Claude Code Documentation

> Original English documentation from code.claude.com/docs

Claude Code 概述

> ## Documentation Index
>
> Fetch the complete documentation index at: https://code.claude.com/docs/llms.txt
> Use this file to discover all available pages before exploring further.

Claude Code 概述

> 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.

开始使用

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.

````
 The full-featured CLI for working with Claude Code directly in your term

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



    **macOS, Linux, WSL:**

    ```bash  theme={null}
    curl -fsSL https://claude.ai/install.sh | bash
    ```

    **Windows PowerShell:**

    ```powershell  theme={null}
    irm https://claude.ai/install.ps1 | iex
    ```

    **Windows CMD:**

    ```batch  theme={null}
    curl -fsSL https://claude.ai/install.cmd -o install.cmd && install.cm
    ```

    If you see `The token '&&' is not a valid statement separator`, you'

    **Windows requires [Git for Windows](https://git-scm.com/downloads/wi


      Native installations automatically update in the background to keep




    ```bash  theme={null}
    brew install --cask claude-code
    ```
````

Homebrew installations do not auto-update. Run `brew upgrade claude

```powershell  theme={null}
winget install Anthropic.ClaudeCode
```

WinGet installations do not auto-update. Run `winget upgrade Anthro

Then start Claude Code in any project:

```bash  theme={null}
cd your-project
claude
```

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

See [advanced setup](/en/setup) for installation options, manual update

The VS Code extension provides inline diffs, @-mentions, plan review, and

* [Install for VS Code](vscode:extension/anthropic.claude-code)
* [Install for Cursor](cursor:extension/anthropic.claude-code)

```
Or search for "Claude Code" in the Extensions view (`Cmd+Shift+X` on Mac

[Get started with VS Code →](/en/vs-code#get-started)



A standalone app for running Claude Code outside your IDE or terminal. Re

Download and install:

* [macOS](https://claude.ai/api/desktop/darwin/universal/dmg/latest/redi
* [Windows](https://claude.ai/api/desktop/win32/x64/setup/latest/redirec
* [Windows ARM64](https://claude.ai/api/desktop/win32/arm64/setup/latest/

After installing, launch Claude, sign in, and click the **Code** tab to s

[Learn more about the desktop app →](/en/desktop-quickstart)



Run Claude Code in your browser with no local setup. Kick off long-runnin

Start coding at [claude.ai/code](https://claude.ai/code).

[Get started on the web →](/en/claude-code-on-the-web#getting-started)



A plugin for IntelliJ IDEA, PyCharm, WebStorm, and other JetBrains IDEs w

Install the [Claude Code plugin](https://plugins.jetbrains.com/plugin/27

[Get started with JetBrains →](/en/jetbrains)
```

您可以做什么

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

Claude Code handles the tedious tasks that eat up your day: writing test

```bash  theme={null}
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

For bugs, paste an error message or describe the symptom. Claude Code tra

Claude Code works directly with git. It stages changes, writes commit mes

```bash  theme={null}
claude "commit my changes with a descriptive message"
```

In CI, you can automate code review and issue triage with [GitHub Actions

The [Model Context Protocol (MCP)](/en/mcp) is an open standard for conne

[`CLAUDE.md`](/en/memory) is a markdown file you add to your project roo

Create [custom commands](/en/skills) to package repeatable workflows you

[Hooks](/en/hooks) let you run shell commands before or after Claude Cod

Spawn [multiple Claude Code agents](/en/sub-agents) that work on differe

````
For fully custom workflows, the [Agent SDK](https://platform.claude.com/d



Claude Code is composable and follows the Unix philosophy. Pipe logs into

```bash  theme={null}
# Analyze recent log output
tail -200 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 se
```

See the [CLI reference](/en/cli-reference) for the full set of commands a



Run Claude on a schedule to automate work that repeats: morning PR review

* [Cloud scheduled tasks](/en/web-scheduled-tasks) run on Anthropic-manag
* [Desktop scheduled tasks](/en/desktop#schedule-recurring-tasks) run on
* [`/loop`](/en/scheduled-tasks) repeats a prompt within a CLI session fo



Sessions aren't tied to a single surface. Move work between environments

* Step away from your desk and keep working from your phone or any browse
* Message [Dispatch](/en/desktop#sessions-from-dispatch) a task from you
* Kick off a long-running task on the [web](/en/claude-code-on-the-web) e
* Hand off a terminal session to the [Desktop app](/en/desktop) with `/de
* Route tasks from team chat: mention `@Claude` in [Slack](/en/slack) wit
````

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
Push events from Telegram, Discord, iMessage, or my own webhooks into a session	Channels
Start a task locally, continue on mobile	Web or Claude iOS app
Run Claude on a recurring schedule	Cloud scheduled tasks or Desktop scheduled tasks
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

下一步

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

快速入门

> ## Documentation Index
>
> Fetch the complete documentation index at: https://code.claude.com/docs/llms.txt
> Use this file to discover all available pages before exploring further.

快速入门

> Welcome to Claude Code!

```
export const InstallConfigurator = () => {
const TERM = {
mac: {
label: 'macOS / Linux',
cmd: 'curl -fsSL https://claude.ai/install.sh | bash'
},
win: {
label: 'Windows'
},
brew: {
label: 'Homebrew',
cmd: 'brew install --cask claude-code'
},
```

```
winget: {
label: 'WinGet',
cmd: 'winget install Anthropic.ClaudeCode'
}
};
const WIN_VARIANTS = {
ps: 'irm https://claude.ai/install.ps1 | iex',
cmd: 'curl -fsSL https://claude.ai/install.cmd -o install.cmd && install.cmd && del
install.cmd'
};
const TABS = [{
key: 'terminal',
label: 'Terminal'
}, {
key: 'desktop',
label: 'Desktop'
}, {
key: 'vscode',
label: 'VS Code'
}, {
key: 'jetbrains',
label: 'JetBrains'
}];
const ALT_TARGETS = {
desktop: {
name: 'Desktop',
installLabel: 'Download the app',
installHref: 'https://claude.com/download?
utm_source=claude_code&utm_medium=docs&utm_content=configurator_desktop_download',
guideHref: '/en/desktop-quickstart'
},
vscode: {
name: 'VS Code',
installLabel: 'Install from Marketplace',
installHref: 'https://marketplace.visualstudio.com/items?itemName=anthropic.claude-
code',
altCmd: 'code --install-extension anthropic.claude-code',
```

guideHref: '/en/vs-code'
},
jetbrains: {
name: 'JetBrains',
installLabel: 'Install from Marketplace',
installHref: 'https://plugins.jetbrains.com/plugin/27310-claude-code-beta-',
guideHref: '/en/jetbrains'
}
};
const PROVIDERS = [{
key: 'anthropic',
label: 'Anthropic'
}, {
key: 'bedrock',
label: 'Amazon Bedrock'
}, {
key: 'foundry',
label: 'Microsoft Foundry'
}, {
key: 'vertex',
label: 'Google Vertex AI'
}];
const PROVIDER_NOTICE = {
bedrock: <>
Configure your AWS account first. Running on Bedrock
requires model access enabled in the AWS console and IAM credentials.{' '}
Bedrock setup guide →
,
vertex: <>
Configure your GCP project first. Running on Vertex AI
requires the Vertex API enabled and a service account with the right
permissions.{' '}
Vertex setup guide →
,
foundry: <>
Configure your Azure resources first. Running on
Microsoft Foundry requires an Azure subscription with a Foundry resource

and model deployments provisioned.{' '}
Foundry setup guide →

};
const iconCheck = (size = 14) =>

```
;
```

const iconCopy = (size = 14) =>

```
;
```

const iconArrowRight = (size = 13) =>

```
;
```

const iconArrowUpRight = (size = 14) =>

```
;
```

const iconInfo = (size = 16) =>

```
;
```

const [target, setTarget] = useState('terminal');
const [team, setTeam] = useState(false);
const [provider, setProvider] = useState('anthropic');
const [pkg, setPkg] = useState(() => (/Win/).test(navigator.userAgent) ? 'win' : 'mac');
const [winCmd, setWinCmd] = useState(false);
const [copied, setCopied] = useState(null);
const copyTimer = useRef(null);
const handleCopy = async (text, key) => {
try {
await navigator.clipboard.writeText(text);
} catch {
const ta = document.createElement('textarea');
ta.value = text;
document.body.appendChild(ta);

```
ta.select();
document.execCommand('copy');
document.body.removeChild(ta);
}
clearTimeout(copyTimer.current);
setCopied(key);
copyTimer.current = setTimeout(() => setCopied(null), 1800);
};
const cardBodyCmd = (cmd, prompt) => {
const on = copied === 'term';
return
{prompt || '$'}
{cmd}
handleCopy(cmd, 'term')}>
{on ? iconCheck(13) : iconCopy(13)}
{on ? 'Copied' : 'Copy'}
```

```
    ;
```

```
};
const isWinInstaller = pkg === 'win';
const isWinPrompt = pkg === 'win' || pkg === 'winget';
const terminalCmd = isWinInstaller ? WIN_VARIANTS[winCmd ? 'cmd' : 'ps'] :
TERM[pkg].cmd;
const alt = ALT_TARGETS[target];
const showNotice = team && provider !== 'anthropic';
const STYLES = `
.cc-ic {
--ic-slate: #141413;
--ic-clay: #d97757;
--ic-clay-deep: #c6613f;
--ic-gray-000: #ffffff;
--ic-gray-150: #f0eee6;
--ic-gray-550: #73726c;
--ic-gray-700: #3d3d3a;
--ic-border-subtle: rgba(31, 30, 29, 0.08);
--ic-border-default: rgba(31, 30, 29, 0.15);
```

```
--ic-border-strong: rgba(31, 30, 29, 0.3);
--ic-font-mono: ui-monospace, SFMono-Regular, Menlo, Monaco, 'Courier New',
monospace;
font-family: 'Anthropic Sans', -apple-system, BlinkMacSystemFont, 'Segoe UI', sans-serif;
font-size: 14px; line-height: 1.5; color: var(--ic-slate);
margin: 8px 0 32px;
}
.dark .cc-ic {
--ic-slate: #f0eee6;
--ic-gray-000: #262624;
--ic-gray-150: #1f1e1d;
--ic-gray-550: #91908a;
--ic-gray-700: #bfbdb4;
--ic-border-subtle: rgba(240, 238, 230, 0.08);
--ic-border-default: rgba(240, 238, 230, 0.14);
--ic-border-strong: rgba(240, 238, 230, 0.28);
}
.dark .cc-ic-check { background: transparent; }
.dark .cc-ic-card { border: 0.5px solid var(--ic-border-subtle); }
.dark .cc-ic-p-pill.cc-ic-active { box-shadow: 0 1px 2px rgba(0, 0, 0, 0.3); }
.cc-ic , .cc-ic ::before, .cc-ic *::after { box-sizing: border-box; }
.cc-ic a { text-decoration: none; }
.cc-ic a:not([class]) { color: inherit; }
.cc-ic button { font-family: inherit; cursor: pointer; }

.cc-ic-tab-strip {
display: inline-flex; gap: 2px;
padding: 4px; background: var(--ic-gray-150);
border-radius: 10px; overflow-x: auto;
max-width: 100%;
}
.cc-ic-tab {
appearance: none; background: none; border: none;
padding: 10px 18px; font-size: 15px; font-weight: 430;
color: var(--ic-gray-550); border-radius: 7px;
white-space: nowrap;
transition: color 0.12s, background-color 0.12s;
}
```

```
.cc-ic-tab:hover { color: var(--ic-gray-700); }
.cc-ic-tab.cc-ic-active {
color: var(--ic-slate); font-weight: 500;
background: var(--ic-gray-000);
box-shadow: 0 1px 3px rgba(0, 0, 0, 0.08);
}
.dark .cc-ic-tab.cc-ic-active { box-shadow: 0 1px 3px rgba(0, 0, 0, 0.4); }

.cc-ic-team-wrap { padding: 16px 0 20px; }
.cc-ic-team-toggle {
display: flex; align-items: center; gap: 12px; font-family: inherit;
padding: 12px 16px; font-size: 14px; font-weight: 430;
color: var(--ic-gray-700); cursor: pointer; user-select: none;
width: fit-content; background: var(--ic-gray-150);
border: 0.5px solid var(--ic-border-subtle); border-radius: 8px;
transition: border-color 0.15s;
}
.cc-ic-team-toggle:hover { border-color: var(--ic-border-default); }
.cc-ic-team-toggle.cc-ic-checked {
background: rgba(217, 119, 87, 0.08);
border-color: rgba(217, 119, 87, 0.25);
}
.cc-ic-check {
width: 16px; height: 16px;
border: 1px solid var(--ic-border-strong); border-radius: 4px;
background: var(--ic-gray-000);
display: flex; align-items: center; justify-content: center;
flex-shrink: 0;
}
.cc-ic-check svg { color: #fff; display: none; }
.cc-ic-team-toggle.cc-ic-checked .cc-ic-check { background: var(--ic-clay-deep); border-
color: var(--ic-clay-deep); }
.cc-ic-team-toggle.cc-ic-checked .cc-ic-check svg { display: block; }

.cc-ic-team-reveal { display: flex; flex-direction: column; gap: 12px; margin-bottom:
16px; }
.cc-ic-sales {
display: flex; align-items: center; justify-content: space-between;
```

```
gap: 16px; padding: 14px 16px;
background: var(--ic-gray-000); border: 0.5px solid var(--ic-border-default);
border-radius: 8px; flex-wrap: wrap;
}
.cc-ic-sales-text { font-size: 13px; color: var(--ic-gray-700); line-height: 1.5; flex: 1; min-
width: 200px; }
.cc-ic-sales-text strong { font-weight: 550; color: var(--ic-slate); }
.cc-ic-sales-actions { display: flex; align-items: center; gap: 8px; flex-shrink: 0; }
.cc-ic-btn-clay {
display: inline-flex; align-items: center; gap: 8px;
background: var(--ic-clay-deep); color: #fff; border: none;
border-radius: 8px; padding: 8px 14px;
font-size: 13px; font-weight: 500;
transition: background-color 0.15s; white-space: nowrap;
}
.cc-ic-btn-clay:hover { background: var(--ic-clay); }
.cc-ic-btn-ghost {
display: inline-flex; align-items: center; gap: 8px;
background: transparent; color: var(--ic-gray-700);
border: 0.5px solid var(--ic-border-default);
border-radius: 8px; padding: 8px 14px;
font-size: 13px; font-weight: 500;
}
.cc-ic-btn-ghost:hover { background: rgba(0, 0, 0, 0.04); }

.cc-ic-provider-bar {
display: flex; align-items: center; gap: 12px;
padding: 14px 16px; background: var(--ic-gray-150);
border-radius: 8px; font-size: 13px; flex-wrap: wrap;
}
.cc-ic-provider-bar .cc-ic-label { color: var(--ic-gray-550); flex-shrink: 0; }
.cc-ic-provider-pills { display: flex; gap: 4px; flex-wrap: wrap; }
.cc-ic-p-pill {
appearance: none; border: none; background: transparent;
padding: 6px 12px; border-radius: 6px;
font-size: 13px; font-weight: 430; color: var(--ic-gray-700);
white-space: nowrap;
}
```

```
.cc-ic-p-pill:hover { background: rgba(0, 0, 0, 0.04); }
.cc-ic-p-pill.cc-ic-active {
background: var(--ic-gray-000); color: var(--ic-slate);
font-weight: 500; box-shadow: 0 1px 2px rgba(0, 0, 0, 0.05);
}
.cc-ic-provider-notice {
display: flex; padding: 16px 18px;
background: var(--ic-gray-000); border: 0.5px solid var(--ic-border-default);
border-radius: 8px; gap: 14px; align-items: flex-start;
}
.cc-ic-provider-notice > svg { color: var(--ic-gray-550); margin-top: 2px; flex-shrink: 0; }
.cc-ic-provider-notice-body { font-size: 14px; line-height: 1.55; color: var(--ic-gray-700); }
.cc-ic-provider-notice-body strong { font-weight: 550; color: var(--ic-slate); }
.cc-ic-provider-notice-body a { color: var(--ic-clay-deep); font-weight: 500; }
.cc-ic-provider-notice-body a:hover { text-decoration: underline; }

.cc-ic-card { background: #141413; border-radius: 12px; overflow: hidden; }
.cc-ic-subtabs {
display: flex; align-items: center;
background: #1a1918;
border-bottom: 0.5px solid rgba(255, 255, 255, 0.08);
padding: 0 8px; overflow-x: auto;
}
.cc-ic-subtab-spacer { flex: 1; }
.cc-ic-subtab {
appearance: none; background: none; border: none;
padding: 12px 16px; font-size: 12px;
color: rgba(255, 255, 255, 0.5);
position: relative; white-space: nowrap;
}
.cc-ic-subtab:hover { color: rgba(255, 255, 255, 0.75); }
.cc-ic-subtab.cc-ic-active { color: #fff; }
.cc-ic-subtab.cc-ic-active::after {
content: ''; position: absolute;
left: 12px; right: 12px; bottom: -0.5px;
height: 2px; background: var(--ic-clay);
}
.cc-ic-cmd-toggle {
```

```
display: flex; align-items: center; gap: 8px; font-family: inherit;
background: none; border: none;
padding: 0 12px; font-size: 11px;
color: rgba(255, 255, 255, 0.5);
cursor: pointer; user-select: none; white-space: nowrap;
}
.cc-ic-cmd-toggle:hover { color: rgba(255, 255, 255, 0.75); }
.cc-ic-mini-check {
width: 12px; height: 12px;
border: 1px solid rgba(255, 255, 255, 0.3); border-radius: 3px;
display: flex; align-items: center; justify-content: center;
flex-shrink: 0;
}
.cc-ic-mini-check svg { color: #fff; display: none; }
.cc-ic-cmd-toggle.cc-ic-checked .cc-ic-mini-check { background: var(--ic-clay-deep);
border-color: var(--ic-clay-deep); }
.cc-ic-cmd-toggle.cc-ic-checked .cc-ic-mini-check svg { display: block; }

.cc-ic-card-body { padding: 24px 26px; display: flex; align-items: flex-start; gap: 14px; }
.cc-ic-prompt {
color: var(--ic-clay); font-family: var(--ic-font-mono);
font-size: 17px; user-select: none; padding-top: 2px;
}
.cc-ic-cmd {
flex: 1; font-family: var(--ic-font-mono);
font-size: 17px; color: #f0eee6;
line-height: 1.55; white-space: pre-wrap; word-break: break-word;
}
.cc-ic-copy {
display: inline-flex; align-items: center; gap: 6px;
background: rgba(255, 255, 255, 0.08);
border: 0.5px solid rgba(255, 255, 255, 0.12);
color: rgba(255, 255, 255, 0.85);
padding: 7px 13px; border-radius: 8px;
font-size: 13px; font-weight: 500; flex-shrink: 0;
}
.cc-ic-copy:hover { background: rgba(255, 255, 255, 0.14); }
```

```
.cc-ic-copy.cc-ic-copied { background: var(--ic-clay-deep); border-color: var(--ic-clay-deep); color: #fff; }

.cc-ic-below {
margin-top: 12px; font-size: 13px; color: var(--ic-gray-550);
display: flex; gap: 16px; flex-wrap: wrap; align-items: baseline;
}
.cc-ic-below a { color: var(--ic-gray-700); border-bottom: 0.5px solid var(--ic-border-default); }
.cc-ic-below a:hover { color: var(--ic-clay-deep); border-bottom-color: var(--ic-clay-deep); }
.cc-ic-handoff {
padding: 20px 22px;
background: var(--ic-gray-000);
border: 0.5px solid var(--ic-border-default);
border-radius: 12px;
}
.cc-ic-handoff-head {
font-size: 14px; line-height: 1.55; color: var(--ic-gray-700);
margin-bottom: 14px;
}
.cc-ic-handoff-head strong { font-weight: 550; color: var(--ic-slate); }
.cc-ic-handoff-actions { display: flex; gap: 10px; flex-wrap: wrap; }
.cc-ic-handoff-alt {
margin-top: 12px; font-size: 12px; color: var(--ic-gray-550);
}
.cc-ic-handoff-alt code {
font-family: var(--ic-font-mono); font-size: 11px;
background: var(--ic-gray-150); padding: 2px 6px;
border-radius: 4px; color: var(--ic-gray-700);
}
.cc-ic-copy-sm {
appearance: none; border: none;
display: inline-flex; align-items: center; justify-content: center;
width: 22px; height: 22px;
margin-left: 4px; vertical-align: middle;
background: var(--ic-gray-150); color: var(--ic-gray-550);
border-radius: 4px;
```

```
transition: color 0.1s, background-color 0.1s;
}
.cc-ic-copy-sm:hover { color: var(--ic-gray-700); background: var(--ic-border-default); }
.cc-ic-copy-sm.cc-ic-copied { background: var(--ic-clay-deep); color: #fff; }

@media (max-width: 720px) {
.cc-ic-tab { padding: 12px 14px; font-size: 14px; }
.cc-ic-sales-actions { width: 100%; }
.cc-ic-card-body { padding: 20px; }
.cc-ic-cmd { font-size: 15px; }
}
`;
return
{STYLES}
```

```
 {}

  {TABS.map(t =>  setTarget(t.key)}>
      {t.label}
    )}


{}

   setTeam(!team)}>
    {iconCheck(11)}

      I’m buying for a team or company (SSO, AWS/Azure/GCP, central bi




{}
{team &&

        Set up your team: self-serve or talk to sales.


          Get started


          Contact sales {iconArrowRight()}




      Run on

        {PROVIDERS.map(p =>  setProvider(p.key)}>
```

```
            {p.label}
          )}



    {showNotice &&
        {iconInfo()}

          {PROVIDER_NOTICE[provider]}

      }
  }

{}
{target === 'terminal' &&

      {Object.keys(TERM).map(k =>  setPkg(k)}>
          {TERM[k].label}
        )}

      {isWinInstaller &&  setWinCmd(!winCmd)}>
          {iconCheck(9)}
          CMD instead of PowerShell
        }

    {cardBodyCmd(terminalCmd, isWinPrompt ? '>' : '$')}
  }

{}
{target === 'terminal' &&
    {isWinInstaller &&
        Requires{' '}

          Git for Windows
        .
      }
    {(pkg === 'brew' || pkg === 'winget') &&
```

```
          Does not auto-update. Run{' '}
          {pkg === 'brew' ? 'brew upgrade claude-code' : 'winget upgrade
          periodically.
        }
      Troubleshooting
    }

  {alt &&

        The steps below use the command line.{' '}
        Prefer {alt.name}? Install here, then follow the {alt.name} guid



          {alt.installLabel} {iconArrowUpRight(13)}


          {alt.name} guide {iconArrowRight(12)}


      {alt.altCmd &&
          or run {alt.altCmd}
           handleCopy(alt.altCmd, 'alt')} aria-label="Copy command">
            {copied === 'alt' ? iconCheck(11) : iconCopy(11)}

        }
    }
;
```

};

export const Experiment = ({flag, treatment, children}) => {
const VID_KEY = 'exp_vid';
const CONSENT_COUNTRIES = new Set(['AT', 'BE', 'BG', 'HR', 'CY', 'CZ', 'DK', 'EE', 'FI', 'FR', 'DE', 'GR', 'HU', 'IE', 'IT', 'LV', 'LT', 'LU', 'MT', 'NL', 'PL', 'PT', 'RO', 'SK', 'SI', 'ES', 'SE', 'RE', 'GP', 'MQ', 'GF', 'YT', 'BL', 'MF', 'PM', 'WF', 'PF', 'NC', 'AW', 'CW', 'SX', 'FO', 'GL', 'AX', 'GB', 'UK', 'AI', 'BM', 'IO', 'VG', 'KY', 'FK', 'GI', 'MS', 'PN', 'SH', 'TC', 'GG', 'JE', 'IM', 'CA', 'BR', 'IN']);

```
const fnv1a = s => {
let h = 0x811c9dc5;
for (let i = 0; i >> 0;
};
const bucket = (seed, vid) => fnv1a(fnv1a(seed + vid) + '') % 10000 {
const params = new URLSearchParams(location.search);
const force = params.get('gb-force');
if (force) {
for (const p of force.split(',')) {
const [k, v] = p.split(':');
if (k === flag) return {
variant: v || 'treatment',
track: false
};
}
}
if (navigator.globalPrivacyControl) {
return {
variant: 'control',
track: false
};
}
const prefsMatch = document.cookie.match(/(?:^|; )anthropic-consent-preferences=([^;]
+)/);
if (prefsMatch) {
try {
if (JSON.parse(decodeURIComponent(prefsMatch[1])).analytics !== true) {
return {
variant: 'control',
track: false
};
}
} catch {
return {
variant: 'control',
track: false
};
```

```
}
} else {
const country = params.get('country')?.toUpperCase() || (document.cookie.match(/
(?:^|; )cf_geo=([A-Z]{2})/) || [])[1];
if (!country || CONSENT_COUNTRIES.has(country)) {
return {
variant: 'control',
track: false
};
}
}
let vid;
try {
const ajsMatch = document.cookie.match(/(?:^|; )ajs_anonymous_id=([^;]+)/);
if (ajsMatch) {
vid = decodeURIComponent(ajsMatch[1]).replace(/^"|"$/g, '');
} else {
vid = localStorage.getItem(VID_KEY);
if (!vid) {
vid = crypto.randomUUID();
}
document.cookie = ajs_anonymous_id=${vid}; domain=.claude.com;
path=/; Secure; SameSite=Lax; max-age=31536000;
}
try {
localStorage.setItem(VID_KEY, vid);
} catch {}
} catch {
return {
variant: 'control',
track: false
};
}
return {
variant: bucket(flag, vid),
track: true,
vid
```

```
};
});
useEffect(() => {
if (!decision.track) return;
fetch('https://api.anthropic.com/api/event_logging/v2/batch', {
method: 'POST',
headers: {
'Content-Type': 'application/json',
'x-service-name': 'claude_code_docs'
},
body: JSON.stringify({
events: [{
event_type: 'GrowthbookExperimentEvent',
event_data: {
device_id: decision.vid,
anonymous_id: decision.vid,
timestamp: new Date().toISOString(),
experiment_id: flag,
variation_id: decision.variant === 'treatment' ? 1 : 0,
environment: 'production'
}
}]
}),
keepalive: true
}).catch(() => {});
}, []);
return decision.variant === 'treatment' ? treatment : children;
};
```

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, Team, or Enterprise), Claude Console account, or access through a supported cloud provider

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:

macOS, Linux, WSL:

```bash  theme={null}
curl -fsSL https://claude.ai/install.sh | bash
```

Windows PowerShell:

```powershell  theme={null}
irm https://claude.ai/install.ps1 | iex
```

Windows CMD:

```batch  theme={null}
curl -fsSL https://claude.ai/install.cmd -o install.cmd && install.cmd &&
```

If you see `The token '&&' is not a valid statement separator`, you're in

Windows requires [Git for Windows](https://git-scm.com/downloads/win).

Native installations automatically update in the background to keep you

```bash  theme={null}
brew install --cask claude-code
```

Homebrew installations do not auto-update. Run `brew upgrade claude-cod

```powershell  theme={null}
winget install Anthropic.ClaudeCode
```

  WinGet installations do not auto-update. Run `winget upgrade Anthropic

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:

```bash theme={null}
claude

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

```bash  theme={null}
/login
# Follow the prompts to log in with your account
```

You can log in using any of these account types:

- Claude Pro, Max, Team, 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:

```bash theme={null}
cd /path/to/your/project
claude
```

```
You'll see the Claude Code welcome screen with your session information,

  After logging in (Step 2), your credentials are stored on your system.

## Step 4: Ask your first question

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

```text  theme={null}
what does this project do?
```

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

```text theme={null}
what technologies does this project use?
```

```
```text  theme={null}
where is the main entry point?
```

```text theme={null}
explain the folder structure
```

```
You can also ask Claude about its own capabilities:

```text  theme={null}
what can Claude Code do?
```

```text theme={null}
how do I create custom skills in Claude Code?
```

```
```text  theme={null}
can Claude Code work with Docker?
```

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:

```text theme={null}
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

  Claude Code always asks for permission before modifying files. You can

## Step 6: Use Git with Claude Code

Claude Code makes Git operations conversational:

```text  theme={null}
what files have I changed?
```

```text theme={null}
commit my changes with a descriptive message
```

```
You can also prompt for more complex Git operations:

```text  theme={null}
create a new branch called feature/quickstart
```

```text theme={null}
show me the last 5 commits
```

```
```text  theme={null}
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:

```text theme={null}
add input validation to the user registration form
```

Or fix existing issues:

```text theme={null}
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

```text theme={null}
refactor the authentication module to use async/await instead of callbacks
```

Write tests

```text theme={null}
write unit tests for the calculator functions
```

Update documentation

```text theme={null}
update the README with installation instructions
```

```
**Code review**

```text  theme={null}
review my changes and suggest improvements
```

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"`
`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`
`/clear`	Clear conversation history	`/clear`
`/help`	Show available commands	`/help`
`exit` or Ctrl+D	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 wr



Break complex tasks into steps:

```text  theme={null}
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:

```text  theme={null}
analyze the database schema
```

```text  theme={null}
build a dashboard showing products that are most frequently returned by
```



* 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:

```
Understand the agentic loop, built-in tools, and how Claude Code interac


Get better results with effective prompting and project setup


Step-by-step guides for common tasks


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

记忆与指令

> ## Documentation Index
>
> Fetch the complete documentation index at: https://code.claude.com/docs/llms.txt
> Use this file to discover all available pages before exploring further.

Claude 如何记住您的项目

> 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	

	CLAUDE.md files	Auto memory
		Every session (first 200 lines or 25KB)
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	Team members

Scope	Location	Purpose	Use case examples	Shared with
			standards, common workflows	via source control
User instructions	`~/.claude/CLAUDE.md`	Personal preferences for all projects	Code styling preferences, personal tooling shortcuts	Just you (all projects)
Local instructions	`./CLAUDE.local.md`	Personal project-specific preferences; add to `.gitignore`	Your sandbox URLs, preferred test data	Just you (current project)

CLAUDE.md and CLAUDE.local.md files in the directory hierarchy above the working directory are loaded in full at launch. 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.

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.

Set `CLAUDE_CODE_NEW_INIT=1` to enable an interactive multi-phase flow. `/init` asks which artifacts to set up: CLAUDE.md files, skills, and hooks. It then explores your codebase with a subagent, fills in gaps via follow-up questions, and presents a reviewable proposal before writing any files.

Write effective instructions

CLAUDE.md files are loaded into the context window at the start of every session, consuming tokens alongside your conversation. The context window visualization shows where CLAUDE.md loads relative to the rest of the startup context. 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:

```text theme={null}
See @README for project overview and @package.json for available npm commands for this project.

# Additional Instructions

- git workflow @docs/git-instructions.md
```

```
For private per-project preferences that shouldn't be checked into versi

If you work across multiple git worktrees of the same repository, a giti

```text  theme={null}
# Individual Preferences
- @~/.claude/my-project-instructions.md
```

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/`.
```

AGENTS.md

Claude Code reads `CLAUDE.md`, not `AGENTS.md`. If your repository already uses `AGENTS.md` for other coding agents, create a `CLAUDE.md` that imports it so both tools read the same instructions without duplicating them. You can also add Claude-specific instructions below the import. Claude loads the imported file at session start, then appends the rest:

```markdown CLAUDE.md theme={null}
@AGENTS.md

## Claude Code

Use plan mode for changes under `src/billing/`.
```

```
### How CLAUDE.md files load

Claude Code reads CLAUDE.md files by walking up the directory tree from y

All discovered files are concatenated into context rather than overriding

Claude also discovers `CLAUDE.md` and `CLAUDE.local.md` files in subdirec

If you work in a large monorepo where other teams' CLAUDE.md files get pi

Block-level HTML comments (``) in CLAUDE.md files are stripped before the

#### Load from additional directories

The `--add-dir` flag gives Claude access to additional directories outsi

To also load CLAUDE.md files from additional directories, including `CLA
```

```bash  theme={null}
CLAUDE_CODE_ADDITIONAL_DIRECTORIES_CLAUDE_MD=1 claude --add-dir ../share
```

`CLAUDE.local.md` files in additional directories are not loaded.

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.

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/`:

```text theme={null}
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](#path-specific-rules) are loaded at

#### Path-specific rules

Rules can be scoped to specific files using YAML frontmatter with the `pa

```markdown  theme={null}
---
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
`*.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:
```

```markdown theme={null}

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 s

This example links both a shared directory and an individual file:

```bash  theme={null}
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:

```text theme={null}
~/.claude/rules/
├── preferences.md # Your personal coding preferences
└── workflows.md # Your preferred workflows
```
```

```
User-level rules are loaded before project rules, giving project rules hi

### Manage CLAUDE.md for large teams

For organizations deploying Claude Code across teams, you can centralize

#### Deploy organization-wide CLAUDE.md

Organizations can deploy a centrally managed CLAUDE.md that applies to al


    * macOS: `/Library/Application Support/ClaudeCode/CLAUDE.md`
    * Linux and WSL: `/etc/claude-code/CLAUDE.md`
    * Windows: `C:\Program Files\ClaudeCode\CLAUDE.md`



    Use MDM, Group Policy, Ansible, or similar tools to distribute the fi


A managed CLAUDE.md and [managed settings](/en/settings#settings-files) s

| Concern                                          | Configure in
| :------------------------------------------------ | :-------------------
| Block specific tools, commands, or file paths  | Managed settings: `pe
| Enforce sandbox isolation                        | Managed settings: `sa
| Environment variables and API provider routing | Managed settings: `en
| Authentication method and organization lock    | Managed settings: `fo
| Code style and quality guidelines                | Managed CLAUDE.md
| Data handling and compliance reminders         | Managed CLAUDE.md
| Behavioral instructions for Claude               | Managed CLAUDE.md

Settings rules are enforced by the client regardless of what Claude deci

#### Exclude specific CLAUDE.md files
```

```
In large monorepos, ancestor CLAUDE.md files may contain instructions tha

This example excludes a top-level CLAUDE.md and a rules directory from a

```json  theme={null}
{
  "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.

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:

```json theme={null}
{
"autoMemoryEnabled": false
}
```
```

````
To disable auto memory via environment variable, set `CLAUDE_CODE_DISABL

### Storage location

Each project gets its own memory directory at `~/.claude/projects//memor

To store auto memory in a different location, set `autoMemoryDirectory` i

```json  theme={null}
{
  "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:

```text theme={null}
~/.claude/projects//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 w

Auto memory is machine-local. All worktrees and subdirectories within th

### How it works

The first 200 lines of `MEMORY.md`, or the first 25KB, whichever comes f

This limit applies only to `MEMORY.md`. CLAUDE.md files are loaded in fu

Topic files like `debugging.md` or `patterns.md` are not loaded at start

Claude reads and writes memory files during your session. When you see "

### Audit and edit your memory

Auto memory files are plain markdown you can edit or delete at any time.

## View and edit with `/memory`

The `/memory` command lists all CLAUDE.md, CLAUDE.local.md, and rules fi

When you ask Claude to remember something, like "always use pnpm, not npm

## Troubleshoot memory issues

These are the most common issues with CLAUDE.md and auto memory, along wi

### Claude isn't following my CLAUDE.md

CLAUDE.md content is delivered as a user message after the system prompt

To debug:

* Run `/memory` to verify your CLAUDE.md and CLAUDE.local.md files are be
* Check that the relevant CLAUDE.md is in a location that gets loaded fo
```

```
* Make instructions more specific. "Use 2-space indentation" works bette
* Look for conflicting instructions across CLAUDE.md files. If two files

For instructions you want at the system prompt level, use [`--append-syst

  Use the [`InstructionsLoaded` hook](/en/hooks#instructionsloaded) to lo

### I don't know what auto memory saved

Run `/memory` and select the auto memory folder to browse what Claude has

### My CLAUDE.md is too large

Files over 200 lines consume more context and may reduce adherence. Move

### Instructions seem lost after `/compact`

CLAUDE.md fully survives compaction. After `/compact`, Claude re-reads yo

See [Write effective instructions](#write-effective-instructions) for gui

## Related resources

* [Skills](/en/skills): package repeatable workflows that load on demand
* [Settings](/en/settings): configure Claude Code behavior with settings
* [Manage sessions](/en/sessions): manage context, resume conversations,
* [Subagent memory](/en/sub-agents#enable-persistent-memory): let subagen

---

# 通用工作流

> ## Documentation Index
> Fetch the complete documentation index at: https://code.claude.com/docs
> Use this file to discover all available pages before exploring further

# 通用工作流
```

````
> Step-by-step guides for exploring codebases, fixing bugs, refactoring,

This page covers practical workflows for everyday development: exploring

## Understand new codebases

### Get a quick codebase overview

Suppose you've just joined a new project and need to understand its struc


    ```bash  theme={null}
    cd /path/to/project
    ```



    ```bash  theme={null}
    claude
    ```



    ```text  theme={null}
    give me an overview of this codebase
    ```



    ```text  theme={null}
    explain the main architecture patterns used here
    ```

    ```text  theme={null}
    what are the key data models?
    ```
````

````
    ```text  theme={null}
    how is authentication handled?
    ```


  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 functio


    ```text  theme={null}
    find the files that handle user authentication
    ```



    ```text  theme={null}
    how do these authentication files work together?
    ```



    ```text  theme={null}
    trace the login process from front-end to database
    ```


  Tips:

  * Be specific about what you're looking for
````

```
  * Use domain language from the project
  * Install a [code intelligence plugin](/en/discover-plugins#code-intel

***

## Fix bugs efficiently

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


    ```text  theme={null}
    I'm seeing an error when I run npm test
    ```



    ```text  theme={null}
    suggest a few ways to fix the @ts-ignore in user.ts
    ```



    ```text  theme={null}
    update user.ts to add the null check you suggested
    ```


  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

    ```text  theme={null}
    find deprecated API usage in our codebase
    ```

    ```text  theme={null}
    suggest how to refactor utils.js to use modern JavaScript features
    ```

    ```text  theme={null}
    refactor utils.js to use ES2024 features while maintaining the same b
    ```

    ```text  theme={null}
    run tests for the refactored code
    ```

  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
````

```text  theme={null}
/agents
```

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

Claude Code automatically delegates appropriate tasks to specialized

```text  theme={null}
review my recent code changes for security issues
```

```text  theme={null}
run all tests and fix any failures
```

```text  theme={null}
use the code-reviewer subagent to check the auth module
```

```text  theme={null}
have the debugger subagent investigate why users can't log in
```

```text  theme={null}
/agents
```

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

```
    * A unique identifier that describes the subagent's purpose (for exa
    * When Claude should use this agent
    * Which tools it can access
    * A system prompt describing the agent's role and behavior


  Tips:

  * Create project-specific subagents in `.claude/agents/` for team shar
  * Use descriptive `description` fields to enable automatic delegation
  * Limit tool access to what each subagent actually needs
  * Check the [subagents documentation](/en/sub-agents) for detailed exa

***

## Use Plan Mode for safe code analysis

Plan Mode instructs Claude to create a plan by analyzing the codebase wi

### When to use Plan Mode

* **Multi-step implementation**: When your feature requires making edits
* **Code exploration**: When you want to research the codebase thoroughl
* **Interactive development**: When you want to iterate on the direction

### 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 cy

If you are in Normal Mode, **Shift+Tab** first switches into Auto-Accept

**Start a new session in Plan Mode**

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

```bash  theme={null}
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"):

```bash theme={null}
claude --permission-mode plan -p "Analyze the authentication system and suggest improvements"
```

Example: Planning a complex refactor

```bash  theme={null}
claude --permission-mode plan
```

```text theme={null}
I need to refactor our authentication system to use OAuth2. Create a detailed migration plan.
```

```
Claude analyzes the current implementation and create a comprehensive pla
```

```text  theme={null}
What about backward compatibility?
```

```text theme={null}
How should we handle database migration?
```

```
Press `Ctrl+G` to open the plan in your default text editor, where you ca

When you accept a plan, Claude automatically names the session from the p

### Configure Plan Mode as default

```json  theme={null}
// .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.
```

```text  theme={null}
find functions in NotificationsService.swift that are not covered by tes
```

```text  theme={null}
add tests for the notification service
```

```text  theme={null}
add test cases for edge conditions in the notification service
```

```text  theme={null}
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:

````
```text  theme={null}
summarize the changes I've made to the authentication module
```



```text  theme={null}
create a pr
```



```text  theme={null}
enhance the PR description with more context about the security improveme
```
````

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`.

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.

```text  theme={null}
find functions without proper JSDoc comments in the auth module
```

```text  theme={null}
add JSDoc comments to the undocumented functions in auth.js
```

```text  theme={null}
improve the generated documentation with more context and examples
```

```text  theme={null}
check if the documentation follows our project standards
```

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.

````
 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/y



```text  theme={null}
What does this image show?
```

```text  theme={null}
Describe the UI elements in this screenshot
```

```text  theme={null}
Are there any problematic elements in this diagram?
```



```text  theme={null}
Here's a screenshot of the error. What's causing it?
```

```text  theme={null}
This is our current database schema. How should we modify it for the new
```



```text  theme={null}
Generate CSS to match this design mockup
```

```text  theme={null}
````
````

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

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.
```

````
```text  theme={null}
Explain the logic in @src/utils/auth.js
```

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



```text  theme={null}
What's the structure of @src/components?
```

This provides a directory listing with file information.



```text  theme={null}
Show me the data from @github:repos/owner/repo/issues
```

This fetches data from connected MCP servers using the format @server:re
````

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.

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`
Limit token budget	Set `MAX_THINKING_TOKENS` environment variable	Limit the thinking budget to a specific number of tokens. On

Scope	How to configure	Details
		Opus 4.6 and Sonnet 4.6, only `0` applies unless adaptive reasoning is disabled. 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 token budget drawn from your output allocation. The budget varies by model; see `MAX_THINKING_TOKENS` for per-model ceilings. You can limit the budget with that environment variable, or disable thinking entirely via `/config` or the `Option+T` / `Alt+T` toggle.

On Opus 4.6 and Sonnet 4.6, adaptive reasoning controls thinking depth, so `MAX_THINKING_TOKENS` only applies when set to `0` to disable thinking, or when `CLAUDE_CODE_DISABLE_ADAPTIVE_THINKING=1` reverts these models to the fixed budget. See environment variables.

You're charged for all thinking tokens used even when thinking summaries are redacted. In interactive mode, thinking appears as a collapsed stub by default. Set `showThinkingSummaries: true` in `settings.json` to show full summaries.

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 interactive sessions from the same git repository, including worktrees. Sessions created by `claude -p` or SDK invocations do not appear in the picker, but you can still resume one by passing its session ID directly to `claude --resume`.

Name your sessions

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

````
Name a session at startup with `-n`:

```bash  theme={null}
claude -n auth-refactor
```

Or use `/rename` during a session, which also shows the name on the prom

```text  theme={null}
/rename auth-refactor
```

You can also rename any session from the picker: run `/resume`, navigate



From the command line:

```bash  theme={null}
claude --resume auth-refactor
```

Or from inside an active session:

```text  theme={null}
/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 `/branch`, `/rewind`, or `--fork-session`) are grouped together under their root session, making it easier to find related conversations.

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:

```bash theme={null}
```

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:

```bash  theme={null}
# Auto-generates a name like "bright-running-fox"
claude --worktree
```

Worktrees are created at `/.claude/worktrees/` and branch from the default remote branch, which is where `origin/HEAD` points. The worktree branch is named `worktree-`.

The base branch is not configurable through a Claude Code flag or setting. `origin/HEAD` is a reference stored in your local `.git` directory that Git set once when you cloned. If the repository's default branch later changes on GitHub or GitLab, your local `origin/HEAD` keeps pointing at the old one, and worktrees will branch from there. To re-sync your local reference with whatever the remote currently considers its default:
```

```bash theme={null}
git remote set-head origin -a
```

```
This is a standard Git command that only updates your local `.git` direc

For full control over how worktrees are created, including choosing a di

You can also ask Claude to "work in a worktree" or "start a worktree" du

### Subagent worktrees

Subagents can also use worktree isolation to work in parallel without co

### Worktree cleanup

When you exit a worktree session, Claude handles cleanup based on whethe

* **No changes**: the worktree and its branch are removed automatically
* **Changes or commits exist**: Claude prompts you to keep or remove the

Subagent worktrees orphaned by a crash or an interrupted parallel run ar

To clean up worktrees outside of a Claude session, use [manual worktree r

  Add `.claude/worktrees/` to your `.gitignore` to prevent worktree conte

### Copy gitignored files to worktrees

Git worktrees are fresh checkouts, so they don't include untracked files

The file uses `.gitignore` syntax to list which files to copy. Only files

```text .worktreeinclude theme={null}
.env
.env.local
config/secrets.json
```
```

This applies to worktrees created with `--worktree`, subagent worktrees, and parallel sessions in the desktop app.

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.

```bash theme={null}
# 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](https://git-scm.c

  Remember to initialize your development environment in each new worktre

### Non-git version control

Worktree isolation works with git by default. For other version control s

For automated coordination of parallel sessions with shared tasks and mes

***

## Get notified when Claude needs your attention

When you kick off a long-running task and switch to another window, you c


    Open `~/.claude/settings.json` and add a `Notification` hook that cal


        ```json  theme={null}
        {
          "hooks": {
            "Notification": [
              {
                "matcher": "",
                "hooks": [
                  {
                    "type": "command",
                    "command": "osascript -e 'display notification \"Clau
                  }
                ]
              }
            ]
          }
````
````

````
}
```

```json  theme={null}
{
  "hooks": {
    "Notification": [
      {
        "matcher": "",
        "hooks": [
          {
            "type": "command",
            "command": "notify-send 'Claude Code' 'Claude Code ne
          }
        ]
      }
    ]
  }
}
```

```json  theme={null}
{
  "hooks": {
    "Notification": [
      {
        "matcher": "",
        "hooks": [
          {
            "type": "command",
            "command": "powershell.exe -Command \"[System.Reflec
          }
        ]
````

```
            }
          ]
        }
      }
      ```

    If your settings file already has a `hooks` key, merge the `Notifica

    By default the hook fires on all notification types. To fire only fo

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

    Type `/hooks` and select `Notification` to confirm the hook appears.

For the complete event schema and notification types, see the [Notificat

***

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:

```
```json  theme={null}
// package.json
{
    ...
    "scripts": {
        ...
        "lint:claude": "claude -p 'you are a linter. please look at the
    }
}
```

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:

```
```bash theme={null}
cat build-error.txt | claude -p 'concisely explain the root cause of this build error' >
output.txt
```

````
  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 in



    ```bash  theme={null}
    cat data.txt | claude -p 'summarize this data' --output-format text >
    ```

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



    ```bash  theme={null}
    cat code.py | claude -p 'analyze this code for bugs' --output-format
    ```

    This outputs a JSON array of messages with metadata including cost an



    ```bash  theme={null}
    cat log.txt | claude -p 'parse this log file for errors' --output-fo
    ```

    This outputs a series of JSON objects in real-time as Claude processe


  Tips:
````

```
  * Use `--output-format text` for simple integrations where you just nee
  * Use `--output-format json` when you need the full conversation log
  * Use `--output-format stream-json` for real-time output of each conve

***

## Run Claude on a schedule

Suppose you want Claude to handle a task automatically on a recurring bas

Pick a scheduling option based on where you want the task to run:

| Option                                                          | Where
| :-------------------------------------------------------------- | :---
| [Cloud scheduled tasks](/en/web-scheduled-tasks)                | Anth
| [Desktop scheduled tasks](/en/desktop#schedule-recurring-tasks) | Your
| [GitHub Actions](/en/github-actions)                            | Your
| [`/loop`](/en/scheduled-tasks)                                  | The

  When writing prompts for scheduled tasks, be explicit about what succes

***

## Ask Claude about its capabilities

Claude has built-in access to its documentation and can answer questions

### Example questions

```text  theme={null}
can Claude Code create pull requests?
```

```text theme={null}
how does Claude Code handle permissions?
```
```

```text  theme={null}
what skills are available?
```

```text theme={null}
how do I use MCP with Claude Code?
```

```text  theme={null}
how do I configure Claude Code for Amazon Bedrock?
```

```text theme={null}
what are the limitations of Claude Code?
```

```
  Claude provides documentation-based answers to these questions. For har

  Tips:

  * Claude always has access to the latest Claude Code documentation, reg
  * Ask specific questions to get detailed answers
  * Claude can explain complex features like MCP integration, enterprise

***

## 下一步


    Patterns for getting the most out of Claude Code



    Understand the agentic loop and context management



    Add skills, hooks, MCP, subagents, and plugins



    Clone the development container reference implementation

---

# 最佳实践

> ## Documentation Index
> Fetch the complete documentation index at: https://code.claude.com/docs
> Use this file to discover all available pages before exploring further

# Claude Code 最佳实践
```

```
> Tips and patterns for getting the most out of Claude Code, from configu

Claude Code is an agentic coding environment. Unlike a chatbot that answe

This changes how you work. Instead of writing code yourself and asking Cl

But this autonomy still comes with a learning curve. Claude works within

This guide covers patterns that have proven effective across Anthropic's

***

Most best practices are based on one constraint: Claude's context window

Claude's context window holds your entire conversation, including every m

This matters since LLM performance degrades as context fills. When the co

***

## Give Claude a way to verify its work

  Include tests, screenshots, or expected outputs so Claude can check its

Claude performs dramatically better when it can verify its own work, like

Without clear success criteria, it might produce something that looks rig

| Strategy                                | Before
| --------------------------------------- | ------------------------------
| **Provide verification criteria**       | *"implement a function that val
| **Verify UI changes visually**          | *"make the dashboard look bette
| **Address root causes, not symptoms**   | *"the build is failing"*

UI changes can be verified using the [Claude in Chrome extension](/en/ch
```

````
Your verification can also be a test suite, a linter, or a Bash command t

***

## Explore first, then plan, then code

  Separate research and planning from implementation to avoid solving the

Letting Claude jump straight to coding can produce code that solves the w

The recommended workflow has four phases:


    Enter Plan Mode. Claude reads files and answers questions without mak

    ```txt claude (Plan Mode) theme={null}
    read /src/auth and understand how we handle sessions and login.
    also look at how we manage environment variables for secrets.
    ```



    Ask Claude to create a detailed implementation plan.

    ```txt claude (Plan Mode) theme={null}
    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 editin



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

    ```txt claude (Normal Mode) theme={null}
    implement the OAuth flow from your plan. write tests for the
````
````

````
    callback handler, run the test suite and fix any failures.
    ```



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

    ```txt claude (Normal Mode) theme={null}
    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

  Planning is most useful when you're uncertain about the approach, when

***

## Provide specific context in your prompts

  The more precise your instructions, the fewer corrections you'll need.

Claude can infer intent, but it can't read your mind. Reference specific

| Strategy
| ----------------------------------------------------------------------
| **Scope the task.** Specify which file, what scenario, and testing pre
| **Point to sources.** Direct Claude to the source that can answer a qu
| **Reference existing patterns.** Point Claude to patterns in your code
| **Describe the symptom.** Provide the symptom, the likely location, an

Vague prompts can be useful when you're exploring and can afford to cours

### Provide rich content
````
````

```
  Use `@` to reference files, paste screenshots/images, or pipe data dir

You can provide rich data to Claude in several ways:

* **Reference files with `@`** instead of describing where code lives. C
* **Paste images directly**. Copy/paste or drag and drop images into the
* **Give URLs** for documentation and API references. Use `/permissions`
* **Pipe in data** by running `cat error.log | claude` to send file cont
* **Let Claude fetch what it needs**. Tell Claude to pull context itself

***

## Configure your environment

A few setup steps make Claude Code significantly more effective across a

### Write an effective CLAUDE.md

  Run `/init` to generate a starter CLAUDE.md file based on your current

CLAUDE.md is a special file that Claude reads at the start of every conv

The `/init` command analyzes your codebase to detect build systems, test

There's no required format for CLAUDE.md files, but keep it short and hu

```markdown CLAUDE.md theme={null}
# 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 performa
```
```

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:

```markdown CLAUDE.md theme={null}
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 tea
* **Project root (`./CLAUDE.local.md`)**: personal project-specific notes
* **Parent directories**: useful for monorepos where both `root/CLAUDE.md
* **Child directories**: Claude pulls in child CLAUDE.md files on demand

### Configure permissions

  Use [auto mode](/en/permission-modes#eliminate-prompts-with-auto-mode)

By default, Claude Code requests permission for actions that might modify

* **Auto mode**: a separate classifier model reviews commands and blocks
* **Permission allowlists**: permit specific tools you know are safe, lik
* **Sandboxing**: enable OS-level isolation that restricts filesystem and

Read more about [permission modes](/en/permission-modes), [permission rul

### Use CLI tools

  Tell Claude Code to use CLI tools like `gh`, `aws`, `gcloud`, and `sent

CLI tools are the most context-efficient way to interact with external se

Claude is also effective at learning CLI tools it doesn't already know. T

### Connect MCP servers

  Run `claude mcp add` to connect external tools like Notion, Figma, or y

With [MCP servers](/en/mcp), you can ask Claude to implement features fro

### Set up hooks
```

````
  Use hooks for actions that must happen every time with zero exceptions

[Hooks](/en/hooks-guide) run scripts automatically at specific points in

Claude can write hooks for you. Try prompts like *"Write a hook that runs

### Create skills

  Create `SKILL.md` files in `.claude/skills/` to give Claude domain know

[Skills](/en/skills) extend Claude's knowledge with information specific

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

```markdown .claude/skills/api-conventions/SKILL.md theme={null}
---
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:

```markdown .claude/skills/fix-issue/SKILL.md theme={null}

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`

### Create custom subagents

  Define specialized assistants in `.claude/agents/` that Claude can dele

[Subagents](/en/sub-agents) run in their own context with their own set o

```markdown .claude/agents/security-reviewer.md theme={null}
---
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

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

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

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.

```text theme={null}
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 s

***

## Manage your session

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

### Course-correct early and often

  Correct Claude as soon as you notice it going off track.

The best results come from tight feedback loops. Though Claude occasiona

* **`Esc`**: stop Claude mid-action with the `Esc` key. Context is prese
* **`Esc + Esc` or `/rewind`**: press `Esc` twice or run `/rewind` to op
* **`"Undo that"`**: have Claude revert its changes.
* **`/clear`**: reset context between unrelated tasks. Long sessions wit

If you've corrected Claude more than twice on the same issue in one sessi

### Manage context aggressively

  Run `/clear` between unrelated tasks to reset context.

Claude Code automatically compacts conversation history when you approac

During long sessions, Claude's context window can fill with irrelevant co

* Use `/clear` frequently between tasks to reset the context window enti
* When auto compaction triggers, Claude summarizes what matters most, in
* For more control, run `/compact `, like `/compact Focus on the API cha
* To compact only part of the conversation, use `Esc + Esc` or `/rewind`
* Customize compaction behavior in CLAUDE.md with instructions like `"Wh
* For quick questions that don't need to stay in context, use [`/btw`](/
```

````
### Use subagents for investigation

  Delegate research with `"use subagents to investigate X"`. They explore

Since context is your fundamental constraint, subagents are one of the mo

```text  theme={null}
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:

```text theme={null}
use a subagent to review this code for edge cases
```

````
### Rewind with checkpoints

  Every action Claude makes creates a checkpoint. You can restore convers

Claude automatically checkpoints before changes. Double-tap `Escape` or

Instead of carefully planning every move, you can tell Claude to try some

  Checkpoints only track changes made *by Claude*, not external processes

### Resume conversations

  Run `claude --continue` to pick up where you left off, or `--resume` to

Claude Code saves conversations locally. When a task spans multiple sessi

```bash  theme={null}
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

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.

```bash theme={null}

# 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

  Run multiple Claude sessions in parallel to speed up development, run i

There are three main ways to run parallel sessions:

* [Claude Code desktop app](/en/desktop#work-in-parallel-with-sessions):
* [Claude Code on the web](/en/claude-code-on-the-web): Run on Anthropic
* [Agent teams](/en/agent-teams): Automated coordination of multiple sess

Beyond parallelizing work, multiple sessions enable quality-focused workf

For example, use a Writer/Reviewer pattern:

| Session A (Writer)
| ---------------------------------------------------------------------
| `Implement a rate limiter for our API endpoints`
|
| `Here's the review feedback: [Session B output]. Address these issues.`

You can do something similar with tests: have one Claude write tests, the

### Fan out across files

  Loop through tasks calling `claude -p` for each. Use `--allowedTools` f

For large migrations or analyses, you can distribute work across many pa


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



    ```bash  theme={null}
    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
    ```


    Refine your prompt based on what goes wrong with the first 2-3 files

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

```bash  theme={null}
claude -p "" --output-format json | your_command
```

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

Run autonomously with auto mode

For uninterrupted execution with background safety checks, use auto mode. A classifier model reviews commands before they run, blocking scope escalation, unknown infrastructure, and hostile-content-driven actions while letting routine work proceed without prompts.

```bash theme={null}
claude --permission-mode auto -p "fix all lint errors"
```

```
For non-interactive runs with the `-p` flag, auto mode aborts if the clas

***

## 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
  > **Fix**: `/clear` between unrelated tasks.
* **Correcting over and over.** Claude does something wrong, you correct
  > **Fix**: After two failed corrections, `/clear` and write a better in
* **The over-specified CLAUDE.md.** If your CLAUDE.md is too long, Claude
  > **Fix**: Ruthlessly prune. If Claude already does something correctly
* **The trust-then-verify gap.** Claude produces a plausible-looking impl
  > **Fix**: Always provide verification (tests, scripts, screenshots). I
* **The infinite exploration.** You ask Claude to "investigate" something
  > **Fix**: Scope investigations narrowly or use subagents so the explo

***

## Develop your intuition

The patterns in this guide aren't set in stone. They're starting points f

Sometimes you *should* let context accumulate because you're deep in one

Pay attention to what works. When Claude produces great output, notice wh

Over time, you'll develop intuition that no guide can capture. You'll kno

## Related resources

* [How Claude Code works](/en/how-claude-code-works): the agentic loop, t
* [Extend Claude Code](/en/features-overview): skills, hooks, MCP, subage
* [Common workflows](/en/common-workflows): step-by-step recipes for debu
```

```
* [CLAUDE.md](/en/memory): store project conventions and persistent conte

---

# 设置

> ## Documentation Index
> Fetch the complete documentation index at: https://code.claude.com/docs
> Use this file to discover all available pages before exploring further

# Claude Code settings

> Configure Claude Code with global and project-level settings, and envi

Claude Code offers a variety of settings to configure its behavior to me

## 配置 scopes

Claude Code uses a **scope system** to determine where configurations ap

### Available scopes

| Scope       | Location
| :---------- | :-------------------------------------------------------
| **Managed** | Server-managed settings, plist / registry, or system-leve
| **User**    | `~/.claude/` directory
| **Project** | `.claude/` in repository
| **Local**   | `.claude/settings.local.json`

### 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 sc

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 p

### What uses scopes

Scopes apply to many Claude Code features:

| Feature         | User location             | Project location
| :-------------- | :------------------------ | :----------------------
| **Settings**    | `~/.claude/settings.json` | `.claude/settings.json`
```

```
| **Subagents**   | `~/.claude/agents/`       | `.claude/agents/`
| **MCP servers** | `~/.claude.json`          | `.mcp.json`
| **Plugins**     | `~/.claude/settings.json` | `.claude/settings.json`
| **CLAUDE.md**   | `~/.claude/CLAUDE.md`     | `CLAUDE.md` or `.claude/C

***

## 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
  projects.
* **Project settings** are saved in your project directory:
  * `.claude/settings.json` for settings that are checked into source con
  * `.claude/settings.local.json` for settings that are not checked in, c
* **Managed settings**: For organizations that need centralized control,

  * **Server-managed settings**: delivered from Anthropic's servers via t
  * **MDM/OS-level policies**: delivered through native device managemen
    * macOS: `com.anthropic.claudecode` managed preferences domain (deplo
    * Windows: `HKLM\SOFTWARE\Policies\ClaudeCode` registry key with a `S
    * Windows (user-level): `HKCU\SOFTWARE\Policies\ClaudeCode` (lowest p
  * **File-based**: `managed-settings.json` and `managed-mcp.json` deploy

    * macOS: `/Library/Application Support/ClaudeCode/`
    * Linux and WSL: `/etc/claude-code/`
    * Windows: `C:\Program Files\ClaudeCode\`


      The legacy Windows path `C:\ProgramData\ClaudeCode\managed-settings


    File-based managed settings also support a drop-in directory at `mana

    Following the systemd convention, `managed-settings.json` is merged t
```

```
    Use numeric prefixes to control merge order, for example `10-telemet

  See [managed settings](/en/permissions#managed-only-settings) and [Mana


    Managed deployments can also restrict **plugin marketplace additions
    `strictKnownMarketplaces`. For more information, see [Managed marketp

* **Other configuration** is stored in `~/.claude.json`. This file conta

  Claude Code automatically creates timestamped backups of configuration

```JSON Example settings.json theme={null}
{
  "$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
`agent`	Run the main thread as a named subagent. Applies that subagent's system prompt, tool restrictions, and model. See Invoke subagents explicitly
`allowedChannelPlugins`	(Managed settings only) Allowlist of channel plugins that may push messages. Replaces the default Anthropic allowlist when set. Undefined = fall back to the default, empty array = block all channel plugins. Requires `channelsEnabled: true`. See Restrict which channel plugins can run
`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
`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

Key	Description
`allowManagedHooksOnly`	(Managed settings only) Prevent loading of user, project, and plugin hooks. Only allows managed hooks and SDK hooks. See Hook configuration
`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
`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
`alwaysThinkingEnabled`	Enable extended thinking by default for all sessions. Typically configured via the `/config` command rather than editing directly
`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
`attribution`	Customize attribution for git commits and pull requests. See Attribution settings
`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
`autoMode`	

Key	Description
	Customize what the auto mode classifier blocks and allows. Contains `environment`, `allow`, and `soft_deny` arrays of prose rules. See Configure the auto mode classifier. Not read from shared project settings
`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
`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
`awsAuthRefresh`	Custom script that modifies the `.aws` directory (see advanced credential configuration)
`awsCredentialExport`	Custom script that outputs JSON with AWS credentials (see advanced credential configuration)
`blockedMarketplaces`	(Managed settings only) Blocklist of marketplace sources. Blocked sources are checked before downloading, so they never touch the filesystem. See Managed marketplace restrictions
`channelsEnabled`	(Managed settings only) Allow channels for Team and Enterprise users. Unset or `false` blocks channel message delivery regardless of what users pass to `--channels`
`cleanupPeriodDays`	Sessions inactive for longer than this period are deleted at startup (default: 30 days, minimum 1). Setting to `0` is rejected with a validation error.

Key	Description
	Also controls the age cutoff for automatic removal of orphaned subagent worktrees at startup. To disable transcript writes entirely in non-interactive mode (`-p`), use the `--no-session-persistence` flag or the `persistSession: false` SDK option; there is no interactive-mode equivalent.
`companyAnnouncements`	Announcement to display to users at startup. If multiple announcements are provided, they will be cycled through at random.
`defaultShell`	Default shell for input-box `!` commands. Accepts `"bash"` (default) or `"powershell"`. Setting `"powershell"` routes interactive `!` commands through PowerShell on Windows. Requires `CLAUDE_CODE_USE_POWERSHELL_TOOL=1`. See PowerShell tool
`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
`disableAllHooks`	Disable all hooks and any custom status line
`disableAutoMode`	Set to `"disable"` to prevent auto mode from being activated. Removes `auto` from the `Shift+Tab` cycle and rejects `--permission-mode auto` at startup. Most useful in managed settings where users cannot override it
`disableDeepLinkRegistration`	

Key	Description
	Set to `"disable"` to prevent Claude Code from registering the `claude-cli://` protocol handler with the operating system on startup. Deep links let external tools open a Claude Code session with a pre-filled prompt via `claude-cli://open?q=...`. The `q` parameter supports multi-line prompts using URL-encoded newlines (`%0A`). Useful in environments where protocol handler registration is restricted or managed separately
`disabledMcpjsonServers`	List of specific MCP servers from `.mcp.json` files to reject
`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
`enableAllProjectMcpServers`	Automatically approve all MCP servers defined in project `.mcp.json` files
`enabledMcpjsonServers`	List of specific MCP servers from `.mcp.json` files to approve
`env`	Environment variables that will be applied to every session
`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
`feedbackSurveyRate`	Probability (0–1) that the session quality survey appears when eligible. Set to `0` to suppress

Key	Description
	entirely. Useful when using Bedrock, Vertex, or Foundry where the default sample rate does not apply
`fileSuggestion`	Configure a custom script for `@` file autocomplete. See File suggestion settings
`forceLoginMethod`	Use `claudeai` to restrict login to Claude.ai accounts, `console` to restrict login to Claude Console (API usage billing) accounts
`forceLoginOrgUUID`	Require login to belong to a specific organization. Accepts a single UUID string, which also pre-selects that organization during login, or an array of UUIDs where any listed organization is accepted without pre-selection. When set in managed settings, login fails if the authenticated account does not belong to a listed organization; an empty array fails closed and blocks login with a misconfiguration message
`hooks`	Configure custom commands to run at lifecycle events. See hooks documentation for format
`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
`includeCoAuthoredBy`	**Deprecated**: Use `attribution` instead. Whether to include the `co-authored-by Claude` byline in git commits and pull requests (default: `true`)

Key	Description
`includeGitInstructions`	Include built-in commit and PR workflow instructions and the git status snapshot in Claude's system prompt (default: `true`). Set to `false` to remove both, for example when using your own git workflow skills. The `CLAUDE_CODE_DISABLE_GIT_INSTRUCTIONS` environment variable takes precedence over this setting when set
`language`	Configure Claude's preferred response language (e.g., `"japanese"`, `"spanish"`, `"french"`). Claude will respond in this language by default. Also sets the voice dictation language
`model`	Override the default model to use for Claude Code
`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
`otelHeadersHelper`	Script to generate dynamic OpenTelemetry headers. Runs at startup and periodically (see Dynamic headers)
`outputStyle`	Configure an output style to adjust the system prompt. See output styles documentation
`permissions`	See table below for structure of permissions.
`plansDirectory`	Customize where plan files are stored. Path is relative to project root. Default: `~/.claude/plans`
`pluginTrustMessage`	

Key	Description
	(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.
`prefersReducedMotion`	Reduce or disable UI animations (spinners, shimmer, flash effects) for accessibility
`respectGitignore`	Control whether the `@` file picker respects `.gitignore` patterns. When `true` (default), files matching `.gitignore` patterns are excluded from suggestions
`showClearContextOnPlanAccept`	Show the "clear context" option on the plan accept screen. Defaults to `false`. Set to `true` to restore the option
`showThinkingSummaries`	Show extended thinking summaries in interactive sessions. When unset or `false` (default in interactive mode), thinking blocks are redacted by the API and shown as a collapsed stub. Redaction only changes what you see, not what the model generates: to reduce thinking spend, lower the budget or disable thinking instead. Non-interactive mode (`-p`) and SDK callers always receive summaries regardless of this setting
`spinnerTipsEnabled`	Show tips in the spinner while Claude is working. Set to `false` to disable tips (default: `true`)
`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
`spinnerVerbs`	

Key	Description
	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
`statusLine`	Configure a custom status line to display context. See `statusLine` documentation
`strictKnownMarketplaces`	(Managed settings only) Allowlist of plugin marketplaces users can add. Undefined = no restrictions, empty array = lockdown. Applies to marketplace additions only. See Managed marketplace restrictions
`useAutoModeDuringPlan`	Whether plan mode uses auto mode semantics when auto mode is available. Default: `true`. Not read from shared project settings. Appears in `/config` as "Use auto mode during plan"
`voiceEnabled`	Enable push-to-talk voice dictation. Written automatically when you run `/voice`. Requires a Claude.ai account

Global config settings

These settings are stored in `~/.claude.json` rather than `settings.json`. Adding them to `settings.json` will trigger a schema validation error.

Key	Description	Example
`autoConnectIde`	Automatically connect to a running IDE when Claude Code starts from an external terminal. Default: `false`. Appears in `/config` as **Auto-connect to IDE (external terminal)**	`true`

Key	Description	Example
	when running outside a VS Code or JetBrains terminal	
`autoInstallIdeExtension`	Automatically install the Claude Code IDE extension when running from a VS Code terminal. Default: `true`. Appears in `/config` as **Auto-install IDE extension** when running inside a VS Code or JetBrains terminal. You can also set the `CLAUDE_CODE_IDE_SKIP_AUTO_INSTALL` environment variable	`false`
`editorMode`	Key binding mode for the input prompt: `"normal"` or `"vim"`. Default: `"normal"`. Written automatically when you run `/vim`. Appears in `/config` as **Key binding mode**	`"vim"`
`showTurnDuration`	Show turn duration messages after responses, e.g. "Cooked for 1m 6s". Default: `true`. Appears in `/config` as **Show turn duration**	`false`
`terminalProgressBarEnabled`	Show the terminal progress bar in supported terminals: ConEmu, Ghostty 1.2.0+, and iTerm2 3.6.6+. Default: `true`. Appears in `/config` as **Terminal progress bar**	`false`
`teammateMode`	How agent team teammates display: `auto` (picks split panes in tmux or iTerm2, in-process otherwise), `in-process`, or `tmux`. See choose a display mode	`"in-process"`

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"]`

To copy gitignored files like `.env` into new worktrees, use a `.worktreeinclude` file in your project root instead of a setting.

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	`[ "Bash(git push *)" ]`

Keys	Description	Example
	Permission rule syntax below	
`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 for file access. Most `.claude/` configuration is not discovered from these directories	`[ "../docs/" ]`
`defaultMode`	Default permission mode when opening Claude Code. Valid values: `default`, `acceptEdits`, `plan`, `auto`, `dontAsk`, `bypassPermissions`. The `--permission-mode` CLI flag overrides this setting for a single session	`"acceptEdits"`
`disableBypassPermissionsMode`	Set to `"disable"` to prevent `bypassPermissions` mode from being activated. This disables the `--dangerously-skip-`	`"disable"`

Keys	Description	Example
	`permissions` command-line flag. Typically placed in managed settings to enforce organizational policy, but works from any scope	
`skipDangerousModePermissionPrompt`	Skip the confirmation prompt shown before entering bypass permissions mode via `--dangerously-skip-permissions` or `defaultMode: "bypassPermissions"`. Ignored when set in project settings (`.claude/settings.json`) to prevent untrusted repositories from auto-bypassing the prompt	`true`

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`

Rule	Effect
`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`
`failIfUnavailable`	Exit with an error at startup if `sandbox.enabled` is true but the sandbox cannot start (missing dependencies, unsupported platform, or platform restrictions). When false (default), a warning is shown and commands run unsandboxed. Intended for managed settings deployments that require sandboxing as a hard gate	`true`
`autoAllowBashIfSandboxed`	Auto-approve bash commands when sandboxed. Default: true	`true`
`excludedCommands`	Commands that should run outside of the sandbox	`["dock`
`allowUnsandboxedCommands`	Allow commands to run outside the sandbox via the	`false`

Keys	Description	Example
	`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	
`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` `"~/.ku`
`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/lo` `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.	`["~/.a` `creden`
`filesystem.allowRead`	Paths to re-allow reading within `denyRead` regions. Takes	`["."]`

Keys	Description	Example
	precedence over `denyRead`. Arrays are merged across all settings scopes. Use this to create workspace-only read access patterns.	
`filesystem.allowManagedReadPathsOnly`	(Managed settings only) Only `filesystem.allowRead` paths from managed settings are respected. `denyRead` still merges from all sources. Default: false	`true`
`network.allowUnixSockets`	Unix socket paths accessible in sandbox (for SSH agents, etc.)	`["~/.s` `agent-`
`network.allowAllUnixSockets`	Allow all Unix socket connections in sandbox. Default: false	`true`
`network.allowLocalBinding`	Allow binding to localhost ports (macOS only). Default: false	`true`
`network.allowedDomains`	Array of domains to allow for outbound network traffic. Supports wildcards (e.g., `*.example.com`).	`["gith` `"*.npm`
`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`

Keys	Description	Example
`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` , `filesystem.denyRead` , and `filesystem.allowRead` support these prefixes:

Prefix	Meaning	Example
`/`	Absolute path from filesystem root	`/tmp/build` stays `/tmp/build`

Prefix	Meaning	Example
`~/`	Relative to home directory	`~/.kube` becomes `$HOME/.kube`
`./` or no prefix	Relative to the project root for project settings, or to `~/.claude` for user settings	`./output` in `.claude/settings.json` resolves to `/output`

The older `//path` prefix for absolute paths still works. If you previously used single-slash `/path` expecting project-relative resolution, switch to `./path`. This syntax differs from Read and Edit permission rules, which use `//path` for absolute and `/path` for project-relative. Sandbox filesystem paths use standard conventions: `/tmp/build` is an absolute path.

Configuration example:

```json theme={null}
{
"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 tha

* **`sandbox.filesystem` settings** (shown above): Control paths at the
* **Permission rules**: Use `Edit` allow/deny rules to control Claude's

### Attribution settings

Claude Code adds attribution to git commits and pull requests. These are

* Commits use [git trailers](https://git-scm.com/docs/git-interpret-trai
* Pull request descriptions are plain text

| Keys     | Description
| :------- | :-----------------------------------------------------
| `commit` | Attribution for git commits, including any trailers. Empty
| `pr`     | Attribution for pull request descriptions. Empty string hid

**Default commit attribution:**

```text  theme={null}
 Generated with [Claude Code](https://claude.com/claude-code)

   Co-Authored-By: Claude Sonnet 4.6
```

Default pull request attribution:

```text theme={null}
 Generated with Claude Code
```
```

```
**Example:**

```json  theme={null}
{
  "attribution": {
    "commit": "Generated with AI\n\nCo-Authored-By: AI ",
    "pr": ""
  }
}
```

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.

```json theme={null}
{
"fileSuggestion": {
"type": "command",
"command": "~/.claude/file-suggestion.sh"
}
}
```

```
The command runs with the same environment variables as [hooks](/en/hooks

```json  theme={null}
{"query": "src/comp"}
```

Output newline-separated file paths to stdout (currently limited to 15):

```text theme={null}
src/components/Button.tsx
src/components/Modal.tsx
src/components/Form.tsx
```

> **Example:**
>
> ```bash  theme={null}
> #!/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.

```json theme={null}
{
"allowedHttpHookUrls": ["https://hooks.example.com/*", "http://localhost:*"]
}
```

```
**Restrict HTTP hook environment variables:**

Limit which environment variable names HTTP hooks can interpolate into he

```json  theme={null}
{
  "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)
2. Policies deployed by IT through server delivery, MDM configuration profiles, registry policies, or managed settings files
3. Cannot be overridden by any other level, including command line arguments
4. Within the managed tier, precedence is: server-managed > MDM/OS-level policies > file-based (`managed-settings.d/*.json` + `managed-settings.json`) > HKCU registry (Windows only). Only one managed source is used; sources do not merge across tiers. Within the file-based tier, drop-in files and the base file are merged together.
5. **Command line arguments**
6. Temporary overrides for a specific session
7. **Local project settings** (`.claude/settings.local.json`)
8. Personal project-specific settings
9. **Shared project settings** (`.claude/settings.json`)
10. Team-shared project settings in source control
11. **User settings** (`~/.claude/settings.json`)
12. Personal global settings
```

This hierarchy ensures that organizational policies are always enforced while still allowing teams and individuals to customize their experience. The same precedence applies whether you run Claude Code from the CLI, the VS Code extension, or a JetBrains IDE.

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.

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:

```json theme={null}
{
"permissions": {
"deny": [
"Read(./.env)",
"Read(./.env.*)",
"Read(./secrets/**)",
"Read(./config/credentials.json)",
"Read(./build)"
]
}
}
```

````
This replaces the deprecated `ignorePatterns` configuration. Files matchi

## Subagent configuration

Claude Code supports custom AI subagents that can be configured at both u

* **User subagents**: `~/.claude/agents/` - Available across all your pro
* **Project subagents**: `.claude/agents/` - Specific to your project and

Subagent files define specialized AI assistants with custom prompts and t

## Plugin configuration

Claude Code supports a plugin system that lets you extend functionality w

### Plugin settings

Plugin-related settings in `settings.json`:

```json  theme={null}
{
  "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)
- **Managed settings** (`managed-settings.json`): Organization-wide policy overrides that block installation at all scopes and hide the plugin from the marketplace

Example:

```json theme={null}
{
"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 re

**When a repository includes `extraKnownMarketplaces`**:

1. Team members are prompted to install the marketplace when they trust
2. Team members are then prompted to install plugins from that marketpla
3. Users can skip unwanted marketplaces or plugins (stored in user setti
4. Installation respects trust boundaries and requires explicit consent

**Example**:

```json  theme={null}
{
  "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`)
- `settings`: inline marketplace declared directly in settings.json without a separate hosted repository (uses `name` and `plugins`)

Use `source: 'settings'` to declare a small set of plugins inline without setting up a hosted marketplace repository. Plugins listed here must reference external sources such as GitHub or npm. You still need to enable each plugin separately in `enabledPlugins`.

```json theme={null}
{
"extraKnownMarketplaces": {
"team-tools": {
"source": {
"source": "settings",
"name": "team-tools",
"plugins": [
{
"name": "code-formatter",
"source": {
"source": "github",
"repo": "acme-corp/code-formatter"
}
}
]
}
}
}
}
```

```
#### `strictKnownMarketplaces`

**Managed settings only**: Controls which plugin marketplaces users are a

**Managed settings file locations**:

* **macOS**: `/Library/Application Support/ClaudeCode/managed-settings.js
* **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 ex
* Uses exact matching for source specifications (including `ref`, `path`

**Allowlist behavior**:

* `undefined` (default): No restrictions - users can add any marketplace
* Empty array `[]`: Complete lockdown - users cannot add any new marketp
* List of sources: Users can only add marketplaces that match exactly

**All supported source types**:

The allowlist supports multiple marketplace source types. Most sources us

1. **GitHub repositories**:

```json  theme={null}
{ "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"
```
```

Fields: `repo` (required), `ref` (optional: branch/tag/SHA), `path` (optional: subdirectory)

1. **Git repositories**:

```json theme={null}
{ "source": "git", "url": "https://gitlab.example.com/tools/plugins.git" }
{ "source": "git", "url": "https://bitbucket.org/acme-corp/plugins.git", "ref": "production" }
{ "source": "git", "url": "ssh://git@git.example.com/plugins.git", "ref": "v3.1", "path": "approved" }
```

```
Fields: `url` (required), `ref` (optional: branch/tag/SHA), `path` (optio

3. **URL-based marketplaces**:

```json  theme={null}
{ "source": "url", "url": "https://plugins.example.com/marketplace.json"
{ "source": "url", "url": "https://cdn.example.com/marketplace.json", "he
```

Fields: `url` (required), `headers` (optional: HTTP headers for authenticated access)

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**:

```json theme={null}
{ "source": "npm", "package": "@acme-corp/claude-plugins" }
{ "source": "npm", "package": "@acme-corp/approved-marketplace" }
```
```

```
Fields: `package` (required, supports scoped packages)

5. **File paths**:

```json  theme={null}
{ "source": "file", "path": "/usr/local/share/claude/acme-marketplace.jso
{ "source": "file", "path": "/opt/acme-corp/plugins/marketplace.json" }
```

Fields: `path` (required: absolute path to marketplace.json file)

1. **Directory paths**:

```json theme={null}
{ "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

7. **Host pattern matching**:

```json  theme={null}
{ "source": "hostPattern", "hostPattern": "^github\\.example\\.com$" }
{ "source": "hostPattern", "hostPattern": "^gitlab\\.internal\\.example\\
```

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:

```json theme={null}
{
"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:
>
> ```json  theme={null}
> {
>   "strictKnownMarketplaces": []
> }
> ```

Example: allow all marketplaces from an internal git server:

```json theme={null}
{
"strictKnownMarketplaces": [
{
"source": "hostPattern",
"hostPattern": "^github\.example\.com$"
}
]
}
```

```
**Exact matching requirements**:

Marketplace sources must match **exactly** for a user's addition to be a

* 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**:

```json  theme={null}
// 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		Any settings file
```

Aspect	`strictKnownMarketplaces`	`extraKnownMarketplaces`
	`managed-settings.json` only	
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:

```json theme={null}
{
"strictKnownMarketplaces": [
{ "source": "github", "repo": "acme-corp/plugins" }
]
}
```

````
`extraKnownMarketplaces` requires named marketplaces:

```json  theme={null}
{
  "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`:

```json theme={null}
{
"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

**Important notes**:

* Restrictions are checked BEFORE any network requests or filesystem ope
* When blocked, users see clear error messages indicating the source is
* The restriction applies only to adding NEW marketplaces; previously ins
* Managed settings have the highest precedence and cannot be overridden

See [Managed marketplace restrictions](/en/plugin-marketplaces#managed-ma

### 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](/en/plu

## Environment variables

Environment variables let you control Claude Code behavior without editin

See the [environment variables reference](/en/env-vars) for the full list

## Tools available to Claude

Claude Code has access to a set of tools for reading, editing, searching

See the [tools reference](/en/tools-reference) for the full list and Bash

## See also
```

````
* [Permissions](/en/permissions): permission system, rule syntax, tool-sp
* [Authentication](/en/authentication): set up user access to Claude Code
* [Troubleshooting](/en/troubleshooting): solutions for common configurat

---

# 故障排除

> ## Documentation Index
> Fetch the complete documentation index at: https://code.claude.com/docs
> Use this file to discover all available pages before exploring further

# Troubleshooting

> Discover solutions to common issues with Claude Code installation and c

## Troubleshoot installation issues

  If you'd rather skip the terminal entirely, the [Claude Code Desktop ap

Find the error message or symptom you're seeing:

| What you see                                                  | Solution
| :------------------------------------------------------------ | :-------
| `command not found: claude` or `'claude' is not recognized` | [Fix you
| `syntax error near unexpected token '

    ```bash  theme={null}
    echo $PATH | tr ':' '\n' | grep local/bin
    ```

    If there's no output, the directory is missing. Add it to your shell

    ```bash  theme={null}
    # 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:

```bash  theme={null}
claude --version
```



```powershell  theme={null}
$env:PATH -split ';' | Select-String 'local\\bin'
```

If there's no output, add the install directory to your User PATH:

```powershell  theme={null}
$currentPath = [Environment]::GetEnvironmentVariable('PATH', 'User')
[Environment]::SetEnvironmentVariable('PATH', "$currentPath;$env:USE
```

Restart your terminal for the change to take effect.

Verify the fix worked:

```powershell  theme={null}
claude --version
```
````
````

````
    ```batch  theme={null}
    echo %PATH% | findstr /i "local\bin"
    ```

    If there's no output, open System Settings, go to Environment Variab

    Verify the fix worked:

    ```batch  theme={null}
    claude --version
    ```


### Check for conflicting installations

Multiple Claude Code installations can cause version mismatches or unexpe


    List all `claude` binaries found in your PATH:

    ```bash  theme={null}
    which -a claude
    ```

    Check whether the native installer and npm versions are present:

    ```bash  theme={null}
    ls -la ~/.local/bin/claude
    ```

    ```bash  theme={null}
    ls -la ~/.claude/local/
    ```

    ```bash  theme={null}
    npm -g ls @anthropic-ai/claude-code 2>/dev/null
````
````

````
    ```



    ```powershell  theme={null}
    where.exe claude
    Test-Path "$env:LOCALAPPDATA\Claude Code\claude.exe"
    ```


If you find multiple installations, keep only one. The native install at

Uninstall an npm global install:

```bash  theme={null}
npm uninstall -g @anthropic-ai/claude-code
````

Remove a Homebrew install on macOS:

```bash theme={null}
brew uninstall --cask claude-code
```

````
### Check directory permissions

The installer needs write access to `~/.local/bin/` and `~/.claude/`. If

```bash  theme={null}
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:

```bash theme={null}
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 chec

Confirm the binary exists and is executable:

```bash  theme={null}
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.

```bash theme={null}
ldd $(which claude) | grep "not found"
```

Run a quick sanity check that the binary can execute:

```bash  theme={null}
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:

``text theme={null} bash: line 1: syntax error near unexpected token'

```
On PowerShell, the same problem appears as:

```text  theme={null}
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:

`bash theme={null} brew install --cask claude-code`

On Windows, install via WinGet:

`powershell theme={null} 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:

Platform	Error message
macOS	`zsh: command not found: claude`
Linux	`bash: claude: command not found`
Windows CMD	`'claude' is not recognized as an internal or external command`
```

Platform	Error message
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:
   `bash theme={null} 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.
2. **Try an alternative install method**:

On macOS or Linux:

`bash theme={null} brew install --cask claude-code`

On Windows:

`powershell theme={null} 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:

```
bash theme={null} sudo apt-get update && sudo apt-get install ca-certificates
```

On macOS via Homebrew:

```
bash theme={null} brew install ca-certificates
```

1. **On Windows, enable TLS 1.2** in PowerShell before running the installer:
   ```
   powershell theme={null} [Net.ServicePointManager]::SecurityProtocol = [Net.SecurityProtocolType]::Tls12 irm https://claude.ai/install.ps1 | iex
   ```

2. **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:
   ```
   bash theme={null} 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**:
   ```
   bash theme={null} curl -sI https://storage.googleapis.com
   ```

2. **If behind a proxy**, set `HTTPS_PROXY` so the installer can route through it. See proxy configuration for details.
   `bash theme={null} export HTTPS_PROXY=http://proxy.example.com:8080 curl -fsSL https://claude.ai/install.sh | bash`
3. **If on a restricted network**, try a different network or VPN, or use an alternative install method:

On macOS or Linux:

`bash theme={null} brew install --cask claude-code`

On Windows:

`powershell theme={null} 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:

`powershell theme={null} irm https://claude.ai/install.ps1 | iex`

Or stay in CMD and use the CMD installer instead:

`batch theme={null} 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:
  `powershell theme={null} 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:

```text theme={null}
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 o

**Solutions:**

1. **Add swap space** if your server has limited RAM. Swap uses disk spa

   Create a 2 GB swap file and enable it:

   ```bash  theme={null}
   sudo fallocate -l 2G /swapfile
   sudo chmod 600 /swapfile
   sudo mkswap /swapfile
   sudo swapon /swapfile
   ```

   Then retry the installation:

   ```bash  theme={null}
   curl -fsSL https://claude.ai/install.sh | bash
   ```

2. **Close other processes** to free memory before installing.

3. **Use a larger instance** if possible. Claude Code requires at least 4

### Install hangs in Docker

When installing Claude Code in a Docker container, installing as root in

**Solutions:**

1. **Set a working directory** before running the installer. When run fr
   ```dockerfile  theme={null}
   WORKDIR /tmp
   RUN curl -fsSL https://claude.ai/install.sh | bash
   ```
````

````
2. **Increase Docker memory limits** if using Docker Desktop:
   ```bash  theme={null}
   docker build --memory=4g .
   ```

### Windows: Claude Desktop overrides `claude` CLI command

If you installed an older version of Claude Desktop, it may register a `

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](https://git-scm.co

**If Git is not installed**, download and install it from [git-scm.com/d

**If Git is already installed** but Claude Code still can't find it, set

```json  theme={null}
{
  "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.

```text theme={null}
Error loading shared library libstdc++.so.6: No such file or directory
```
```

````
This can happen on glibc-based systems that have musl cross-compilation p

**Solutions:**

1. **Check which libc your system uses**:
   ```bash  theme={null}
   ldd /bin/ls | head -1
   ```
   If it shows `linux-vdso.so` or references to `/lib/x86_64-linux-gnu/`

2. **If you're on glibc but got the musl binary**, remove the installati

3. **If you're actually on musl** (Alpine Linux), install the required pa
   ```bash  theme={null}
   apk add libgcc libstdc++ ripgrep
   ```

### `Illegal instruction` on Linux

If the installer prints `Illegal instruction` instead of the OOM `Killed

```text  theme={null}
bash: line 142: 2238232 Illegal instruction    "$binary_path" install ${
````

Solutions:

1. **Verify your architecture**:
   `bash theme={null} 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.

2. **Try an alternative install method** while the architecture issue is resolved:
   `bash theme={null} 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.

```text theme={null}
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 late

2. **Update macOS** if you're on an older version. The binary uses load

3. **Try Homebrew** as an alternative install method:
   ```bash  theme={null}
   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 installa

* Run `npm config set os linux` before installation
* Install with `npm install -g @anthropic-ai/claude-code --force --no-os

**Node not found errors**: if you see `exec: node: not found` when runnin

**nvm version conflicts**: if you have nvm installed in both WSL and Win

You can identify this issue by:

* Running `which npm` and `which node` - if they point to Windows paths
* Experiencing broken functionality after switching Node versions with nv

To resolve this issue, fix your Linux PATH to ensure the Linux node/npm v

**Primary solution: Ensure nvm is properly loaded in your shell**

The most common cause is that nvm isn't loaded in non-interactive shells

```bash  theme={null}
````
````

```
# 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:

```bash theme={null}
source ~/.nvm/nvm.sh
```

```
**Alternative: Adjust PATH order**

If nvm is properly loaded but Windows paths still take priority, you can

```bash  theme={null}
export PATH="$HOME/.nvm/versions/node/$(node -v)/bin:$PATH"
```
```

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 about missing `bubblewrap` or `socat` when running `/sandbox`, install the dependencies:

```
 ```bash  theme={null}
sudo apt-get install bubblewrap socat
```



```bash  theme={null}
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:

```bash theme={null}
curl -fsSL https://claude.ai/install.sh | bash
```

```
## Permissions and authentication

These sections address login failures, token issues, and permission prom

### Repeated permission prompts

If you find yourself repeatedly approving the same commands, you can all
to run without approval using the `/permissions` command. See [Permissio

### 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 cop

### OAuth error: Invalid code

If you see `OAuth error: Invalid code. Please make sure the full code wa

**Solutions:**

* Press Enter to retry and complete the login quickly after the browser
* Type `c` to copy the full URL if the browser doesn't open automatical
* If using a remote/SSH session, the browser may open on the wrong machi

### 403 Forbidden after login

If you see `API Error: 403 {"error":{"type":"forbidden","message":"Reque

* **Claude Pro/Max users**: verify your subscription is active at [claud
* **Console users**: confirm your account has the "Claude Code" or "Deve
* **Behind a proxy**: corporate proxies can interfere with API requests.
```

````
### "This organization has been disabled" with an active subscription

If you see `API Error: 400 ... "This organization has been disabled"` des

When `ANTHROPIC_API_KEY` is present and you have approved it, Claude Code

To use your subscription instead, unset the environment variable and remo

```bash  theme={null}
unset ANTHROPIC_API_KEY
claude
````

Check `~/.zshrc`, `~/.bashrc`, or `~/.profile` for `export ANTHROPIC_API_KEY=...` lines and remove them to make the change permanent. Run `/status` inside Claude Code to confirm which authentication method is active.

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:

```bash theme={null}
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 co

### "Not logged in" or token expired

If Claude Code prompts you to log in again after a session, your OAuth t

Run `/login` to re-authenticate. If this happens frequently, check that

## 配置 file locations

Claude Code stores configuration in several locations:

| File                          | Purpose
| :---------------------------- | :-------------------------------------
| `~/.claude/settings.json`     | User settings (permissions, hooks, mod
| `.claude/settings.json`       | Project settings (checked into source
| `.claude/settings.local.json` | Local project settings (not committed)
| `~/.claude.json`              | Global state (theme, OAuth, MCP server
| `.mcp.json`                   | Project MCP servers (checked into sour
| `managed-mcp.json`            | [Managed MCP servers](/en/mcp#managed-
| Managed settings              | [Managed settings](/en/settings#settin

On Windows, `~` refers to your user home directory, such as `C:\Users\Yo

For details on configuring these files, see [Settings](/en/settings) and

### Resetting configuration

To reset Claude Code to default settings, you can remove the configurati

```bash  theme={null}
# Reset all user settings and state
rm ~/.claude.json
rm -rf ~/.claude/

# Reset project-specific settings
```
```

```
rm -rf .claude/
rm .mcp.json
```

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`:

```bash theme={null}

# 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](/en/env-vars).

### Slow or incomplete search results on WSL

Disk read performance penalties when [working across file systems on WSL

  `/doctor` will show Search as OK in this case.

**Solutions:**

1. **Submit more specific searches**: reduce the number of files searched

2. **Move project to Linux filesystem**: if possible, ensure your projec

3. **Use native Windows instead**: consider running Claude Code natively

## IDE integration issues

If Claude Code does not connect to your IDE or behaves unexpectedly with

### JetBrains IDE not detected on WSL2

If you're using Claude Code on WSL2 with JetBrains IDEs and getting "No a

#### WSL2 networking modes

WSL2 uses NAT networking by default, which can prevent IDE detection. You

**Option 1: Configure Windows Firewall** (recommended)

1. Find your WSL2 IP address:
   ```bash  theme={null}
   wsl hostname -I
   # Example output: 172.21.123.45
   ```
````

```
2. Open PowerShell as Administrator and create a firewall rule:
   ```powershell  theme={null}
   New-NetFirewallRule -DisplayName "Allow WSL2 Internal Traffic" -Direc
   ```
   Adjust the IP range based on your WSL2 subnet from step 1.

3. Restart both your IDE and Claude Code

Option 2: Switch to mirrored networking

Add to `.wslconfig` in your Windows user directory:

```ini  theme={null}
[wsl2]
networkingMode=mirrored
```

Then restart WSL with `wsl --shutdown` from PowerShell.

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:
3. Uncheck "Move focus to the editor with Escape", or
4. Click "Configure terminal keybindings" and delete the "Switch focus to Editor" shortcut
5. 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:

````markdown theme={null}
```
function example() {
  return "hello";
}
```
````

````
Instead of properly tagged blocks like:

```markdown  theme={null}
```javascript
function example() {
  return "hello";
}
```
````
````

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 `/feedback` 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:
4. Installation type, version, and search functionality
5. Auto-update status and available versions
6. Invalid settings files (malformed JSON, incorrect types)
7. MCP server configuration errors
8. Keybinding configuration problems
9. Context usage warnings (large CLAUDE.md files, high MCP token usage, unreachable permission rules)
10. Plugin and agent loading errors
11. Ask Claude directly about its capabilities and features - Claude has built-in access to its documentation

---

CLI 参考

> ## Documentation Index
>
> Fetch the complete documentation index at: https://code.claude.com/docs/llms.txt
> Use this file to discover all available pages before exploring further.

CLI reference

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

CLI 命令

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 "" "query"`	Resume session by ID or name	`claude -r "auth-refactor" "Finish this PR"`
	Update to latest version	`claude update`

Command	Description	Example
`claude update`		
`claude auth login`	Sign in to your Anthropic account. Use `--email` to pre-fill your email address, `--sso` to force SSO authentication, and `--console` to sign in with Anthropic Console for API usage billing instead of a Claude subscription	`claude auth login --console`
`claude auth logout`	Log out from your Anthropic account	`claude auth logout`
`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 auto-mode defaults`	Print the built-in auto mode classifier rules as JSON. Use `claude auto-mode config` to see your effective config with settings applied	`claude auto-mode defaults > rules.json`
`claude mcp`	Configure Model Context Protocol (MCP) servers	See the Claude Code MCP documentation.
`claude plugin`	Manage Claude Code plugins. Alias: `claude plugins`. See plugin reference for subcommands	`claude plugin install code-review@claude-plugins-official`
	Start a Remote Control server to control Claude Code from Claude.ai or	

Command	Description	Example
`claude remote-control`	the Claude app. Runs in server mode (no local interactive session). See Server mode flags	`claude remote-control --name "My Project"`

CLI 标志

Customize Claude Code's behavior with these command-line flags. `claude --help` does not list every flag, so a flag's absence from `--help` does not mean it is unavailable.

Flag	Description	Example
`--add-dir`	Add additional working directories for Claude to read and edit files. Grants file access; most `.claude/` configuration is not discovered from these directories. 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-a`
`--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 '{"reviewe {"description":"Reviews code","prompt":"You are a c reviewer"}}'`
`--allow-dangerously-skip-permissions`	Add `bypassPermissions` to the `Shift+Tab` mode cycle without starting in it. Lets you begin in a different mode like `plan` and	`claude --permission-mode p --allow-dangerously-skip-permissions`

Flag	Description	Example
	switch to `bypassPermissions` later. See permission modes	
`--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(gi diff *)" "Read"`
`--append-system-prompt`	Append custom text to the end of the default system prompt	`claude --append-system-pro "Always use TypeScript"`
`--append-system-prompt-file`	Load additional system prompt text from a file and append to the default prompt	`claude --append-system-pro file ./extra-rules.txt`
`--bare`	Minimal mode: skip auto-discovery of hooks, skills, plugins, MCP servers, auto memory, and CLAUDE.md so scripted calls start faster. Claude has access to Bash, file read, and file edit tools. Sets `CLAUDE_CODE_SIMPLE`. See bare mode	`claude --bare -p "query"`
`--betas`	Beta headers to include in API requests (API key users only)	`claude --betas interleaved thinking`
`--channels`	(Research preview) MCP servers whose channel notifications Claude should listen for in this session. Space-separated list of `plugin:@` entries. Requires Claude.ai authentication	`claude --channels plugin:m notifier@my-marketplace`
`--chrome`	Enable Chrome browser integration for web automation and testing	`claude --chrome`

Flag	Description	Example
`--continue`, `-c`	Load the most recent conversation in the current directory	`claude --continue`
`--dangerously-load-development-channels`	Enable channels that are not on the approved allowlist, for local development. Accepts `plugin:@` and `server:` entries. Prompts for confirmation	`claude --dangerously-load-development-channels server:webhook`
`--dangerously-skip-permissions`	Skip permission prompts. Equivalent to `--permission-mode bypassPermissions`. See permission modes for what this does and does not skip	`claude --dangerously-skip-permissions`
`--debug`	Enable debug mode with optional category filtering (for example, `"api,hooks"` or `"!statsig,!file"`)	`claude --debug "api,mcp"`
`--debug-file`	Write debug logs to a specific file path. Implicitly enables debug mode. Takes precedence over `CLAUDE_CODE_DEBUG_LOGS_DIR`	`claude --debug-file /tmp/claude-debug.log`
`--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"`
`--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`

Flag	Description	Example
`--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 --f 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-hook-events`	Include all hook lifecycle events in the output stream. Requires `--output-format stream-json`	`claude -p --output-format stream-json --include-hook-events "query"`
`--include-partial-messages`	Include partial streaming events in output. Requires `--print` and `--output-format stream-json`	`claude -p --output-format stream-json --include-parti messages "query"`
`--input-format`	Specify input format for print mode (options: `text`, `stream-json`)	`claude -p --output-format --input-format stream-json`
`--json-schema`	Get validated JSON output matching a JSON Schema after agent	

Flag	Description	Example
	completes its workflow (print mode only, see structured outputs)	`claude -p --json-schema '{"type":"object","properti {...}}' "query"`
`--maintenance`	Run maintenance hooks and start interactive mode	`claude --maintenance`
`--max-budget-usd`	Maximum dollar amount to spend on API calls before stopping (print mode only)	`claude -p --max-budget-usd "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.`
`--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` . `/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	`claude -p --no-session-persistence "query"`

Flag	Description	Example
	cannot be resumed (print mode only)	
`--output-format`	Specify output format for print mode (options: `text`, `json`, `stream-json`)	`claude -p "query" --output-format json`
`--enable-auto-mode`	Unlock auto mode in the `Shift+Tab` cycle. Requires a Team, Enterprise, or API plan and Claude Sonnet 4.6 or Opus 4.6	`claude --enable-auto-mode`
`--permission-mode`	Begin in a specified permission mode. Accepts `default`, `acceptEdits`, `plan`, `auto`, `dontAsk`, or `bypassPermissions`. Overrides `defaultMode` from settings files	`claude --permission-mode p`
`--permission-prompt-tool`	Specify an MCP tool to handle permission prompts in non-interactive mode	`claude -p --permission-pro 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 l bug"`

Flag	Description	Example
`--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"`
`--replay-user-messages`	Re-emit user messages from stdin back on stdout for acknowledgment. Requires `--input-format stream-json` and `--output-format stream-json`	`claude -p --input-format stream-json --output-format stream-json --replay-user-messages`
`--resume`, `-r`	Resume a specific session by ID or name, or show an interactive picker to choose a session	`claude --resume auth-refac`
`--session-id`	Use a specific session ID for the conversation (must be a valid UUID)	`claude --session-id "550e8 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`
`--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 a Python expert"`
`--system-prompt-file`	Load system prompt from a file, replacing the default prompt	`claude --system-prompt-file custom-prompt.txt`

Flag	Description	Example
`--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 Choose a display mode	`claude --teammate-mode in-process`
`--tmux`	Create a tmux session for the worktree. Requires `--worktree`. Uses iTerm2 native panes when available; pass `--tmux=classic` for traditional tmux	`claude -w feature-auth --t`
`--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 `/.claude/worktrees/`. 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"`
`--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

---

技能扩展

> ## Documentation Index
>
> Fetch the complete documentation index at: https://code.claude.com/docs/llms.txt
> Use this file to discover all available pages before exploring further.

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`.

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, `` ` indicates a required argument and [arg]` `` indicates an optional one.

Skill	Purpose
`/batch`	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]`	Enable debug logging for the current session and troubleshoot issues by reading the session debug log. Debug logging is off by default unless you started with `claude --debug`, so running `/debug` mid-session starts capturing logs from that point forward. Optionally describe the issue to focus the analysis
`/loop [interval]`	Run a prompt repeatedly on an interval while the session stays open. Useful for polling a deployment, babysitting a PR, or

Skill	Purpose
	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`.

````
Create a directory for the skill in your personal skills folder. Persona

```bash  theme={null}
mkdir -p ~/.claude/skills/explain-code
```



Every skill needs a `SKILL.md` file with two parts: YAML frontmatter (be

Create `~/.claude/skills/explain-code/SKILL.md`:

```yaml  theme={null}
---
name: explain-code
description: Explains code with visual diagrams and analogies. Use when
---

When explaining code, always include:

1. **Start with an analogy**: Compare the code to something from everyda
2. **Draw a diagram**: Use ASCII art to show the flow, structure, or rel
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 ana
```



You can test it two ways:

**Let Claude invoke it automatically** by asking something that matches

```text  theme={null}
How does this code work?
```
````

````
**Or invoke it directly** with the skill name:

```text  theme={null}
/explain-code src/auth/login.ts
```

Either way, Claude should include an analogy and ASCII diagram in its ex
````

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.md`	All your projects
Project	`.claude/skills//SKILL.md`	This project only
Plugin	`/skills//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:

```text theme={null}
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 file

  Files in `.claude/commands/` still work and support the same [frontmatt

#### Skills from additional directories

The `--add-dir` flag [grants file access](/en/permissions#additional-dire

Other `.claude/` configuration such as subagents, commands, and output st

  CLAUDE.md files from `--add-dir` directories are not loaded by default

## Configure skills

Skills are configured through YAML frontmatter at the top of `SKILL.md` a

### Types of skill content

Skill files can contain any instructions, but thinking about how you want

**Reference content** adds knowledge Claude applies to your current work

```yaml  theme={null}
---
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.

```yaml theme={null}

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 t

### Frontmatter reference

Beyond the markdown content, you can configure skill behavior using YAML

```yaml  theme={null}
---
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. Front-load the key use case: descriptions longer than 250 characters are truncated in the skill listing to reduce context usage.
`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. Accepts a space-separated string or a YAML list.
`model`	No	Model to use when this skill is active.
`effort`	No	Effort level when this skill is active. Overrides the session effort level. Default: inherits from session. Options: `low`, `medium`, `high`, `max` (Opus 4.6 only).
`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.
`paths`	No	Glob patterns that limit when this skill is activated. Accepts a comma-separated string or a YAML list. When set, Claude loads the skill automatically only when working with files matching the patterns. Uses the same format as path-specific rules.
`shell`	No	Shell to use for `!`command`` blocks in this skill. Accepts `bash` (default) or `powershell`. Setting `powershell` runs inline shell commands via PowerShell on Windows. Requires `CLAUDE_CODE_USE_POWERSHELL_TOOL=1`.

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:`.
`$ARGUMENTS[N]`	Access a specific argument by 0-based index, such as `$ARGUMENTS[0]` for the first argument.
`$N`	Shorthand for `$ARGUMENTS[N]`, such as `$0` for the first argument or `$1` for the second.

Variable	Description
`${CLAUDE_SESSION_ID}`	The current session ID. Useful for logging, creating session-specific files, or correlating skill output with sessions.
`${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:

```yaml theme={null}

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.m

```text  theme={null}
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:
```

```markdown theme={null}

## Additional resources

- For complete API details, see reference.md
- For usage examples, see examples.md
```

````
Keep `SKILL.md` under 500 lines. Move detailed reference material to sepa

### Control who invokes a skill

By default, both you and Claude can invoke any skill. You can type `/skil

* **`disable-model-invocation: true`**: Only you can invoke the skill. Us

* **`user-invocable: false`**: Only Claude can invoke the skill. Use this

This example creates a deploy skill that only you can trigger. The `disab

```yaml  theme={null}
---
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:
````

Frontmatter	You can invoke	Claude can invoke	When loaded into context
(default)	Yes	Yes	Description always in context, full skill loads when invoked
`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

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:

```yaml theme={null}

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 a

This skill fixes a GitHub issue by number. The `$ARGUMENTS` placeholder g

```yaml  theme={null}
---
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:` 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`:

```yaml theme={null}

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]

```yaml  theme={null}
---
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 `!` `` 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:

```yaml theme={null}

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 `` !`` `` 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 th

  To enable [extended thinking](/en/common-workflows#use-extended-thinkin

### Run skills in a subagent

Add `context: fork` to your frontmatter when you want a skill to run in i

  `context: fork` only makes sense for skills with explicit instructions

Skills and [subagents](/en/sub-agents) work together in two directions:

| Approach                     | System prompt
| :--------------------------- | :-------------------------------------
| Skill with `context: fork`   | From agent type (`Explore`, `Plan`, etc
| Subagent with `skills` field | Subagent's markdown body

With `context: fork`, you write the task in your skill and pick an agent

#### Example: Research skill using Explore agent

This skill runs research in a forked Explore agent. The skill content be

```yaml  theme={null}
---
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` :

```text theme={null}

# Add to deny rules:

Skill
```

```
**Allow or deny specific skills** using [permission rules](/en/permissio

```text  theme={null}
# 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.

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:

```bash theme={null}
mkdir -p ~/.claude/skills/codebase-visualizer/scripts
```

```
Create `~/.claude/skills/codebase-visualizer/SKILL.md`. The description

````yaml  theme={null}
---
name: codebase-visualizer
description: Generate an interactive collapsible tree visualization of yo
allowed-tools: Bash(python *)
---

# Codebase Visualizer

Generate an interactive HTML tree view that shows your project's file st

## 使用方法

Run the visualization script from your project root:

```bash
python ~/.claude/skills/codebase-visualizer/scripts/visualize.py .
```

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:

```python expandable theme={null}
#!/usr/bin/env python3
"""Generate an interactive collapsible tree visualization of a codebase."""

import json
import sys
import webbrowser
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><b>{ext}</b></div>'
f"<div>
f'{(size/total_size)*100:.1f}%'
for ext, size in sorted_exts
)
def fmt(b):
if b
```

Codebase Explorer

```
 body {{ font: 14px/1.5 system-ui, sans-serif; margin: 0; background: #1a
.container {{ display: flex; height: 100vh; }}
.sidebar {{ width: 280px; background: #252542; padding: 20px; border-rig
.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-transfor
.stat {{ display: flex; justify-content: space-between; padding: 8px 0; 
.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-ra
.file:hover {{ background: #2d2d44; }}
.size {{ color: #888; margin-left: auto; font-size: 12px; }}
.dot {{ width: 8px; height: 8px; border-radius: 50%; margin-right: 8px; 




  Summary
  Files{stats["files"]:,}
  Directories{stats["dirs"]:,}
  Total size{fmt(data["size"])}
  File types{len(stats["extensions"])}
  By file type
  {lang_bars}


  {data["name"]}
```

```
const data = {json.dumps(data)};
const colors = {json.dumps(colors)};
function fmt(b) {{ if (b □ ${{node.name}}${{fmt(node.size)}}`;
    const ul = document.createElement('ul'); ul.className = 'tree';
    node.children.sort((a,b) => (b.children?1:0)-(a.children?1:0) || a.na
    node.children.forEach(c => render(c, ul));
    det.appendChild(ul);
    const li = document.createElement('li'); li.appendChild(det); parent
  }} else {{
    const li = document.createElement('li'); li.className = 'file';
    li.innerHTML = `${{node.name}}${{fmt(node.size)}}`;
    parent.appendChild(li);
  }}
}}
data.children.forEach(c => render(c, document.getElementById('root')));
```

'''

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 codebas

This pattern works for any visual output: dependency graphs, test covera

## 故障排除

### 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 invocati

### Skill descriptions are cut short

Skill descriptions are loaded into context so Claude knows what's availa

To raise the limit, set the `SLASH_COMMAND_TOOL_CHAR_BUDGET` environment

## Related resources

* **[Subagents](/en/sub-agents)**: delegate tasks to specialized agents
* **[Plugins](/en/plugins)**: package and distribute skills with other e
* **[Hooks](/en/hooks)**: automate workflows around tool events
* **[Memory](/en/memory)**: manage CLAUDE.md files for persistent contex
* **[Built-in commands](/en/commands)**: reference for built-in `/` comma
* **[Permissions](/en/permissions)**: control tool and skill access
```

```
---

# 钩子参考

> ## Documentation Index
> Fetch the complete documentation index at: https://code.claude.com/docs
> Use this file to discover all available pages before exploring further

# Hooks reference

> Reference for Claude Code hook events, configuration schema, JSON input

  For a quickstart guide with examples, see [Automate workflows with hook

Hooks are user-defined shell commands, HTTP endpoints, or LLM prompts tha

## Hook lifecycle

Hooks fire at specific points during a Claude Code session. When an event




The table below summarizes when each event fires. The [Hook events](#hook

| Event                | When it fires
| :------------------- | :-------------------------------------------
| `SessionStart`       | When a session begins or resumes
| `UserPromptSubmit`   | When you submit a prompt, before Claude processe
| `PreToolUse`         | Before a tool call executes. Can block it
| `PermissionRequest`  | When a permission dialog appears
| `PermissionDenied`   | When a tool call is denied by the auto mode clas
| `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
| `TaskCreated`        | When a task is being created via `TaskCreate`
| `TaskCompleted`      | When a task is being marked as completed
| `Stop`               | When Claude finishes responding
| `StopFailure`        | When the turn ends due to an API error. Output a
| `TeammateIdle`       | When an [agent team](/en/agent-teams) teammate i
| `InstructionsLoaded` | When a CLAUDE.md or `.claude/rules/*.md` file is
| `ConfigChange`       | When a configuration file changes during a sessi
| `CwdChanged`         | When the working directory changes, for example
| `FileChanged`        | When a watched file changes on disk. The `matche
| `WorktreeCreate`     | When a worktree is being created via `--worktree
| `WorktreeRemove`     | When a worktree is being removed, either at sess
| `PreCompact`         | Before context compaction
| `PostCompact`        | After context compaction completes
| `Elicitation`        | When an MCP server requests user input during a
| `ElicitationResult`  | After a user responds to an MCP elicitation, bef
| `SessionEnd`         | When a session terminates

### How a hook resolves

To see how these pieces fit together, consider this `PreToolUse` hook tha

```json  theme={null}
{
  "hooks": {
    "PreToolUse": [
      {
        "matcher": "Bash",
        "hooks": [
          {
            "type": "command",
            "if": "Bash(rm *)",
            "command": "\"$CLAUDE_PROJECT_DIR\"/.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`:

```bash theme={null}

# !/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

    The `PreToolUse` event fires. Claude Code sends the tool input as JS

    ```json  theme={null}
    { "tool_name": "Bash", "tool_input": { "command": "rm -rf /tmp/build
    ```

    The matcher `"Bash"` matches the tool name, so this hook group activa

    The `if` condition `"Bash(rm *)"` matches because the command starts

    The script inspects the full command and finds `rm -rf`, so it prints

    ```json  theme={null}
    {
      "hookSpecificOutput": {
        "hookEventName": "PreToolUse",
        "permissionDecision": "deny",
        "permissionDecisionReason": "Destructive command blocked by hook
      }
    }
    ```

    If the command had been a safer `rm` variant like `rm file.txt`, the
````

```
    Claude Code reads the JSON decision, blocks the tool call, and shows


The [Configuration](#configuration) section below documents the full sch

## 配置

Hooks are defined in JSON settings files. The configuration has three le

1. Choose a [hook event](#hook-events) to respond to, like `PreToolUse` 
2. Add a [matcher group](#matcher-patterns) to filter when it fires, lik
3. Define one or more [hook handlers](#hook-handler-fields) to run when 

See [How a hook resolves](#how-a-hook-resolves) above for a complete wal

  This page uses specific terms for each level: **hook event** for the l

### Hook locations

Where you define a hook determines its scope:

| Location                                                    | Scope    
| :---------------------------------------------------------- | :--------
| `~/.claude/settings.json`                                   | All your 
| `.claude/settings.json`                                     | Single pr
| `.claude/settings.local.json`                               | Single pr
| Managed policy settings                                     | Organizat
| [Plugin](/en/plugins) `hooks/hooks.json`                    | When plug
| [Skill](/en/skills) or [agent](/en/sub-agents) frontmatter | While the

For details on settings file resolution, see [settings](/en/settings). E

### Matcher patterns

The `matcher` field is a regex string that filters when hooks fire. Use 
```

````
| Event
| :-----------------------------------------------------------------
| `PreToolUse`, `PostToolUse`, `PostToolUseFailure`, `PermissionRequest`
| `SessionStart`
| `SessionEnd`
| `Notification`
| `SubagentStart`
| `PreCompact`, `PostCompact`
| `SubagentStop`
| `ConfigChange`
| `CwdChanged`
| `FileChanged`
| `StopFailure`
| `InstructionsLoaded`
| `Elicitation`
| `ElicitationResult`
| `UserPromptSubmit`, `Stop`, `TeammateIdle`, `TaskCreated`, `TaskComple

The matcher is a regex, so `Edit|Write` matches either tool and `Noteboo

This example runs a linting script only when Claude writes or edits a fi

```json  theme={null}
{
  "hooks": {
    "PostToolUse": [
      {
        "matcher": "Edit|Write",
        "hooks": [
          {
            "type": "command",
            "command": "/path/to/lint-check.sh"
          }
        ]
      }
    ]
````
````

```
    }
  }
```

`UserPromptSubmit`, `Stop`, `TeammateIdle`, `TaskCreated`, `TaskCompleted`, `WorktreeCreate`, `WorktreeRemove`, and `CwdChanged` don't support matchers and always fire on every occurrence. If you add a `matcher` field to these events, it is silently ignored.

For tool events, you can filter more narrowly by setting the `if` field on individual hook handlers. `if` uses permission rule syntax to match against the tool name and arguments together, so `"Bash(git *)"` runs only for `git` commands and `"Edit(*.ts)"` runs only for TypeScript files.

Match MCP tools

MCP server tools appear as regular tools in tool events (`PreToolUse`, `PostToolUse`, `PostToolUseFailure`, `PermissionRequest`, `PermissionDenied`), so you can match them the same way you match any other tool name.

MCP tools follow the naming pattern `mcp____`, 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:

```json theme={null}
{
"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 comma

* **[Command hooks](#command-hook-fields)** (`type: "command"`): run a sh
* **[HTTP hooks](#http-hook-fields)** (`type: "http"`): send the event's
* **[Prompt hooks](#prompt-and-agent-hook-fields)** (`type: "prompt"`): s
* **[Agent hooks](#prompt-and-agent-hook-fields)** (`type: "agent"`): spa

#### Common fields

These fields apply to all hook types:

| Field           | Required | Description
| :-------------- | :------- | :-----------------------------------
| `type`          | yes      | `"command"`, `"http"`, `"prompt"`, or `"ag
| `if`            | no       | Permission rule syntax to filter when this
| `timeout`       | no       | Seconds before canceling. Defaults: 600 fo
| `statusMessage` | no       | Custom spinner message displayed while the
| `once`          | no       | If `true`, runs only once per session then

#### Command hook fields

In addition to the [common fields](#common-fields), command hooks accept

| Field     | Required | Description
| :-------- | :------- | :-----------------------------------------
| `command` | yes      | Shell command to execute
| `async`   | no       | If `true`, runs in the background without blocki
| `shell`   | no       | Shell to use for this hook. Accepts `"bash"` (de

#### HTTP hook fields

In addition to the [common fields](#common-fields), HTTP hooks accept the

| Field           | Required | Description
```

```
| :--------------- | :------- | :-------------------------------------
| `url`            | yes      | URL to send the POST request to
| `headers`        | no       | Additional HTTP headers as key-value pai
| `allowedEnvVars` | no       | List of environment variable names that

Claude Code sends the hook's [JSON input](#hook-input-and-output) as the

Error handling differs from command hooks: non-2xx responses, connection

This example sends `PreToolUse` events to a local validation service, au

```json  theme={null}
{
  "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 installation directory, for scripts bundled with a plugin. Changes on each plugin update.
- `${CLAUDE_PLUGIN_DATA}` : the plugin's persistent data directory, for dependencies and state that should survive plugin updates.

  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:

  `json theme={null} { "hooks": { "PostToolUse": [ { "matcher": "Write|Edit", "hooks": [ { "type": "command", "command": "\"$CLAUDE_PROJECT_DIR\"/.claude/hooks/check-style.sh" } ] } ] } }`

  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:

`json theme={null} { "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:

```yaml theme={null}

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 configu

The menu displays all four hook types: `command`, `prompt`, `agent`, and

* `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, so

### 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 `"disableAllH

The `disableAllHooks` setting respects the managed settings hierarchy. If

Direct edits to hooks in settings files are normally picked up automatica

## Hook input and output

Command hooks receive JSON data via stdin and communicate results through

### Common input fields

Hook events receive these fields as JSON, in addition to event-specific f

| 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](/en/permissions#permissio
| `hook_event_name` | Name of the event that fired

When running with `--agent` or inside a subagent, two additional fields a

| Field         | Description
| :------------ | :-------------------------------------------------
| `agent_id`    | Unique identifier for the subagent. Present only when th
| `agent_type` | Agent name (for example, `"Explore"` or `"security-revie

For example, a `PreToolUse` hook for a Bash command receives this on stdi

```json  theme={null}
{
  "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:

```bash theme={null}

# !/bin/bash

---

# Reads JSON input from stdin, checks the command

---

command=$(jq -r '.tool_input.command' &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 d

| 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
| `Stop`               | Yes        | Prevents Claude from stopping, cont
| `SubagentStop`       | Yes        | Prevents the subagent from stopping
| `TeammateIdle`       | Yes        | Prevents the teammate from going id
| `TaskCreated`        | Yes        | Rolls back the task creation
| `TaskCompleted`      | Yes        | Prevents the task from being marked
| `ConfigChange`       | Yes        | Blocks the configuration change fro
| `StopFailure`        | No         | Output and exit code are ignored
| `PostToolUse`        | No         | Shows stderr to Claude (tool alread
| `PostToolUseFailure` | No         | Shows stderr to Claude (tool alread
| `PermissionDenied`   | No         | Exit code and stderr are ignored (d
| `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
| `CwdChanged`         | No         | Shows stderr to user only
| `FileChanged`        | 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
| `WorktreeCreate`     | Yes        | Any non-zero exit code causes workt
| `WorktreeRemove`     | No         | Failures are logged in debug mode o
| `InstructionsLoaded` | No         | Exit code is ignored

### HTTP response handling

HTTP hooks use HTTP status codes and response bodies instead of exit code
```

```
* **2xx with an empty body**: success, equivalent to exit code 0 with no
* **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]
* **Non-2xx status**: non-blocking error, execution continues
* **Connection failure or timeout**: non-blocking error, execution contin

Unlike command hooks, HTTP hooks cannot signal a blocking error through s

### JSON output

Exit codes let you allow or block, but JSON output gives you finer-graine

  You must choose one approach per hook, not both: either use exit codes

Your hook's stdout must contain only the JSON object. If your shell profi

Hook output injected into context (`additionalContext`, `systemMessage`,

The JSON object supports three kinds of fields:

* **Universal fields** like `continue` work across all events. These are
* **Top-level `decision` and `reason`** are used by some events to block
* **`hookSpecificOutput`** is a nested object for events that need riche

| Field            | Default | Description
| :--------------- | :------ | :-----------------------------------
| `continue`       | `true`  | If `false`, Claude stops processing entir
| `stopReason`     | none    | Message shown to the user when `continue`
| `suppressOutput` | `false` | If `true`, hides stdout from verbose mode
| `systemMessage`  | none    | Warning message shown to the user

To stop Claude entirely regardless of event type:

```json  theme={null}
{ "continue": false, "stopReason": "Build failed, fix errors before conti
```
```

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, TaskCreated, 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/defer), `permissionDecisionReason`
PermissionRequest	`hookSpecificOutput`	`decision.behavior` (allow/deny)
PermissionDenied	`hookSpecificOutput`	`retry: true` tells the model it may retry the denied tool call
WorktreeCreate	path return	Command hook prints path on stdout; HTTP hook returns `hookSpecificOutput.worktreePath`. Hook failure or missing path 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)

Events	Decision pattern	Key fields
WorktreeRemove, Notification, SessionEnd, PreCompact, PostCompact, InstructionsLoaded, StopFailure, CwdChanged, FileChanged	None	No decision control. Used for side effects like logging or cleanup

Here are examples of each pattern in action:

Used by `UserPromptSubmit`, `PostToolUse`, `PostToolUseFailure`, `Stop`,

```json  theme={null}
{
  "decision": "block",
  "reason": "Test suite must pass before proceeding"
}
```

Uses `hookSpecificOutput` for richer control: allow, deny, or escalate t

```json  theme={null}
{
  "hookSpecificOutput": {
    "hookEventName": "PreToolUse",
    "permissionDecision": "deny",
    "permissionDecisionReason": "Database writes are not allowed"
  }
}
```

Uses `hookSpecificOutput` to allow or deny a permission request on behal

```json  theme={null}
{
  "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`, an `agent_type` field contains the agent name.

```json theme={null}
{
"session_id": "abc123",
"transcript_path": "/Users/.../.claude/projects/.../
00893aaf-19fa-41d2-8238-13269b9b3ca0.jsonl",
"cwd": "/Users/...",
"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

| Field               | Description
| :------------------ | :-------------------------------------------
| `additionalContext` | String added to Claude's context. Multiple hooks

```json  theme={null}
{
  "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:

```bash theme={null}
# !/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 expo

```bash  theme={null}
#!/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 > "$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.

`CLAUDE_ENV_FILE` is available for SessionStart, CwdChanged, and FileChanged 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.

The matcher runs against `load_reason`. For example, use `"matcher": "session_start"` to fire only for files loaded at session start, or `"matcher": "path_glob_match|nested_traversal"` to fire only for lazy loads.
```

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
`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"`, `"include"`, or `"compact"`. The `"compact"` value fires when instruction files are re-loaded after a compaction event
`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

```json theme={null}
{
"session_id": "abc123",
"transcript_path": "/Users/.../.claude/projects/.../transcript.jsonl",
"cwd": "/Users/my-project",
"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 r

### UserPromptSubmit

Runs when the user submits a prompt, before Claude processes it. This al
to add additional context based on the prompt/conversation, validate pro
block certain types of prompts.

#### UserPromptSubmit input

In addition to the [common input fields](#common-input-fields), UserProm

```json  theme={null}
{
  "session_id": "abc123",
  "transcript_path": "/Users/.../.claude/projects/.../00893aaf-19fa-41d2
  "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

```json theme={null}
{
"decision": "block",
"reason": "Explanation for decision",
"hookSpecificOutput": {
"hookEventName": "UserPromptSubmit",
"additionalContext": "My additional context here"
}
}
```

```
  The JSON format isn't required for simple use cases. To add context, yo
  block prompts or want more structured control.

### PreToolUse

Runs after Claude creates tool parameters and before processing the tool

Use [PreToolUse decision control](#pretooluse-decision-control) to allow

#### PreToolUse input

In addition to the [common input fields](#common-input-fields), PreToolUs

##### Bash

Executes shell commands.

| Field               | Type    | Example            | Description
| :------------------ | :------ | :----------------- | :----------
| `command`           | string  | `"npm test"`       | The shell command
| `description`       | string  | `"Run test suite"` | Optional descripti
| `timeout`           | number  | `120000`           | Optional timeout i
| `run_in_background` | boolean | `false`            | Whether to run the

##### Write

Creates or overwrites a file.

| Field       | Type   | Example               | Description
| :---------- | :----- | :-------------------- | :----------
| `file_path` | string | `"/path/to/file.txt"` | Absolute path to the fi
| `content`   | string | `"file content"`      | Content to write to the

##### Edit

Replaces a string in an existing file.
```

```
| Field         | Type    | Example                | Description
| :------------ | :------ | :--------------------- | :------------------
| `file_path`   | string  | `"/path/to/file.txt"` | Absolute path to the
| `old_string`  | string  | `"original text"`     | Text to find and rep
| `new_string`  | string  | `"replacement text"`  | Replacement text
| `replace_all` | boolean | `false`               | Whether to replace a

##### Read

Reads file contents.

| Field       | Type   | Example                | Description
| :---------- | :----- | :--------------------- | :----------------------
| `file_path` | string | `"/path/to/file.txt"` | Absolute path to the fi
| `offset`    | number | `10`                   | Optional line number to
| `limit`     | number | `50`                   | Optional number of lines

##### Glob

Finds files matching a glob pattern.

| Field     | Type   | Example          | Description
| :-------- | :----- | :--------------- | :----------------------------
| `pattern` | string | `"**/*.ts"`      | Glob pattern to match files aga
| `path`    | string | `"/path/to/dir"` | Optional directory to search in

##### Grep

Searches file contents with regular expressions.

| Field         | Type    | Example          | Description
| :------------ | :------ | :--------------- | :------------------------
| `pattern`     | string  | `"TODO.*fix"`    | Regular expression pattern
| `path`        | string  | `"/path/to/dir"` | Optional file or directory
| `glob`        | string  | `"*.ts"`         | Optional glob pattern to
| `output_mode` | string  | `"content"`      | `"content"`, `"files_with_
```

```
| `-i`          | boolean | `true`           | Case insensitive search
| `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 conte
| `prompt` | string | `"Extract the API endpoints"` | Prompt to run on t

##### WebSearch

Searches the web.

| Field             | Type   | Example                       | Descripti
| :---------------- | :----- | :---------------------------- | :-------
| `query`           | string | `"react hooks best practices"` | Search q
| `allowed_domains` | array  | `["docs.example.com"]`        | Optional
| `blocked_domains` | array  | `["spam.example.com"]`        | Optional

##### Agent

Spawns a [subagent](/en/sub-agents).

| Field           | Type   | Example                     | Description
| :-------------- | :----- | :-------------------------- | :-------------
| `prompt`        | string | `"Find all API endpoints"` | The task for t
| `description`   | string | `"Find API endpoints"`     | Short descripti
| `subagent_type` | string | `"Explore"`                | Type of specia
| `model`         | string | `"sonnet"`                 | Optional model

##### AskUserQuestion

Asks the user one to four multiple-choice questions.
```

```
| Field       | Type   | Example
| :---------- | :----- | :-----------------------------------------
| `questions` | array  | `[{"question": "Which framework?", "header": "F
| `answers`   | object | `{"Which framework?": "React"}`

#### PreToolUse decision control

`PreToolUse` hooks can control whether a tool call proceeds. Unlike othe

| Field                      | Description
| :------------------------- | :-----------------------------------------
| `permissionDecision`       | `"allow"` skips the permission prompt. `"
| `permissionDecisionReason` | For `"allow"` and `"ask"`, shown to the us
| `updatedInput`             | Modifies the tool's input parameters befo
| `additionalContext`        | String added to Claude's context before th

When multiple PreToolUse hooks return different decisions, precedence is

When a hook returns `"ask"`, the permission prompt displayed to the user

```json  theme={null}
{
  "hookSpecificOutput": {
    "hookEventName": "PreToolUse",
    "permissionDecision": "allow",
    "permissionDecisionReason": "My reason here",
    "updatedInput": {
      "field_to_modify": "new value"
    },
    "additionalContext": "Current environment: production. Proceed with
  }
}
```

`AskUserQuestion` and `ExitPlanMode` require user interaction and normally block in non-interactive mode with the `-p` flag. Returning `permissionDecision: "allow"` together with `updatedInput` satisfies that requirement: the hook reads the tool's input from stdin, collects the answer through your own UI, and returns it in
```

`updatedInput` so the tool runs without prompting. Returning `"allow"` alone is not sufficient for these tools. For `AskUserQuestion`, echo back the original `questions` array and add an `answers` object mapping each question's text to the chosen answer.

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.

Defer a tool call for later

`"defer"` is for integrations that run `claude -p` as a subprocess and read its JSON output, such as an Agent SDK app or a custom UI built on top of Claude Code. It lets that calling process pause Claude at a tool call, collect input through its own interface, and resume where it left off. Claude Code honors this value only in non-interactive mode with the `-p` flag. In interactive sessions it logs a warning and ignores the hook result.

The `defer` value requires Claude Code v2.1.89 or later. Earlier versions do not recognize it and the tool proceeds through the normal permission flow.

The `AskUserQuestion` tool is the typical case: Claude wants to ask the user something, but there is no terminal to answer in. The round trip works like this:

1. Claude calls `AskUserQuestion`. The `PreToolUse` hook fires.
2. The hook returns `permissionDecision: "defer"`. The tool does not execute. The process exits with `stop_reason: "tool_deferred"` and the pending tool call preserved in the transcript.
3. The calling process reads `deferred_tool_use` from the SDK result, surfaces the question in its own UI, and waits for an answer.
4. The calling process runs `claude -p --resume`. The same tool call fires `PreToolUse` again.
5. The hook returns `permissionDecision: "allow"` with the answer in `updatedInput`. The tool executes and Claude continues.

The `deferred_tool_use` field carries the tool's `id`, `name`, and `input`. The `input` is the parameters Claude generated for the tool call, captured before execution:

```json theme={null}
{
"type": "result",
"subtype": "success",
"stop_reason": "tool_deferred",
"session_id": "abc123",
"deferred_tool_use": {
"id": "toolu_01abc",
"name": "AskUserQuestion",
"input": { "questions": [{ "question": "Which framework?", "header": "Framework",
"options": [{"label": "React"}, {"label": "Vue"}], "multiSelect": false }] }
}
}
```

```
There is no timeout or retry limit. The session remains on disk until you

`"defer"` only works when Claude makes a single tool call in the turn. If

If the deferred tool is no longer available when you resume, the process

  `--resume` does not restore the permission mode from the prior session

### PermissionRequest

Runs when the user is shown a permission dialog.
Use [PermissionRequest decision control](#permissionrequest-decision-cont

Matches on tool name, same values as PreToolUse.

#### PermissionRequest input

PermissionRequest hooks receive `tool_name` and `tool_input` fields like

```json  theme={null}
{
  "session_id": "abc123",
  "transcript_path": "/Users/.../.claude/projects/.../00893aaf-19fa-41d2
  "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. Replaces the entire input object, so include unchanged fields alongside modified ones
`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

```json theme={null}
{
"hookSpecificOutput": {
"hookEventName": "PermissionRequest",
"decision": {
"behavior": "allow",
"updatedInput": {
"command": "npm run lint"
}
}
```

```
}
}
```

```
#### Permission update entries

The `updatedPermissions` output field and the [`permission_suggestions` i

| `type`              | Fields                              | Effect
| :------------------ | :---------------------------------- | :----------
| `addRules`          | `rules`, `behavior`, `destination` | Adds permiss
| `replaceRules`      | `rules`, `behavior`, `destination` | Replaces al
| `removeRules`       | `rules`, `behavior`, `destination` | Removes mat
| `setMode`           | `mode`, `destination`               | Changes the
| `addDirectories`    | `directories`, `destination`       | Adds workin
| `removeDirectories` | `directories`, `destination`       | Removes wor

The `destination` field on every entry determines whether the change sta

| `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 ow

### 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.

```json  theme={null}
{
```
```

```
  "session_id": "abc123",
  "transcript_path": "/Users/.../.claude/projects/.../00893aaf-19fa-41d2
  "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

```json theme={null}
{
```

"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 th

Matches on tool name, same values as PreToolUse.

#### PostToolUseFailure input

PostToolUseFailure hooks receive the same `tool_name` and `tool_input` fi

```json  theme={null}
{
  "session_id": "abc123",
  "transcript_path": "/Users/.../.claude/projects/.../00893aaf-19fa-41d2
  "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

```json theme={null}
{
"hookSpecificOutput": {
"hookEventName": "PostToolUseFailure",
"additionalContext": "Additional information about the failure for Claude"
}
}
```

```
### PermissionDenied

Runs when the [auto mode](/en/permission-modes#eliminate-prompts-with-au

Matches on tool name, same values as PreToolUse.

#### PermissionDenied input

In addition to the [common input fields](#common-input-fields), Permissi

```json  theme={null}
{
  "session_id": "abc123",
  "transcript_path": "/Users/.../.claude/projects/.../00893aaf-19fa-41d2
  "cwd": "/Users/...",
  "permission_mode": "auto",
  "hook_event_name": "PermissionDenied",
  "tool_name": "Bash",
  "tool_input": {
    "command": "rm -rf /tmp/build",
    "description": "Clean build directory"
  },
  "tool_use_id": "toolu_01ABC123...",
  "reason": "Auto mode denied: command targets a path outside the projec
}
```

Field	Description
`reason`	The classifier's explanation for why the tool call was denied

PermissionDenied decision control

PermissionDenied hooks can tell the model it may retry the denied tool call. Return a JSON object with `hookSpecificOutput.retry` set to `true`:
```

```json theme={null}
{
"hookSpecificOutput": {
"hookEventName": "PermissionDenied",
"retry": true
}
}
```

````
When `retry` is `true`, Claude Code adds a message to the conversation te

### Notification

Runs when Claude Code sends notifications. Matches on notification type:

Use separate matchers to run different handlers depending on the notifica

```json  theme={null}
{
  "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.

```json theme={null}
{
"session_id": "abc123",
"transcript_path": "/Users/.../.claude/projects/.../
00893aaf-19fa-41d2-8238-13269b9b3ca0.jsonl",
"cwd": "/Users/...",
"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 t

| Field               | Description                     |
| :------------------ | :------------------------------ |
| `additionalContext` | String added to Claude's context |

### SubagentStart

Runs when a Claude Code subagent is spawned via the Agent tool. Supports

#### SubagentStart input

In addition to the [common input fields](#common-input-fields), SubagentS

```json  theme={null}
{
  "session_id": "abc123",
  "transcript_path": "/Users/.../.claude/projects/.../00893aaf-19fa-41d2
  "cwd": "/Users/...",
  "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

```json theme={null}
{
"hookSpecificOutput": {
"hookEventName": "SubagentStart",
"additionalContext": "Follow security guidelines for this task"

```
}
}
```

```
### SubagentStop

Runs when a Claude Code subagent has finished responding. Matches on agen

#### SubagentStop input

In addition to the [common input fields](#common-input-fields), SubagentS

```json  theme={null}
{
  "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
  "last_assistant_message": "Analysis complete. Found 3 potential issues
}
```

SubagentStop hooks use the same decision control format as Stop hooks.

TaskCreated

Runs when a task is being created via the `TaskCreate` tool. Use this to enforce naming conventions, require task descriptions, or prevent certain tasks from being created.

When a `TaskCreated` hook exits with code 2, the task is not created 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": "..."}`. TaskCreated hooks do not support matchers and fire on every occurrence.
```

TaskCreated input

In addition to the common input fields, TaskCreated hooks receive `task_id`, `task_subject`, and optionally `task_description`, `teammate_name`, and `team_name`.

```json theme={null}
{
"session_id": "abc123",
"transcript_path": "/Users/.../.claude/projects/.../
00893aaf-19fa-41d2-8238-13269b9b3ca0.jsonl",
"cwd": "/Users/...",
"permission_mode": "default",
"hook_event_name": "TaskCreated",
"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 created
| `task_subject`     | Title of the task
| `task_description` | Detailed description of the task. May be absent
| `teammate_name`    | Name of the teammate creating the task. May be abs
| `team_name`        | Name of the team. May be absent

#### TaskCreated decision control

TaskCreated hooks support two ways to control task creation:

* **Exit code 2**: the task is not created and the stderr message is fed
* **JSON `{"continue": false, "stopReason": "..."}`**: stops the teammate

This example blocks tasks whose subjects don't follow the required forma

```bash  theme={null}
#!/bin/bash
INPUT=$(cat)
TASK_SUBJECT=$(echo "$INPUT" | jq -r '.task_subject')

if [[ ! "$TASK_SUBJECT" =~ ^\[TICKET-[0-9]+\] ]]; then
  echo "Task subject must start with a ticket number, e.g. '[TICKET-123]
  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`.

```json theme={null}
{
"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 a
| `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 mes
* **JSON `{"continue": false, "stopReason": "..."}`**: stops the teammate

This example runs tests and blocks task completion if they fail:

```bash  theme={null}
#!/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_SU
  exit 2
fi

exit 0
```

Stop

Runs when the main Claude Code agent has finished responding. Does not run if the stoppage occurred due to a user interrupt. API errors fire StopFailure instead.
```

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.

```json theme={null}
{
"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

| Field      | Description
| :--------- | :-----------------------------------------------------
| `decision` | `"block"` prevents Claude from stopping. Omit to allow Cl
| `reason`   | Required when `decision` is `"block"`. Tells Claude why i

```json  theme={null}
{
  "decision": "block",
  "reason": "Must be provided when Claude is blocked from stopping"
}
```
```

StopFailure

Runs instead of Stop when the turn ends due to an API error. Output and exit code are ignored. Use this to log failures, send alerts, or take recovery actions when Claude cannot complete a response due to rate limits, authentication problems, or other API errors.

StopFailure input

In addition to the common input fields, StopFailure hooks receive `error`, optional `error_details`, and optional `last_assistant_message`. The `error` field identifies the error type and is used for matcher filtering.

Field	Description
`error`	Error type: `rate_limit`, `authentication_failed`, `billing_error`, `invalid_request`, `server_error`, `max_output_tokens`, or `unknown`
`error_details`	Additional details about the error, when available
`last_assistant_message`	The rendered error text shown in the conversation. Unlike `Stop` and `SubagentStop`, where this field holds Claude's conversational output, for `StopFailure` it contains the API error string itself, such as `"API Error: Rate limit reached"`

```json theme={null}
{
"session_id": "abc123",
"transcript_path": "/Users/.../.claude/projects/.../
00893aaf-19fa-41d2-8238-13269b9b3ca0.jsonl",
"cwd": "/Users/...",
"hook_event_name": "StopFailure",
"error": "rate_limit",
"error_details": "429 Too Many Requests",
```

"last_assistant_message": "API Error: Rate limit reached"
}

```
StopFailure hooks have no decision control. They run for notification and

### TeammateIdle

Runs when an [agent team](/en/agent-teams) teammate is about to go idle a

When a `TeammateIdle` hook exits with code 2, the teammate receives the s

#### TeammateIdle input

In addition to the [common input fields](#common-input-fields), Teammate

```json  theme={null}
{
  "session_id": "abc123",
  "transcript_path": "/Users/.../.claude/projects/.../00893aaf-19fa-41d2
  "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:

```bash theme={null}
#!/bin/bash

if [ ! -f "./dist/output.js" ]; then
echo "Build artifact missing. Run the build before stopping." >&2
exit 2
fi

exit 0
```

````
### ConfigChange

Runs when a configuration file changes during a session. Use this to audi

ConfigChange hooks fire for changes to settings files, managed policy se

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:

```json  theme={null}
{
  "hooks": {
    "ConfigChange": [
      {
        "hooks": [
          {
            "type": "command",
            "command": "\"$CLAUDE_PROJECT_DIR\"/.claude/hooks/audit-confi
          }
        ]
      }
    ]
  }
}
````
````

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.

```json theme={null}
{
"session_id": "abc123",
"transcript_path": "/Users/.../.claude/projects/.../
00893aaf-19fa-41d2-8238-13269b9b3ca0.jsonl",
"cwd": "/Users/...",
"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. Us

| Field      | Description
| :--------- | :-----------------------------------------------------
| `decision` | `"block"` prevents the configuration change from being app
| `reason`   | Explanation shown to the user when `decision` is `"block"

```json  theme={null}
{
  "decision": "block",
  "reason": "Configuration changes to project settings require admin app
}
```

`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.
```

CwdChanged

Runs when the working directory changes during a session, for example when Claude executes a `cd` command. Use this to react to directory changes: reload environment variables, activate project-specific toolchains, or run setup scripts automatically. Pairs with FileChanged for tools like direnv that manage per-directory environment.

CwdChanged hooks have access to `CLAUDE_ENV_FILE`. Variables written to that file persist into subsequent Bash commands for the session, just as in SessionStart hooks. Only `type: "command"` hooks are supported.

CwdChanged does not support matchers and fires on every directory change.

CwdChanged input

In addition to the common input fields, CwdChanged hooks receive `old_cwd` and `new_cwd`.

```json theme={null}
{
"session_id": "abc123",
"transcript_path": "/Users/.../.claude/projects/.../transcript.jsonl",
"cwd": "/Users/my-project/src",
"hook_event_name": "CwdChanged",
"old_cwd": "/Users/my-project",
"new_cwd": "/Users/my-project/src"
}
```

````
#### CwdChanged output

In addition to the [JSON output fields](#json-output) available to all ho

| Field        | Description
| :----------- | :-----------------------------------------------------
| `watchPaths` | Array of absolute paths. Replaces the current dynamic wa

CwdChanged hooks have no decision control. They cannot block the directo

### FileChanged

Runs when a watched file changes on disk. The `matcher` field in your hoo

FileChanged hooks have access to `CLAUDE_ENV_FILE`. Variables written to

#### FileChanged input

In addition to the [common input fields](#common-input-fields), FileChang

| Field       | Description
| :---------- | :-----------------------------------------------------
| `file_path` | Absolute path to the file that changed
| `event`     | What happened: `"change"` (file modified), `"add"` (file

```json  theme={null}
{
  "session_id": "abc123",
  "transcript_path": "/Users/.../.claude/projects/.../transcript.jsonl",
  "cwd": "/Users/my-project",
  "hook_event_name": "FileChanged",
  "file_path": "/Users/my-project/.envrc",
  "event": "change"
}
````
````

FileChanged output

In addition to the JSON output fields available to all hooks, FileChanged hooks can return `watchPaths` to dynamically update which file paths are watched:

Field	Description
`watchPaths`	Array of absolute paths. Replaces the current dynamic watch list (paths from your `matcher` configuration are always watched). Use this when your hook script discovers additional files to watch based on the changed file

FileChanged hooks have no decision control. They cannot block the file change from occurring.

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.

Because the hook replaces the default behavior entirely, `.worktreeinclude` is not processed. If you need to copy local configuration files like `.env` into the new worktree, do it inside your hook script.

The hook must return the absolute path to the created worktree directory. Claude Code uses this path as the working directory for the isolated session. Command hooks print it on stdout; HTTP hooks return it via `hookSpecificOutput.worktreePath`.

This example creates an SVN working copy and prints the path for Claude Code to use. Replace the repository URL with your own:

```json theme={null}
{
"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

#### WorktreeCreate input

In addition to the [common input fields](#common-input-fields), Worktree

```json  theme={null}
{
  "session_id": "abc123",
  "transcript_path": "/Users/.../.claude/projects/.../00893aaf-19fa-41d2
  "cwd": "/Users/...",
  "hook_event_name": "WorktreeCreate",
  "name": "feature-auth"
}
```

WorktreeCreate output

WorktreeCreate hooks do not use the standard allow/block decision model. Instead, the hook's success or failure determines the outcome. The hook must return the absolute path to the created worktree directory:

- **Command hooks** (`type: "command"`): print the path on stdout.
- **HTTP hooks** (`type: "http"`): return `{ "hookSpecificOutput": { "hookEventName": "WorktreeCreate", "worktreePath": "/absolute/path" } }` in the response body.
```

If the hook fails or produces no path, worktree creation fails with an error.

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 returned by WorktreeCreate as `worktree_path` in the hook input. This example reads that path and removes the directory:

```json theme={null}
{
"hooks": {
"WorktreeRemove": [
{
"hooks": [
{
"type": "command",
"command": "bash -c 'jq -r .worktree_path | xargs rm -rf'"
}
]
}
]
}
}
```

```
#### WorktreeRemove input

In addition to the [common input fields](#common-input-fields), Worktree

```json  theme={null}
{
  "session_id": "abc123",
  "transcript_path": "/Users/.../.claude/projects/.../00893aaf-19fa-41d2
  "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.

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.

```json theme={null}
{

"session_id": "abc123",
"transcript_path": "/Users/.../.claude/projects/.../
00893aaf-19fa-41d2-8238-13269b9b3ca0.jsonl",
"cwd": "/Users/...",
"hook_event_name": "PreCompact",
"trigger": "manual",
"custom_instructions": ""
}

````
PostCompact

Runs after Claude Code completes a compact operation. Use this event to

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](#common-input-fields), PostCompa

```json  theme={null}
{
  "session_id": "abc123",
  "transcript_path": "/Users/.../.claude/projects/.../00893aaf-19fa-41d2
  "cwd": "/Users/...",
  "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
`resume`	Session switched via interactive `/resume`
`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.

```json theme={null}
{
"session_id": "abc123",
"transcript_path": "/Users/.../.claude/projects/.../
00893aaf-19fa-41d2-8238-13269b9b3ca0.jsonl",
"cwd": "/Users/...",
"hook_event_name": "SessionEnd",
"reason": "other"
}
```

```
SessionEnd hooks have no decision control. They cannot block session ter

SessionEnd hooks have a default timeout of 1.5 seconds. This applies to

```bash  theme={null}
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):

```json theme={null}
{
"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):

```json  theme={null}
{
  "session_id": "abc123",
  "transcript_path": "/Users/.../.claude/projects/.../00893aaf-19fa-41d2
  "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`:

```json theme={null}
{
"hookSpecificOutput": {
"hookEventName": "Elicitation",
"action": "accept",
"content": {
"username": "alice"
}
}
}
```
```

```
| Field     | Values                          | Description
| :-------- | :------------------------------ | :-----------------------
| `action`  | `accept`, `decline`, `cancel` | Whether to accept, decline
| `content` | object                          | Form field values to submi

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, modi

The matcher field matches against the MCP server name.

#### ElicitationResult input

In addition to the [common input fields](#common-input-fields), Elicitati

```json  theme={null}
{
  "session_id": "abc123",
  "transcript_path": "/Users/.../.claude/projects/.../00893aaf-19fa-41d2
  "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`:
```

```json theme={null}
{
"hookSpecificOutput": {
"hookEventName": "ElicitationResult",
"action": "decline",
"content": {}
}
}
```

```
| Field     | Values                          | Description
| :-------- | :------------------------------ | :-----------------------
| `action`  | `accept`, `decline`, `cancel` | Overrides the user's actio
| `content` | object                          | Overrides form field value

Exit code 2 blocks the response, changing the effective action to `decli

## Prompt-based hooks

In addition to command and HTTP hooks, Claude Code supports prompt-based

Events that support all four hook types (`command`, `http`, `prompt`, an

* `PermissionRequest`
* `PostToolUse`
* `PostToolUseFailure`
* `PreToolUse`
* `Stop`
* `SubagentStop`
* `TaskCompleted`
* `TaskCreated`
* `UserPromptSubmit`

Events that support `command` and `http` hooks but not `prompt` or `agen

* `ConfigChange`
* `CwdChanged`
* `Elicitation`
* `ElicitationResult`
* `FileChanged`
* `InstructionsLoaded`
* `Notification`
* `PermissionDenied`
* `PostCompact`
* `PreCompact`
* `SessionEnd`
```

````
* `StopFailure`
* `SubagentStart`
* `TeammateIdle`
* `WorktreeCreate`
* `WorktreeRemove`

`SessionStart` supports only `command` hooks.

### 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 defau
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 `comm

This `Stop` hook asks the LLM to evaluate whether all tasks are complete

```json  theme={null}
{
  "hooks": {
    "Stop": [
      {
        "hooks": [
          {
            "type": "prompt",
            "prompt": "Evaluate if Claude should stop: $ARGUMENTS. Check
          }
        ]
      }
    ]
  }
}
````
````

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
`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:

```json theme={null}
{
"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

```json  theme={null}
{
  "hooks": {
    "Stop": [
      {
        "hooks": [
          {
            "type": "prompt",
            "prompt": "You are evaluating whether Claude should stop work
            "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:

```json theme={null}
{
"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

### Configure an async hook

Add `"async": true` to a command hook's configuration to run it in the ba

This hook runs a test script after every `Write` tool call. Claude contin

```json  theme={null}
{
  "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`:

```bash theme={null}

# !/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 r

```json  theme={null}
{
  "hooks": {
    "PostToolUse": [
      {
        "matcher": "Write|Edit",
        "hooks": [
          {
            "type": "command",
            "command": "\"$CLAUDE_PROJECT_DIR\"/.claude/hooks/run-tests-
            "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.

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.

Windows PowerShell tool

On Windows, you can run individual hooks in PowerShell by setting `"shell": "powershell"` on a command hook. Hooks spawn PowerShell directly, so this works regardless of whether `CLAUDE_CODE_USE_POWERSHELL_TOOL` is set. Claude Code auto-detects `pwsh.exe` (PowerShell 7+) with a fallback to `powershell.exe` (5.1).

```json theme={null}
{
"hooks": {
"PostToolUse": [
{
"matcher": "Write",
"hooks": [
{
"type": "command",
"shell": "powershell",
```

"command": "Write-Host 'File written'"
}
]
}
]
}
}

```
## Debug hooks

Run `claude --debug` to see hook execution details, including which hooks

```text  theme={null}
[DEBUG] Executing hooks for PostToolUse:Write
[DEBUG] Found 1 hook commands to execute
[DEBUG] Executing hook command:  with timeout 600000ms
[DEBUG] Hook command completed with status 0:
```

For more granular hook matching details, set `CLAUDE_CODE_DEBUG_LOG_LEVEL=verbose` to see additional log lines such as hook matcher counts and query matching.

For troubleshooting common issues like hooks not firing, infinite Stop hook loops, or configuration errors, see Limitations and troubleshooting in the guide.
```