SMALL_FAST_MODEL='claude-haiku-4-5@20251001'
```

IAM configuration

Assign the required IAM permissions:

The `roles/aiplatform.user` role includes the required permissions:

- `aiplatform.endpoints.predict` - Required for model invocation and token counting

For more restrictive permissions, create a custom role with only the permissions above.

For details, see Vertex IAM documentation.

> **Note:**
>
> Create a dedicated GCP project for Claude Code to simplify cost tracking and access control.

1M token context window

Claude Opus 4.6, Sonnet 4.6, Sonnet 4.5, and Sonnet 4 support the 1M token context window on Vertex AI. Claude Code automatically enables the extended context window when you select a 1M model variant.

To enable the 1M context window for your pinned model, append `[1m]` to the model ID. See Pin models for third-party deployments for details.

Troubleshooting

If you encounter quota issues:

- Check current quotas or request quota increase through Cloud Console

If you encounter "model not found" 404 errors:

- Confirm model is Enabled in Model Garden
- Verify you have access to the specified region
- If using `CLOUD_ML_REGION=global`, check that your models support global endpoints in Model Garden under "Supported features". For models that don't support global endpoints, either:
  - Specify a supported model via `ANTHROPIC_MODEL` or `ANTHROPIC_SMALL_FAST_MODEL`, or
  - Set a regional endpoint using `VERTEX_REGION_<MODEL_NAME>` environment variables

If you encounter 429 errors:

- For regional endpoints, ensure the primary model and small/fast model are supported in your selected region
- Consider switching to `CLOUD_ML_REGION=global` for better availability

Additional resources

- Vertex AI documentation
- Vertex AI pricing
- Vertex AI quotas and limits

Claude Code on Microsoft Foundry

> *Learn about configuring Claude Code through Microsoft Foundry, including setup, configuration, and troubleshooting.*

Prerequisites

Before configuring Claude Code with Microsoft Foundry, ensure you have:

- An Azure subscription with access to Microsoft Foundry
- RBAC permissions to create Microsoft Foundry resources and deployments
- Azure CLI installed and configured (optional - only needed if you don't have another mechanism for getting credentials)

> **Note:**
>
> If you are deploying Claude Code to multiple users, pin your model versions to prevent breakage when Anthropic releases new models.

Setup

1. Provision Microsoft Foundry resource

First, create a Claude resource in Azure:

1. Navigate to the Microsoft Foundry portal
2. Create a new resource, noting your resource name
3. Create deployments for the Claude models:
    - Claude Opus
    - Claude Sonnet
    - Claude Haiku

2. Configure Azure credentials

Claude Code supports two authentication methods for Microsoft Foundry. Choose the method that best fits your security requirements.

Option A: API key authentication

1. Navigate to your resource in the Microsoft Foundry portal
2. Go to the **Endpoints and keys** section
3. Copy **API Key**
4. Set the environment variable:

```
export ANTHROPIC_FOUNDRY_API_KEY=your-azure-api-key
```

Option B: Microsoft Entra ID authentication

When `ANTHROPIC_FOUNDRY_API_KEY` is not set, Claude Code automatically uses the Azure SDK default credential chain. This supports a variety of methods for authenticating local and remote workloads.

On local environments, you commonly may use the Azure CLI:

```
az login
```

> **Note:**
>
> When using Microsoft Foundry, the `/login` and `/logout` commands are disabled since authentication is handled through Azure credentials.

3. Configure Claude Code

Set the following environment variables to enable Microsoft Foundry:

```
## Enable Microsoft Foundry integration
export CLAUDE_CODE_USE_FOUNDRY=1

## Azure resource name (replace {resource} with your resource name)
export ANTHROPIC_FOUNDRY_RESOURCE={resource}
## Or provide the full base URL:
## export ANTHROPIC_FOUNDRY_BASE_URL=https://{resource}.services.ai.azure.com/
anthropic
```

4. Pin model versions

> **Warning:**
>
> Pin specific model versions for every deployment. If you use model aliases (`sonnet` , `opus` , `haiku`) without pinning, Claude Code may attempt to use a newer model version that isn't available in your Foundry account, breaking existing users when Anthropic releases updates. When you create Azure deployments, select a specific model version rather than "auto-update to latest."

Set the model variables to match the deployment names you created in step 1:

```
export ANTHROPIC_DEFAULT_OPUS_MODEL='claude-opus-4-6'
export ANTHROPIC_DEFAULT_SONNET_MODEL='claude-sonnet-4-6'
export ANTHROPIC_DEFAULT_HAIKU_MODEL='claude-haiku-4-5'
```

For current and legacy model IDs, see Models overview. See Model configuration for the full list of environment variables.

Azure RBAC configuration

The `Azure AI User` and `Cognitive Services User` default roles include all required permissions for invoking Claude models.

For more restrictive permissions, create a custom role with the following:

```
{
  "permissions": [
    {
      "dataActions": [
        "Microsoft.CognitiveServices/accounts/providers/*"
      ]
    }
  ]
}
```

For details, see Microsoft Foundry RBAC documentation.

Troubleshooting

If you receive an error “Failed to get token from azureADTokenProvider: ChainedTokenCredential authentication failed”:

- Configure Entra ID on the environment, or set `ANTHROPIC_FOUNDRY_API_KEY`.

Additional resources

- Microsoft Foundry documentation
- Microsoft Foundry models
- Microsoft Foundry pricing

LLM gateway configuration

> *Learn how to configure Claude Code to work with LLM gateway solutions. Covers gateway requirements, authentication configuration, model selection, and provider-specific endpoint setup.*

LLM gateways provide a centralized proxy layer between Claude Code and model providers, often providing:

- **Centralized authentication** - Single point for API key management
- **Usage tracking** - Monitor usage across teams and projects
- **Cost controls** - Implement budgets and rate limits
- **Audit logging** - Track all model interactions for compliance
- **Model routing** - Switch between providers without code changes

Gateway requirements

For an LLM gateway to work with Claude Code, it must meet the following requirements:

API format

The gateway must expose to clients at least one of the following API formats:

1. **Anthropic Messages**: `/v1/messages`, `/v1/messages/count_tokens`
   - Must forward request headers: `anthropic-beta`, `anthropic-version`
2. **Bedrock InvokeModel**: `/invoke`, `/invoke-with-response-stream`
   - Must preserve request body fields: `anthropic_beta`, `anthropic_version`
3. **Vertex rawPredict**: `:rawPredict`, `:streamRawPredict`, `/count-tokens:rawPredict`
   - Must forward request headers: `anthropic-beta`, `anthropic-version`

Failure to forward headers or preserve body fields may result in reduced functionality or inability to use Claude Code features.

> **Note:**

> Claude Code determines which features to enable based on the API format. When using the Anthropic Messages format with Bedrock or Vertex, you may need to set environment variable `CLAUDE_CODE_DISABLE_EXPERIMENTAL_BETAS=1`.

Configuration

Model selection

By default, Claude Code will use standard model names for the selected API format.

If you have configured custom model names in your gateway, use the environment variables documented in Model configuration to match your custom names.

LiteLLM configuration

> **Note:**
>
> LiteLLM is a third-party proxy service. Anthropic doesn't endorse, maintain, or audit LiteLLM's security or functionality. This guide is provided for informational purposes and may become outdated. Use at your own discretion.

Prerequisites

- Claude Code updated to the latest version
- LiteLLM Proxy Server deployed and accessible
- Access to Claude models through your chosen provider

Basic LiteLLM setup

Configure Claude Code:

Authentication methods

Static API key

Simplest method using a fixed API key:

```
## Set in environment
export ANTHROPIC_AUTH_TOKEN=sk-litellm-static-key

## Or in Claude Code settings
{
  "env": {
    "ANTHROPIC_AUTH_TOKEN": "sk-litellm-static-key"
  }
}
```

This value will be sent as the `Authorization` header.

Dynamic API key with helper

For rotating keys or per-user authentication:

1. Create an API key helper script:

```
#!/bin/bash
## ~/bin/get-litellm-key.sh

## Example: Fetch key from vault
vault kv get -field=api_key secret/litellm/claude-code

## Example: Generate JWT token
jwt encode \
  --secret="${JWT_SECRET}" \
  --exp="+1h" \
  '{"user":"'${USER}'","team":"engineering"}'
```

1. Configure Claude Code settings to use the helper:

```
{
  "apiKeyHelper": "~/bin/get-litellm-key.sh"
}
```

1. Set token refresh interval:

```
## Refresh every hour (3600000 ms)
export CLAUDE_CODE_API_KEY_HELPER_TTL_MS=3600000
```

This value will be sent as `Authorization` and `X-Api-Key` headers. The `apiKeyHelper` has lower precedence than `ANTHROPIC_AUTH_TOKEN` or `ANTHROPIC_API_KEY`.

Unified endpoint (recommended)

Using LiteLLM's Anthropic format endpoint:

```
export ANTHROPIC_BASE_URL=https://litellm-server:4000
```

Benefits of the unified endpoint over pass-through endpoints:

- Load balancing
- Fallbacks
- Consistent support for cost tracking and end-user tracking

Provider-specific pass-through endpoints (alternative)

Claude API through LiteLLM

Using pass-through endpoint:

```
export ANTHROPIC_BASE_URL=https://litellm-server:4000/anthropic
```

Amazon Bedrock through LiteLLM

Using pass-through endpoint:

```
export ANTHROPIC_BEDROCK_BASE_URL=https://litellm-server:4000/bedrock
export CLAUDE_CODE_SKIP_BEDROCK_AUTH=1
export CLAUDE_CODE_USE_BEDROCK=1
```

Google Vertex AI through LiteLLM

Using pass-through endpoint:

```
export ANTHROPIC_VERTEX_BASE_URL=https://litellm-server:4000/vertex_ai/v1
export ANTHROPIC_VERTEX_PROJECT_ID=your-gcp-project-id
export CLAUDE_CODE_SKIP_VERTEX_AUTH=1
export CLAUDE_CODE_USE_VERTEX=1
export CLOUD_ML_REGION=us-east5
```

For more detailed information, refer to the LiteLLM documentation.

Additional resources

- LiteLLM documentation
- Claude Code settings
- Enterprise network configuration
- Third-party integrations overview

Enterprise network configuration

> *Configure Claude Code for enterprise environments with proxy servers, custom Certificate Authorities (CA), and mutual Transport Layer Security (mTLS) authentication.*

Claude Code supports various enterprise network and security configurations through environment variables. This includes routing traffic through corporate proxy servers, trusting custom Certificate Authorities (CA), and authenticating with mutual Transport Layer Security (mTLS) certificates for enhanced security.

> **Note:**
>
> All environment variables shown on this page can also be configured in `settings.json`.

Proxy configuration

Environment variables

Claude Code respects standard proxy environment variables:

```
## HTTPS proxy (recommended)
export HTTPS_PROXY=https://proxy.example.com:8080

## HTTP proxy (if HTTPS not available)
export HTTP_PROXY=http://proxy.example.com:8080

## Bypass proxy for specific requests - space-separated format
export NO_PROXY="localhost 192.168.1.1 example.com .example.com"
## Bypass proxy for specific requests - comma-separated format
export NO_PROXY="localhost,192.168.1.1,example.com,.example.com"
## Bypass proxy for all requests
export NO_PROXY="*"
```

> **Note:**

Claude Code does not support SOCKS proxies.

Basic authentication

If your proxy requires basic authentication, include credentials in the proxy URL:

```
export HTTPS_PROXY=http://username:password@proxy.example.com:8080
```

Warning:

Avoid hardcoding passwords in scripts. Use environment variables or secure credential storage instead.

Tip:

For proxies requiring advanced authentication (NTLM, Kerberos, etc.), consider using an LLM Gateway service that supports your authentication method.

Custom CA certificates

If your enterprise environment uses custom CAs for HTTPS connections (whether through a proxy or direct API access), configure Claude Code to trust them:

```
export NODE_EXTRA_CA_CERTS=/path/to/ca-cert.pem
```

mTLS authentication

For enterprise environments requiring client certificate authentication:

```
## Client certificate for authentication
export CLAUDE_CODE_CLIENT_CERT=/path/to/client-cert.pem

## Client private key
export CLAUDE_CODE_CLIENT_KEY=/path/to/client-key.pem

## Optional: Passphrase for encrypted private key
export CLAUDE_CODE_CLIENT_KEY_PASSPHRASE="your-passphrase"
```

Network access requirements

Claude Code requires access to the following URLs:

- `api.anthropic.com` : Claude API endpoints
- `claude.ai` : authentication for claude.ai accounts
- `platform.claude.com` : authentication for Anthropic Console accounts

Ensure these URLs are allowlisted in your proxy configuration and firewall rules. This is especially important when using Claude Code in containerized or restricted network environments.

Claude Code on the web and Code Review connect to your repositories from Anthropic-managed infrastructure. If your GitHub Enterprise Cloud organization restricts access by IP address, enable IP allow list inheritance for installed GitHub Apps. The Claude GitHub App registers its IP ranges, so enabling this setting allows access without manual configuration. To add the ranges to your allow list manually instead, or to configure other firewalls, see the Anthropic API IP addresses.

Additional resources

- Claude Code settings
- Environment variables reference
- Troubleshooting guide

Part 12: Environment Setup

Development containers

> *Learn about the Claude Code development container for teams that need consistent, secure environments.*

The reference devcontainer setup and associated Dockerfile offer a preconfigured development container that you can use as is, or customize for your needs. This devcontainer works with the Visual Studio Code Dev Containers extension and similar tools.

The container's enhanced security measures (isolation and firewall rules) allow you to run `claude --dangerously-skip-permissions` to bypass permission prompts for unattended operation.

> **Warning:**
>
> While the devcontainer provides substantial protections, no system is completely immune to all attacks. When executed with `--dangerously-skip-permissions`, devcontainers don't prevent a malicious project from exfiltrating anything accessible in the devcontainer including Claude Code credentials. We recommend only using devcontainers when developing with trusted repositories. Always maintain good security practices and monitor Claude's activities.

Key features

- **Production-ready Node.js**: Built on Node.js 20 with essential development dependencies
- **Security by design**: Custom firewall restricting network access to only necessary services
- **Developer-friendly tools**: Includes git, ZSH with productivity enhancements, fzf, and more
- **Seamless VS Code integration**: Pre-configured extensions and optimized settings
- **Session persistence**: Preserves command history and configurations between container restarts
- **Works everywhere**: Compatible with macOS, Windows, and Linux development environments

Getting started in 4 steps

1. Install VS Code and the Remote - Containers extension
2. Clone the Claude Code reference implementation repository
3. Open the repository in VS Code
4. When prompted, click “Reopen in Container” (or use Command Palette: Cmd+Shift+P → “Remote-Containers: Reopen in Container”)

Configuration breakdown

The devcontainer setup consists of three primary components:

- **devcontainer.json**: Controls container settings, extensions, and volume mounts
- **Dockerfile**: Defines the container image and installed tools
- **init-firewall.sh**: Establishes network security rules

Security features

The container implements a multi-layered security approach with its firewall configuration:

- **Precise access control**: Restricts outbound connections to whitelisted domains only (npm registry, GitHub, Claude API, etc.)
- **Allowed outbound connections**: The firewall permits outbound DNS and SSH connections
- **Default-deny policy**: Blocks all other external network access
- **Startup verification**: Validates firewall rules when the container initializes
- **Isolation**: Creates a secure development environment separated from your main system

Customization options

The devcontainer configuration is designed to be adaptable to your needs:

- Add or remove VS Code extensions based on your workflow
- Modify resource allocations for different hardware environments
- Adjust network access permissions
- Customize shell configurations and developer tooling

Example use cases

Secure client work

Use devcontainers to isolate different client projects, ensuring code and credentials never mix between environments.

Team onboarding

New team members can get a fully configured development environment in minutes, with all necessary tools and settings pre-installed.

Consistent CI/CD environments

Mirror your devcontainer configuration in CI/CD pipelines to ensure development and production environments match.

Related resources

- VS Code devcontainers documentation
- Claude Code security best practices
- Enterprise network configuration

Optimize your terminal setup

> *Claude Code works best when your terminal is properly configured. Follow these guidelines to optimize your experience.*

Themes and appearance

Claude cannot control the theme of your terminal. That's handled by your terminal application. You can match Claude Code's theme to your terminal any time via the `/config` command.

For additional customization of the Claude Code interface itself, you can configure a custom status line to display contextual information like the current model, working directory, or git branch at the bottom of your terminal.

Line breaks

You have several options for entering line breaks into Claude Code:

- **Quick escape**: Type `\` followed by Enter to create a newline
- **Shift+Enter**: Works out of the box in iTerm2, WezTerm, Ghostty, and Kitty
- **Keyboard shortcut**: Set up a keybinding to insert a newline in other terminals

Set up Shift+Enter for other terminals

Run `/terminal-setup` within Claude Code to automatically configure Shift+Enter for VS Code, Alacritty, Zed, and Warp.

> **Note:**
>
> The `/terminal-setup` command is only visible in terminals that require manual configuration. If you're using iTerm2, WezTerm, Ghostty, or Kitty, you won't see this command because Shift+Enter already works natively.

Set up Option+Enter (VS Code, iTerm2 or macOS Terminal.app)

For Mac Terminal.app:

1. Open Settings → Profiles → Keyboard
2. Check "Use Option as Meta Key"

For iTerm2 and VS Code terminal:

1. Open Settings → Profiles → Keys
2. Under General, set Left/Right Option key to "Esc+"

Notification setup

When Claude finishes working and is waiting for your input, it fires a notification event. You can surface this event as a desktop notification through your terminal or run custom logic with notification hooks.

Terminal notifications

Kitty and Ghostty support desktop notifications without additional configuration. iTerm 2 requires setup:

1. Open iTerm 2 Settings → Profiles → Terminal
2. Enable "Notification Center Alerts"
3. Click "Filter Alerts" and check "Send escape sequence-generated alerts"

If notifications aren't appearing, verify that your terminal app has notification permissions in your OS settings.

Other terminals, including the default macOS Terminal, do not support native notifications. Use notification hooks instead.

Notification hooks

To add custom behavior when notifications fire, such as playing a sound or sending a message, configure a notification hook. Hooks run alongside terminal notifications, not as a replacement.

Handling large inputs

When working with extensive code or long instructions:

- **Avoid direct pasting**: Claude Code may struggle with very long pasted content
- **Use file-based workflows**: Write content to a file and ask Claude to read it
- **Be aware of VS Code limitations**: The VS Code terminal is particularly prone to truncating long pastes

Vim Mode

Claude Code supports a subset of Vim keybindings that can be enabled with `/vim` or configured via `/config`.

The supported subset includes:

- Mode switching: `Esc` (to NORMAL), `i` / `I`, `a` / `A`, `o` / `O` (to INSERT)
- Navigation: `h` / `j` / `k` / `l`, `w` / `e` / `b`, `0` / `$` / `^`, `gg` / `G`, `f` / `F` / `t` / `T` with `;` / `,` repeat
- Editing: `x`, `dw` / `de` / `db` / `dd` / `D`, `cw` / `ce` / `cb` / `cc` / `C`, `.` (repeat)
- Yank/paste: `yy` / `Y`, `yw` / `ye` / `yb`, `p` / `P`
- Text objects: `iw` / `aw`, `iW` / `aW`, `i"` / `a"`, `i'` / `a'`, `i(` / `a(`, `i[` / `a[`, `i{` / `a{`
- Indentation: `>>` / `<<`
- Line operations: `J` (join lines)

See Interactive mode for the complete reference.

Customize your status line

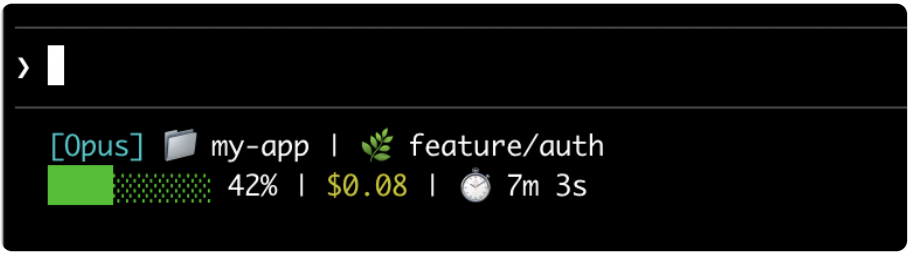
> *Configure a custom status bar to monitor context window usage, costs, and git status in Claude Code*

The status line is a customizable bar at the bottom of Claude Code that runs any shell script you configure. It receives JSON session data on stdin and displays whatever your script prints, giving you a persistent, at-a-glance view of context usage, costs, git status, or anything else you want to track.

Status lines are useful when you:

- Want to monitor context window usage as you work
- Need to track session costs
- Work across multiple sessions and need to distinguish them
- Want git branch and status always visible

Here's an example of a multi-line status line that displays git info on the first line and a color-coded context bar on the second.



A multi-line status line showing model name, directory, git branch on the first line, and a context usage progress bar with cost and duration on the second line

This page walks through setting up a basic status line, explains how the data flows from Claude Code to your script, lists all the fields you can display, and provides ready-to-use examples for common patterns like git status, cost tracking, and progress bars.

Set up a status line

Use the `/statusline` command to have Claude Code generate a script for you, or manually create a script and add it to your settings.

Use the /statusline command

The `/statusline` command accepts natural language instructions describing what you want displayed. Claude Code generates a script file in `~/.claude/` and updates your settings automatically:

```
/statusline show model name and context percentage with a progress bar
```

Manually configure a status line

Add a `statusLine` field to your user settings (`~/.claude/settings.json` , where `~` is your home directory) or project settings. Set `type` to `"command"` and point `command` to a script path or an inline shell command. For a full walkthrough of creating a script, see Build a status line step by step.

```
{
  "statusLine": {
    "type": "command",
    "command": "~/.claude/statusline.sh",
    "padding": 2
  }
}
```

The `command` field runs in a shell, so you can also use inline commands instead of a script file. This example uses `jq` to parse the JSON input and display the model name and context percentage:

```
{
  "statusLine": {
    "type": "command",
    "command": "jq -r '\"[\\(.model.display_name)] \\(.context_window.used_percent
age // 0)% context\"'"
  }
}
```

The optional `padding` field adds extra horizontal spacing (in characters) to the status line content. Defaults to `0` . This padding is in addition to the interface's built-in spacing, so it controls relative indentation rather than absolute distance from the terminal edge.

Disable the status line

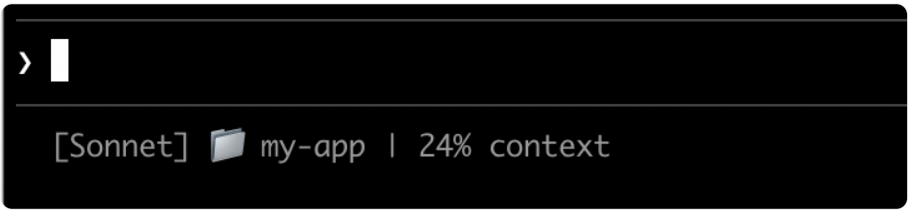
Run `/statusline` and ask it to remove or clear your status line (e.g., `/statusline delete`, `/statusline clear`, `/statusline remove it`). You can also manually delete the `statusLine` field from your settings.json.

Build a status line step by step

This walkthrough shows what's happening under the hood by manually creating a status line that displays the current model, working directory, and context window usage percentage.

> **Note:**
>
> Running `/statusline` with a description of what you want configures all of this for you automatically.

These examples use Bash scripts, which work on macOS and Linux. On Windows, see Windows configuration for PowerShell and Git Bash examples.



A status line showing model name, directory, and context percentage

Step 1: Create a script that reads JSON and prints output

Claude Code sends JSON data to your script via stdin. This script uses `jq`, a command-line JSON parser you may need to install, to extract the model name, directory, and context percentage, then prints a formatted line.

Save this to `~/.claude/statusline.sh` (where `~` is your home directory, such as `/Users/username` on macOS or `/home/username` on Linux):

```
#!/bin/bash
## Read JSON data that Claude Code sends to stdin
input=$(cat)

## Extract fields using jq
MODEL=$(echo "$input" | jq -r '.model.display_name')
DIR=$(echo "$input" | jq -r '.workspace.current_dir')
## The "// 0" provides a fallback if the field is null
PCT=$(echo "$input" | jq -r '.context_window.used_percentage // 0' | cut -d. -f1)

## Output the status line - ${DIR##*/} extracts just the folder name
echo "[$MODEL] 📁 ${DIR##*/} | ${PCT}% context"
```

Step 2: Make it executable

Mark the script as executable so your shell can run it:

```
chmod +x ~/.claude/statusline.sh
```

Step 3: Add to settings

Tell Claude Code to run your script as the status line. Add this configuration to `~/.claude/settings.json`, which sets `type` to `"command"` (meaning "run this shell command") and points `command` to your script:

```
{
  "statusLine": {
    "type": "command",
    "command": "~/.claude/statusline.sh"
  }
}
```

Your status line appears at the bottom of the interface. Settings reload automatically, but changes won't appear until your next interaction with Claude Code.

How status lines work

Claude Code runs your script and pipes JSON session data to it via stdin. Your script reads the JSON, extracts what it needs, and prints text to stdout. Claude Code displays whatever your script prints.

When it updates

Your script runs after each new assistant message, when the permission mode changes, or when vim mode toggles. Updates are debounced at 300ms, meaning rapid changes batch together and your script runs once things settle. If a new update triggers while your script is still running, the in-flight execution is cancelled. If you edit your script, the changes won't appear until your next interaction with Claude Code triggers an update.

What your script can output

- **Multiple lines**: each `echo` or `print` statement displays as a separate row. See the multi-line example.
- **Colors**: use ANSI escape codes like `\033[32m` for green (terminal must support them). See the git status example.
- **Links**: use OSC 8 escape sequences to make text clickable (Cmd+click on macOS, Ctrl+click on Windows/Linux). Requires a terminal that supports hyperlinks like iTerm2, Kitty, or WezTerm. See the clickable links example.

> **Note:**
>
> The status line runs locally and does not consume API tokens. It temporarily hides during certain UI interactions, including autocomplete suggestions, the help menu, and permission prompts.

Available data

Claude Code sends the following JSON fields to your script via stdin:

Field	Description
`model.id`, `model.display_name`	Current model identifier and display name
`cwd`, `workspace.current_dir`	Current working directory. Both fields contain the same value; `workspace.current_dir` is preferred for consistency with `workspace.project_dir`.

Field	Description
`workspace.project_dir`	Directory where Claude Code was launched, which may differ from `cwd` if the working directory changes during a session
`cost.total_cost_usd`	Total session cost in USD
`cost.total_duration_ms`	Total wall-clock time since the session started, in milliseconds
`cost.total_api_duration_ms`	Total time spent waiting for API responses in milliseconds
`cost.total_lines_added`, `cost.total_lines_removed`	Lines of code changed
`context_window.total_input_tokens`, `context_window.total_output_tokens`	Cumulative token counts across the session
`context_window.context_window_size`	Maximum context window size in tokens. 200000 by default, or 1000000 for models with extended context.
`context_window.used_percentage`	Pre-calculated percentage of context window used
`context_window.remaining_percentage`	Pre-calculated percentage of context window remaining
`context_window.current_usage`	Token counts from the last API call, described in context window fields
`exceeds_200k_tokens`	Whether the total token count (input, cache, and output tokens combined) from the most recent API response exceeds 200k. This is a fixed threshold regardless of actual context window size.
`session_id`	Unique session identifier
`transcript_path`	Path to conversation transcript file
`version`	Claude Code version
`output_style.name`	Name of the current output style

Field	Description
`vim.mode`	Current vim mode (`NORMAL` or `INSERT`) when vim mode is enabled
`agent.name`	Agent name when running with the `--agent` flag or agent settings configured
`worktree.name`	Name of the active worktree. Present only during `--worktree` sessions
`worktree.path`	Absolute path to the worktree directory
`worktree.branch`	Git branch name for the worktree (for example, `"worktree-my-feature"`). Absent for hook-based worktrees
`worktree.original_cwd`	The directory Claude was in before entering the worktree
`worktree.original_branch`	Git branch checked out before entering the worktree. Absent for hook-based worktrees

Your status line command receives this JSON structure via stdin:

```
{
  "cwd": "/current/working/directory",
  "session_id": "abc123...",
  "transcript_path": "/path/to/transcript.jsonl",
  "model": {
    "id": "claude-opus-4-6",
    "display_name": "Opus"
  },
  "workspace": {
    "current_dir": "/current/working/directory",
    "project_dir": "/original/project/directory"
  },
  "version": "1.0.80",
  "output_style": {
    "name": "default"
  },
  "cost": {
    "total_cost_usd": 0.01234,
    "total_duration_ms": 45000,
    "total_api_duration_ms": 2300,
    "total_lines_added": 156,
    "total_lines_removed": 23
  },
  "context_window": {
    "total_input_tokens": 15234,
    "total_output_tokens": 4521,
    "context_window_size": 200000,
    "used_percentage": 8,
    "remaining_percentage": 92,
    "current_usage": {
      "input_tokens": 8500,
      "output_tokens": 1200,
      "cache_creation_input_tokens": 5000,
      "cache_read_input_tokens": 2000
    }
  },
  "exceeds_200k_tokens": false,
  "vim": {
```

```
    "mode": "NORMAL"
  },
  "agent": {
    "name": "security-reviewer"
  },
  "worktree": {
    "name": "my-feature",
    "path": "/path/to/.claude/worktrees/my-feature",
    "branch": "worktree-my-feature",
    "original_cwd": "/path/to/project",
    "original_branch": "main"
  }
}
```

Fields that may be absent (not present in JSON):

- `vim` : appears only when vim mode is enabled
- `agent` : appears only when running with the `--agent` flag or agent settings configured
- `worktree` : appears only during `--worktree` sessions. When present, `branch` and `original_branch` may also be absent for hook-based worktrees

Fields that may be `null` :

- `context_window.current_usage` : `null` before the first API call in a session
- `context_window.used_percentage` , `context_window.remaining_percentage` : may be `null` early in the session

Handle missing fields with conditional access and null values with fallback defaults in your scripts.

Context window fields

The `context_window` object provides two ways to track context usage:

- **Cumulative totals** (`total_input_tokens` , `total_output_tokens`): sum of all tokens across the entire session, useful for tracking total consumption
- **Current usage** (`current_usage`): token counts from the most recent API call, use this for accurate context percentage since it reflects the actual context state

The `current_usage` object contains:

- `input_tokens` : input tokens in current context

- `output_tokens`: output tokens generated
- `cache_creation_input_tokens`: tokens written to cache
- `cache_read_input_tokens`: tokens read from cache

The `used_percentage` field is calculated from input tokens only: `input_tokens + cache_creation_input_tokens + cache_read_input_tokens`. It does not include `output_tokens`.

If you calculate context percentage manually from `current_usage`, use the same input-only formula to match `used_percentage`.

The `current_usage` object is `null` before the first API call in a session.

Examples

These examples show common status line patterns. To use any example:

1. Save the script to a file like `~/.claude/statusline.sh` (or `.py` / `.js`)
2. Make it executable: `chmod +x ~/.claude/statusline.sh`
3. Add the path to your settings

The Bash examples use `jq` to parse JSON. Python and Node.js have built-in JSON parsing.

Context window usage

Display the current model and context window usage with a visual progress bar. Each script reads JSON from stdin, extracts the `used_percentage` field, and builds a 10-character bar where filled blocks (▓) represent usage:

A status line showing model name and a progress bar with percentage

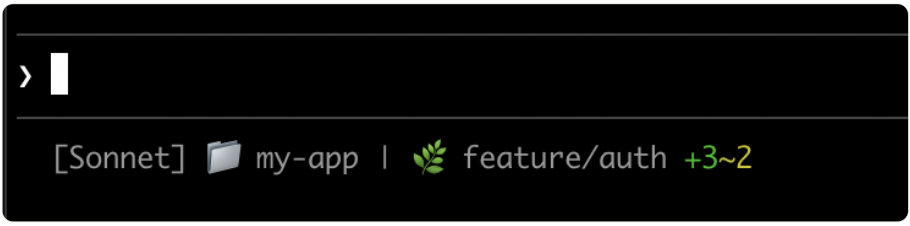
```bash
#!/bin/bash
## Read all of stdin into a variable
input=$(cat)

## Extract fields with jq, "// 0" provides fallback for null
MODEL=$(echo "$input" | jq -r '.model.display_name')
PCT=$(echo "$input" | jq -r '.context_window.used_percentage // 0' | cut -d. -f1)

## Build progress bar: printf -v creates a run of spaces, then
## ${var// /▓} replaces each space with a block character
BAR_WIDTH=10
FILLED=$((PCT * BAR_WIDTH / 100))
EMPTY=$((BAR_WIDTH - FILLED))
BAR=""
[ "$FILLED" -gt 0 ] && printf -v FILL "%${FILLED}s" && BAR="${FILL// /▓}"
[ "$EMPTY" -gt 0 ] && printf -v PAD "%${EMPTY}s" && BAR="${BAR}${PAD// /░}"

echo "[$MODEL] $BAR $PCT%"
```

```python
#!/usr/bin/env python3
import json, sys

## json.load reads and parses stdin in one step
data = json.load(sys.stdin)
model = data['model']['display_name']
## "or 0" handles null values
pct = int(data.get('context_window', {}).get('used_percentage', 0) or 0)

## String multiplication builds the bar
filled = pct * 10 // 100
bar = '▓' * filled + '░' * (10 - filled)

print(f"[{model}] {bar} {pct}%")
```

```
#!/usr/bin/env node
// Node.js reads stdin asynchronously with events
let input = '';
process.stdin.on('data', chunk => input += chunk);
process.stdin.on('end', () => {
    const data = JSON.parse(input);
    const model = data.model.display_name;
    // Optional chaining (?.) safely handles null fields
    const pct = Math.floor(data.context_window?.used_percentage || 0);

    // String.repeat() builds the bar
    const filled = Math.floor(pct * 10 / 100);
    const bar = '▓'.repeat(filled) + '░'.repeat(10 - filled);

    console.log(`[${model}] ${bar} ${pct}%`);
});
```

Git status with colors

Show git branch with color-coded indicators for staged and modified files. This script uses ANSI escape codes for terminal colors: `\033[32m` is green, `\033[33m` is yellow, and `\033[0m` resets to default.



A status line showing model, directory, git branch, and colored indicators for staged and modified files

Each script checks if the current directory is a git repository, counts staged and modified files, and displays color-coded indicators:

```
#!/bin/bash
input=$(cat)

MODEL=$(echo "$input" | jq -r '.model.display_name')
DIR=$(echo "$input" | jq -r '.workspace.current_dir')

GREEN='\033[32m'
YELLOW='\033[33m'
RESET='\033[0m'

if git rev-parse --git-dir > /dev/null 2>&1; then
    BRANCH=$(git branch --show-current 2>/dev/null)
    STAGED=$(git diff --cached --numstat 2>/dev/null | wc -l | tr -d ' ')
    MODIFIED=$(git diff --numstat 2>/dev/null | wc -l | tr -d ' ')

    GIT_STATUS=""
    [ "$STAGED" -gt 0 ] && GIT_STATUS="${GREEN}+${STAGED}${RESET}"
    [ "$MODIFIED" -gt 0 ] && GIT_STATUS="${GIT_STATUS}${YELLOW}~${MODIFIED}$
{RESET}"

    echo -e "[$MODEL] 📁 ${DIR##*/} | 🌿 $BRANCH $GIT_STATUS"
else
    echo "[$MODEL] 📁 ${DIR##*/}"
fi
```

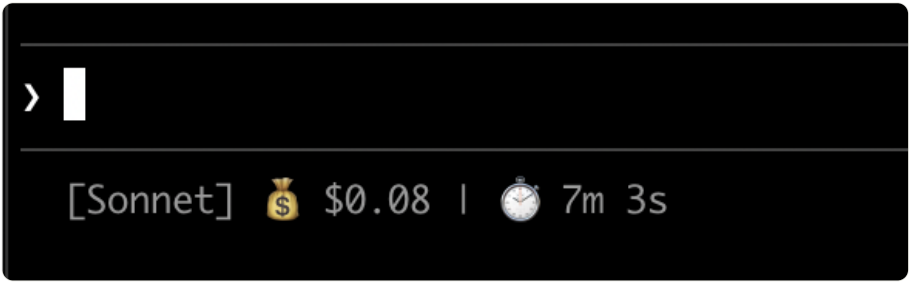
```
#!/usr/bin/env python3
import json, sys, subprocess, os

data = json.load(sys.stdin)
model = data['model']['display_name']
directory = os.path.basename(data['workspace']['current_dir'])

GREEN, YELLOW, RESET = '\033[32m', '\033[33m', '\033[0m'

try:
    subprocess.check_output(['git', 'rev-parse', '--git-dir'], stderr=subprocess.D
EVNULL)
    branch = subprocess.check_output(['git', 'branch', '--show-current'], text=Tru
e).strip()
    staged_output = subprocess.check_output(['git', 'diff', '--cached', '--
numstat'], text=True).strip()
    modified_output = subprocess.check_output(['git', 'diff', '--numstat'], text=T
rue).strip()
    staged = len(staged_output.split('\n')) if staged_output else 0
    modified = len(modified_output.split('\n')) if modified_output else 0

    git_status = f"{GREEN}+{staged}{RESET}" if staged else ""
    git_status += f"{YELLOW}~{modified}{RESET}" if modified else ""

    print(f"[{model}] 📁 {directory} | 🌿 {branch} {git_status}")
except:
    print(f"[{model}] 📁 {directory}")
```

```
#!/usr/bin/env node
const { execSync } = require('child_process');
const path = require('path');

let input = '';
process.stdin.on('data', chunk => input += chunk);
process.stdin.on('end', () => {
    const data = JSON.parse(input);
    const model = data.model.display_name;
    const dir = path.basename(data.workspace.current_dir);

    const GREEN = '\x1b[32m', YELLOW = '\x1b[33m', RESET = '\x1b[0m';

    try {
        execSync('git rev-parse --git-dir', { stdio: 'ignore' });
        const branch = execSync('git branch --show-current', { encoding:
'utf8' }).trim();
        const staged = execSync('git diff --cached --numstat', { encoding: 'utf8'
}).trim().split('\n').filter(Boolean).length;
        const modified = execSync('git diff --numstat', { encoding:
'utf8' }).trim().split('\n').filter(Boolean).length;

        let gitStatus = staged ? `${GREEN}+${staged}${RESET}` : '';
        gitStatus += modified ? `${YELLOW}~${modified}${RESET}` : '';

        console.log(`[${model}] 📁 ${dir} | 🌿 ${branch} ${gitStatus}`);
    } catch {
        console.log(`[${model}] 📁 ${dir}`);
    }
});
```

Cost and duration tracking

Track your session's API costs and elapsed time. The `cost.total_cost_usd` field accumulates the cost of all API calls in the current session. The `cost.total_duration_ms` field measures total elapsed time since the session started, while `cost.total_api_duration_ms` tracks only the time spent waiting for API responses.

Each script formats cost as currency and converts milliseconds to minutes and seconds:



A status line showing model name, session cost, and duration

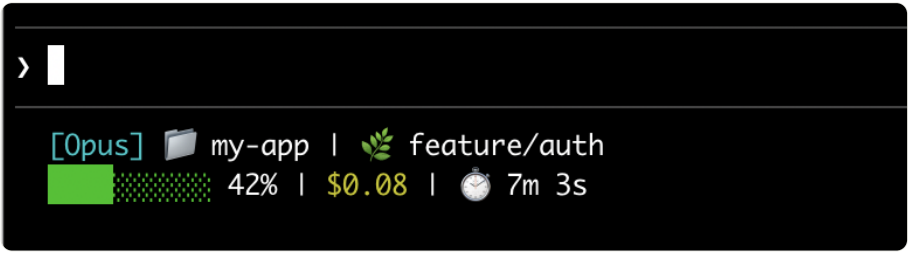
```bash
#!/bin/bash
input=$(cat)

MODEL=$(echo "$input" | jq -r '.model.display_name')
COST=$(echo "$input" | jq -r '.cost.total_cost_usd // 0')
DURATION_MS=$(echo "$input" | jq -r '.cost.total_duration_ms // 0')

COST_FMT=$(printf '$%.2f' "$COST")
DURATION_SEC=$((DURATION_MS / 1000))
MINS=$((DURATION_SEC / 60))
SECS=$((DURATION_SEC % 60))

echo "[$MODEL] 💰 $COST_FMT | ⏱️ ${MINS}m ${SECS}s"
```

```python
#!/usr/bin/env python3
import json, sys

data = json.load(sys.stdin)
model = data['model']['display_name']
cost = data.get('cost', {}).get('total_cost_usd', 0) or 0
duration_ms = data.get('cost', {}).get('total_duration_ms', 0) or 0

duration_sec = duration_ms // 1000
mins, secs = duration_sec // 60, duration_sec % 60

print(f"[{model}] 💰 ${cost:.2f} | ⏱️ {mins}m {secs}s")
```

```
#!/usr/bin/env node
let input = '';
process.stdin.on('data', chunk => input += chunk);
process.stdin.on('end', () => {
    const data = JSON.parse(input);
    const model = data.model.display_name;
    const cost = data.cost?.total_cost_usd || 0;
    const durationMs = data.cost?.total_duration_ms || 0;

    const durationSec = Math.floor(durationMs / 1000);
    const mins = Math.floor(durationSec / 60);
    const secs = durationSec % 60;

    console.log(`[${model}] 💰 $${cost.toFixed(2)} | ⏱️ ${mins}m ${secs}s`);
});
```

Display multiple lines

Your script can output multiple lines to create a richer display. Each `echo` statement produces a separate row in the status area.



A multi-line status line showing model name, directory, git branch on the first line, and a context usage progress bar with cost and duration on the second line

This example combines several techniques: threshold-based colors (green under 70%, yellow 70-89%, red 90%+), a progress bar, and git branch info. Each `print` or `echo` statement creates a separate row:

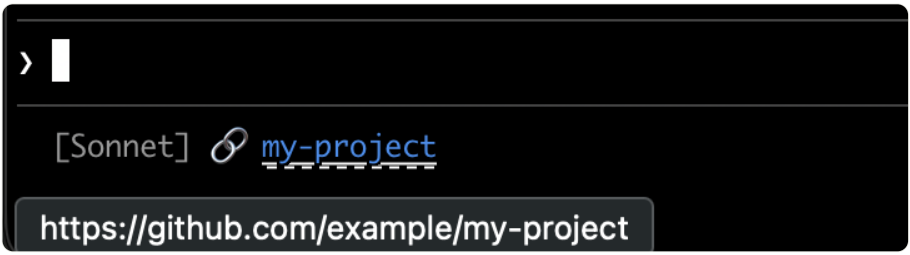
```
#!/bin/bash
input=$(cat)

MODEL=$(echo "$input" | jq -r '.model.display_name')
DIR=$(echo "$input" | jq -r '.workspace.current_dir')
COST=$(echo "$input" | jq -r '.cost.total_cost_usd // 0')
PCT=$(echo "$input" | jq -r '.context_window.used_percentage // 0' | cut -d. -f1)
DURATION_MS=$(echo "$input" | jq -r '.cost.total_duration_ms // 0')

CYAN='\033[36m'; GREEN='\033[32m'; YELLOW='\033[33m'; RED='\033[31m'; RESET='\033[
0m'

## Pick bar color based on context usage
if [ "$PCT" -ge 90 ]; then BAR_COLOR="$RED"
elif [ "$PCT" -ge 70 ]; then BAR_COLOR="$YELLOW"
else BAR_COLOR="$GREEN"; fi

FILLED=$((PCT / 10)); EMPTY=$((10 - FILLED))
printf -v FILL "%${FILLED}s"; printf -v PAD "%${EMPTY}s"
BAR="${FILL// /▓}${PAD// /░}"

MINS=$((DURATION_MS / 60000)); SECS=$(((DURATION_MS % 60000) / 1000))

BRANCH=""
git rev-parse --git-dir > /dev/null 2>&1 && BRANCH=" | 🌿 $(git branch --show-
current 2>/dev/null)"

echo -e "${CYAN}[$MODEL]${RESET} 📁 ${DIR##*/}$BRANCH"
COST_FMT=$(printf '$%.2f' "$COST")
echo -e "${BAR_COLOR}${BAR}${RESET} ${PCT}% | ${YELLOW}${COST_FMT}${RESET} | ⏱️ $
{MINS}m ${SECS}s"
```

```
#!/usr/bin/env python3
import json, sys, subprocess, os

data = json.load(sys.stdin)
model = data['model']['display_name']
directory = os.path.basename(data['workspace']['current_dir'])
cost = data.get('cost', {}).get('total_cost_usd', 0) or 0
pct = int(data.get('context_window', {}).get('used_percentage', 0) or 0)
duration_ms = data.get('cost', {}).get('total_duration_ms', 0) or 0

CYAN, GREEN, YELLOW, RED, RESET = '\033[36m', '\033[32m', '\033[33m', '\033[31m',
'\033[0m'

bar_color = RED if pct ≥ 90 else YELLOW if pct ≥ 70 else GREEN
filled = pct // 10
bar = '█' * filled + '░' * (10 - filled)

mins, secs = duration_ms // 60000, (duration_ms % 60000) // 1000

try:
    branch = subprocess.check_output(['git', 'branch', '--show-current'], text=Tru
e, stderr=subprocess.DEVNULL).strip()
    branch = f" | 🌿 {branch}" if branch else ""
except:
    branch = ""

print(f"{CYAN}[{model}]{RESET} 📁 {directory}{branch}")
print(f"{bar_color}{bar}{RESET} {pct}% | {YELLOW}${cost:.2f}{RESET} | ⏱️ {mins}m
{secs}s")
```

```
#!/usr/bin/env node
const { execSync } = require('child_process');
const path = require('path');

let input = '';
process.stdin.on('data', chunk => input += chunk);
process.stdin.on('end', () => {
    const data = JSON.parse(input);
    const model = data.model.display_name;
    const dir = path.basename(data.workspace.current_dir);
    const cost = data.cost?.total_cost_usd || 0;
    const pct = Math.floor(data.context_window?.used_percentage || 0);
    const durationMs = data.cost?.total_duration_ms || 0;

    const CYAN = '\x1b[36m', GREEN = '\x1b[32m', YELLOW = '\x1b[33m', RED = '\x1b[
31m', RESET = '\x1b[0m';

    const barColor = pct >= 90 ? RED : pct >= 70 ? YELLOW : GREEN;
    const filled = Math.floor(pct / 10);
    const bar = '▓'.repeat(filled) + '░'.repeat(10 - filled);

    const mins = Math.floor(durationMs / 60000);
    const secs = Math.floor((durationMs % 60000) / 1000);

    let branch = '';
    try {
        branch = execSync('git branch --show-current', { encoding: 'utf8', stdio:
['pipe', 'pipe', 'ignore'] }).trim();
        branch = branch ? ` | 🌿 ${branch}` : '';
    } catch {}

    console.log(`${CYAN}[${model}]${RESET} 📁 ${dir}${branch}`);
    console.log(`${barColor}${bar}${RESET} ${pct}% | ${YELLOW}$${cost.toFixed(2)}$
{RESET} | ⏱️ ${mins}m ${secs}s`);
});
```

Clickable links

This example creates a clickable link to your GitHub repository. It reads the git remote URL, converts SSH format to HTTPS with `sed`, and wraps the repo name in OSC 8 escape codes. Hold Cmd (macOS) or Ctrl (Windows/Linux) and click to open the link in your browser.



A status line showing a clickable link to a GitHub repository

Each script gets the git remote URL, converts SSH format to HTTPS, and wraps the repo name in OSC 8 escape codes. The Bash version uses `printf '%b'` which interprets backslash escapes more reliably than `echo -e` across different shells:

```
#!/bin/bash
input=$(cat)

MODEL=$(echo "$input" | jq -r '.model.display_name')

## Convert git SSH URL to HTTPS
REMOTE=$(git remote get-url origin 2>/dev/null | sed 's/git@github.com:/https:\/\/
github.com\//' | sed 's/\.git$//')

if [ -n "$REMOTE" ]; then
    REPO_NAME=$(basename "$REMOTE")
    # OSC 8 format: \e]8;;URL\a then TEXT then \e]8;;\a
    # printf %b interprets escape sequences reliably across shells
    printf '%b' "[$MODEL] 🔗 \e]8;;${REMOTE}\a${REPO_NAME}\e]8;;\a\n"
else
    echo "[$MODEL]"
fi
```

```python
#!/usr/bin/env python3
import json, sys, subprocess, re, os

data = json.load(sys.stdin)
model = data['model']['display_name']

## Get git remote URL
try:
    remote = subprocess.check_output(
        ['git', 'remote', 'get-url', 'origin'],
        stderr=subprocess.DEVNULL, text=True
    ).strip()
    # Convert SSH to HTTPS format
    remote = re.sub(r'^git@github\.com:', 'https://github.com/', remote)
    remote = re.sub(r'\.git$', '', remote)
    repo_name = os.path.basename(remote)
    # OSC 8 escape sequences
    link = f"\033]8;;{remote}\a{repo_name}\033]8;;\a"
    print(f"[{model}] 🔗 {link}")
except:
    print(f"[{model}]")
```

```
#!/usr/bin/env node
const { execSync } = require('child_process');
const path = require('path');

let input = '';
process.stdin.on('data', chunk => input += chunk);
process.stdin.on('end', () => {
    const data = JSON.parse(input);
    const model = data.model.display_name;

    try {
        let remote = execSync('git remote get-url origin', { encoding: 'utf8', std
io: ['pipe', 'pipe', 'ignore'] }).trim();
        // Convert SSH to HTTPS format
        remote = remote.replace(/^git@github\.com:/, 'https://github.com/').replac
e(/\.git$/, '');
        const repoName = path.basename(remote);
        // OSC 8 escape sequences
        const link = `\x1b]8;;${remote}\x07${repoName}\x1b]8;;\x07`;
        console.log(`[${model}] 🔗 ${link}`);
    } catch {
        console.log(`[${model}]`);
    }
});
```

Cache expensive operations

Your status line script runs frequently during active sessions. Commands like `git status` or `git diff` can be slow, especially in large repositories. This example caches git information to a temp file and only refreshes it every 5 seconds.

Use a stable, fixed filename for the cache file like `/tmp/statusline-git-cache`. Each status line invocation runs as a new process, so process-based identifiers like `$$`, `os.getpid()`, or `process.pid` produce a different value every time and the cache is never reused.

Each script checks if the cache file is missing or older than 5 seconds before running git commands:

```
#!/bin/bash
input=$(cat)

MODEL=$(echo "$input" | jq -r '.model.display_name')
DIR=$(echo "$input" | jq -r '.workspace.current_dir')

CACHE_FILE="/tmp/statusline-git-cache"
CACHE_MAX_AGE=5  # seconds

cache_is_stale() {
    [ ! -f "$CACHE_FILE" ] || \
    # stat -f %m is macOS, stat -c %Y is Linux
    [ $(($(date +%s) - $(stat -f %m "$CACHE_FILE" 2>/dev/null || stat -c %Y "$CACH
E_FILE" 2>/dev/null || echo 0))) -gt $CACHE_MAX_AGE ]
}

if cache_is_stale; then
    if git rev-parse --git-dir > /dev/null 2>&1; then
        BRANCH=$(git branch --show-current 2>/dev/null)
        STAGED=$(git diff --cached --numstat 2>/dev/null | wc -l | tr -d ' ')
        MODIFIED=$(git diff --numstat 2>/dev/null | wc -l | tr -d ' ')
        echo "$BRANCH|$STAGED|$MODIFIED" > "$CACHE_FILE"
    else
        echo "||" > "$CACHE_FILE"
    fi
fi

IFS='|' read -r BRANCH STAGED MODIFIED < "$CACHE_FILE"

if [ -n "$BRANCH" ]; then
    echo "[$MODEL] 📁 ${DIR##*/} | 🌿 $BRANCH +$STAGED ~$MODIFIED"
else
    echo "[$MODEL] 📁 ${DIR##*/}"
fi
```

```
#!/usr/bin/env python3
import json, sys, subprocess, os, time

data = json.load(sys.stdin)
model = data['model']['display_name']
directory = os.path.basename(data['workspace']['current_dir'])

CACHE_FILE = "/tmp/statusline-git-cache"
CACHE_MAX_AGE = 5  # seconds

def cache_is_stale():
    if not os.path.exists(CACHE_FILE):
        return True
    return time.time() - os.path.getmtime(CACHE_FILE) > CACHE_MAX_AGE

if cache_is_stale():
    try:
        subprocess.check_output(['git', 'rev-parse', '--git-dir'], stderr=subproce
ss.DEVNULL)
        branch = subprocess.check_output(['git', 'branch', '--show-current'],
text=True).strip()
        staged = subprocess.check_output(['git', 'diff', '--cached', '--
numstat'], text=True).strip()
        modified = subprocess.check_output(['git', 'diff', '--numstat'],
text=True).strip()
        staged_count = len(staged.split('\n')) if staged else 0
        modified_count = len(modified.split('\n')) if modified else 0
        with open(CACHE_FILE, 'w') as f:
            f.write(f"{branch}|{staged_count}|{modified_count}")
    except:
        with open(CACHE_FILE, 'w') as f:
            f.write("||")

with open(CACHE_FILE) as f:
    branch, staged, modified = f.read().strip().split('|')

if branch:
```

```
    print(f"[{model}] 📁 {directory} | 🌿 {branch} +{staged} ~{modified}")
else:
    print(f"[{model}] 📁 {directory}")
```

```
#!/usr/bin/env node
const { execSync } = require('child_process');
const fs = require('fs');
const path = require('path');

let input = '';
process.stdin.on('data', chunk => input += chunk);
process.stdin.on('end', () => {
    const data = JSON.parse(input);
    const model = data.model.display_name;
    const dir = path.basename(data.workspace.current_dir);

    const CACHE_FILE = '/tmp/statusline-git-cache';
    const CACHE_MAX_AGE = 5; // seconds

    const cacheIsStale = () => {
        if (!fs.existsSync(CACHE_FILE)) return true;
        return (Date.now() / 1000) - fs.statSync(CACHE_FILE).mtimeMs / 1000 >
CACHE_MAX_AGE;
    };

    if (cacheIsStale()) {
        try {
            execSync('git rev-parse --git-dir', { stdio: 'ignore' });
            const branch = execSync('git branch --show-current', { encoding: 'utf8
' }).trim();
            const staged = execSync('git diff --cached --numstat', { encoding: 'ut
f8' }).trim().split('\n').filter(Boolean).length;
            const modified = execSync('git diff --numstat', { encoding:
'utf8' }).trim().split('\n').filter(Boolean).length;
            fs.writeFileSync(CACHE_FILE, `${branch}|${staged}|${modified}`);
        } catch {
            fs.writeFileSync(CACHE_FILE, '||');
        }
    }

    const [branch, staged, modified] = fs.readFileSync(CACHE_FILE,
'utf8').trim().split('|');
```

```
    if (branch) {
        console.log(`[${model}] 📁 ${dir} | 🌿 ${branch} +${staged} ~${modified}
`);
    } else {
        console.log(`[${model}] 📁 ${dir}`);
    }
});
```

Windows configuration

On Windows, Claude Code runs status line commands through Git Bash. You can invoke PowerShell from that shell:

```
{
  "statusLine": {
    "type": "command",
    "command": "powershell -NoProfile -File C:/Users/username/.claude/
statusline.ps1"
  }
}
```

```
$input_json = $input | Out-String | ConvertFrom-Json
$cwd = $input_json.cwd
$model = $input_json.model.display_name
$used = $input_json.context_window.used_percentage
$dirname = Split-Path $cwd -Leaf

if ($used) {
    Write-Host "$dirname [$model] ctx: $used%"
} else {
    Write-Host "$dirname [$model]"
}
```

Or run a Bash script directly:

```
{
  "statusLine": {
    "type": "command",
    "command": "~/.claude/statusline.sh"
  }
}
```

```
#!/usr/bin/env bash
input=$(cat)
cwd=$(echo "$input" | grep -o '"cwd":"[^"]*"' | cut -d'"' -f4)
model=$(echo "$input" | grep -o '"display_name":"[^"]*"' | cut -d'"' -f4)
dirname="${cwd##*[/\\]}"
echo "$dirname [$model]"
```

Tips

- **Test with mock input**: `echo '{"model":{"display_name":"Opus"},"context_window":{"used_percentage":25}}' | ./statusline.sh`
- **Keep output short**: the status bar has limited width, so long output may get truncated or wrap awkwardly
- **Cache slow operations**: your script runs frequently during active sessions, so commands like `git status` can cause lag. See the caching example for how to handle this.

Community projects like ccstatusline and starship-claude provide pre-built configurations with themes and additional features.

Troubleshooting

Status line not appearing

- Verify your script is executable: `chmod +x ~/.claude/statusline.sh`
- Check that your script outputs to stdout, not stderr
- Run your script manually to verify it produces output
- If `disableAllHooks` is set to `true` in your settings, the status line is also disabled. Remove this setting or set it to `false` to re-enable.

- Run `claude --debug` to log the exit code and stderr from the first status line invocation in a session
- Ask Claude to read your settings file and execute the `statusLine` command directly to surface errors

Status line shows `--` or empty values

- Fields may be `null` before the first API response completes
- Handle null values in your script with fallbacks such as `// 0` in jq
- Restart Claude Code if values remain empty after multiple messages

Context percentage shows unexpected values

- Use `used_percentage` for accurate context state rather than cumulative totals
- The `total_input_tokens` and `total_output_tokens` are cumulative across the session and may exceed the context window size
- Context percentage may differ from `/context` output due to when each is calculated

OSC 8 links not clickable

- Verify your terminal supports OSC 8 hyperlinks (iTerm2, Kitty, WezTerm)
- Terminal.app does not support clickable links
- SSH and tmux sessions may strip OSC sequences depending on configuration
- If escape sequences appear as literal text like `\e]8;;`, use `printf '%b'` instead of `echo -e` for more reliable escape handling

Display glitches with escape sequences

- Complex escape sequences (ANSI colors, OSC 8 links) can occasionally cause garbled output if they overlap with other UI updates
- If you see corrupted text, try simplifying your script to plain text output
- Multi-line status lines with escape codes are more prone to rendering issues than single-line plain text

Script errors or hangs

- Scripts that exit with non-zero codes or produce no output cause the status line to go blank
- Slow scripts block the status line from updating until they complete. Keep scripts fast to avoid stale output.
- If a new update triggers while a slow script is running, the in-flight script is cancelled
- Test your script independently with mock input before configuring it

Notifications share the status line row

- System notifications like MCP server errors, auto-updates, and token warnings display on the right side of the same row as your status line
- Enabling verbose mode adds a token counter to this area
- On narrow terminals, these notifications may truncate your status line output

Part 13: Troubleshooting & Changelog

Troubleshooting

Discover solutions to common issues with Claude Code installation and usage.

Troubleshoot installation issues

Tip:

If you'd rather skip the terminal entirely, the Claude Code Desktop app lets you install and use Claude Code through a graphical interface. Download it for macOS or Windows and start coding without any command-line setup.

Find the error message or symptom you're seeing:

What you see	Solution
`command not found: claude` or `'claude' is not recognized`	Fix your PATH
`syntax error near unexpected token '<'`	Install script returns HTML
`curl: (56) Failure writing output to destination`	Download script first, then run it
`Killed` during install on Linux	Add swap space for low-memory servers
`TLS connect error` or `SSL/TLS secure channel`	Update CA certificates

What you see	Solution
`Failed to fetch version` or can't reach download server	Check network and proxy settings
`irm is not recognized` or `&& is not valid`	Use the right command for your shell
`Claude Code on Windows requires git-bash`	Install or configure Git Bash
`Error loading shared library`	Wrong binary variant for your system
`Illegal instruction` on Linux	Architecture mismatch
`dyld: cannot load` or `Abort trap` on macOS	Binary incompatibility
`Invoke-Expression: Missing argument in parameter list`	Install script returns HTML
`App unavailable in region`	Claude Code is not available in your country. See supported countries.
`unable to get local issuer certificate`	Configure corporate CA certificates
`OAuth error` or `403 Forbidden`	Fix authentication

If your issue isn't listed, work through these diagnostic steps.

Debug installation problems

Check network connectivity

The installer downloads from `storage.googleapis.com`. Verify you can reach it:

```
curl -sI https://storage.googleapis.com
```

If this fails, your network may be blocking the connection. Common causes:

- Corporate firewalls or proxies blocking Google Cloud Storage
- Regional network restrictions: try a VPN or alternative network
- TLS/SSL issues: update your system's CA certificates, or check if `HTTPS_PROXY` is configured

If you're behind a corporate proxy, set `HTTPS_PROXY` and `HTTP_PROXY` to your proxy's address before installing. Ask your IT team for the proxy URL if you don't know it, or check your browser's proxy settings.

This example sets both proxy variables, then runs the installer through your proxy:

```
export HTTP_PROXY=http://proxy.example.com:8080
export HTTPS_PROXY=http://proxy.example.com:8080
curl -fsSL https://claude.ai/install.sh | bash
```

Verify your PATH

If installation succeeded but you get a `command not found` or `not recognized` error when running `claude`, the install directory isn't in your PATH. Your shell searches for programs in directories listed in PATH, and the installer places `claude` at `~/.local/bin/claude` on macOS/Linux or `%USERPROFILE%\.local\bin\claude.exe` on Windows.

Check if the install directory is in your PATH by listing your PATH entries and filtering for `local/bin`:

macOS/Linux

```
echo $PATH | tr ':' '\n' | grep local/bin
```

If there's no output, the directory is missing. Add it to your shell configuration:

```
## Zsh (macOS default)
echo 'export PATH="$HOME/.local/bin:$PATH"' >> ~/.zshrc
source ~/.zshrc

## Bash (Linux default)
echo 'export PATH="$HOME/.local/bin:$PATH"' >> ~/.bashrc
source ~/.bashrc
```

Alternatively, close and reopen your terminal.

Verify the fix worked:

```
claude --version
```

Windows PowerShell

```
$env:PATH -split ';' | Select-String 'local\\bin'
```

If there's no output, add the install directory to your User PATH:

```
$currentPath = [Environment]::GetEnvironmentVariable('PATH', 'User')
[Environment]::SetEnvironmentVariable('PATH', "$currentPath;$env:USERPROFILE\.local\bin", 'User')
```

Restart your terminal for the change to take effect.

Verify the fix worked:

```
claude --version
```

Windows CMD

```
echo %PATH% | findstr /i "local\bin"
```

If there's no output, open System Settings, go to Environment Variables, and add `%USERPROFILE%\.local\bin` to your User PATH variable. Restart your terminal.

Verify the fix worked:

```
claude --version
```

Check for conflicting installations

Multiple Claude Code installations can cause version mismatches or unexpected behavior. Check what's installed:

macOS/Linux

List all `claude` binaries found in your PATH:

```
which -a claude
```

Check whether the native installer and npm versions are present:

```
ls -la ~/.local/bin/claude
```

```
ls -la ~/.claude/local/
```

```
npm -g ls @anthropic-ai/claude-code 2>/dev/null
```

Windows PowerShell

```
where.exe claude
Test-Path "$env:LOCALAPPDATA\Claude Code\claude.exe"
```

If you find multiple installations, keep only one. The native install at `~/.local/bin/claude` is recommended. Remove any extra installations:

Uninstall an npm global install:

```
npm uninstall -g @anthropic-ai/claude-code
```

Remove a Homebrew install on macOS:

```
brew uninstall --cask claude-code
```

Check directory permissions

The installer needs write access to `~/.local/bin/` and `~/.claude/`. If installation fails with permission errors, check whether these directories are writable:

```
test -w ~/.local/bin && echo "writable" || echo "not writable"
test -w ~/.claude && echo "writable" || echo "not writable"
```

If either directory isn't writable, create the install directory and set your user as the owner:

```
sudo mkdir -p ~/.local/bin
sudo chown -R $(whoami) ~/.local
```

Verify the binary works

If `claude` is installed but crashes or hangs on startup, run these checks to narrow down the cause.

Confirm the binary exists and is executable:

```
ls -la $(which claude)
```

On Linux, check for missing shared libraries. If `ldd` shows missing libraries, you may need to install system packages. On Alpine Linux and other musl-based distributions, see Alpine Linux setup.

```
ldd $(which claude) | grep "not found"
```

Run a quick sanity check that the binary can execute:

```
claude --version
```

Common installation issues

These are the most frequently encountered installation problems and their solutions.

Install script returns HTML instead of a shell script

When running the install command, you may see one of these errors:

```
bash: line 1: syntax error near unexpected token `<'
bash: line 1: `<!DOCTYPE html>'
```

On PowerShell, the same problem appears as:

```
Invoke-Expression: Missing argument in parameter list.
```

This means the install URL returned an HTML page instead of the install script. If the HTML page says “App unavailable in region,” Claude Code is not available in your country. See supported countries.

Otherwise, this can happen due to network issues, regional routing, or a temporary service disruption.

Solutions:

1. **Use an alternative install method**:
   On macOS or Linux, install via Homebrew:

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

On Windows, install via WinGet:

```
winget install Anthropic.ClaudeCode
```

1. **Retry after a few minutes**: the issue is often temporary. Wait and try the original command again.

`command not found: claude` after installation

The install finished but `claude` doesn’t work. The exact error varies by platform:

Plat-form	Error message
macOS	`zsh: command not found: claude`

Platform	Error message
Linux	`bash: claude: command not found`
Windows CMD	`'claude' is not recognized as an internal or external command`
PowerShell	`claude : The term 'claude' is not recognized as the name of a cmdlet`

This means the install directory isn't in your shell's search path. See Verify your PATH for the fix on each platform.

`curl: (56) Failure writing output to destination`

The `curl ... | bash` command downloads the script and passes it directly to Bash for execution using a pipe (`|`). This error means the connection broke before the script finished downloading. Common causes include network interruptions, the download being blocked mid-stream, or system resource limits.

Solutions:

1. **Check network stability**: Claude Code binaries are hosted on Google Cloud Storage. Test that you can reach it:

```
curl -fsSL https://storage.googleapis.com -o /dev/null
```

If the command completes silently, your connection is fine and the issue is likely intermittent. Retry the install command. If you see an error, your network may be blocking the download.

1. **Try an alternative install method**:
   On macOS or Linux:

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

On Windows:

```
winget install Anthropic.ClaudeCode
```

TLS or SSL connection errors

Errors like `curl: (35) TLS connect error`, `schannel: next InitializeSecurityContext failed`, or PowerShell's `Could not establish trust relationship for the SSL/TLS secure channel` indicate TLS handshake failures.

Solutions:

1. **Update your system CA certificates**:
   On Ubuntu/Debian:

```
sudo apt-get update && sudo apt-get install ca-certificates
```

On macOS via Homebrew:

```
brew install ca-certificates
```

1. **On Windows, enable TLS 1.2** in PowerShell before running the installer:

```
[Net.ServicePointManager]::SecurityProtocol = [Net.SecurityProtocolType]::Tls12
irm https://claude.ai/install.ps1 | iex
```

1. **Check for proxy or firewall interference**: corporate proxies that perform TLS inspection can cause these errors, including `unable to get local issuer certificate`. Set `NODE_EXTRA_CA_CERTS` to your corporate CA certificate bundle:

```
export NODE_EXTRA_CA_CERTS=/path/to/corporate-ca.pem
```

Ask your IT team for the certificate file if you don't have it. You can also try on a direct connection to confirm the proxy is the cause.

`Failed to fetch version from storage.googleapis.com`

The installer couldn't reach the download server. This typically means `storage.googleapis.com` is blocked on your network.

Solutions:

1. **Test connectivity directly**:

```
curl -sI https://storage.googleapis.com
```

1. **If behind a proxy**, set `HTTPS_PROXY` so the installer can route through it. See proxy configuration for details.

```
export HTTPS_PROXY=http://proxy.example.com:8080
curl -fsSL https://claude.ai/install.sh | bash
```

1. **If on a restricted network**, try a different network or VPN, or use an alternative install method:

   On macOS or Linux:

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

On Windows:

```
winget install Anthropic.ClaudeCode
```

Windows: `irm` or `&&` not recognized

If you see `'irm' is not recognized` or `The token '&&' is not valid`, you're running the wrong command for your shell.

- `irm` **not recognized**: you're in CMD, not PowerShell. You have two options:

  Open PowerShell by searching for "PowerShell" in the Start menu, then run the original install command:

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

Or stay in CMD and use the CMD installer instead:

```
curl -fsSL https://claude.ai/install.cmd -o install.cmd && install.cmd && del
install.cmd
```

- `&&` **not valid**: you're in PowerShell but ran the CMD installer command. Use the PowerShell installer:

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

Install killed on low-memory Linux servers

If you see `Killed` during installation on a VPS or cloud instance:

```
Setting up Claude Code...
Installing Claude Code native build latest...
bash: line 142: 34803 Killed    "$binary_path" install ${TARGET:+"$TARGET"}
```

The Linux OOM killer terminated the process because the system ran out of memory. Claude Code requires at least 4 GB of available RAM.

Solutions:

1. **Add swap space** if your server has limited RAM. Swap uses disk space as overflow memory, letting the install complete even with low physical RAM.
   Create a 2 GB swap file and enable it:

```
sudo fallocate -l 2G /swapfile
sudo chmod 600 /swapfile
sudo mkswap /swapfile
sudo swapon /swapfile
```

Then retry the installation:

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

1. **Close other processes** to free memory before installing.
2. **Use a larger instance** if possible. Claude Code requires at least 4 GB of RAM.

Install hangs in Docker

When installing Claude Code in a Docker container, installing as root into `/` can cause hangs.

Solutions:

1. **Set a working directory** before running the installer. When run from `/`, the installer scans the entire filesystem, which causes excessive memory usage. Setting `WORKDIR` limits the scan to a small directory:

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

1. **Increase Docker memory limits** if using Docker Desktop:

```
docker build --memory=4g .
```

Windows: Claude Desktop overrides `claude` CLI command

If you installed an older version of Claude Desktop, it may register a `Claude.exe` in the `WindowsApps` directory that takes PATH priority over Claude Code CLI. Running `claude` opens the Desktop app instead of the CLI.

Update Claude Desktop to the latest version to fix this issue.

Windows: "Claude Code on Windows requires git-bash"

Claude Code on native Windows needs Git for Windows, which includes Git Bash.

If Git is not installed, download and install it from git-scm.com/downloads/win. During setup, select "Add to PATH." Restart your terminal after installing.

If Git is already installed but Claude Code still can't find it, set the path in your settings.json file:

```
{
  "env": {
    "CLAUDE_CODE_GIT_BASH_PATH": "C:\\Program Files\\Git\\bin\\bash.exe"
  }
}
```

If your Git is installed somewhere else, find the path by running `where.exe git` in PowerShell and use the `bin\bash.exe` path from that directory.

Linux: wrong binary variant installed (musl/glibc mismatch)

If you see errors about missing shared libraries like `libstdc++.so.6` or `libgcc_s.so.1` after installation, the installer may have downloaded the wrong binary variant for your system.

```
Error loading shared library libstdc++.so.6: No such file or directory
```

This can happen on glibc-based systems that have musl cross-compilation packages installed, causing the installer to misdetect the system as musl.

Solutions:

1. **Check which libc your system uses**:

```
ldd /bin/ls | head -1
```

If it shows `linux-vdso.so` or references to `/lib/x86_64-linux-gnu/`, you're on glibc. If it shows `musl`, you're on musl.

1. **If you're on glibc but got the musl binary**, remove the installation and reinstall. You can also manually download the correct binary from the GCS bucket at `https://storage.googleapis.com/claude-code-dist-86c565f3-f756-42ad-8dfa-d59b1c096819/claude-code-releases/{VERSION}/manifest.json`. File a GitHub issue with the output of `ldd /bin/ls` and `ls /lib/libc.musl*`.
2. **If you're actually on musl** (Alpine Linux), install the required packages:

```
apk add libgcc libstdc++ ripgrep
```

`Illegal instruction` on Linux

If the installer prints `Illegal instruction` instead of the OOM `Killed` message, the downloaded binary doesn't match your CPU architecture. This commonly happens on ARM servers that receive an x86 binary, or on older CPUs that lack required instruction sets.

```
bash: line 142: 2238232 Illegal instruction    "$binary_path" install ${TARGET:
+"$TARGET"}
```

Solutions:

1. **Verify your architecture**:

```
uname -m
```

`x86_64` means 64-bit Intel/AMD, `aarch64` means ARM64. If the binary doesn't match, file a GitHub issue with the output.

1. **Try an alternative install method** while the architecture issue is resolved:

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

`dyld: cannot load` on macOS

If you see `dyld: cannot load` or `Abort trap: 6` during installation, the binary is incompatible with your macOS version or hardware.

```
dyld: cannot load 'claude-2.1.42-darwin-x64' (load command 0x80000034 is unknown)
Abort trap: 6
```

Solutions:

1. **Check your macOS version**: Claude Code requires macOS 13.0 or later. Open the Apple menu and select About This Mac to check your version.
2. **Update macOS** if you're on an older version. The binary uses load commands that older macOS versions don't support.
3. **Try Homebrew** as an alternative install method:

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

Windows installation issues: errors in WSL

You might encounter the following issues in WSL:

OS/platform detection issues: if you receive an error during installation, WSL may be using Windows `npm`. Try:

- Run `npm config set os linux` before installation

- Install with `npm install -g @anthropic-ai/claude-code --force --no-os-check`. Do not use `sudo`.

Node not found errors: if you see `exec: node: not found` when running `claude`, your WSL environment may be using a Windows installation of Node.js. You can confirm this with `which npm` and `which node`, which should point to Linux paths starting with `/usr/` rather than `/mnt/c/`. To fix this, try installing Node via your Linux distribution's package manager or via `nvm`.

nvm version conflicts: if you have nvm installed in both WSL and Windows, you may experience version conflicts when switching Node versions in WSL. This happens because WSL imports the Windows PATH by default, causing Windows nvm/npm to take priority over the WSL installation.

You can identify this issue by:

- Running `which npm` and `which node` - if they point to Windows paths (starting with `/mnt/c/`), Windows versions are being used
- Experiencing broken functionality after switching Node versions with nvm in WSL

To resolve this issue, fix your Linux PATH to ensure the Linux node/npm versions take priority:

Primary solution: Ensure nvm is properly loaded in your shell

The most common cause is that nvm isn't loaded in non-interactive shells. Add the following to your shell configuration file (`~/.bashrc`, `~/.zshrc`, etc.):

```
## Load nvm if it exists
export NVM_DIR="$HOME/.nvm"
[ -s "$NVM_DIR/nvm.sh" ] && \. "$NVM_DIR/nvm.sh"
[ -s "$NVM_DIR/bash_completion" ] && \. "$NVM_DIR/bash_completion"
```

Or run directly in your current session:

```
source ~/.nvm/nvm.sh
```

Alternative: Adjust PATH order

If nvm is properly loaded but Windows paths still take priority, you can explicitly prepend your Linux paths to PATH in your shell configuration:

```
export PATH="$HOME/.nvm/versions/node/$(node -v)/bin:$PATH"
```

> **Warning:**
>
> Avoid disabling Windows PATH importing via `appendWindowsPath = false` as this breaks the ability to call Windows executables from WSL. Similarly, avoid uninstalling Node.js from Windows if you use it for Windows development.

WSL2 sandbox setup

Sandboxing is supported on WSL2 but requires installing additional packages. If you see an error like “Sandbox requires socat and bubblewrap” when running `/sandbox`, install the dependencies:

Ubuntu/Debian

```
sudo apt-get install bubblewrap socat
```

Fedora

```
sudo dnf install bubblewrap socat
```

WSL1 does not support sandboxing. If you see “Sandboxing requires WSL2”, you need to upgrade to WSL2 or run Claude Code without sandboxing.

Permission errors during installation

If the native installer fails with permission errors, the target directory may not be writable. See Check directory permissions.

If you previously installed with npm and are hitting npm-specific permission errors, switch to the native installer:

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

Permissions and authentication

These sections address login failures, token issues, and permission prompt behavior.

Repeated permission prompts

If you find yourself repeatedly approving the same commands, you can allow specific tools to run without approval using the `/permissions` command. See Permissions docs.

Authentication issues

If you're experiencing authentication problems:

1. Run `/logout` to sign out completely
2. Close Claude Code
3. Restart with `claude` and complete the authentication process again

If the browser doesn't open automatically during login, press `c` to copy the OAuth URL to your clipboard, then paste it into your browser manually.

OAuth error: Invalid code

If you see `OAuth error: Invalid code. Please make sure the full code was copied`, the login code expired or was truncated during copy-paste.

Solutions:

- Press Enter to retry and complete the login quickly after the browser opens
- Type `c` to copy the full URL if the browser doesn't open automatically
- If using a remote/SSH session, the browser may open on the wrong machine. Copy the URL displayed in the terminal and open it in your local browser instead.

403 Forbidden after login

If you see `API Error: 403 {"error":{"type":"forbidden","message":"Request not allowed"}}` after logging in:

- **Claude Pro/Max users**: verify your subscription is active at claude.ai/settings
- **Console users**: confirm your account has the "Claude Code" or "Developer" role assigned by your admin
- **Behind a proxy**: corporate proxies can interfere with API requests. See network configuration for proxy setup.

OAuth login fails in WSL2

Browser-based login in WSL2 may fail if WSL can't open your Windows browser. Set the `BROWSER` environment variable:

```
export BROWSER="/mnt/c/Program Files/Google/Chrome/Application/chrome.exe"
claude
```

Or copy the URL manually: when the login prompt appears, press `c` to copy the OAuth URL, then paste it into your Windows browser.

"Not logged in" or token expired

If Claude Code prompts you to log in again after a session, your OAuth token may have expired.

Run `/login` to re-authenticate. If this happens frequently, check that your system clock is accurate, as token validation depends on correct timestamps.

Configuration file locations

Claude Code stores configuration in several locations:

File	Purpose
`~/.claude/settings.json`	User settings (permissions, hooks, model overrides)
`.claude/settings.json`	Project settings (checked into source control)
`.claude/settings.local.json`	Local project settings (not committed)
`~/.claude.json`	Global state (theme, OAuth, MCP servers)
`.mcp.json`	Project MCP servers (checked into source control)
`managed-mcp.json`	Managed MCP servers
Managed settings	Managed settings (server-managed, MDM/OS-level policies, or file-based)

On Windows, `~` refers to your user home directory, such as `C:\Users\YourName`.

For details on configuring these files, see Settings and MCP.

Resetting configuration

To reset Claude Code to default settings, you can remove the configuration files:

```
## Reset all user settings and state
rm ~/.claude.json
rm -rf ~/.claude/

## Reset project-specific settings
rm -rf .claude/
rm .mcp.json
```

> **Warning:**
>
> This will remove all your settings, MCP server configurations, and session history.

Performance and stability

These sections cover issues related to resource usage, responsiveness, and search behavior.

High CPU or memory usage

Claude Code is designed to work with most development environments, but may consume significant resources when processing large codebases. If you're experiencing performance issues:

1. Use `/compact` regularly to reduce context size
2. Close and restart Claude Code between major tasks
3. Consider adding large build directories to your `.gitignore` file

Command hangs or freezes

If Claude Code seems unresponsive:

1. Press Ctrl+C to attempt to cancel the current operation
2. If unresponsive, you may need to close the terminal and restart

Search and discovery issues

If Search tool, `@file` mentions, custom agents, and custom skills aren't working, install system `ripgrep`:

```
## macOS (Homebrew)
brew install ripgrep

## Windows (winget)
winget install BurntSushi.ripgrep.MSVC

## Ubuntu/Debian
sudo apt install ripgrep

## Alpine Linux
apk add ripgrep

## Arch Linux
pacman -S ripgrep
```

Then set `USE_BUILTIN_RIPGREP=0` in your environment.

Slow or incomplete search results on WSL

Disk read performance penalties when working across file systems on WSL may result in fewer-than-expected matches when using Claude Code on WSL. Search still functions, but returns fewer results than on a native filesystem.

> **Note:**
>
> `/doctor` will show Search as OK in this case.

Solutions:

1. **Submit more specific searches**: reduce the number of files searched by specifying directories or file types: "Search for JWT validation logic in the auth-service package" or "Find use of md5 hash in JS files".
2. **Move project to Linux filesystem**: if possible, ensure your project is located on the Linux filesystem (`/home/`) rather than the Windows filesystem (`/mnt/c/`).
3. **Use native Windows instead**: consider running Claude Code natively on Windows instead of through WSL, for better file system performance.

IDE integration issues

If Claude Code does not connect to your IDE or behaves unexpectedly within an IDE terminal, try the solutions below.

JetBrains IDE not detected on WSL2

If you're using Claude Code on WSL2 with JetBrains IDEs and getting "No available IDEs detected" errors, this is likely due to WSL2's networking configuration or Windows Firewall blocking the connection.

WSL2 networking modes

WSL2 uses NAT networking by default, which can prevent IDE detection. You have two options:

Option 1: Configure Windows Firewall (recommended)

1. Find your WSL2 IP address:

```
wsl hostname -I
## Example output: 172.21.123.45
```

1. Open PowerShell as Administrator and create a firewall rule:

```
New-NetFirewallRule -DisplayName "Allow WSL2 Internal Traffic" -Direction Inbound
-Protocol TCP -Action Allow -RemoteAddress 172.21.0.0/16 -LocalAddress 172.21.0.0/
16
```

Adjust the IP range based on your WSL2 subnet from step 1.

1. Restart both your IDE and Claude Code

Option 2: Switch to mirrored networking

Add to `.wslconfig` in your Windows user directory:

```
[wsl2]
networkingMode=mirrored
```

Then restart WSL with `wsl --shutdown` from PowerShell.

> **Note:**
>
> These networking issues only affect WSL2. WSL1 uses the host's network directly and doesn't require these configurations.

For additional JetBrains configuration tips, see the JetBrains IDE guide.

Report Windows IDE integration issues

If you're experiencing IDE integration problems on Windows, create an issue with the following information:

- Environment type: native Windows (Git Bash) or WSL1/WSL2
- WSL networking mode, if applicable: NAT or mirrored
- IDE name and version
- Claude Code extension/plugin version
- Shell type: Bash, Zsh, PowerShell, etc.

Escape key not working in JetBrains IDE terminals

If you're using Claude Code in JetBrains terminals and the `Esc` key doesn't interrupt the agent as expected, this is likely due to a keybinding clash with JetBrains' default shortcuts.

To fix this issue:

1. Go to Settings → Tools → Terminal
2. Either:
   - Uncheck "Move focus to the editor with Escape", or
   - Click "Configure terminal keybindings" and delete the "Switch focus to Editor" shortcut
3. Apply the changes

This allows the `Esc` key to properly interrupt Claude Code operations.

Markdown formatting issues

Claude Code sometimes generates markdown files with missing language tags on code fences, which can affect syntax highlighting and readability in GitHub, editors, and documentation tools.

Missing language tags in code blocks

If you notice code blocks like this in generated markdown:

````
```
function example() {
  return "hello";
}
```text
````

Instead of properly tagged blocks like:

````
```javascript
function example() {
  return "hello";
}
```text
````

Solutions:

1. **Ask Claude to add language tags**: request "Add appropriate language tags to all code blocks in this markdown file."
2. **Use post-processing hooks**: set up automatic formatting hooks to detect and add missing language tags. See Auto-format code after edits for an example of a PostToolUse formatting hook.
3. **Manual verification**: after generating markdown files, review them for proper code block formatting and request corrections if needed.

Inconsistent spacing and formatting

If generated markdown has excessive blank lines or inconsistent spacing:

Solutions:

1. **Request formatting corrections**: ask Claude to "Fix spacing and formatting issues in this markdown file."
2. **Use formatting tools**: set up hooks to run markdown formatters like `prettier` or custom formatting scripts on generated markdown files.
3. **Specify formatting preferences**: include formatting requirements in your prompts or project memory files.

Reduce markdown formatting issues

To minimize formatting issues:

- **Be explicit in requests**: ask for "properly formatted markdown with language-tagged code blocks"
- **Use project conventions**: document your preferred markdown style in `CLAUDE.md`
- **Set up validation hooks**: use post-processing hooks to automatically verify and fix common formatting issues

Get more help

If you're experiencing issues not covered here:

1. Use the `/bug` command within Claude Code to report problems directly to Anthropic
2. Check the GitHub repository for known issues
3. Run `/doctor` to diagnose issues. It checks:
    - Installation type, version, and search functionality
    - Auto-update status and available versions
    - Invalid settings files (malformed JSON, incorrect types)
    - MCP server configuration errors
    - Keybinding configuration problems
    - Context usage warnings (large CLAUDE.md files, high MCP token usage, unreachable permission rules)
    - Plugin and agent loading errors
4. Ask Claude directly about its capabilities and features - Claude has built-in access to its documentation

Changelog

> *Release notes for Claude Code, including new features, improvements, and bug fixes by version.*

This page is generated from the CHANGELOG.md on GitHub.

Run `claude --version` to check your installed version.

- Increased default maximum output token limits for Claude Opus 4.6 to 64k tokens, and the upper bound for Opus 4.6 and Sonnet 4.6 models to 128k tokens
- Added `allowRead` sandbox filesystem setting to re-allow read access within `denyRead` regions
- `/copy` now accepts an optional index: `/copy N` copies the Nth-latest assistant response
- Fixed "Always Allow" on compound bash commands (e.g. `cd src && npm test`) saving a single rule for the full string instead of per-subcommand, leading to dead rules and repeated permission prompts
- Fixed auto-updater starting overlapping binary downloads when the slash-command overlay repeatedly opened and closed, accumulating tens of gigabytes of memory
- Fixed `--resume` silently truncating recent conversation history due to a race between memory-extraction writes and the main transcript
- Fixed PreToolUse hooks returning `"allow"` bypassing `deny` permission rules, including enterprise managed settings
- Fixed Write tool silently converting line endings when overwriting CRLF files or creating files in CRLF directories
- Fixed memory growth in long-running sessions from progress messages surviving compaction
- Fixed cost and token usage not being tracked when the API falls back to non-streaming mode
- Fixed `CLAUDE_CODE_DISABLE_EXPERIMENTAL_BETAS` not stripping beta tool-schema fields, causing proxy gateways to reject requests
- Fixed Bash tool reporting errors for successful commands when the system temp directory path contains spaces
- Fixed paste being lost when typing immediately after pasting

- Fixed Ctrl+D in `/feedback` text input deleting forward instead of the second press exiting the session
- Fixed API error when dragging a 0-byte image file into the prompt
- Fixed Claude Desktop sessions incorrectly using the terminal CLI's configured API key instead of OAuth
- Fixed `git-subdir` plugins at different subdirectories of the same monorepo commit colliding in the plugin cache
- Fixed ordered list numbers not rendering in terminal UI
- Fixed a race condition where stale-worktree cleanup could delete an agent worktree just resumed from a previous crash
- Fixed input deadlock when opening `/mcp` or similar dialogs while the agent is running
- Fixed Backspace and Delete keys not working in vim NORMAL mode
- Fixed status line not updating when vim mode is toggled on or off
- Fixed hyperlinks opening twice on Cmd+click in VS Code, Cursor, and other xterm.js-based terminals
- Fixed background colors rendering as terminal-default inside tmux with default configuration
- Fixed iTerm2 session crash when selecting text inside tmux over SSH
- Fixed clipboard copy silently failing in tmux sessions; copy toast now indicates whether to paste with `⌘V` or tmux `prefix+]`
- Fixed `←` / `→` accidentally switching tabs in settings, permissions, and sandbox dialogs while navigating lists
- Fixed IDE integration not auto-connecting when Claude Code is launched inside tmux or screen
- Fixed CJK characters visually bleeding into adjacent UI elements when clipped at the right edge
- Fixed teammate panes not closing when the leader exits
- Fixed iTerm2 auto mode not detecting iTerm2 for native split-pane teammates
- Faster startup on macOS (~60ms) by reading keychain credentials in parallel with module loading
- Faster `--resume` on fork-heavy and very large sessions — up to 45% faster loading and ~100-150MB less peak memory
- Improved Esc to abort in-flight non-streaming API requests
- Improved `claude plugin validate` to check skill, agent, and command frontmatter plus `hooks/hooks.json`, catching YAML parse errors and schema violations

- Background bash tasks are now killed if output exceeds 5GB, preventing runaway processes from filling disk
- Sessions are now auto-named from plan content when you accept a plan
- Improved headless mode plugin installation to compose correctly with `CLAUDE_CODE_PLUGIN_SEED_DIR`
- Show a notice when `apiKeyHelper` takes longer than 10s, preventing it from blocking the main loop
- The Agent tool no longer accepts a `resume` parameter — use `SendMessage({to: agentId})` to continue a previously spawned agent
- `SendMessage` now auto-resumes stopped agents in the background instead of returning an error
- Renamed `/fork` to `/branch` (`/fork` still works as an alias)
- [VSCode] Improved plan preview tab titles to use the plan's heading instead of "Claude's Plan"
- [VSCode] When option+click doesn't trigger native selection on macOS, the footer now points to the `macOptionClickForcesSelection` setting
- Added MCP elicitation support — MCP servers can now request structured input mid-task via an interactive dialog (form fields or browser URL)
- Added new `Elicitation` and `ElicitationResult` hooks to intercept and override responses before they're sent back
- Added `-n` / `--name <name>` CLI flag to set a display name for the session at startup
- Added `worktree.sparsePaths` setting for `claude --worktree` in large monorepos to check out only the directories you need via git sparse-checkout
- Added `PostCompact` hook that fires after compaction completes
- Added `/effort` slash command to set model effort level
- Added session quality survey — enterprise admins can configure the sample rate via the `feedbackSurveyRate` setting
- Fixed deferred tools (loaded via `ToolSearch`) losing their input schemas after conversation compaction, causing array and number parameters to be rejected with type errors
- Fixed slash commands showing "Unknown skill"
- Fixed plan mode asking for re-approval after the plan was already accepted
- Fixed voice mode swallowing keypresses while a permission dialog or plan editor was open
- Fixed `/voice` not working on Windows when installed via npm

- Fixed spurious “Context limit reached” when invoking a skill with `model:` frontmatter on a 1M-context session
- Fixed “adaptive thinking is not supported on this model” error when using non-standard model strings
- Fixed `Bash(cmd:*)` permission rules not matching when a quoted argument contains `#`
- Fixed “don’t ask again” in the Bash permission dialog showing the full raw command for pipes and compound commands
- Fixed auto-compaction retrying indefinitely after consecutive failures — a circuit breaker now stops after 3 attempts
- Fixed MCP reconnect spinner persisting after successful reconnection
- Fixed LSP plugins not registering servers when the LSP Manager initialized before marketplaces were reconciled
- Fixed clipboard copying in tmux over SSH — now attempts both direct terminal write and tmux clipboard integration
- Fixed `/export` showing only the filename instead of the full file path in the success message
- Fixed transcript not auto-scrolling to new messages after selecting text
- Fixed Escape key not working to exit the login method selection screen
- Fixed several Remote Control issues: sessions silently dying when the server reaps an idle environment, rapid messages being queued one-at-a-time instead of batched, and stale work items causing redelivery after JWT refresh
- Fixed bridge sessions failing to recover after extended WebSocket disconnects
- Fixed slash commands not found when typing the exact name of a soft-hidden command
- Improved `--worktree` startup performance by reading git refs directly and skipping redundant `git fetch` when the remote branch is already available locally
- Improved background agent behavior — killing a background agent now preserves its partial results in the conversation context
- Improved model fallback notifications — now always visible instead of hidden behind verbose mode, with human-friendly model names
- Improved blockquote readability on dark terminal themes — text is now italic with a left bar instead of dim
- Improved stale worktree cleanup — worktrees left behind after an interrupted parallel run are now automatically cleaned up

- Improved Remote Control session titles — now derived from your first prompt instead of showing "Interactive session"
- Improved `/voice` to show your dictation language on enable and warn when your `language` setting isn't supported for voice input
- Updated `--plugin-dir` to only accept one path to support subcommands — use repeated `--plugin-dir` for multiple directories
- [VSCode] Fixed gitignore patterns containing commas silently excluding entire filetypes from the @-mention file picker
- Added 1M context window for Opus 4.6 by default for Max, Team, and Enterprise plans (previously required extra usage)
- Added `/color` command for all users to set a prompt-bar color for your session
- Added session name display on the prompt bar when using `/rename`
- Added last-modified timestamps to memory files, helping Claude reason about which memories are fresh vs. stale
- Added hook source display (settings/plugin/skill) in permission prompts when a hook requires confirmation
- Fixed voice mode not activating correctly on fresh installs without toggling `/voice` twice
- Fixed the Claude Code header not updating the displayed model name after switching models with `/model` or Option+P
- Fixed session crash when an attachment message computation returns undefined values
- Fixed Bash tool mangling `!` in piped commands (e.g., `jq 'select(.x != .y)'` now works correctly)
- Fixed managed-disabled plugins showing up in the `/plugin` Installed tab — plugins force-disabled by your organization are now hidden
- Fixed token estimation over-counting for thinking and `tool_use` blocks, preventing premature context compaction
- Fixed corrupted marketplace config path handling
- Fixed `/resume` losing session names after resuming a forked or continued session
- Fixed Esc not closing the `/status` dialog after visiting the Config tab
- Fixed input handling when accepting or rejecting a plan
- Fixed footer hint in agent teams showing "↓ to expand" instead of the correct "shift + ↓ to expand"

- Improved startup performance on macOS non-MDM machines by skipping unnecessary subprocess spawns
- Suppressed async hook completion messages by default (visible with `--verbose` or transcript mode)
- Breaking change: Removed deprecated Windows managed settings fallback at `C:\ProgramData\ClaudeCode\managed-settings.json` — use `C:\Program Files\ClaudeCode\managed-settings.json`
- Added actionable suggestions to `/context` command — identifies context-heavy tools, memory bloat, and capacity warnings with specific optimization tips
- Added `autoMemoryDirectory` setting to configure a custom directory for auto-memory storage
- Fixed memory leak where streaming API response buffers were not released when the generator was terminated early, causing unbounded RSS growth on the Node.js/npm code path
- Fixed managed policy `ask` rules being bypassed by user `allow` rules or skill `allowed-tools`
- Fixed full model IDs (e.g., `claude-opus-4-5`) being silently ignored in agent frontmatter `model:` field and `--agents` JSON config — agents now accept the same model values as `--model`
- Fixed MCP OAuth authentication hanging when the callback port is already in use
- Fixed MCP OAuth refresh never prompting for re-auth after the refresh token expires, for OAuth servers that return errors with HTTP 200 (e.g. Slack)
- Fixed voice mode silently failing on the macOS native binary for users whose terminal had never been granted microphone permission — the binary now includes the `audio-input` entitlement so macOS prompts correctly
- Fixed `SessionEnd` hooks being killed after 1.5 s on exit regardless of `hook.timeout` — now configurable via `CLAUDE_CODE_SESSIONEND_HOOKS_TIMEOUT_MS`
- Fixed `/plugin install` failing inside the REPL for marketplace plugins with local sources
- Fixed marketplace update not syncing git submodules — plugin sources in submodules no longer break after update
- Fixed unknown slash commands with arguments silently dropping input — now shows your input as a warning
- Fixed Hebrew, Arabic, and other RTL text not rendering correctly in Windows Terminal, conhost, and VS Code integrated terminal
- Fixed LSP servers not working on Windows due to malformed file URIs

- Changed `--plugin-dir` so local dev copies now override installed marketplace plugins with the same name (unless that plugin is force-enabled by managed settings)
- [VSCode] Fixed delete button not working for Untitled sessions
- [VSCode] Improved scroll wheel responsiveness in the integrated terminal with terminal-aware acceleration
- Added `modelOverrides` setting to map model picker entries to custom provider model IDs (e.g. Bedrock inference profile ARNs)
- Added actionable guidance when OAuth login or connectivity checks fail due to SSL certificate errors (corporate proxies, `NODE_EXTRA_CA_CERTS`)
- Fixed freezes and 100% CPU loops triggered by permission prompts for complex bash commands
- Fixed a deadlock that could freeze Claude Code when many skill files changed at once (e.g. during `git pull` in a repo with a large `.claude/skills/` directory)
- Fixed Bash tool output being lost when running multiple Claude Code sessions in the same project directory
- Fixed subagents with `model: opus` / `sonnet` / `haiku` being silently downgraded to older model versions on Bedrock, Vertex, and Microsoft Foundry
- Fixed background bash processes spawned by subagents not being cleaned up when the agent exits
- Fixed `/resume` showing the current session in the picker
- Fixed `/ide` crashing with `onInstall is not defined` when auto-installing the extension
- Fixed `/loop` not being available on Bedrock/Vertex/Foundry and when telemetry was disabled
- Fixed SessionStart hooks firing twice when resuming a session via `--resume` or `--continue`
- Fixed JSON-output hooks injecting no-op system-reminder messages into the model's context on every turn
- Fixed voice mode session corruption when a slow connection overlaps a new recording
- Fixed Linux sandbox failing to start with "ripgrep (rg) not found" on native builds
- Fixed Linux native modules not loading on Amazon Linux 2 and other glibc 2.26 systems
- Fixed "media_type: Field required" API error when receiving images via Remote Control

- Fixed `/heapdump` failing on Windows with `EEXIST` error when the Desktop folder already exists
- Improved Up arrow after interrupting Claude — now restores the interrupted prompt and rewinds the conversation in one step
- Improved IDE detection speed at startup
- Improved clipboard image pasting performance on macOS
- Improved `/effort` to work while Claude is responding, matching `/model` behavior
- Improved voice mode to automatically retry transient connection failures during rapid push-to-talk re-press
- Improved the Remote Control spawn mode selection prompt with better context
- Changed default Opus model on Bedrock, Vertex, and Microsoft Foundry to Opus 4.6 (was Opus 4.1)
- Deprecated `/output-style` command — use `/config` instead. Output style is now fixed at session start for better prompt caching
- VSCode: Fixed HTTP 400 errors for users behind proxies or on Bedrock/Vertex with Claude 4.5 models
- Fixed tool search to activate even with `ANTHROPIC_BASE_URL` as long as `ENABLE_TOOL_SEARCH` is set.
- Added `w` key in `/copy` to write the focused selection directly to a file, bypassing the clipboard (useful over SSH)
- Added optional description argument to `/plan` (e.g., `/plan fix the auth bug`) that enters plan mode and immediately starts
- Added `ExitWorktree` tool to leave an `EnterWorktree` session
- Added `CLAUDE_CODE_DISABLE_CRON` environment variable to immediately stop scheduled cron jobs mid-session
- Added `lsof`, `pgrep`, `tput`, `ss`, `fd`, and `fdfind` to the bash auto-approval allowlist, reducing permission prompts for common read-only operations
- Restored the `model` parameter on the Agent tool for per-invocation model overrides
- Simplified effort levels to low/medium/high (removed max) with new symbols (○ ◐ ●) and a brief notification instead of a persistent icon. Use `/effort auto` to reset to default
- Improved `/config` — Escape now cancels changes, Enter saves and closes, Space toggles settings
- Improved up-arrow history to show current session's messages first when running multiple concurrent sessions

- Improved voice input transcription accuracy for repo names and common dev terms (regex, OAuth, JSON)
- Improved bash command parsing by switching to a native module — faster initialization and no memory leak
- Reduced bundle size by ~510 KB
- Changed CLAUDE.md HTML comments (``) to be hidden from Claude when auto-injected. Comments remain visible when read with the Read tool
- Fixed slow exits when background tasks or hooks were slow to respond
- Fixed agent task progress stuck on "Initializing…"
- Fixed skill hooks firing twice per event when a hooks-enabled skill is invoked by the model
- Fixed several voice mode issues: occasional input lag, false "No speech detected" errors after releasing push-to-talk, and stale transcripts re-filling the prompt after submission
- Fixed `--continue` not resuming from the most recent point after `--compact`
- Fixed bash security parsing edge cases
- Added support for marketplace git URLs without `.git` suffix (Azure DevOps, AWS CodeCommit)
- Improved marketplace clone failure messages to show diagnostic info even when git produces no stderr
- Fixed several plugin issues: installation failing on Windows with `EEXIST` error in OneDrive folders, marketplace blocking user-scope installs when a project-scope install exists, `CLAUDE_CODE_PLUGIN_CACHE_DIR` creating literal `~` directories, and `plugin.json` with marketplace-only fields failing to load
- Fixed feedback survey appearing too frequently in long sessions
- Fixed `--effort` CLI flag being reset by unrelated settings writes on startup
- Fixed backgrounded Ctrl+B queries losing their transcript or corrupting the new conversation after `/clear`
- Fixed `/clear` killing background agent/bash tasks — only foreground tasks are now cleared
- Fixed worktree isolation issues: Task tool resume not restoring cwd, and background task notifications missing `worktreePath` and `worktreeBranch`
- Fixed `/model` not displaying results when run while Claude is working
- Fixed digit keys selecting menu options instead of typing in plan mode permission prompt's text input

- Fixed sandbox permission issues: certain file write operations incorrectly allowed without prompting, and output redirections to allowlisted directories (like `/tmp/claude/`) prompting unnecessarily
- Improved CPU utilization in long sessions
- Fixed prompt cache invalidation in SDK `query()` calls, reducing input token costs up to 12x
- Fixed Escape key becoming unresponsive after cancelling a query
- Fixed double Ctrl+C not exiting when background agents or tasks are running
- Fixed team agents to inherit the leader's model
- Fixed "Always Allow" saving permission rules that never match again
- Fixed several hooks issues: `transcript_path` pointing to the wrong directory for resumed/forked sessions, agent `prompt` being silently deleted from settings.json on every settings write, PostToolUse block reason displaying twice, async hooks not receiving stdin with bash `read -r`, and validation error message showing an example that fails validation
- Fixed session crashes in Desktop/SDK when Read returned files containing U+2028/U+2029 characters
- Fixed terminal title being cleared on exit even when `CLAUDE_CODE_DISABLE_TERMINAL_TITLE` was set
- Fixed several permission rule matching issues: wildcard rules not matching commands with heredocs, embedded newlines, or no arguments; `sandbox.excludedCommands` failing with env var prefixes; "always allow" suggesting overly broad prefixes for nested CLI tools; and deny rules not applying to all command forms
- Fixed oversized and truncated images from Bash data-URL output
- Fixed a crash when resuming sessions that contained Bedrock API errors
- Fixed intermittent "expected boolean, received string" validation errors on Edit, Bash, and Grep tool inputs
- Fixed multi-line session titles when forking from a conversation whose first message contained newlines
- Fixed queued messages not showing attached images, and images being lost when pressing ↑ to edit a queued message
- Fixed parallel tool calls where a failed Read/WebFetch/Glob would cancel its siblings — only Bash errors now cascade
- VSCode: Fixed scroll speed in integrated terminals not matching native terminals

Earlier changelog entries omitted. See full changelog for complete history.