

# Building with Claude Code

# Claude Code overview

> Learn about Claude Code, Anthropic's agentic coding tool that lives in your terminal and helps you turn ideas into code faster than ever before.

## Get started in 30 seconds

Prerequisites:

\* [Node.js 18 or newer](https://nodejs.org/en/download/)

\* A [Claude.ai](https://claude.ai) (recommended) or [Anthropic Console](https://console.anthropic.com/) account

```
```bash
```

# Install Claude Code

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

# Navigate to your project

```
cd your-awesome-project
```

# Start coding with Claude

```
claude
```

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

```
...
```

That's it! You're ready to start coding with Claude. [Continue with Quickstart (5 mins)](/en/docs/claude-code/quickstart)

(Got specific setup needs or hit issues? See [advanced setup](/en/docs/claude-code/setup) or [troubleshooting](/en/docs/claude-code/troubleshooting).)

## ## What Claude Code does for you

**Build features from descriptions**: Tell Claude what you want to build in plain English. It will make a plan, write the code, and ensure it works.

**Debug and fix issues**: Describe a bug or paste an error message. Claude Code will analyze your codebase, identify the problem, and implement a fix.

**Navigate any codebase**: Ask anything about your team's codebase, and get a thoughtful answer back. Claude Code maintains awareness of your entire project structure, can find up-to-date information from the web, and with [MCP](/en/docs/claude-code/mcp) can pull from external datasources like Google Drive, Figma, and Slack.

**Automate tedious tasks**: Fix fiddly lint issues, resolve merge conflicts, and write release notes. Do all this in a single command from your developer machines, or automatically in CI.

## ## Why developers love Claude Code

**Works in your terminal**: Not another chat window. Not another IDE. Claude Code meets you where you already work, with the tools you already love.

**Takes action**: Claude Code can directly edit files, run commands, and create commits. Need more? [MCP](/en/docs/claude-code/mcp) lets Claude read your design docs in Google Drive, update your tickets in Jira, or use *your* custom developer tooling.

**Unix philosophy**: Claude Code is composable and scriptable. ``tail -f app.log | claude -p "Slack me if you see any anomalies appear in this log stream" *works*``. Your CI can run ``claude -p "If there are new text strings, translate them into French and raise a PR for @lang-fr-team to review"`.`

**Enterprise-ready**: Use Anthropic's API, or host on AWS or GCP. Enterprise-grade [security](/en/docs/claude-code/security), [privacy](/en/docs/claude-code/data-usage), and [compliance](https://trust.anthropic.com/) is built-in.

## ## Next steps

<CardGroup>

<Card title="Quickstart" icon="rocket" href="/en/docs/claude-code/quickstart">

See Claude Code in action with practical examples

</Card>

<Card title="Common workflows" icon="graduation-cap" href="/en/docs/claude-code/common-workflows">

Step-by-step guides for common workflows

</Card>

<Card title="Troubleshooting" icon="wrench" href="/en/docs/claude-code/troubleshooting">

Solutions for common issues with Claude Code

</Card>

<Card title="IDE setup" icon="laptop" href="/en/docs/claude-code/ide-integrations">

Add Claude Code to your IDE

</Card>

</CardGroup>

## Additional resources

<CardGroup>

<Card title="Host on AWS or GCP" icon="cloud" href="/en/docs/claude-code/third-party-integrations">

Configure Claude Code with Amazon Bedrock or Google Vertex AI

</Card>

<Card title="Settings" icon="gear" href="/en/docs/claude-code/settings">

Customize Claude Code for your workflow

</Card>

<Card title="Commands" icon="terminal" href="/en/docs/claude-code/cli-reference">

Learn about CLI commands and controls

</Card>

<Card title="Reference implementation" icon="code"  
href="https://github.com/anthropics/claude-code/tree/main/.devcontainer">

Clone our development container reference implementation

</Card>

<Card title="Security" icon="shield" href="/en/docs/claude-code/security">

Discover Claude Code's safeguards and best practices for safe usage

</Card>

<Card title="Privacy and data usage" icon="lock" href="/en/docs/claude-code/data-usage">

Understand how Claude Code handles your data

</Card>

</CardGroup>

# Claude Code overview

> Learn about Claude Code, Anthropic's agentic coding tool that lives in your terminal and helps you turn ideas into code faster than ever before.

## Get started in 30 seconds

Prerequisites:

\* [Node.js 18 or newer](https://nodejs.org/en/download/)

\* A [Claude.ai](https://claude.ai) (recommended) or [Anthropic Console](https://console.anthropic.com/) account

```
```bash
```

```
# Install Claude Code
```

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

```
# Navigate to your project
```

```
cd your-awesome-project
```

```
# Start coding with Claude
```

```
claude
```

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

```
...
```

That's it! You're ready to start coding with Claude. [Continue with Quickstart (5 mins)](/en/docs/claude-code/quickstart)

(Got specific setup needs or hit issues? See [advanced setup](/en/docs/claude-code/setup) or [troubleshooting](/en/docs/claude-code/troubleshooting).)

## ## What Claude Code does for you

**\*\*Build features from descriptions\*\*:** Tell Claude what you want to build in plain English. It will make a plan, write the code, and ensure it works.

**\*\*Debug and fix issues\*\*:** Describe a bug or paste an error message. Claude Code will analyze your codebase, identify the problem, and implement a fix.

\* \*\*Navigate any codebase\*\*:

 Ask anything about your team's codebase, and get a thoughtful answer back. Claude Code maintains awareness of your entire project structure, can find up-to-date information from the web, and with [MCP](/en/docs/claude-code/mcp) can pull from external datasources like Google Drive, Figma, and Slack.

\* \*\*Automate tedious tasks\*\*:

 Fix fiddly lint issues, resolve merge conflicts, and write release notes. Do all this in a single command from your developer machines, or automatically in CI.

## ## Why developers love Claude Code

\* \*\*Works in your terminal\*\*:

 Not another chat window. Not another IDE. Claude Code meets you where you already work, with the tools you already love.

\* \*\*Takes action\*\*:

 Claude Code can directly edit files, run commands, and create commits. Need more? [MCP](/en/docs/claude-code/mcp) lets Claude read your design docs in Google Drive, update your tickets in Jira, or use *your* custom developer tooling.

\* \*\*Unix philosophy\*\*:

 Claude Code is composable and scriptable. ``tail -f app.log | claude -p "Slack me if you see any anomalies appear in this log stream"`` *works*. Your CI can run ``claude -p "If there are new text strings, translate them into French and raise a PR for @lang-fr-team to review"``.

\* \*\*Enterprise-ready\*\*:

 Use Anthropic's API, or host on AWS or GCP. Enterprise-grade [security](/en/docs/claude-code/security), [privacy](/en/docs/claude-code/data-usage), and [compliance](https://trust.anthropic.com/) is built-in.

## ## Next steps

<CardGroup>

<Card title="Quickstart" icon="rocket" href="/en/docs/claude-code/quickstart">

See Claude Code in action with practical examples

</Card>

<Card title="Common workflows" icon="graduation-cap" href="/en/docs/claude-code/common-workflows">

Step-by-step guides for common workflows

</Card>

<Card title="Troubleshooting" icon="wrench" href="/en/docs/claude-code/troubleshooting">

Solutions for common issues with Claude Code

</Card>

<Card title="IDE setup" icon="laptop" href="/en/docs/claude-code/ide-integrations">

Add Claude Code to your IDE

</Card>

</CardGroup>

## Additional resources

<CardGroup>

<Card title="Host on AWS or GCP" icon="cloud" href="/en/docs/claude-code/third-party-integrations">

Configure Claude Code with Amazon Bedrock or Google Vertex AI

</Card>

<Card title="Settings" icon="gear" href="/en/docs/claude-code/settings">

Customize Claude Code for your workflow

</Card>

<Card title="Commands" icon="terminal" href="/en/docs/claude-code/cli-reference">

Learn about CLI commands and controls

</Card>

<Card title="Reference implementation" icon="code"  
href="https://github.com/anthropics/claude-code/tree/main/.devcontainer">

Clone our development container reference implementation

</Card>

<Card title="Security" icon="shield" href="/en/docs/claude-code/security">

Discover Claude Code's safeguards and best practices for safe usage

</Card>

<Card title="Privacy and data usage" icon="lock" href="/en/docs/claude-code/data-usage">

Understand how Claude Code handles your data

</Card>

</CardGroup>

## # Claude Code overview

> Learn about Claude Code, Anthropic's agentic coding tool that lives in your terminal and helps you turn ideas into code faster than ever before.

### ## Get started in 30 seconds

Prerequisites:

\* [Node.js 18 or newer](https://nodejs.org/en/download/)

\* A [Claude.ai](https://claude.ai) (recommended) or [Anthropic Console](https://console.anthropic.com/) account

```
```bash
```

```
# Install Claude Code
```

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

```
# Navigate to your project
```

```
cd your-awesome-project
```



# Start coding with Claude

claude

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

...

That's it! You're ready to start coding with Claude. [Continue with Quickstart (5 mins)](</en/docs/claude-code/quickstart>)

(Got specific setup needs or hit issues? See [advanced setup](</en/docs/claude-code/setup>) or [troubleshooting](</en/docs/claude-code/troubleshooting>).)

## ## What Claude Code does for you

**Build features from descriptions**: Tell Claude what you want to build in plain English. It will make a plan, write the code, and ensure it works.

**Debug and fix issues**: Describe a bug or paste an error message. Claude Code will analyze your codebase, identify the problem, and implement a fix.

**Navigate any codebase**: Ask anything about your team's codebase, and get a thoughtful answer back. Claude Code maintains awareness of your entire project structure, can find up-to-date information from the web, and with [MCP](</en/docs/claude-code/mcp>) can pull from external datasources like Google Drive, Figma, and Slack.

**Automate tedious tasks**: Fix fiddly lint issues, resolve merge conflicts, and write release notes. Do all this in a single command from your developer machines, or automatically in CI.

## ## Why developers love Claude Code

**Works in your terminal**: Not another chat window. Not another IDE. Claude Code meets you where you already work, with the tools you already love.

**Takes action**: Claude Code can directly edit files, run commands, and create commits. Need more? [MCP](</en/docs/claude-code/mcp>) lets Claude read your design docs in Google Drive, update your tickets in Jira, or use *your* custom developer tooling.

**Unix philosophy**: Claude Code is composable and scriptable. `tail -f app.log | claude -p "Slack me if you see any anomalies appear in this log stream" *works*`. Your CI can run `claude -p "If there are new text strings, translate them into French and raise a PR for @lang-fr-team to review"`.

\* \*\*Enterprise-ready\*\*: Use Anthropic's API, or host on AWS or GCP. Enterprise-grade [security](/en/docs/claude-code/security), [privacy](/en/docs/claude-code/data-usage), and [compliance](https://trust.anthropic.com/) is built-in.

## ## Next steps

<CardGroup>

<Card title="Quickstart" icon="rocket" href="/en/docs/claude-code/quickstart">

See Claude Code in action with practical examples

</Card>

<Card title="Common workflows" icon="graduation-cap" href="/en/docs/claude-code/common-workflows">

Step-by-step guides for common workflows

</Card>

<Card title="Troubleshooting" icon="wrench" href="/en/docs/claude-code/troubleshooting">

Solutions for common issues with Claude Code

</Card>

<Card title="IDE setup" icon="laptop" href="/en/docs/claude-code/ide-integrations">

Add Claude Code to your IDE

</Card>

</CardGroup>

## ## Additional resources

<CardGroup>

<Card title="Host on AWS or GCP" icon="cloud" href="/en/docs/claude-code/third-party-integrations">

Configure Claude Code with Amazon Bedrock or Google Vertex AI

</Card>

<Card title="Settings" icon="gear" href="/en/docs/claude-code/settings">

Customize Claude Code for your workflow

</Card>

<Card title="Commands" icon="terminal" href="/en/docs/claude-code/cli-reference">

Learn about CLI commands and controls

</Card>

<Card title="Reference implementation" icon="code"  
href="https://github.com/anthropics/claude-code/tree/main/.devcontainer">

Clone our development container reference implementation

</Card>

<Card title="Security" icon="shield" href="/en/docs/claude-code/security">

Discover Claude Code's safeguards and best practices for safe usage

</Card>

<Card title="Privacy and data usage" icon="lock" href="/en/docs/claude-code/data-usage">

Understand how Claude Code handles your data

</Card>

</CardGroup>

# Subagents

> Create and use specialized AI subagents in Claude Code for task-specific workflows and improved context management.

Custom subagents in Claude Code are specialized AI assistants that can be invoked to handle specific types of tasks. They enable more efficient problem-solving by providing task-specific configurations with customized system prompts, tools and a separate context window.

### ## What are subagents?

Subagents are pre-configured AI personalities that Claude Code can delegate tasks to. Each subagent:

- \* Has a specific purpose and expertise area
- \* Uses its own context window separate from the main conversation
- \* Can be configured with specific tools it's allowed to use
- \* Includes a custom system prompt that guides its behavior

When Claude Code encounters a task that matches a subagent's expertise, it can delegate that task to the specialized subagent, which works independently and returns results.

### ## Key benefits

<CardGroup cols={2}>

<Card title="Context preservation" icon="layer-group">

Each subagent operates in its own context, preventing pollution of the main conversation and keeping it focused on high-level objectives.

</Card>

<Card title="Specialized expertise" icon="brain">

Subagents can be fine-tuned with detailed instructions for specific domains, leading to higher success rates on designated tasks.

</Card>

<Card title="Reusability" icon="rotate">

Once created, subagents can be used across different projects and shared with your team for consistent workflows.

</Card>

<Card title="Flexible permissions" icon="shield-check">

Each subagent can have different tool access levels, allowing you to limit powerful tools to specific subagent types.

</Card>

</CardGroup>

## ## Quick start

To create your first subagent:

<Steps>

<Step title="Open the subagents interface">

Run the following command:

...

/agents

...

</Step>

<Step title="Select 'Create New Agent'">

Choose whether to create a project-level or user-level subagent

</Step>

<Step title="Define the subagent">

- \* **Recommended**: Generate with Claude first, then customize to make it yours
- \* Describe your subagent in detail and when it should be used
- \* Select the tools you want to grant access to (or leave blank to inherit all tools)
- \* The interface shows all available tools, making selection easy
- \* If you're generating with Claude, you can also edit the system prompt in your own editor by pressing `e`

</Step>

<Step title="Save and use">

Your subagent is now available! Claude will use it automatically when appropriate, or you can invoke it explicitly:

...

> Use the code-reviewer subagent to check my recent changes

...

</Step>

</Steps>

## Subagent configuration

### File locations

Subagents are stored as Markdown files with YAML frontmatter in two possible locations:

Type	Location	Scope	Priority
:-----	:-----	:-----	:-----
<b>Project subagents</b>	<code>`.claude/agents/`</code>	Available in current project	Highest

| **\*\*User subagents\*\*** | `~/claude/agents/` | Available across all projects | Lower |

When subagent names conflict, project-level subagents take precedence over user-level subagents.

### File format

Each subagent is defined in a Markdown file with this structure:

```
```markdown
---
name: your-sub-agent-name
description: Description of when this subagent should be invoked
tools: tool1, tool2, tool3 # Optional - inherits all tools if omitted
---
```

Your subagent's system prompt goes here. This can be multiple paragraphs and should clearly define the subagent's role, capabilities, and approach to solving problems.

Include specific instructions, best practices, and any constraints the subagent should follow.

...

#### Configuration fields

Field	Required	Description
:-----	:-----	:-----

`name`	Yes	Unique identifier using lowercase letters and hyphens	
`description`	Yes	Natural language description of the subagent's purpose	
`tools`	No	Comma-separated list of specific tools. If omitted, inherits all tools from the main thread	

### ### Available tools

Subagents can be granted access to any of Claude Code's internal tools. See the [tools documentation](/en/docs/claude-code/settings#tools-available-to-claude) for a complete list of available tools.

<Tip>

**\*\*Recommended:\*\*** Use the `/agents` command to modify tool access - it provides an interactive interface that lists all available tools, including any connected MCP server tools, making it easier to select the ones you need.

</Tip>

You have two options for configuring tools:

**\* \*\*Omit the `tools` field\*\*** to inherit all tools from the main thread (default), including MCP tools

**\* \*\*Specify individual tools\*\*** as a comma-separated list for more granular control (can be edited manually or via `/agents`)

**\*\*MCP Tools\*\***: Subagents can access MCP tools from configured MCP servers. When the `tools` field is omitted, subagents inherit all MCP tools available to the main thread.

## ## Managing subagents

### ### Using the /agents command (Recommended)

The `/agents` command provides a comprehensive interface for subagent management:



...

/agents

...

This opens an interactive menu where you can:

- \* View all available subagents (built-in, user, and project)
- \* Create new subagents with guided setup
- \* Edit existing custom subagents, including their tool access
- \* Delete custom subagents
- \* See which subagents are active when duplicates exist
- \* **Easily manage tool permissions** with a complete list of available tools

### Direct file management

You can also manage subagents by working directly with their files:

```
```bash
```

```
# Create a project subagent
```

```
mkdir -p .claude/agents
```

```
echo '---
```

```
name: test-runner
```

```
description: Use proactively to run tests and fix failures
```

```
---
```

You are a test automation expert. When you see code changes, proactively run the appropriate tests. If tests fail, analyze the failures and fix them while preserving the original test intent.' > .claude/agents/test-runner.md

# Create a user subagent

```
mkdir -p ~/.claude/agents
```

# ... create subagent file

...

## Using subagents effectively

### Automatic delegation

Claude Code proactively delegates tasks based on:

- \* The task description in your request
- \* The `description` field in subagent configurations
- \* Current context and available tools

<Tip>

To encourage more proactive subagent use, include phrases like "use PROACTIVELY" or "MUST BE USED" in your `description` field.

</Tip>

### Explicit invocation

Request a specific subagent by mentioning it in your command:

...

- > Use the test-runner subagent to fix failing tests
- > Have the code-reviewer subagent look at my recent changes
- > Ask the debugger subagent to investigate this error

...

## ## Example subagents

### ### Code reviewer

```markdown

---

name: code-reviewer

description: Expert code review specialist. Proactively reviews code for quality, security, and maintainability. Use immediately after writing or modifying code.

tools: Read, Grep, Glob, Bash

---

You are a senior code reviewer ensuring high standards of code quality and security.

When invoked:

1. Run git diff to see recent changes
2. Focus on modified files
3. Begin review immediately

Review checklist:

- Code is simple and readable
- Functions and variables are well-named
- No duplicated code
- Proper error handling
- No exposed secrets or API keys
- Input validation implemented

- Good test coverage
- Performance considerations addressed

Provide feedback organized by priority:

- Critical issues (must fix)
- Warnings (should fix)
- Suggestions (consider improving)

Include specific examples of how to fix issues.

...

### Debugger

```markdown

---

name: debugger

description: Debugging specialist for errors, test failures, and unexpected behavior. Use proactively when encountering any issues.

tools: Read, Edit, Bash, Grep, Glob

---

You are an expert debugger specializing in root cause analysis.

When invoked:

1. Capture error message and stack trace
2. Identify reproduction steps
3. Isolate the failure location
4. Implement minimal fix

## 5. Verify solution works

Debugging process:

- Analyze error messages and logs
- Check recent code changes
- Form and test hypotheses
- Add strategic debug logging
- Inspect variable states

For each issue, provide:

- Root cause explanation
- Evidence supporting the diagnosis
- Specific code fix
- Testing approach
- Prevention recommendations

Focus on fixing the underlying issue, not just symptoms.

...

### Data scientist

```markdown

---

name: data-scientist

description: Data analysis expert for SQL queries, BigQuery operations, and data insights. Use proactively for data analysis tasks and queries.

tools: Bash, Read, Write

---

You are a data scientist specializing in SQL and BigQuery analysis.

When invoked:

1. Understand the data analysis requirement
2. Write efficient SQL queries
3. Use BigQuery command line tools (bq) when appropriate
4. Analyze and summarize results
5. Present findings clearly

Key practices:

- Write optimized SQL queries with proper filters
- Use appropriate aggregations and joins
- Include comments explaining complex logic
- Format results for readability
- Provide data-driven recommendations

For each analysis:

- Explain the query approach
- Document any assumptions
- Highlight key findings
- Suggest next steps based on data

Always ensure queries are efficient and cost-effective.

...

## Best practices

\* \*\*Start with Claude-generated agents\*\*:

We highly recommend generating your initial subagent with Claude and then iterating on it to make it personally yours. This approach gives you the best results - a solid foundation that you can customize to your specific needs.

\* \*\*Design focused subagents\*\*:

Create subagents with single, clear responsibilities rather than trying to make one subagent do everything. This improves performance and makes subagents more predictable.

\* \*\*Write detailed prompts\*\*:

Include specific instructions, examples, and constraints in your system prompts. The more guidance you provide, the better the subagent will perform.

\* \*\*Limit tool access\*\*:

Only grant tools that are necessary for the subagent's purpose. This improves security and helps the subagent focus on relevant actions.

\* \*\*Version control\*\*:

Check project subagents into version control so your team can benefit from and improve them collaboratively.

## ## Advanced usage

### ### Chaining subagents

For complex workflows, you can chain multiple subagents:

...

> First use the code-analyzer subagent to find performance issues, then use the optimizer subagent to fix them

...

### ### Dynamic subagent selection

Claude Code intelligently selects subagents based on context. Make your `description` fields specific and action-oriented for best results.

### ## Performance considerations

\* **Context efficiency**: Agents help preserve main context, enabling longer overall sessions

\* **Latency**: Subagents start off with a clean slate each time they are invoked and may add latency as they gather context that they require to do their job effectively.

### ## Related documentation

\* [\[Slash commands\]](/en/docs/claude-code/slash-commands) - Learn about other built-in commands

\* [\[Settings\]](/en/docs/claude-code/settings) - Configure Claude Code behavior

\* [\[Hooks\]](/en/docs/claude-code/hooks) - Automate workflows with event handlers

## # Subagents

> Create and use specialized AI subagents in Claude Code for task-specific workflows and improved context management.

Custom subagents in Claude Code are specialized AI assistants that can be invoked to handle specific types of tasks. They enable more efficient problem-solving by providing task-specific configurations with customized system prompts, tools and a separate context window.

### ## What are subagents?

Subagents are pre-configured AI personalities that Claude Code can delegate tasks to. Each subagent:

\* Has a specific purpose and expertise area

\* Uses its own context window separate from the main conversation

\* Can be configured with specific tools it's allowed to use



\* Includes a custom system prompt that guides its behavior

When Claude Code encounters a task that matches a subagent's expertise, it can delegate that task to the specialized subagent, which works independently and returns results.

## ## Key benefits

<CardGroup cols={2}>

<Card title="Context preservation" icon="layer-group">

Each subagent operates in its own context, preventing pollution of the main conversation and keeping it focused on high-level objectives.

</Card>

<Card title="Specialized expertise" icon="brain">

Subagents can be fine-tuned with detailed instructions for specific domains, leading to higher success rates on designated tasks.

</Card>

<Card title="Reusability" icon="rotate">

Once created, subagents can be used across different projects and shared with your team for consistent workflows.

</Card>

<Card title="Flexible permissions" icon="shield-check">

Each subagent can have different tool access levels, allowing you to limit powerful tools to specific subagent types.

</Card>

</CardGroup>

## ## Quick start

To create your first subagent:

<Steps>

<Step title="Open the subagents interface">

Run the following command:

...

/agents

...

</Step>

<Step title="Select 'Create New Agent'">

Choose whether to create a project-level or user-level subagent

</Step>

<Step title="Define the subagent">

\* **Recommended**: Generate with Claude first, then customize to make it yours

\* Describe your subagent in detail and when it should be used

\* Select the tools you want to grant access to (or leave blank to inherit all tools)

\* The interface shows all available tools, making selection easy

\* If you're generating with Claude, you can also edit the system prompt in your own editor by pressing `e`

</Step>

<Step title="Save and use">

Your subagent is now available! Claude will use it automatically when appropriate, or you can invoke it explicitly:

```
...

> Use the code-reviewer subagent to check my recent changes

...

</Step>

</Steps>
```

## ## Subagent configuration

### ### File locations

Subagents are stored as Markdown files with YAML frontmatter in two possible locations:

Type	Location	Scope	Priority
Project subagents	`.claude/agents/`	Available in current project	Highest
User subagents	`\${HOME}/.claude/agents/`	Available across all projects	Lower

When subagent names conflict, project-level subagents take precedence over user-level subagents.

### ### File format

Each subagent is defined in a Markdown file with this structure:

```
```markdown
---
name: your-sub-agent-name

description: Description of when this subagent should be invoked
```

tools: tool1, tool2, tool3 # Optional - inherits all tools if omitted

---

Your subagent's system prompt goes here. This can be multiple paragraphs and should clearly define the subagent's role, capabilities, and approach to solving problems.

Include specific instructions, best practices, and any constraints the subagent should follow.

...

#### Configuration fields

Field	Required	Description
:-----	:-----	:-----
`name`	Yes	Unique identifier using lowercase letters and hyphens
`description`	Yes	Natural language description of the subagent's purpose
`tools`	No	Comma-separated list of specific tools. If omitted, inherits all tools from the main thread

### Available tools

Subagents can be granted access to any of Claude Code's internal tools. See the [tools documentation](/en/docs/claude-code/settings#tools-available-to-claude) for a complete list of available tools.

<Tip>

**\*\*Recommended:\*\*** Use the `/agents` command to modify tool access - it provides an interactive interface that lists all available tools, including any connected MCP server tools, making it easier to select the ones you need.

</Tip>

You have two options for configuring tools:

\* **Omit the `tools` field** to inherit all tools from the main thread (default), including MCP tools

\* **Specify individual tools** as a comma-separated list for more granular control (can be edited manually or via `/agents`)

**MCP Tools**: Subagents can access MCP tools from configured MCP servers. When the `tools` field is omitted, subagents inherit all MCP tools available to the main thread.

## ## Managing subagents

### ### Using the /agents command (Recommended)

The `/agents` command provides a comprehensive interface for subagent management:

...

/agents

...

This opens an interactive menu where you can:

\* View all available subagents (built-in, user, and project)

\* Create new subagents with guided setup

\* Edit existing custom subagents, including their tool access

\* Delete custom subagents

\* See which subagents are active when duplicates exist

\* **Easily manage tool permissions** with a complete list of available tools

### ### Direct file management

You can also manage subagents by working directly with their files:

```
```bash
```

```
# Create a project subagent
```

```
mkdir -p .claude/agents
```

```
echo '---
```

```
name: test-runner
```

```
description: Use proactively to run tests and fix failures
```

```
---
```

You are a test automation expert. When you see code changes, proactively run the appropriate tests. If tests fail, analyze the failures and fix them while preserving the original test intent.' > .claude/agents/test-runner.md

```
# Create a user subagent
```

```
mkdir -p ~/.claude/agents
```

```
# ... create subagent file
```

```
...
```

## ## Using subagents effectively

### ### Automatic delegation

Claude Code proactively delegates tasks based on:

- \* The task description in your request

\* The `description` field in subagent configurations

\* Current context and available tools

<Tip>

To encourage more proactive subagent use, include phrases like "use PROACTIVELY" or "MUST BE USED" in your `description` field.

</Tip>

### Explicit invocation

Request a specific subagent by mentioning it in your command:

...

> Use the test-runner subagent to fix failing tests

> Have the code-reviewer subagent look at my recent changes

> Ask the debugger subagent to investigate this error

...

## Example subagents

### Code reviewer

```markdown

---

name: code-reviewer

description: Expert code review specialist. Proactively reviews code for quality, security, and maintainability. Use immediately after writing or modifying code.

tools: Read, Grep, Glob, Bash

---

You are a senior code reviewer ensuring high standards of code quality and security.

When invoked:

1. Run git diff to see recent changes
2. Focus on modified files
3. Begin review immediately

Review checklist:

- Code is simple and readable
- Functions and variables are well-named
- No duplicated code
- Proper error handling
- No exposed secrets or API keys
- Input validation implemented
- Good test coverage
- Performance considerations addressed

Provide feedback organized by priority:

- Critical issues (must fix)
- Warnings (should fix)
- Suggestions (consider improving)

Include specific examples of how to fix issues.

...



### ### Debugger

```markdown

---

name: debugger

description: Debugging specialist for errors, test failures, and unexpected behavior. Use proactively when encountering any issues.

tools: Read, Edit, Bash, Grep, Glob

---

You are an expert debugger specializing in root cause analysis.

When invoked:

1. Capture error message and stack trace
2. Identify reproduction steps
3. Isolate the failure location
4. Implement minimal fix
5. Verify solution works

Debugging process:

- Analyze error messages and logs
- Check recent code changes
- Form and test hypotheses
- Add strategic debug logging
- Inspect variable states

For each issue, provide:

- Root cause explanation

- Evidence supporting the diagnosis
- Specific code fix
- Testing approach
- Prevention recommendations

Focus on fixing the underlying issue, not just symptoms.

...

### Data scientist

```markdown

---

name: data-scientist

description: Data analysis expert for SQL queries, BigQuery operations, and data insights. Use proactively for data analysis tasks and queries.

tools: Bash, Read, Write

---

You are a data scientist specializing in SQL and BigQuery analysis.

When invoked:

1. Understand the data analysis requirement
2. Write efficient SQL queries
3. Use BigQuery command line tools (bq) when appropriate
4. Analyze and summarize results
5. Present findings clearly

Key practices:

- Write optimized SQL queries with proper filters
- Use appropriate aggregations and joins
- Include comments explaining complex logic
- Format results for readability
- Provide data-driven recommendations

For each analysis:

- Explain the query approach
- Document any assumptions
- Highlight key findings
- Suggest next steps based on data

Always ensure queries are efficient and cost-effective.

...

### ## Best practices

\* \*\*Start with Claude-generated agents\*\*:

We highly recommend generating your initial subagent with Claude and then iterating on it to make it personally yours. This approach gives you the best results - a solid foundation that you can customize to your specific needs.

\* \*\*Design focused subagents\*\*:

Create subagents with single, clear responsibilities rather than trying to make one subagent do everything. This improves performance and makes subagents more predictable.

\* \*\*Write detailed prompts\*\*:

Include specific instructions, examples, and constraints in your system prompts. The more guidance you provide, the better the subagent will perform.

\* \*\*Limit tool access\*\*:

Only grant tools that are necessary for the subagent's purpose. This improves security and helps the subagent focus on relevant actions.

\* \*\*Version control\*\*: Check project subagents into version control so your team can benefit from and improve them collaboratively.

## ## Advanced usage

### ### Chaining subagents

For complex workflows, you can chain multiple subagents:

...

> First use the code-analyzer subagent to find performance issues, then use the optimizer subagent to fix them

...

### ### Dynamic subagent selection

Claude Code intelligently selects subagents based on context. Make your `description` fields specific and action-oriented for best results.

## ## Performance considerations

\* \*\*Context efficiency\*\*: Agents help preserve main context, enabling longer overall sessions

\* \*\*Latency\*\*: Subagents start off with a clean slate each time they are invoked and may add latency as they gather context that they require to do their job effectively.

## ## Related documentation

\* [Slash commands](/en/docs/claude-code/slash-commands) - Learn about other built-in commands

\* [Settings](/en/docs/claude-code/settings) - Configure Claude Code behavior

\* [Hooks](/en/docs/claude-code/hooks) - Automate workflows with event handlers

## # Subagents

> Create and use specialized AI subagents in Claude Code for task-specific workflows and improved context management.

Custom subagents in Claude Code are specialized AI assistants that can be invoked to handle specific types of tasks. They enable more efficient problem-solving by providing task-specific configurations with customized system prompts, tools and a separate context window.

### ## What are subagents?

Subagents are pre-configured AI personalities that Claude Code can delegate tasks to. Each subagent:

- \* Has a specific purpose and expertise area
- \* Uses its own context window separate from the main conversation
- \* Can be configured with specific tools it's allowed to use
- \* Includes a custom system prompt that guides its behavior

When Claude Code encounters a task that matches a subagent's expertise, it can delegate that task to the specialized subagent, which works independently and returns results.

### ## Key benefits

<CardGroup cols={2}>

<Card title="Context preservation" icon="layer-group">

Each subagent operates in its own context, preventing pollution of the main conversation and keeping it focused on high-level objectives.

</Card>

<Card title="Specialized expertise" icon="brain">

Subagents can be fine-tuned with detailed instructions for specific domains, leading to higher success rates on designated tasks.

</Card>

<Card title="Reusability" icon="rotate">

Once created, subagents can be used across different projects and shared with your team for consistent workflows.

</Card>

<Card title="Flexible permissions" icon="shield-check">

Each subagent can have different tool access levels, allowing you to limit powerful tools to specific subagent types.

</Card>

</CardGroup>

## ## Quick start

To create your first subagent:

<Steps>

<Step title="Open the subagents interface">

Run the following command:

...

/agents

...

</Step>

<Step title="Select 'Create New Agent'">

Choose whether to create a project-level or user-level subagent

</Step>

<Step title="Define the subagent">

\* **Recommended**: Generate with Claude first, then customize to make it yours

\* Describe your subagent in detail and when it should be used

\* Select the tools you want to grant access to (or leave blank to inherit all tools)

\* The interface shows all available tools, making selection easy

\* If you're generating with Claude, you can also edit the system prompt in your own editor by pressing `e`

</Step>

<Step title="Save and use">

Your subagent is now available! Claude will use it automatically when appropriate, or you can invoke it explicitly:

...

> Use the code-reviewer subagent to check my recent changes

...

</Step>

</Steps>

## Subagent configuration

### File locations

Subagents are stored as Markdown files with YAML frontmatter in two possible locations:

| Type                         | Location                             | Scope                         | Priority |
|------------------------------|--------------------------------------|-------------------------------|----------|
| :-----                       | :-----                               | :-----                        | :-----   |
| <b>**Project subagents**</b> | <code>`.claude/agents/`</code>       | Available in current project  | Highest  |
| <b>**User subagents**</b>    | <code>`\${~}/.claude/agents/`</code> | Available across all projects | Lower    |

When subagent names conflict, project-level subagents take precedence over user-level subagents.

### ### File format

Each subagent is defined in a Markdown file with this structure:

```
```markdown
---

name: your-sub-agent-name

description: Description of when this subagent should be invoked

tools: tool1, tool2, tool3 # Optional - inherits all tools if omitted

---
```

Your subagent's system prompt goes here. This can be multiple paragraphs and should clearly define the subagent's role, capabilities, and approach to solving problems.

Include specific instructions, best practices, and any constraints the subagent should follow.

```
...
```

### #### Configuration fields



Field	Required	Description
name	Yes	Unique identifier using lowercase letters and hyphens
description	Yes	Natural language description of the subagent's purpose
tools	No	Comma-separated list of specific tools. If omitted, inherits all tools from the main thread

### Available tools

Subagents can be granted access to any of Claude Code's internal tools. See the [tools documentation](/en/docs/claude-code/settings#tools-available-to-claude) for a complete list of available tools.

<Tip>

**Recommended:** Use the `/agents` command to modify tool access - it provides an interactive interface that lists all available tools, including any connected MCP server tools, making it easier to select the ones you need.

</Tip>

You have two options for configuring tools:

**Omit the `tools` field** to inherit all tools from the main thread (default), including MCP tools

**Specify individual tools** as a comma-separated list for more granular control (can be edited manually or via `/agents`)

**MCP Tools:** Subagents can access MCP tools from configured MCP servers. When the `tools` field is omitted, subagents inherit all MCP tools available to the main thread.

## Managing subagents

### Using the `/agents` command (Recommended)

The `/agents` command provides a comprehensive interface for subagent management:

```
...
```

```
/agents
```

```
...
```

This opens an interactive menu where you can:

- \* View all available subagents (built-in, user, and project)
- \* Create new subagents with guided setup
- \* Edit existing custom subagents, including their tool access
- \* Delete custom subagents
- \* See which subagents are active when duplicates exist
- \* **Easily manage tool permissions** with a complete list of available tools

### Direct file management

You can also manage subagents by working directly with their files:

```
```bash
```

```
# Create a project subagent
```

```
mkdir -p .claude/agents
```

```
echo '---
```

```
name: test-runner
```

```
description: Use proactively to run tests and fix failures
```

```
---
```

You are a test automation expert. When you see code changes, proactively run the appropriate tests. If tests fail, analyze the failures and fix them while preserving the original test intent.' > .claude/agents/test-runner.md

# Create a user subagent

```
mkdir -p ~/.claude/agents
```

# ... create subagent file

...

## Using subagents effectively

### Automatic delegation

Claude Code proactively delegates tasks based on:

- \* The task description in your request
- \* The `description` field in subagent configurations
- \* Current context and available tools

<Tip>

To encourage more proactive subagent use, include phrases like "use PROACTIVELY" or "MUST BE USED" in your `description` field.

</Tip>

### Explicit invocation

Request a specific subagent by mentioning it in your command:

...

- > Use the test-runner subagent to fix failing tests
- > Have the code-reviewer subagent look at my recent changes
- > Ask the debugger subagent to investigate this error

...

## ## Example subagents

### ### Code reviewer

```markdown

---

name: code-reviewer

description: Expert code review specialist. Proactively reviews code for quality, security, and maintainability. Use immediately after writing or modifying code.

tools: Read, Grep, Glob, Bash

---

You are a senior code reviewer ensuring high standards of code quality and security.

When invoked:

1. Run git diff to see recent changes
2. Focus on modified files
3. Begin review immediately

Review checklist:

- Code is simple and readable
- Functions and variables are well-named

- No duplicated code
- Proper error handling
- No exposed secrets or API keys
- Input validation implemented
- Good test coverage
- Performance considerations addressed

Provide feedback organized by priority:

- Critical issues (must fix)
- Warnings (should fix)
- Suggestions (consider improving)

Include specific examples of how to fix issues.

...

### Debugger

```markdown

---

name: debugger

description: Debugging specialist for errors, test failures, and unexpected behavior. Use proactively when encountering any issues.

tools: Read, Edit, Bash, Grep, Glob

---

You are an expert debugger specializing in root cause analysis.

When invoked:

1. Capture error message and stack trace
2. Identify reproduction steps
3. Isolate the failure location
4. Implement minimal fix
5. Verify solution works

Debugging process:

- Analyze error messages and logs
- Check recent code changes
- Form and test hypotheses
- Add strategic debug logging
- Inspect variable states

For each issue, provide:

- Root cause explanation
- Evidence supporting the diagnosis
- Specific code fix
- Testing approach
- Prevention recommendations

Focus on fixing the underlying issue, not just symptoms.

...

### Data scientist

```markdown

---

name: data-scientist

description: Data analysis expert for SQL queries, BigQuery operations, and data insights. Use proactively for data analysis tasks and queries.

tools: Bash, Read, Write

---

You are a data scientist specializing in SQL and BigQuery analysis.

When invoked:

1. Understand the data analysis requirement
2. Write efficient SQL queries
3. Use BigQuery command line tools (bq) when appropriate
4. Analyze and summarize results
5. Present findings clearly

Key practices:

- Write optimized SQL queries with proper filters
- Use appropriate aggregations and joins
- Include comments explaining complex logic
- Format results for readability
- Provide data-driven recommendations

For each analysis:

- Explain the query approach
- Document any assumptions
- Highlight key findings
- Suggest next steps based on data

Always ensure queries are efficient and cost-effective.

...

## ## Best practices

\* \*\*Start with Claude-generated agents\*\*:

We highly recommend generating your initial subagent with Claude and then iterating on it to make it personally yours. This approach gives you the best results - a solid foundation that you can customize to your specific needs.

\* \*\*Design focused subagents\*\*:

Create subagents with single, clear responsibilities rather than trying to make one subagent do everything. This improves performance and makes subagents more predictable.

\* \*\*Write detailed prompts\*\*:

Include specific instructions, examples, and constraints in your system prompts. The more guidance you provide, the better the subagent will perform.

\* \*\*Limit tool access\*\*:

Only grant tools that are necessary for the subagent's purpose. This improves security and helps the subagent focus on relevant actions.

\* \*\*Version control\*\*:

Check project subagents into version control so your team can benefit from and improve them collaboratively.

## ## Advanced usage

### ### Chaining subagents

For complex workflows, you can chain multiple subagents:

...

> First use the code-analyzer subagent to find performance issues, then use the optimizer subagent to fix them

...



### ### Dynamic subagent selection

Claude Code intelligently selects subagents based on context. Make your `description` fields specific and action-oriented for best results.

### ## Performance considerations

\* **Context efficiency**: Agents help preserve main context, enabling longer overall sessions

\* **Latency**: Subagents start off with a clean slate each time they are invoked and may add latency as they gather context that they require to do their job effectively.

### ## Related documentation

\* [\[Slash commands\]](/en/docs/claude-code/slash-commands) - Learn about other built-in commands

\* [\[Settings\]](/en/docs/claude-code/settings) - Configure Claude Code behavior

\* [\[Hooks\]](/en/docs/claude-code/hooks) - Automate workflows with event handlers

### # Subagents

> Create and use specialized AI subagents in Claude Code for task-specific workflows and improved context management.

Custom subagents in Claude Code are specialized AI assistants that can be invoked to handle specific types of tasks. They enable more efficient problem-solving by providing task-specific configurations with customized system prompts, tools and a separate context window.

### ## What are subagents?

Subagents are pre-configured AI personalities that Claude Code can delegate tasks to. Each subagent:

- \* Has a specific purpose and expertise area
- \* Uses its own context window separate from the main conversation
- \* Can be configured with specific tools it's allowed to use
- \* Includes a custom system prompt that guides its behavior

When Claude Code encounters a task that matches a subagent's expertise, it can delegate that task to the specialized subagent, which works independently and returns results.

### ## Key benefits

<CardGroup cols={2}>

<Card title="Context preservation" icon="layer-group">

Each subagent operates in its own context, preventing pollution of the main conversation and keeping it focused on high-level objectives.

</Card>

<Card title="Specialized expertise" icon="brain">

Subagents can be fine-tuned with detailed instructions for specific domains, leading to higher success rates on designated tasks.

</Card>

<Card title="Reusability" icon="rotate">

Once created, subagents can be used across different projects and shared with your team for consistent workflows.

</Card>

<Card title="Flexible permissions" icon="shield-check">

Each subagent can have different tool access levels, allowing you to limit powerful tools to specific subagent types.

</Card>

</CardGroup>

### ## Quick start

To create your first subagent:

<Steps>

<Step title="Open the subagents interface">

Run the following command:

...

/agents

...

</Step>

<Step title="Select 'Create New Agent'">

Choose whether to create a project-level or user-level subagent

</Step>

<Step title="Define the subagent">

\* **Recommended**: Generate with Claude first, then customize to make it yours

\* Describe your subagent in detail and when it should be used

\* Select the tools you want to grant access to (or leave blank to inherit all tools)

\* The interface shows all available tools, making selection easy

\* If you're generating with Claude, you can also edit the system prompt in your own editor by pressing `e`

</Step>

<Step title="Save and use">

Your subagent is now available! Claude will use it automatically when appropriate, or you can invoke it explicitly:

...

> Use the code-reviewer subagent to check my recent changes

...

</Step>

</Steps>

## Subagent configuration

### File locations

Subagents are stored as Markdown files with YAML frontmatter in two possible locations:

| Type                              | Location                             | Scope                         | Priority |
|-----------------------------------|--------------------------------------|-------------------------------|----------|
| :-----   :-----   :-----   :----- |                                      |                               |          |
| <b>Project subagents</b>          | <code>`.claude/agents/`</code>       | Available in current project  | Highest  |
| <b>User subagents</b>             | <code>`\${~}/.claude/agents/`</code> | Available across all projects | Lower    |

When subagent names conflict, project-level subagents take precedence over user-level subagents.

### File format

Each subagent is defined in a Markdown file with this structure:

```markdown

---

name: your-sub-agent-name

description: Description of when this subagent should be invoked

tools: tool1, tool2, tool3 # Optional - inherits all tools if omitted

---

Your subagent's system prompt goes here. This can be multiple paragraphs and should clearly define the subagent's role, capabilities, and approach to solving problems.

Include specific instructions, best practices, and any constraints the subagent should follow.

...

#### #### Configuration fields

| Field         | Required | Description                                                                                 |
|---------------|----------|---------------------------------------------------------------------------------------------|
| :-----        | :-----   | :-----                                                                                      |
| `name`        | Yes      | Unique identifier using lowercase letters and hyphens                                       |
| `description` | Yes      | Natural language description of the subagent's purpose                                      |
| `tools`       | No       | Comma-separated list of specific tools. If omitted, inherits all tools from the main thread |

#### ### Available tools

Subagents can be granted access to any of Claude Code's internal tools. See the [tools documentation](/en/docs/claude-code/settings#tools-available-to-claude) for a complete list of available tools.

<Tip>

**\*\*Recommended:\*\*** Use the `/agents`` command to modify tool access - it provides an interactive interface that lists all available tools, including any connected MCP server tools, making it easier to select the ones you need.

</Tip>

You have two options for configuring tools:

\* **\*\*Omit the ``tools`` field\*\*** to inherit all tools from the main thread (default), including MCP tools

\* **\*\*Specify individual tools\*\*** as a comma-separated list for more granular control (can be edited manually or via `/agents``)

**\*\*MCP Tools\*\***: Subagents can access MCP tools from configured MCP servers. When the ``tools`` field is omitted, subagents inherit all MCP tools available to the main thread.

## ## Managing subagents

### ### Using the `/agents` command (Recommended)

The `/agents`` command provides a comprehensive interface for subagent management:

...

`/agents`

...

This opens an interactive menu where you can:

- \* View all available subagents (built-in, user, and project)
- \* Create new subagents with guided setup
- \* Edit existing custom subagents, including their tool access

- \* Delete custom subagents
- \* See which subagents are active when duplicates exist
- \* \*\*Easily manage tool permissions\*\* with a complete list of available tools

### ### Direct file management

You can also manage subagents by working directly with their files:

```
```bash

# Create a project subagent

mkdir -p .claude/agents

echo '---

name: test-runner

description: Use proactively to run tests and fix failures

---'
```

You are a test automation expert. When you see code changes, proactively run the appropriate tests. If tests fail, analyze the failures and fix them while preserving the original test intent.' > .claude/agents/test-runner.md

### # Create a user subagent

```
mkdir -p ~/.claude/agents

# ... create subagent file

...
```

## ## Using subagents effectively

### ### Automatic delegation

Claude Code proactively delegates tasks based on:

- \* The task description in your request
- \* The `description` field in subagent configurations
- \* Current context and available tools

<Tip>

To encourage more proactive subagent use, include phrases like "use PROACTIVELY" or "MUST BE USED" in your `description` field.

</Tip>

### Explicit invocation

Request a specific subagent by mentioning it in your command:

...

- > Use the test-runner subagent to fix failing tests
- > Have the code-reviewer subagent look at my recent changes
- > Ask the debugger subagent to investigate this error

...

## Example subagents

### Code reviewer

```markdown

---

name: code-reviewer



description: Expert code review specialist. Proactively reviews code for quality, security, and maintainability. Use immediately after writing or modifying code.

tools: Read, Grep, Glob, Bash

---

You are a senior code reviewer ensuring high standards of code quality and security.

When invoked:

1. Run git diff to see recent changes
2. Focus on modified files
3. Begin review immediately

Review checklist:

- Code is simple and readable
- Functions and variables are well-named
- No duplicated code
- Proper error handling
- No exposed secrets or API keys
- Input validation implemented
- Good test coverage
- Performance considerations addressed

Provide feedback organized by priority:

- Critical issues (must fix)
- Warnings (should fix)
- Suggestions (consider improving)

Include specific examples of how to fix issues.

...

### ### Debugger

```markdown

---

name: debugger

description: Debugging specialist for errors, test failures, and unexpected behavior. Use proactively when encountering any issues.

tools: Read, Edit, Bash, Grep, Glob

---

You are an expert debugger specializing in root cause analysis.

When invoked:

1. Capture error message and stack trace
2. Identify reproduction steps
3. Isolate the failure location
4. Implement minimal fix
5. Verify solution works

Debugging process:

- Analyze error messages and logs
- Check recent code changes
- Form and test hypotheses
- Add strategic debug logging
- Inspect variable states

For each issue, provide:

- Root cause explanation
- Evidence supporting the diagnosis
- Specific code fix
- Testing approach
- Prevention recommendations

Focus on fixing the underlying issue, not just symptoms.

...

### Data scientist

```markdown

---

name: data-scientist

description: Data analysis expert for SQL queries, BigQuery operations, and data insights. Use proactively for data analysis tasks and queries.

tools: Bash, Read, Write

---

You are a data scientist specializing in SQL and BigQuery analysis.

When invoked:

1. Understand the data analysis requirement
2. Write efficient SQL queries
3. Use BigQuery command line tools (bq) when appropriate
4. Analyze and summarize results
5. Present findings clearly

Key practices:

- Write optimized SQL queries with proper filters
- Use appropriate aggregations and joins
- Include comments explaining complex logic
- Format results for readability
- Provide data-driven recommendations

For each analysis:

- Explain the query approach
- Document any assumptions
- Highlight key findings
- Suggest next steps based on data

Always ensure queries are efficient and cost-effective.

...

### ## Best practices

\* \*\*Start with Claude-generated agents\*\*: We highly recommend generating your initial subagent with Claude and then iterating on it to make it personally yours. This approach gives you the best results - a solid foundation that you can customize to your specific needs.

\* \*\*Design focused subagents\*\*: Create subagents with single, clear responsibilities rather than trying to make one subagent do everything. This improves performance and makes subagents more predictable.

\* \*\*Write detailed prompts\*\*: Include specific instructions, examples, and constraints in your system prompts. The more guidance you provide, the better the subagent will perform.

\* \*\*Limit tool access\*\*\*: Only grant tools that are necessary for the subagent's purpose. This improves security and helps the subagent focus on relevant actions.

\* \*\*Version control\*\*\*: Check project subagents into version control so your team can benefit from and improve them collaboratively.

## ## Advanced usage

### ### Chaining subagents

For complex workflows, you can chain multiple subagents:

...

> First use the code-analyzer subagent to find performance issues, then use the optimizer subagent to fix them

...

### ### Dynamic subagent selection

Claude Code intelligently selects subagents based on context. Make your `description` fields specific and action-oriented for best results.

## ## Performance considerations

\* \*\*Context efficiency\*\*\*: Agents help preserve main context, enabling longer overall sessions

\* \*\*Latency\*\*\*: Subagents start off with a clean slate each time they are invoked and may add latency as they gather context that they require to do their job effectively.

## ## Related documentation

- \* [Slash commands](/en/docs/claude-code/slash-commands) - Learn about other built-in commands
- \* [Settings](/en/docs/claude-code/settings) - Configure Claude Code behavior
- \* [Hooks](/en/docs/claude-code/hooks) - Automate workflows with event handlers

## # Subagents

> Create and use specialized AI subagents in Claude Code for task-specific workflows and improved context management.

Custom subagents in Claude Code are specialized AI assistants that can be invoked to handle specific types of tasks. They enable more efficient problem-solving by providing task-specific configurations with customized system prompts, tools and a separate context window.

### ## What are subagents?

Subagents are pre-configured AI personalities that Claude Code can delegate tasks to. Each subagent:

- \* Has a specific purpose and expertise area
- \* Uses its own context window separate from the main conversation
- \* Can be configured with specific tools it's allowed to use
- \* Includes a custom system prompt that guides its behavior

When Claude Code encounters a task that matches a subagent's expertise, it can delegate that task to the specialized subagent, which works independently and returns results.

### ## Key benefits

<CardGroup cols={2}>

<Card title="Context preservation" icon="layer-group">

Each subagent operates in its own context, preventing pollution of the main conversation and keeping it focused on high-level objectives.

</Card>

<Card title="Specialized expertise" icon="brain">

Subagents can be fine-tuned with detailed instructions for specific domains, leading to higher success rates on designated tasks.

</Card>

<Card title="Reusability" icon="rotate">

Once created, subagents can be used across different projects and shared with your team for consistent workflows.

</Card>

<Card title="Flexible permissions" icon="shield-check">

Each subagent can have different tool access levels, allowing you to limit powerful tools to specific subagent types.

</Card>

</CardGroup>

## ## Quick start

To create your first subagent:

<Steps>

<Step title="Open the subagents interface">

Run the following command:

...

/agents

...

</Step>

<Step title="Select 'Create New Agent'">

Choose whether to create a project-level or user-level subagent

</Step>

<Step title="Define the subagent">

\* **Recommended**: Generate with Claude first, then customize to make it yours

\* Describe your subagent in detail and when it should be used

\* Select the tools you want to grant access to (or leave blank to inherit all tools)

\* The interface shows all available tools, making selection easy

\* If you're generating with Claude, you can also edit the system prompt in your own editor by pressing `e`

</Step>

<Step title="Save and use">

Your subagent is now available! Claude will use it automatically when appropriate, or you can invoke it explicitly:

...

> Use the code-reviewer subagent to check my recent changes

...

</Step>

</Steps>

## Subagent configuration

### File locations



Subagents are stored as Markdown files with YAML frontmatter in two possible locations:

| Type                  | Location            | Scope                         | Priority |
|-----------------------|---------------------|-------------------------------|----------|
| :-----                | :-----              | :-----                        | :-----   |
| **Project subagents** | `\.claude/agents/`  | Available in current project  | Highest  |
| **User subagents**    | `~/.claude/agents/` | Available across all projects | Lower    |

When subagent names conflict, project-level subagents take precedence over user-level subagents.

### File format

Each subagent is defined in a Markdown file with this structure:

```
```markdown
---
name: your-sub-agent-name

description: Description of when this subagent should be invoked

tools: tool1, tool2, tool3 # Optional - inherits all tools if omitted
---
```

Your subagent's system prompt goes here. This can be multiple paragraphs and should clearly define the subagent's role, capabilities, and approach to solving problems.

Include specific instructions, best practices, and any constraints the subagent should follow.

...

#### Configuration fields

Field	Required	Description
:-----	:-----	:-----
`name`	Yes	Unique identifier using lowercase letters and hyphens
`description`	Yes	Natural language description of the subagent's purpose
`tools`	No	Comma-separated list of specific tools. If omitted, inherits all tools from the main thread

### Available tools

Subagents can be granted access to any of Claude Code's internal tools. See the [tools documentation](/en/docs/claude-code/settings#tools-available-to-claude) for a complete list of available tools.

<Tip>

**Recommended:** Use the `/agents` command to modify tool access - it provides an interactive interface that lists all available tools, including any connected MCP server tools, making it easier to select the ones you need.

</Tip>

You have two options for configuring tools:

**Omit the `tools` field** to inherit all tools from the main thread (default), including MCP tools

**Specify individual tools** as a comma-separated list for more granular control (can be edited manually or via `/agents`)

**MCP Tools:** Subagents can access MCP tools from configured MCP servers. When the `tools` field is omitted, subagents inherit all MCP tools available to the main thread.

## ## Managing subagents

### ### Using the /agents command (Recommended)

The `/agents`` command provides a comprehensive interface for subagent management:

...

`/agents`

...

This opens an interactive menu where you can:

- \* View all available subagents (built-in, user, and project)
- \* Create new subagents with guided setup
- \* Edit existing custom subagents, including their tool access
- \* Delete custom subagents
- \* See which subagents are active when duplicates exist
- \* **Easily manage tool permissions** with a complete list of available tools

### ### Direct file management

You can also manage subagents by working directly with their files:

```
```bash
```

```
# Create a project subagent
```

```
mkdir -p .claude/agents
```

```
echo '---
```

name: test-runner

description: Use proactively to run tests and fix failures

---

You are a test automation expert. When you see code changes, proactively run the appropriate tests. If tests fail, analyze the failures and fix them while preserving the original test intent.' > .claude/agents/test-runner.md

# Create a user subagent

mkdir -p ~/.claude/agents

# ... create subagent file

...

## Using subagents effectively

### Automatic delegation

Claude Code proactively delegates tasks based on:

- \* The task description in your request
- \* The `description` field in subagent configurations
- \* Current context and available tools

<Tip>

To encourage more proactive subagent use, include phrases like "use PROACTIVELY" or "MUST BE USED" in your `description` field.

</Tip>

### Explicit invocation

Request a specific subagent by mentioning it in your command:

...

- > Use the test-runner subagent to fix failing tests
- > Have the code-reviewer subagent look at my recent changes
- > Ask the debugger subagent to investigate this error

...

## ## Example subagents

### ### Code reviewer

```markdown

---

name: code-reviewer

description: Expert code review specialist. Proactively reviews code for quality, security, and maintainability. Use immediately after writing or modifying code.

tools: Read, Grep, Glob, Bash

---

You are a senior code reviewer ensuring high standards of code quality and security.

When invoked:

1. Run git diff to see recent changes
2. Focus on modified files
3. Begin review immediately

Review checklist:

- Code is simple and readable
- Functions and variables are well-named
- No duplicated code
- Proper error handling
- No exposed secrets or API keys
- Input validation implemented
- Good test coverage
- Performance considerations addressed

Provide feedback organized by priority:

- Critical issues (must fix)
- Warnings (should fix)
- Suggestions (consider improving)

Include specific examples of how to fix issues.

...

### Debugger

```markdown

---

name: debugger

description: Debugging specialist for errors, test failures, and unexpected behavior. Use proactively when encountering any issues.

tools: Read, Edit, Bash, Grep, Glob

---

You are an expert debugger specializing in root cause analysis.

When invoked:

1. Capture error message and stack trace
2. Identify reproduction steps
3. Isolate the failure location
4. Implement minimal fix
5. Verify solution works

Debugging process:

- Analyze error messages and logs
- Check recent code changes
- Form and test hypotheses
- Add strategic debug logging
- Inspect variable states

For each issue, provide:

- Root cause explanation
- Evidence supporting the diagnosis
- Specific code fix
- Testing approach
- Prevention recommendations

Focus on fixing the underlying issue, not just symptoms.

...

### Data scientist

```markdown

---

name: data-scientist

description: Data analysis expert for SQL queries, BigQuery operations, and data insights. Use proactively for data analysis tasks and queries.

tools: Bash, Read, Write

---

You are a data scientist specializing in SQL and BigQuery analysis.

When invoked:

1. Understand the data analysis requirement
2. Write efficient SQL queries
3. Use BigQuery command line tools (bq) when appropriate
4. Analyze and summarize results
5. Present findings clearly

Key practices:

- Write optimized SQL queries with proper filters
- Use appropriate aggregations and joins
- Include comments explaining complex logic
- Format results for readability
- Provide data-driven recommendations

For each analysis:

- Explain the query approach
- Document any assumptions



- Highlight key findings
- Suggest next steps based on data

Always ensure queries are efficient and cost-effective.

...

## ## Best practices

\* \*\*Start with Claude-generated agents\*\*:

We highly recommend generating your initial subagent with Claude and then iterating on it to make it personally yours. This approach gives you the best results - a solid foundation that you can customize to your specific needs.

\* \*\*Design focused subagents\*\*:

Create subagents with single, clear responsibilities rather than trying to make one subagent do everything. This improves performance and makes subagents more predictable.

\* \*\*Write detailed prompts\*\*:

Include specific instructions, examples, and constraints in your system prompts. The more guidance you provide, the better the subagent will perform.

\* \*\*Limit tool access\*\*:

Only grant tools that are necessary for the subagent's purpose. This improves security and helps the subagent focus on relevant actions.

\* \*\*Version control\*\*:

Check project subagents into version control so your team can benefit from and improve them collaboratively.

## ## Advanced usage

### ### Chaining subagents

For complex workflows, you can chain multiple subagents:

...

> First use the code-analyzer subagent to find performance issues, then use the optimizer subagent to fix them

...

### ### Dynamic subagent selection

Claude Code intelligently selects subagents based on context. Make your `description` fields specific and action-oriented for best results.

## ## Performance considerations

\* **Context efficiency**: Agents help preserve main context, enabling longer overall sessions

\* **Latency**: Subagents start off with a clean slate each time they are invoked and may add latency as they gather context that they require to do their job effectively.

## ## Related documentation

\* [\[Slash commands\]](/en/docs/claude-code/slash-commands) - Learn about other built-in commands

\* [\[Settings\]](/en/docs/claude-code/settings) - Configure Claude Code behavior

\* [\[Hooks\]](/en/docs/claude-code/hooks) - Automate workflows with event handlers

## # Subagents

> Create and use specialized AI subagents in Claude Code for task-specific workflows and improved context management.

Custom subagents in Claude Code are specialized AI assistants that can be invoked to handle specific types of tasks. They enable more efficient problem-solving by providing task-specific configurations with customized system prompts, tools and a separate context window.

### ## What are subagents?

Subagents are pre-configured AI personalities that Claude Code can delegate tasks to. Each subagent:

- \* Has a specific purpose and expertise area
- \* Uses its own context window separate from the main conversation
- \* Can be configured with specific tools it's allowed to use
- \* Includes a custom system prompt that guides its behavior

When Claude Code encounters a task that matches a subagent's expertise, it can delegate that task to the specialized subagent, which works independently and returns results.

### ## Key benefits

<CardGroup cols={2}>

<Card title="Context preservation" icon="layer-group">

Each subagent operates in its own context, preventing pollution of the main conversation and keeping it focused on high-level objectives.

</Card>

<Card title="Specialized expertise" icon="brain">

Subagents can be fine-tuned with detailed instructions for specific domains, leading to higher success rates on designated tasks.

</Card>

<Card title="Reusability" icon="rotate">

Once created, subagents can be used across different projects and shared with your team for consistent workflows.

</Card>

<Card title="Flexible permissions" icon="shield-check">

Each subagent can have different tool access levels, allowing you to limit powerful tools to specific subagent types.

</Card>

</CardGroup>

## ## Quick start

To create your first subagent:

<Steps>

<Step title="Open the subagents interface">

Run the following command:

...

/agents

...

</Step>

<Step title="Select 'Create New Agent'">

Choose whether to create a project-level or user-level subagent

</Step>

<Step title="Define the subagent">

\* **Recommended**: Generate with Claude first, then customize to make it yours

\* Describe your subagent in detail and when it should be used

\* Select the tools you want to grant access to (or leave blank to inherit all tools)

- \* The interface shows all available tools, making selection easy
- \* If you're generating with Claude, you can also edit the system prompt in your own editor by pressing `e`

</Step>

<Step title="Save and use">

Your subagent is now available! Claude will use it automatically when appropriate, or you can invoke it explicitly:

...

> Use the code-reviewer subagent to check my recent changes

...

</Step>

</Steps>

## Subagent configuration

### File locations

Subagents are stored as Markdown files with YAML frontmatter in two possible locations:

| Type                  | Location                   | Scope                         | Priority |
|-----------------------|----------------------------|-------------------------------|----------|
| :-----                | :-----                     | :-----                        | :-----   |
| **Project subagents** | `.claude/agents/`          | Available in current project  | Highest  |
| **User subagents**    | `\${HOME}/.claude/agents/` | Available across all projects | Lower    |

When subagent names conflict, project-level subagents take precedence over user-level subagents.

### File format

Each subagent is defined in a Markdown file with this structure:

```
```markdown
---
name: your-sub-agent-name

description: Description of when this subagent should be invoked

tools: tool1, tool2, tool3 # Optional - inherits all tools if omitted
---
```

Your subagent's system prompt goes here. This can be multiple paragraphs and should clearly define the subagent's role, capabilities, and approach to solving problems.

Include specific instructions, best practices, and any constraints the subagent should follow.

...

#### Configuration fields

Field	Required	Description
:-----	:-----	:-----
`name`	Yes	Unique identifier using lowercase letters and hyphens
`description`	Yes	Natural language description of the subagent's purpose
`tools`	No	Comma-separated list of specific tools. If omitted, inherits all tools from the main thread

### Available tools

Subagents can be granted access to any of Claude Code's internal tools. See the [tools documentation](/en/docs/claude-code/settings#tools-available-to-claude) for a complete list of available tools.

<Tip>

**\*\*Recommended:\*\*** Use the `/agents`` command to modify tool access - it provides an interactive interface that lists all available tools, including any connected MCP server tools, making it easier to select the ones you need.

</Tip>

You have two options for configuring tools:

\* **\*\*Omit the ``tools`` field\*\*** to inherit all tools from the main thread (default), including MCP tools

\* **\*\*Specify individual tools\*\*** as a comma-separated list for more granular control (can be edited manually or via `/agents``)

**\*\*MCP Tools\*\***: Subagents can access MCP tools from configured MCP servers. When the ``tools`` field is omitted, subagents inherit all MCP tools available to the main thread.

## ## Managing subagents

### ### Using the `/agents` command (Recommended)

The `/agents`` command provides a comprehensive interface for subagent management:

...

`/agents`

...

This opens an interactive menu where you can:

- \* View all available subagents (built-in, user, and project)
- \* Create new subagents with guided setup
- \* Edit existing custom subagents, including their tool access
- \* Delete custom subagents
- \* See which subagents are active when duplicates exist
- \* **Easily manage tool permissions** with a complete list of available tools

### ### Direct file management

You can also manage subagents by working directly with their files:

```
```bash
```

```
# Create a project subagent
```

```
mkdir -p .claude/agents
```

```
echo '---
```

```
name: test-runner
```

```
description: Use proactively to run tests and fix failures
```

```
---
```

You are a test automation expert. When you see code changes, proactively run the appropriate tests. If tests fail, analyze the failures and fix them while preserving the original test intent.' > .claude/agents/test-runner.md

```
# Create a user subagent
```

```
mkdir -p ~/.claude/agents
```

```
# ... create subagent file
```

```
...
```



## ## Using subagents effectively

### ### Automatic delegation

Claude Code proactively delegates tasks based on:

- \* The task description in your request
- \* The `description` field in subagent configurations
- \* Current context and available tools

<Tip>

To encourage more proactive subagent use, include phrases like "use PROACTIVELY" or "MUST BE USED" in your `description` field.

</Tip>

### ### Explicit invocation

Request a specific subagent by mentioning it in your command:

...

- > Use the test-runner subagent to fix failing tests
- > Have the code-reviewer subagent look at my recent changes
- > Ask the debugger subagent to investigate this error

...

## ## Example subagents

### ### Code reviewer

```markdown

---

name: code-reviewer

description: Expert code review specialist. Proactively reviews code for quality, security, and maintainability. Use immediately after writing or modifying code.

tools: Read, Grep, Glob, Bash

---

You are a senior code reviewer ensuring high standards of code quality and security.

When invoked:

1. Run git diff to see recent changes
2. Focus on modified files
3. Begin review immediately

Review checklist:

- Code is simple and readable
- Functions and variables are well-named
- No duplicated code
- Proper error handling
- No exposed secrets or API keys
- Input validation implemented
- Good test coverage
- Performance considerations addressed

Provide feedback organized by priority:

- Critical issues (must fix)

- Warnings (should fix)
- Suggestions (consider improving)

Include specific examples of how to fix issues.

...

### Debugger

```markdown

---

name: debugger

description: Debugging specialist for errors, test failures, and unexpected behavior. Use proactively when encountering any issues.

tools: Read, Edit, Bash, Grep, Glob

---

You are an expert debugger specializing in root cause analysis.

When invoked:

1. Capture error message and stack trace
2. Identify reproduction steps
3. Isolate the failure location
4. Implement minimal fix
5. Verify solution works

Debugging process:

- Analyze error messages and logs
- Check recent code changes

- Form and test hypotheses
- Add strategic debug logging
- Inspect variable states

For each issue, provide:

- Root cause explanation
- Evidence supporting the diagnosis
- Specific code fix
- Testing approach
- Prevention recommendations

Focus on fixing the underlying issue, not just symptoms.

...

### Data scientist

```markdown

---

name: data-scientist

description: Data analysis expert for SQL queries, BigQuery operations, and data insights. Use proactively for data analysis tasks and queries.

tools: Bash, Read, Write

---

You are a data scientist specializing in SQL and BigQuery analysis.

When invoked:

1. Understand the data analysis requirement

2. Write efficient SQL queries
3. Use BigQuery command line tools (bq) when appropriate
4. Analyze and summarize results
5. Present findings clearly

Key practices:

- Write optimized SQL queries with proper filters
- Use appropriate aggregations and joins
- Include comments explaining complex logic
- Format results for readability
- Provide data-driven recommendations

For each analysis:

- Explain the query approach
- Document any assumptions
- Highlight key findings
- Suggest next steps based on data

Always ensure queries are efficient and cost-effective.

...

## ## Best practices

\* \*\*Start with Claude-generated agents\*\*: We highly recommend generating your initial subagent with Claude and then iterating on it to make it personally yours. This approach gives you the best results - a solid foundation that you can customize to your specific needs.

\* \*\*Design focused subagents\*\*: Create subagents with single, clear responsibilities rather than trying to make one subagent do everything. This improves performance and makes subagents more predictable.

\* \*\*Write detailed prompts\*\*\*: Include specific instructions, examples, and constraints in your system prompts. The more guidance you provide, the better the subagent will perform.

\* \*\*Limit tool access\*\*\*: Only grant tools that are necessary for the subagent's purpose. This improves security and helps the subagent focus on relevant actions.

\* \*\*Version control\*\*\*: Check project subagents into version control so your team can benefit from and improve them collaboratively.

## ## Advanced usage

### ### Chaining subagents

For complex workflows, you can chain multiple subagents:

...

> First use the code-analyzer subagent to find performance issues, then use the optimizer subagent to fix them

...

### ### Dynamic subagent selection

Claude Code intelligently selects subagents based on context. Make your `description` fields specific and action-oriented for best results.

## ## Performance considerations

\* \*\*Context efficiency\*\*\*: Agents help preserve main context, enabling longer overall sessions

\* \*\*Latency\*\*\*: Subagents start off with a clean slate each time they are invoked and may add latency as they gather context that they require to do their job effectively.

## ## Related documentation

- \* [Slash commands](/en/docs/claude-code/slash-commands) - Learn about other built-in commands
- \* [Settings](/en/docs/claude-code/settings) - Configure Claude Code behavior
- \* [Hooks](/en/docs/claude-code/hooks) - Automate workflows with event handlers

## # Subagents

**> Create and use specialized AI subagents in Claude Code for task-specific workflows and improved context management.**

Custom subagents in Claude Code are specialized AI assistants that can be invoked to handle specific types of tasks. They enable more efficient problem-solving by providing task-specific configurations with customized system prompts, tools and a separate context window.

## ## What are subagents?

Subagents are pre-configured AI personalities that Claude Code can delegate tasks to. Each subagent:

- \* Has a specific purpose and expertise area
- \* Uses its own context window separate from the main conversation
- \* Can be configured with specific tools it's allowed to use
- \* Includes a custom system prompt that guides its behavior

When Claude Code encounters a task that matches a subagent's expertise, it can delegate that task to the specialized subagent, which works independently and returns results.

## ## Key benefits

**<CardGroup cols={2}>**

**<Card title="Context preservation" icon="layer-group">**

Each subagent operates in its own context, preventing pollution of the main conversation and keeping it focused on high-level objectives.

**</Card>**

**<Card title="Specialized expertise" icon="brain">**

Subagents can be fine-tuned with detailed instructions for specific domains, leading to higher success rates on designated tasks.

**</Card>**

**<Card title="Reusability" icon="rotate">**

Once created, subagents can be used across different projects and shared with your team for consistent workflows.

**</Card>**

**<Card title="Flexible permissions" icon="shield-check">**

Each subagent can have different tool access levels, allowing you to limit powerful tools to specific subagent types.

**</Card>**

**</CardGroup>**

## **## Quick start**

**To create your first subagent:**

**<Steps>**

**<Step title="Open the subagents interface">**



Run the following command:

...

/agents

...

</Step>

<Step title="Select 'Create New Agent'">

Choose whether to create a project-level or user-level subagent

</Step>

<Step title="Define the subagent">

\* **Recommended**: Generate with Claude first, then customize to make it yours

\* Describe your subagent in detail and when it should be used

\* Select the tools you want to grant access to (or leave blank to inherit all tools)

\* The interface shows all available tools, making selection easy

\* If you're generating with Claude, you can also edit the system prompt in your own editor by pressing `e`

</Step>

<Step title="Save and use">

Your subagent is now available! Claude will use it automatically when appropriate, or you can invoke it explicitly:

...

> Use the code-reviewer subagent to check my recent changes

...

</Step>

</Steps>

## Subagent configuration

### File locations

Subagents are stored as Markdown files with YAML frontmatter in two possible locations:

| Type              | Location          | Scope                         | Priority |
|-------------------|-------------------|-------------------------------|----------|
| Project subagents | ~/.claude/agents/ | Available in current project  | Highest  |
| User subagents    | ~/.claude/agents/ | Available across all projects | Lower    |

When subagent names conflict, project-level subagents take precedence over user-level subagents.

### File format

Each subagent is defined in a Markdown file with this structure:

```
```markdown
---
name: your-sub-agent-name

description: Description of when this subagent should be invoked

tools: tool1, tool2, tool3 # Optional - inherits all tools if omitted
---
```

Your subagent's system prompt goes here. This can be multiple paragraphs and should clearly define the subagent's role, capabilities, and approach

to solving problems.

Include specific instructions, best practices, and any constraints the subagent should follow.

...

#### Configuration fields

Field	Required	Description
:-----	:-----	:-----
`name`	Yes	Unique identifier using lowercase letters and hyphens
`description`	Yes	Natural language description of the subagent's purpose
`tools`	No	Comma-separated list of specific tools. If omitted, inherits all tools from the main thread

### Available tools

Subagents can be granted access to any of Claude Code's internal tools. See the [tools documentation](/en/docs/claude-code/settings#tools-available-to-claude) for a complete list of available tools.

<Tip>

**Recommended:** Use the `/agents` command to modify tool access - it provides an interactive interface that lists all available tools, including any connected MCP server tools, making it easier to select the ones you need.

</Tip>

You have two options for configuring tools:

**Omit the ``tools`` field** to inherit all tools from the main thread (default), including MCP tools

\* **Specify individual tools** as a comma-separated list for more granular control (can be edited manually or via `/agents``)

**MCP Tools**: Subagents can access MCP tools from configured MCP servers. When the ``tools`` field is omitted, subagents inherit all MCP tools available to the main thread.

## ## Managing subagents

### ### Using the `/agents` command (Recommended)

The ``/agents`` command provides a comprehensive interface for subagent management:

...

`/agents`

...

This opens an interactive menu where you can:

- \* View all available subagents (built-in, user, and project)
- \* Create new subagents with guided setup
- \* Edit existing custom subagents, including their tool access
- \* Delete custom subagents
- \* See which subagents are active when duplicates exist
- \* **Easily manage tool permissions** with a complete list of available tools

### ### Direct file management

You can also manage subagents by working directly with their files:

```
```bash
```

```
# Create a project subagent
```

```
mkdir -p .claude/agents
```

```
echo '---
```

```
name: test-runner
```

```
description: Use proactively to run tests and fix failures
```

```
---
```

You are a test automation expert. When you see code changes, proactively run the appropriate tests. If tests fail, analyze the failures and fix them while preserving the original test intent.' >

.claude/agents/test-runner.md

```
# Create a user subagent
```

```
mkdir -p ~/.claude/agents
```

```
# ... create subagent file
```

```
...
```

```
## Using subagents effectively
```

```
### Automatic delegation
```

Claude Code proactively delegates tasks based on:

- \* The task description in your request

- \* The `description` field in subagent configurations

- \* Current context and available tools

<Tip>

To encourage more proactive subagent use, include phrases like "use PROACTIVELY" or "MUST BE USED" in your `description` field.

</Tip>

### ### Explicit invocation

Request a specific subagent by mentioning it in your command:

...

> Use the test-runner subagent to fix failing tests

> Have the code-reviewer subagent look at my recent changes

> Ask the debugger subagent to investigate this error

...

## ## Example subagents

### ### Code reviewer

```markdown

---

name: code-reviewer

description: Expert code review specialist. Proactively reviews code for quality, security, and maintainability. Use immediately after writing or modifying code.

tools: Read, Grep, Glob, Bash

---

You are a senior code reviewer ensuring high standards of code quality and security.

**When invoked:**

- 1. Run git diff to see recent changes**
- 2. Focus on modified files**
- 3. Begin review immediately**

**Review checklist:**

- Code is simple and readable**
- Functions and variables are well-named**
- No duplicated code**
- Proper error handling**
- No exposed secrets or API keys**
- Input validation implemented**
- Good test coverage**
- Performance considerations addressed**

**Provide feedback organized by priority:**

- Critical issues (must fix)**
- Warnings (should fix)**
- Suggestions (consider improving)**

**Include specific examples of how to fix issues.**

...

**### Debugger**

```markdown

---

**name: debugger**

**description: Debugging specialist for errors, test failures, and unexpected behavior. Use proactively when encountering any issues.**

**tools: Read, Edit, Bash, Grep, Glob**

---

**You are an expert debugger specializing in root cause analysis.**

**When invoked:**

- 1. Capture error message and stack trace**
- 2. Identify reproduction steps**
- 3. Isolate the failure location**
- 4. Implement minimal fix**
- 5. Verify solution works**

**Debugging process:**

- Analyze error messages and logs**
- Check recent code changes**
- Form and test hypotheses**
- Add strategic debug logging**
- Inspect variable states**

**For each issue, provide:**

- Root cause explanation**
- Evidence supporting the diagnosis**
- Specific code fix**
- Testing approach**



- Prevention recommendations

Focus on fixing the underlying issue, not just symptoms.

...

**### Data scientist**

```markdown

---

**name: data-scientist**

**description:** Data analysis expert for SQL queries, BigQuery operations, and data insights. Use proactively for data analysis tasks and queries.

**tools:** Bash, Read, Write

---

You are a data scientist specializing in SQL and BigQuery analysis.

**When invoked:**

1. Understand the data analysis requirement
2. Write efficient SQL queries
3. Use BigQuery command line tools (bq) when appropriate
4. Analyze and summarize results
5. Present findings clearly

**Key practices:**

- Write optimized SQL queries with proper filters
- Use appropriate aggregations and joins
- Include comments explaining complex logic

- Format results for readability
- Provide data-driven recommendations

For each analysis:

- Explain the query approach
- Document any assumptions
- Highlight key findings
- Suggest next steps based on data

Always ensure queries are efficient and cost-effective.

...

## **## Best practices**

**\* \*\*Start with Claude-generated agents\*\*:** We highly recommend generating your initial subagent with Claude and then iterating on it to make it personally yours. This approach gives you the best results - a solid foundation that you can customize to your specific needs.

**\* \*\*Design focused subagents\*\*:** Create subagents with single, clear responsibilities rather than trying to make one subagent do everything. This improves performance and makes subagents more predictable.

**\* \*\*Write detailed prompts\*\*:** Include specific instructions, examples, and constraints in your system prompts. The more guidance you provide, the better the subagent will perform.

**\* \*\*Limit tool access\*\*:** Only grant tools that are necessary for the subagent's purpose. This improves security and helps the subagent focus on relevant actions.

**\* \*\*Version control\*\*:** Check project subagents into version control so your team can benefit from and improve them collaboratively.

## ## Advanced usage

### ### Chaining subagents

For complex workflows, you can chain multiple subagents:

...

> First use the code-analyzer subagent to find performance issues, then use the optimizer subagent to fix them

...

### ### Dynamic subagent selection

Claude Code intelligently selects subagents based on context. Make your `description` fields specific and action-oriented for best results.

## ## Performance considerations

\* **Context efficiency**: Agents help preserve main context, enabling longer overall sessions

\* **Latency**: Subagents start off with a clean slate each time they are invoked and may add latency as they gather context that they require to do their job effectively.

## ## Related documentation

\* [\[Slash commands\]](/en/docs/claude-code/slash-commands) - Learn about other built-in commands

\* [\[Settings\]](/en/docs/claude-code/settings) - Configure Claude Code behavior

\* [\[Hooks\]](/en/docs/claude-code/hooks) - Automate workflows with event handlers

## # Subagents

> Create and use specialized AI subagents in Claude Code for task-specific workflows and improved context management.

Custom subagents in Claude Code are specialized AI assistants that can be invoked to handle specific types of tasks. They enable more efficient problem-solving by providing task-specific configurations with customized system prompts, tools and a separate context window.

## ## What are subagents?

Subagents are pre-configured AI personalities that Claude Code can delegate tasks to. Each subagent:

- \* Has a specific purpose and expertise area
- \* Uses its own context window separate from the main conversation
- \* Can be configured with specific tools it's allowed to use
- \* Includes a custom system prompt that guides its behavior

When Claude Code encounters a task that matches a subagent's expertise, it can delegate that task to the specialized subagent, which works independently and returns results.

## ## Key benefits

<CardGroup cols={2}>

<Card title="Context preservation" icon="layer-group">

Each subagent operates in its own context, preventing pollution of the main conversation and keeping it focused on high-level objectives.

</Card>

<Card title="Specialized expertise" icon="brain">

Subagents can be fine-tuned with detailed instructions for specific domains, leading to higher success rates on designated tasks.

</Card>

<Card title="Reusability" icon="rotate">

Once created, subagents can be used across different projects and shared with your team for consistent workflows.

</Card>

<Card title="Flexible permissions" icon="shield-check">

Each subagent can have different tool access levels, allowing you to limit powerful tools to specific subagent types.

</Card>

</CardGroup>

## ## Quick start

To create your first subagent:

<Steps>

<Step title="Open the subagents interface">

Run the following command:

...

/agents

...

</Step>

<Step title="Select 'Create New Agent'">

Choose whether to create a project-level or user-level subagent

</Step>

<Step title="Define the subagent">

- \* **Recommended**: Generate with Claude first, then customize to make it yours
- \* Describe your subagent in detail and when it should be used
- \* Select the tools you want to grant access to (or leave blank to inherit all tools)
- \* The interface shows all available tools, making selection easy
- \* If you're generating with Claude, you can also edit the system prompt in your own editor by pressing `e`

</Step>

<Step title="Save and use">

Your subagent is now available! Claude will use it automatically when appropriate, or you can invoke it explicitly:

...

> Use the code-reviewer subagent to check my recent changes

...

</Step>

</Steps>

## Subagent configuration

### File locations

Subagents are stored as Markdown files with YAML frontmatter in two possible locations:

| Type | Location | Scope | Priority |

|                              |                                |                               |         |
|------------------------------|--------------------------------|-------------------------------|---------|
| :-----                       | :-----                         | :-----                        | :-----  |
| <b>**Project subagents**</b> | <code>~/.claude/agents/</code> | Available in current project  | Highest |
| <b>**User subagents**</b>    | <code>~/.claude/agents/</code> | Available across all projects | Lower   |

When subagent names conflict, project-level subagents take precedence over user-level subagents.

### ### File format

Each subagent is defined in a Markdown file with this structure:

```

```markdown
---
name: your-sub-agent-name

description: Description of when this subagent should be invoked

tools: tool1, tool2, tool3 # Optional - inherits all tools if omitted
---

```

Your subagent's system prompt goes here. This can be multiple paragraphs and should clearly define the subagent's role, capabilities, and approach to solving problems.

Include specific instructions, best practices, and any constraints the subagent should follow.

```

...

```

### #### Configuration fields

Field	Required	Description
<code>`name`</code>	Yes	Unique identifier using lowercase letters and hyphens
<code>`description`</code>	Yes	Natural language description of the subagent's purpose
<code>`tools`</code>	No	Comma-separated list of specific tools. If omitted, inherits all tools from the main thread

### ### Available tools

Subagents can be granted access to any of Claude Code's internal tools. See the [tools documentation](/en/docs/claude-code/settings#tools-available-to-claude) for a complete list of available tools.

<Tip>

**Recommended:** Use the `/agents`` command to modify tool access - it provides an interactive interface that lists all available tools, including any connected MCP server tools, making it easier to select the ones you need.

</Tip>

You have two options for configuring tools:

- Omit the ``tools`` field** to inherit all tools from the main thread (default), including MCP tools
- Specify individual tools** as a comma-separated list for more granular control (can be edited manually or via `/agents``)

**MCP Tools:** Subagents can access MCP tools from configured MCP servers. When the ``tools`` field is omitted, subagents inherit all MCP tools available to the main thread.

## ## Managing subagents

### ### Using the `/agents` command (Recommended)



The `/agents` command provides a comprehensive interface for subagent management:

```
---
```

```
/agents
```

```
---
```

This opens an interactive menu where you can:

- \* View all available subagents (built-in, user, and project)
- \* Create new subagents with guided setup
- \* Edit existing custom subagents, including their tool access
- \* Delete custom subagents
- \* See which subagents are active when duplicates exist
- \* **Easily manage tool permissions** with a complete list of available tools

**### Direct file management**

You can also manage subagents by working directly with their files:

```
```bash
```

```
# Create a project subagent
```

```
mkdir -p .claude/agents
```

```
echo '---
```

```
name: test-runner
```

```
description: Use proactively to run tests and fix failures
```

```
---
```

You are a test automation expert. When you see code changes, proactively run the appropriate tests. If tests fail, analyze the failures and fix them while preserving the original test intent.' >  
.claude/agents/test-runner.md

# Create a user subagent

mkdir -p ~/.claude/agents

# ... create subagent file

...

## Using subagents effectively

### Automatic delegation

Claude Code proactively delegates tasks based on:

- \* The task description in your request
- \* The `description` field in subagent configurations
- \* Current context and available tools

<Tip>

To encourage more proactive subagent use, include phrases like "use PROACTIVELY" or "MUST BE USED" in your `description` field.

</Tip>

### Explicit invocation

Request a specific subagent by mentioning it in your command:

...

- > Use the test-runner subagent to fix failing tests
- > Have the code-reviewer subagent look at my recent changes
- > Ask the debugger subagent to investigate this error

...

## ## Example subagents

### ### Code reviewer

```markdown

---

name: code-reviewer

description: Expert code review specialist. Proactively reviews code for quality, security, and maintainability. Use immediately after writing or modifying code.

tools: Read, Grep, Glob, Bash

---

You are a senior code reviewer ensuring high standards of code quality and security.

When invoked:

1. Run git diff to see recent changes
2. Focus on modified files
3. Begin review immediately

Review checklist:

- Code is simple and readable
- Functions and variables are well-named

- No duplicated code
- Proper error handling
- No exposed secrets or API keys
- Input validation implemented
- Good test coverage
- Performance considerations addressed

Provide feedback organized by priority:

- Critical issues (must fix)
- Warnings (should fix)
- Suggestions (consider improving)

Include specific examples of how to fix issues.

...

### Debugger

```markdown

---

name: debugger

description: Debugging specialist for errors, test failures, and unexpected behavior. Use proactively when encountering any issues.

tools: Read, Edit, Bash, Grep, Glob

---

You are an expert debugger specializing in root cause analysis.

When invoked:

1. Capture error message and stack trace
2. Identify reproduction steps
3. Isolate the failure location
4. Implement minimal fix
5. Verify solution works

Debugging process:

- Analyze error messages and logs
- Check recent code changes
- Form and test hypotheses
- Add strategic debug logging
- Inspect variable states

For each issue, provide:

- Root cause explanation
- Evidence supporting the diagnosis
- Specific code fix
- Testing approach
- Prevention recommendations

Focus on fixing the underlying issue, not just symptoms.

...

### Data scientist

```markdown

---

**name: data-scientist**

**description: Data analysis expert for SQL queries, BigQuery operations, and data insights. Use proactively for data analysis tasks and queries.**

**tools: Bash, Read, Write**

---

**You are a data scientist specializing in SQL and BigQuery analysis.**

**When invoked:**

- 1. Understand the data analysis requirement**
- 2. Write efficient SQL queries**
- 3. Use BigQuery command line tools (bq) when appropriate**
- 4. Analyze and summarize results**
- 5. Present findings clearly**

**Key practices:**

- Write optimized SQL queries with proper filters**
- Use appropriate aggregations and joins**
- Include comments explaining complex logic**
- Format results for readability**
- Provide data-driven recommendations**

**For each analysis:**

- Explain the query approach**
- Document any assumptions**
- Highlight key findings**
- Suggest next steps based on data**

**Always ensure queries are efficient and cost-effective.**

...

## **## Best practices**

**\* \*\*Start with Claude-generated agents\*\*:** We highly recommend generating your initial subagent with Claude and then iterating on it to make it personally yours. This approach gives you the best results - a solid foundation that you can customize to your specific needs.

**\* \*\*Design focused subagents\*\*:** Create subagents with single, clear responsibilities rather than trying to make one subagent do everything. This improves performance and makes subagents more predictable.

**\* \*\*Write detailed prompts\*\*:** Include specific instructions, examples, and constraints in your system prompts. The more guidance you provide, the better the subagent will perform.

**\* \*\*Limit tool access\*\*:** Only grant tools that are necessary for the subagent's purpose. This improves security and helps the subagent focus on relevant actions.

**\* \*\*Version control\*\*:** Check project subagents into version control so your team can benefit from and improve them collaboratively.

## **## Advanced usage**

### **### Chaining subagents**

**For complex workflows, you can chain multiple subagents:**

...

**> First use the code-analyzer subagent to find performance issues, then use the optimizer subagent to fix them**

...

### ### Dynamic subagent selection

Claude Code intelligently selects subagents based on context. Make your `description` fields specific and action-oriented for best results.

## ## Performance considerations

\* **Context efficiency**: Agents help preserve main context, enabling longer overall sessions

\* **Latency**: Subagents start off with a clean slate each time they are invoked and may add latency as they gather context that they require to do their job effectively.

## ## Related documentation

\* [\[Slash commands\]](/en/docs/claude-code/slash-commands) - Learn about other built-in commands

\* [\[Settings\]](/en/docs/claude-code/settings) - Configure Claude Code behavior

\* [\[Hooks\]](/en/docs/claude-code/hooks) - Automate workflows with event handlers

## # Python

> Build custom AI agents with the Claude Code Python SDK

## ## Prerequisites

\* Python 3.10+

\* `claude-code-sdk` from PyPI

\* Node.js 18+

\* `@anthropic-ai/claude-code` from NPM



<Note>

To view the Python SDK source code, see the  
[`claude-code-sdk`](https://github.com/anthropics/claude-code-sdk-python) repo.

</Note>

<Tip>

For interactive development, use [IPython](https://ipython.org/): `pip install ipython`

</Tip>

## ## Installation

Install `claude-code-sdk` from PyPI and `@anthropic-ai/claude-code` from NPM:

```
```bash
pip install claude-code-sdk
npm install -g @anthropic-ai/claude-code # Required dependency
```
```

(Optional) Install IPython for interactive development:

```
```bash
pip install ipython
```
```

## ## Quick start

Create your first agent:

```

python

# legal-agent.py

import asyncio

from claude_code_sdk import ClaudeSDKClient, ClaudeCodeOptions

async def main():

    async with ClaudeSDKClient(

        options=ClaudeCodeOptions(

            system_prompt="You are a legal assistant. Identify risks and suggest improvements.",

            max_turns=2

        )

    ) as client:

        # Send the query

        await client.query(

            "Review this contract clause for potential issues: 'The party agrees to unlimited liability...'"

        )

        # Stream the response

        async for message in client.receive_response():

            if hasattr(message, 'content'):

                # Print streaming content as it arrives

                for block in message.content:

                    if hasattr(block, 'text'):

                        print(block.text, end="", flush=True)

if __name__ == "__main__":

```

```
    asyncio.run(main())  
'''
```

Save the code above as `legal-agent.py`, then run:

```
'''bash  
  
python legal-agent.py  
'''
```

For [IPython](<https://ipython.org/>)/Jupyter notebooks, you can run the code directly in a cell:

```
'''python  
  
await main()  
'''
```

<Note>

The Python examples on this page use `asyncio`, but you can also use `anyio`.

</Note>

## ## Basic usage

The Python SDK provides two primary interfaces:

### ### 1. The `ClaudeSDKClient` class (recommended)

Best for streaming responses, multi-turn conversations, and interactive applications:

```

python

import asyncio

from claude_code_sdk import ClaudeSDKClient, ClaudeCodeOptions


async def main():

    async with ClaudeSDKClient(

        options=ClaudeCodeOptions(

            system_prompt="You are a performance engineer",

            allowed_tools=["Bash", "Read", "WebSearch"],

            max_turns=5

        )

    ) as client:

        await client.query("Analyze system performance")


    # Stream responses

    async for message in client.receive_response():

        if hasattr(message, 'content'):

            for block in message.content:

                if hasattr(block, 'text'):

                    print(block.text, end="", flush=True)


# Run as script

asyncio.run(main())


# Or in IPython/Jupyter: await main()
...

```

### ### 2. The `query` function

For simple, one-shot queries:

```
```python
from claude_code_sdk import query, ClaudeCodeOptions

async for message in query(
    prompt="Analyze system performance",
    options=ClaudeCodeOptions(system_prompt="You are a performance engineer")
):
    if type(message).__name__ == "ResultMessage":
        print(message.result)
...

```

### ## Configuration options

The Python SDK accepts all arguments supported by the [\[command line\]](/en/docs/claude-code/cli-reference) through the `ClaudeCodeOptions` class.

### ### ClaudeCodeOptions parameters

```
```python
from claude_code_sdk import ClaudeCodeOptions

options = ClaudeCodeOptions(
    # Core configuration
    system_prompt="You are a helpful assistant",

```

**append\_system\_prompt="Additional system instructions",**

**max\_turns=5,**

**model="claude-3-5-sonnet-20241022",**

**max\_thinking\_tokens=8000,**

#### **# Tool management**

**allowed\_tools=["Bash", "Read", "Write"],**

**disallowed\_tools=["WebSearch"],**

#### **# Session management**

**continue\_conversation=False,**

**resume="session-uuid",**

#### **# Environment**

**cwd="/path/to/working/directory",**

**add\_dirs=["/additional/context/dir"],**

**settings="/path/to/settings.json",**

#### **# Permissions**

**permission\_mode="acceptEdits", # "default", "acceptEdits", "plan", "bypassPermissions"**

**permission\_prompt\_tool\_name="mcp\_\_approval\_tool",**

#### **# MCP integration**

**mcp\_servers={**

**"my\_server": {**

**"command": "npx",**

**"args": ["-y", "@modelcontextprotocol/server-example"],**

```

        "env": {"API_KEY": "your-key"}

    }

},

# Advanced

extra_args={"--verbose": None, "--custom-flag": "value"}

)

...

```

#### #### Parameter details

```

* **`system_prompt`: `str | None` - Custom system prompt defining the agent's role

* **`append_system_prompt`: `str | None` - Additional text appended to system prompt

* **`max_turns`: `int | None` - Maximum conversation turns (unlimited if None)

* **`model`: `str | None` - Specific Claude model to use

* **`max_thinking_tokens`: `int` - Maximum tokens for Claude's thinking process (default: 8000)

* **`allowed_tools`: `list[str]` - Tools specifically allowed for use

* **`disallowed_tools`: `list[str]` - Tools that should not be used

* **`continue_conversation`: `bool` - Continue most recent conversation (default: False)

* **`resume`: `str | None` - Session UUID to resume specific conversation

* **`cwd`: `str | Path | None` - Working directory for the session

* **`add_dirs`: `list[str | Path]` - Additional directories to include in context

* **`settings`: `str | None` - Path to settings file or settings JSON string

* **`permission_mode`: `str | None` - Permission handling mode

* **`permission_prompt_tool_name`: `str | None` - Custom permission prompt tool name

* **`mcp_servers`: `dict | str | Path` - MCP server configurations

* **`extra_args`: `dict[str, str | None]` - Pass arbitrary CLI flags to underlying Claude Code CLI

```

#### #### Permission modes

```
* **"default": CLI prompts for dangerous tools (default behavior)
* **"acceptEdits": Automatically accept file edits without prompting
* **"plan": Plan Mode - analyze without making changes
* **"bypassPermissions": Allow all tools without prompting (use with caution)
```

#### ### Advanced configuration example

```
python
import asyncio

from claude_code_sdk import ClaudeSDKClient, ClaudeCodeOptions

async def advanced_agent():

    """Example showcasing advanced configuration options"""

    async with ClaudeSDKClient(

        options=ClaudeCodeOptions(

            # Custom working directory and additional context

            cwd="/project/root",

            add_dirs=["/shared/libs", "/common/utils"],

            # Model and thinking configuration

            model="claude-3-5-sonnet-20241022",

            max_thinking_tokens=12000,
```



```

# Advanced tool control

allowed_tools=["Read", "Write", "Bash", "Grep"],

disallowed_tools=["WebSearch", "Bash(rm*)"],


# Custom settings and CLI args

settings={"editor": "vim", "theme": "dark"},

extra_args={

    "--verbose": None,

    "--timeout": "300"

}

)

) as client:

    await client.query("Analyze the codebase structure")


    async for message in client.receive_response():

        if hasattr(message, 'content'):

            for block in message.content:

                if hasattr(block, 'text'):

                    print(block.text, end="", flush=True)


    asyncio.run(advanced_agent())
...

## Structured messages and image inputs

```

The SDK supports passing structured messages and image inputs:

```

python

from claude_code_sdk import ClaudeSDKClient, ClaudeCodeOptions

async with ClaudeSDKClient() as client:

    # Text message

    await client.query("Analyze this code for security issues")


    # Message with image reference (image will be read by Claude's Read tool)

    await client.query("Explain what's shown in screenshot.png")


    # Multiple messages in sequence

    messages = [

        "First, analyze the architecture diagram in diagram.png",

        "Now suggest improvements based on the diagram",

        "Finally, generate implementation code"

    ]


    for msg in messages:

        await client.query(msg)

        async for response in client.receive_response():

            # Process each response

            pass


    # The SDK handles image files through Claude's built-in Read tool

    # Supported formats: PNG, JPG, PDF, and other common formats
...

```

## **## Multi-turn conversations**

### **### Method 1: Using ClaudeSDKClient for persistent conversations**

```
```python
import asyncio

from claude_code_sdk import ClaudeSDKClient, ClaudeCodeOptions, query
```

#### **# Method 1: Using ClaudeSDKClient for persistent conversations**

```
async def multi_turn_conversation():

    async with ClaudeSDKClient() as client:

        # First query

        await client.query("Let's refactor the payment module")

        async for msg in client.receive_response():

            # Process first response

            pass

        # Continue in same session

        await client.query("Now add comprehensive error handling")

        async for msg in client.receive_response():

            # Process continuation

            pass

    # The conversation context is maintained throughout

# Method 2: Using query function with session management

async def resume_session():
```

**# Continue most recent conversation**

```
async for message in query(  
    prompt="Now refactor this for better performance",  
    options=ClaudeCodeOptions(continue_conversation=True)  
):  
    if type(message).__name__ == "ResultMessage":  
        print(message.result)
```

**# Resume specific session**

```
async for message in query(  
    prompt="Update the tests",  
    options=ClaudeCodeOptions(  
        resume="550e8400-e29b-41d4-a716-446655440000",  
        max_turns=3  
    )  
):  
    if type(message).__name__ == "ResultMessage":  
        print(message.result)
```

**# Run the examples**

```
asyncio.run(multi_turn_conversation())
```

...

**## Custom system prompts**

**System prompts define your agent's role, expertise, and behavior:**

```

python

import asyncio

from claude_code_sdk import ClaudeSDKClient, ClaudeCodeOptions


async def specialized_agents():

    # SRE incident response agent with streaming

    async with ClaudeSDKClient(

        options=ClaudeCodeOptions(

            system_prompt="You are an SRE expert. Diagnose issues systematically and provide actionable solutions.",

            max_turns=3

        )

    ) as sre_agent:

        await sre_agent.query("API is down, investigate")


    # Stream the diagnostic process

    async for message in sre_agent.receive_response():

        if hasattr(message, 'content'):

            for block in message.content:

                if hasattr(block, 'text'):

                    print(block.text, end="", flush=True)


    # Legal review agent with custom prompt

    async with ClaudeSDKClient(

        options=ClaudeCodeOptions(

            append_system_prompt="Always include comprehensive error handling and unit tests.",

            max_turns=2

        )

    )

```

```

) as dev_agent:

    await dev_agent.query("Refactor this function")


# Collect full response

full_response = []

async for message in dev_agent.receive_response():

    if type(message).__name__ == "ResultMessage":

        print(message.result)


asyncio.run(specialized_agents())

```

...

## ## Custom tools via MCP

The Model Context Protocol (MCP) lets you give your agents custom tools and capabilities:

```

```python

import asyncio

from claude_code_sdk import ClaudeSDKClient, ClaudeCodeOptions


async def mcp_enabled_agent():

    # Legal agent with document access and streaming

    # Note: Configure your MCP servers as needed

    mcp_servers = {

        # Example configuration - uncomment and configure as needed:

        # "docusign": {

        #     "command": "npx",

```

```

# "args": ["-y", "@modelcontextprotocol/server-docusign"],
# "env": {"API_KEY": "your-key"}
# }
}

```

```

async with ClaudeSDKClient(
    options=ClaudeCodeOptions(
        mcp_servers=mcp_servers,
        allowed_tools=["mcp__docusign", "mcp__compliance_db"],
        system_prompt="You are a corporate lawyer specializing in contract review.",
        max_turns=4
    )

```

```

) as client:

```

```

    await client.query("Review this contract for compliance risks")

```

```

# Monitor tool usage and responses

```

```

async for message in client.receive_response():

```

```

    if hasattr(message, 'content'):

```

```

        for block in message.content:

```

```

            if hasattr(block, 'type'):

```

```

                if block.type == 'tool_use':

```

```

                    print(f"\n[Using tool: {block.name}]\n")

```

```

                elif hasattr(block, 'text'):

```

```

                    print(block.text, end="", flush=True)

```

```

            elif hasattr(block, 'text'):

```

```

                print(block.text, end="", flush=True)

```

```

        if type(message).__name__ == "ResultMessage":

            print(f"\n\nReview complete. Total cost: ${message.total_cost_usd:.4f}")

    asyncio.run(mcp_enabled_agent())
    ...

```

### ## Custom permission prompt tool

Implement custom permission handling for tool calls:

```

```python
import asyncio

from claude_code_sdk import ClaudeSDKClient, ClaudeCodeOptions

async def use_permission_prompt():

    """Example using custom permission prompt tool"""

    # MCP server configuration

    mcp_servers = {

        # Example configuration - uncomment and configure as needed:

        # "security": {

        #     "command": "npx",

        #     "args": ["-y", "@modelcontextprotocol/server-security"],

        #     "env": {"API_KEY": "your-key"}

        # }

    }

```



```

async with ClaudeSDKClient(
    options=ClaudeCodeOptions(
        permission_prompt_tool_name="mcp__security__approval_prompt", # Changed from
permission_prompt_tool

        mcp_servers=mcp_servers,

        allowed_tools=["Read", "Grep"],

        disallowed_tools=["Bash(rm*)", "Write"],

        system_prompt="You are a security auditor"

    )
) as client:

    await client.query("Analyze and fix the security issues")


# Monitor tool usage and permissions

async for message in client.receive_response():

    if hasattr(message, 'content'):

        for block in message.content:

            if hasattr(block, 'type'): # Added check for 'type' attribute

                if block.type == 'tool_use':

                    print(f"[Tool: {block.name}] ", end="")

                if hasattr(block, 'text'):

                    print(block.text, end="", flush=True)


# Check for permission denials in error messages

if type(message).__name__ == "ErrorMessage":

    if hasattr(message, 'error') and "Permission denied" in str(message.error):

        print(f"\n ⚠️ Permission denied: {message.error}")

```

**# Example MCP server implementation (Python)**

```

# This would be in your MCP server code

async def approval_prompt(tool_name: str, input: dict, tool_use_id: str = None):

    """Custom permission prompt handler"""

    # Your custom logic here

    if "allow" in str(input):

        return json.dumps({

            "behavior": "allow",

            "updatedInput": input

        })

    else:

        return json.dumps({

            "behavior": "deny",

            "message": f"Permission denied for {tool_name}"

        })

    asyncio.run(use_permission_prompt())
...

```

## ## Output formats

### ### Text output with streaming

```

```python

# Default text output with streaming

async with ClaudeSDKClient() as client:

    await client.query("Explain file src/components/Header.tsx")

```

```

# Stream text as it arrives

async for message in client.receive_response():

    if hasattr(message, 'content'):

        for block in message.content:

            if hasattr(block, 'text'):

                print(block.text, end="", flush=True)

            # Output streams in real-time: This is a React component showing...
...

```

### JSON output with metadata

```

```python

# Collect all messages with metadata

async with ClaudeSDKClient() as client:

    await client.query("How does the data layer work?")

    messages = []

    result_data = None

    async for message in client.receive_messages():

        messages.append(message)

        # Capture result message with metadata

        if type(message).__name__ == "ResultMessage":

            result_data = {

                "result": message.result,

                "cost": message.total_cost_usd,

```

```

        "duration": message.duration_ms,

        "num_turns": message.num_turns,

        "session_id": message.session_id
    }

    break

print(result_data)
'''

## Input formats

'''python

import asyncio

from claude_code_sdk import ClaudeSDKClient

async def process_inputs():

    async with ClaudeSDKClient() as client:

        # Text input

        await client.query("Explain this code")

        async for message in client.receive_response():

            # Process streaming response

            pass

        # Image input (Claude will use Read tool automatically)

        await client.query("What's in this diagram? screenshot.png")

        async for message in client.receive_response():

            # Process image analysis

```

```

    pass

# Multiple inputs with mixed content

inputs = [

    "Analyze the architecture in diagram.png",

    "Compare it with best practices",

    "Generate improved version"

]

for prompt in inputs:

    await client.query(prompt)

    async for message in client.receive_response():

        # Process each response

        pass

asyncio.run(process_inputs())

...

## Agent integration examples

### SRE incident response agent

```python

import asyncio

from claude_code_sdk import ClaudeSDKClient, ClaudeCodeOptions

async def investigate_incident(incident_description: str, severity: str = "medium"):

```

"""Automated incident response agent with real-time streaming"""

# MCP server configuration for monitoring tools

mcp\_servers = {

# Example configuration - uncomment and configure as needed:

# "datadog": {

# "command": "npx",

# "args": ["-y", "@modelcontextprotocol/server-datadog"],

# "env": {"API\_KEY": "your-datadog-key", "APP\_KEY": "your-app-key"}

# }

}

async with ClaudeSDKClient(

options=ClaudeCodeOptions(

system\_prompt="You are an SRE expert. Diagnose issues systematically and provide actionable solutions.",

max\_turns=6,

allowed\_tools=["Bash", "Read", "WebSearch", "mcp\_\_datadog"],

mcp\_servers=mcp\_servers

)

) as client:

# Send the incident details

prompt = f"Incident: {incident\_description} (Severity: {severity})"

print(f"🔍 Investigating: {prompt}\n")

await client.query(prompt)

# Stream the investigation process

investigation\_log = []

```
async for message in client.receive_response():
```

```
    if hasattr(message, 'content'):
```

```
        for block in message.content:
```

```
            if hasattr(block, 'type'):
```

```
                if block.type == 'tool_use':
```

```
                    print(f"[{block.name}] ", end="")
```

```
            if hasattr(block, 'text'):
```

```
                text = block.text
```

```
                print(text, end="", flush=True)
```

```
                investigation_log.append(text)
```

```
# Capture final result
```

```
if type(message).__name__ == "ResultMessage":
```

```
    return {
```

```
        'analysis': ".join(investigation_log),
```

```
        'cost': message.total_cost_usd,
```

```
        'duration_ms': message.duration_ms
```

```
    }
```

```
# Usage
```

```
result = await investigate_incident("Payment API returning 500 errors", "high")
```

```
print(f"\n\nInvestigation complete. Cost: ${result['cost']:.4f}")
```

```
...
```

```
### Automated security review
```

```
```python
```

```

import subprocess

import asyncio

import json

from claude_code_sdk import ClaudeSDKClient, ClaudeCodeOptions


async def audit_pr(pr_number: int):

    """Security audit agent for pull requests with streaming feedback"""

    # Get PR diff

    pr_diff = subprocess.check_output(

        ["gh", "pr", "diff", str(pr_number)],

        text=True

    )


    async with ClaudeSDKClient(

        options=ClaudeCodeOptions(

            system_prompt="You are a security engineer. Review this PR for vulnerabilities, insecure patterns, and compliance issues.",

            max_turns=3,

            allowed_tools=["Read", "Grep", "WebSearch"]

        )

    ) as client:

        print(f"🔍 Auditing PR #{pr_number}\n")

        await client.query(pr_diff)


        findings = []

        async for message in client.receive_response():

            if hasattr(message, 'content'):

                for block in message.content:

```



```

if hasattr(block, 'text'):

    # Stream findings as they're discovered

    print(block.text, end="", flush=True)

    findings.append(block.text)

```

```

if type(message).__name__ == "ResultMessage":

    return {

        'pr_number': pr_number,

        'findings': ".join(findings),

        'metadata': {

            'cost': message.total_cost_usd,

            'duration': message.duration_ms,

            'severity': 'high' if 'vulnerability' in ".join(findings).lower() else 'medium'

        }

    }

```

## # Usage

```

report = await audit_pr(123)

print(f"\n\nAudit complete. Severity: {report['metadata']['severity']}")

print(json.dumps(report, indent=2))

...

```

## ### Multi-turn legal assistant

```

```python

import asyncio

from claude_code_sdk import ClaudeSDKClient, ClaudeCodeOptions

```

```

async def legal_review():

    """Legal document review with persistent session and streaming"""

    async with ClaudeSDKClient(

        options=ClaudeCodeOptions(

            system_prompt="You are a corporate lawyer. Provide detailed legal analysis.",

            max_turns=2

        )

    ) as client:

        # Multi-step review in same session

        steps = [

            "Review contract.pdf for liability clauses",

            "Check compliance with GDPR requirements",

            "Generate executive summary of risks"

        ]

        review_results = []

        for step in steps:

            print(f"\n📋 {step}\n")

            await client.query(step)

            step_result = []

            async for message in client.receive_response():

                if hasattr(message, 'content'):

                    for block in message.content:

```

```
if hasattr(block, 'text'):

    text = block.text

    print(text, end="", flush=True)

    step_result.append(text)
```

```
if type(message).__name__ == "ResultMessage":

    review_results.append({

        'step': step,

        'analysis': ".join(step_result),

        'cost': message.total_cost_usd

    })
```

**# Summary**

```
total_cost = sum(r['cost'] for r in review_results)

print(f"\n\n✅ Legal review complete. Total cost: ${total_cost:.4f}")

return review_results
```

**# Usage**

```
results = await legal_review()

...
```

**## Python-specific best practices**

**### Key patterns**

```
```python
```

```
import asyncio
```

```
from claude_code_sdk import ClaudeSDKClient, ClaudeCodeOptions
```

```
# Always use context managers
```

```
async with ClaudeSDKClient() as client:
```

```
    await client.query("Analyze this code")
```

```
    async for msg in client.receive_response():
```

```
        # Process streaming messages
```

```
    pass
```

```
# Run multiple agents concurrently
```

```
async with ClaudeSDKClient() as reviewer, ClaudeSDKClient() as tester:
```

```
    await asyncio.gather(
```

```
        reviewer.query("Review main.py"),
```

```
        tester.query("Write tests for main.py")
```

```
    )
```

```
# Error handling
```

```
from claude_code_sdk import CLINotFoundError, ProcessError
```

```
try:
```

```
    async with ClaudeSDKClient() as client:
```

```
        # Your code here
```

```
    pass
```

```
except CLINotFoundError:
```

```
    print("Install CLI: npm install -g @anthropic-ai/claude-code")
```

```
except ProcessError as e:
```

```
    print(f"Process error: {e}")
```

**# Collect full response with metadata**

```
async def get_response(client, prompt):

    await client.query(prompt)

    text = []

    async for msg in client.receive_response():

        if hasattr(msg, 'content'):

            for block in msg.content:

                if hasattr(block, 'text'):

                    text.append(block.text)

            if type(msg).__name__ == "ResultMessage":

                return {'text': ''.join(text), 'cost': msg.total_cost_usd}

    ...
```

**### IPython/Jupyter tips**

**```python**

**# In Jupyter, use await directly in cells**

```
client = ClaudeSDKClient()
```

```
await client.connect()
```

```
await client.query("Analyze data.csv")
```

```
async for msg in client.receive_response():
```

```
    print(msg)
```

```
await client.disconnect()
```

**# Create reusable helper functions**

```
async def stream_print(client, prompt):
```

```
await client.query(prompt)

async for msg in client.receive_response():

    if hasattr(msg, 'content'):

        for block in msg.content:

            if hasattr(block, 'text'):

                print(block.text, end="", flush=True)

...

```

### ## Related resources

- \* [\[CLI usage and controls\]\(/en/docs/claude-code/cli-reference\)](#) - Complete CLI documentation
- \* [\[GitHub Actions integration\]\(/en/docs/claude-code/github-actions\)](#) - Automate your GitHub workflow with Claude
- \* [\[Common workflows\]\(/en/docs/claude-code/common-workflows\)](#) - Step-by-step guides for common use cases

### # Python

> Build custom AI agents with the Claude Code Python SDK

### ## Prerequisites

- \* Python 3.10+
- \* `claude-code-sdk` from PyPI`
- \* Node.js 18+
- \* `@anthropic-ai/claude-code` from NPM`

<Note>

To view the Python SDK source code, see the  
[`claude-code-sdk`](https://github.com/anthropics/claude-code-sdk-python) repo.

</Note>

<Tip>

For interactive development, use [IPython](https://ipython.org/): `pip install ipython`

</Tip>

## ## Installation

Install `claude-code-sdk` from PyPI and `@anthropic-ai/claude-code` from NPM:

```
```bash
pip install claude-code-sdk
npm install -g @anthropic-ai/claude-code # Required dependency
```
```

(Optional) Install IPython for interactive development:

```
```bash
pip install ipython
```
```

## ## Quick start

Create your first agent:

```
```python
```

```

# legal-agent.py

import asyncio

from claude_code_sdk import ClaudeSDKClient, ClaudeCodeOptions


async def main():

    async with ClaudeSDKClient(

        options=ClaudeCodeOptions(

            system_prompt="You are a legal assistant. Identify risks and suggest improvements.",

            max_turns=2

        )

    ) as client:

        # Send the query

        await client.query(

            "Review this contract clause for potential issues: 'The party agrees to unlimited liability...'"

        )


        # Stream the response

        async for message in client.receive_response():

            if hasattr(message, 'content'):

                # Print streaming content as it arrives

                for block in message.content:

                    if hasattr(block, 'text'):

                        print(block.text, end="", flush=True)


if __name__ == "__main__":

    asyncio.run(main())

...

```



Save the code above as `legal-agent.py`, then run:

```
```bash

python legal-agent.py

```
```

For [IPython](https://ipython.org/) Jupyter notebooks, you can run the code directly in a cell:

```
```python

await main()

```
```

<Note>

The Python examples on this page use `asyncio`, but you can also use `anyio`.

</Note>

## ## Basic usage

The Python SDK provides two primary interfaces:

### ### 1. The `ClaudeSDKClient` class (recommended)

Best for streaming responses, multi-turn conversations, and interactive applications:

```
```python

import asyncio
```

```
from claude_code_sdk import ClaudeSDKClient, ClaudeCodeOptions
```

```
async def main():
```

```
    async with ClaudeSDKClient(
```

```
        options=ClaudeCodeOptions(
```

```
            system_prompt="You are a performance engineer",
```

```
            allowed_tools=["Bash", "Read", "WebSearch"],
```

```
            max_turns=5
```

```
        )
```

```
    ) as client:
```

```
        await client.query("Analyze system performance")
```

```
    # Stream responses
```

```
    async for message in client.receive_response():
```

```
        if hasattr(message, 'content'):
```

```
            for block in message.content:
```

```
                if hasattr(block, 'text'):
```

```
                    print(block.text, end="", flush=True)
```

```
    # Run as script
```

```
    asyncio.run(main())
```

```
    # Or in IPython/Jupyter: await main()
```

```
    ...
```

```
### 2. The `query` function
```

For simple, one-shot queries:

```
```python
from claude_code_sdk import query, ClaudeCodeOptions

async for message in query(
    prompt="Analyze system performance",
    options=ClaudeCodeOptions(system_prompt="You are a performance engineer")
):
    if type(message).__name__ == "ResultMessage":
        print(message.result)
...

## Configuration options
```

The Python SDK accepts all arguments supported by the [\[command line\]](/en/docs/claude-code/cli-reference) through the `ClaudeCodeOptions` class.

### ClaudeCodeOptions parameters

```
```python
from claude_code_sdk import ClaudeCodeOptions

options = ClaudeCodeOptions(
    # Core configuration
    system_prompt="You are a helpful assistant",
    append_system_prompt="Additional system instructions",
    max_turns=5,
```

**model="claude-3-5-sonnet-20241022",**

**max\_thinking\_tokens=8000,**

#### **# Tool management**

**allowed\_tools=["Bash", "Read", "Write"],**

**disallowed\_tools=["WebSearch"],**

#### **# Session management**

**continue\_conversation=False,**

**resume="session-uuid",**

#### **# Environment**

**cwd="/path/to/working/directory",**

**add\_dirs=["/additional/context/dir"],**

**settings="/path/to/settings.json",**

#### **# Permissions**

**permission\_mode="acceptEdits", # "default", "acceptEdits", "plan", "bypassPermissions"**

**permission\_prompt\_tool\_name="mcp\_\_approval\_tool",**

#### **# MCP integration**

**mcp\_servers={**

**"my\_server": {**

**"command": "npx",**

**"args": ["-y", "@modelcontextprotocol/server-example"],**

**"env": {"API\_KEY": "your-key"}**

**}**

```
},
```

```
# Advanced
```

```
extra_args={"--verbose": None, "--custom-flag": "value"}
```

```
)
```

```
...
```

#### #### Parameter details

```
* **`system_prompt`: `str | None` - Custom system prompt defining the agent's role
```

```
* **`append_system_prompt`: `str | None` - Additional text appended to system prompt
```

```
* **`max_turns`: `int | None` - Maximum conversation turns (unlimited if None)
```

```
* **`model`: `str | None` - Specific Claude model to use
```

```
* **`max_thinking_tokens`: `int` - Maximum tokens for Claude's thinking process (default: 8000)
```

```
* **`allowed_tools`: `list[str]` - Tools specifically allowed for use
```

```
* **`disallowed_tools`: `list[str]` - Tools that should not be used
```

```
* **`continue_conversation`: `bool` - Continue most recent conversation (default: False)
```

```
* **`resume`: `str | None` - Session UUID to resume specific conversation
```

```
* **`cwd`: `str | Path | None` - Working directory for the session
```

```
* **`add_dirs`: `list[str | Path]` - Additional directories to include in context
```

```
* **`settings`: `str | None` - Path to settings file or settings JSON string
```

```
* **`permission_mode`: `str | None` - Permission handling mode
```

```
* **`permission_prompt_tool_name`: `str | None` - Custom permission prompt tool name
```

```
* **`mcp_servers`: `dict | str | Path` - MCP server configurations
```

```
* **`extra_args`: `dict[str, str | None]` - Pass arbitrary CLI flags to underlying Claude Code CLI
```

#### #### Permission modes

**\* \*\*`"default"`\*\*:** CLI prompts for dangerous tools (default behavior)

**\* \*\*`"acceptEdits"`\*\*:** Automatically accept file edits without prompting

**\* \*\*`"plan"`\*\*:** Plan Mode - analyze without making changes

**\* \*\*`"bypassPermissions"`\*\*:** Allow all tools without prompting (use with caution)

### **### Advanced configuration example**

```
```python  
  
import asyncio  
  
from claude_code_sdk import ClaudeSDKClient, ClaudeCodeOptions  
  
async def advanced_agent():  
  
    """Example showcasing advanced configuration options"""  
  
    async with ClaudeSDKClient(  
        options=ClaudeCodeOptions(  
            # Custom working directory and additional context  
  
            cwd="/project/root",  
  
            add_dirs=["/shared/libs", "/common/utils"],  
  
            # Model and thinking configuration  
  
            model="claude-3-5-sonnet-20241022",  
  
            max_thinking_tokens=12000,  
  
            # Advanced tool control  
  
            allowed_tools=["Read", "Write", "Bash", "Grep"],
```

```

disallowed_tools=["WebSearch", "Bash(rm*)"],

# Custom settings and CLI args

settings={"editor": "vim", "theme": "dark"},

extra_args={

    "--verbose": None,

    "--timeout": "300"

}

)

) as client:

    await client.query("Analyze the codebase structure")


    async for message in client.receive_response():

        if hasattr(message, 'content'):

            for block in message.content:

                if hasattr(block, 'text'):

                    print(block.text, end="", flush=True)


    asyncio.run(advanced_agent())
...

## Structured messages and image inputs

The SDK supports passing structured messages and image inputs:

```python
from claude_code_sdk import ClaudeSDKClient, ClaudeCodeOptions

```

**async with ClaudeSDKClient() as client:**

**# Text message**

**await client.query("Analyze this code for security issues")**

**# Message with image reference (image will be read by Claude's Read tool)**

**await client.query("Explain what's shown in screenshot.png")**

**# Multiple messages in sequence**

**messages = [**

**"First, analyze the architecture diagram in diagram.png",**

**"Now suggest improvements based on the diagram",**

**"Finally, generate implementation code"**

**]**

**for msg in messages:**

**await client.query(msg)**

**async for response in client.receive\_response():**

**# Process each response**

**pass**

**# The SDK handles image files through Claude's built-in Read tool**

**# Supported formats: PNG, JPG, PDF, and other common formats**

**...**

**## Multi-turn conversations**



**### Method 1: Using ClaudeSDKClient for persistent conversations**

```
```python

import asyncio

from claude_code_sdk import ClaudeSDKClient, ClaudeCodeOptions, query
```

**# Method 1: Using ClaudeSDKClient for persistent conversations**

```
async def multi_turn_conversation():

    async with ClaudeSDKClient() as client:

        # First query

        await client.query("Let's refactor the payment module")

        async for msg in client.receive_response():

            # Process first response

            pass

        # Continue in same session

        await client.query("Now add comprehensive error handling")

        async for msg in client.receive_response():

            # Process continuation

            pass

    # The conversation context is maintained throughout
```

**# Method 2: Using query function with session management**

```
async def resume_session():

    # Continue most recent conversation

    async for message in query(
```

```

    prompt="Now refactor this for better performance",

    options=ClaudeCodeOptions(continue_conversation=True)

):

    if type(message).__name__ == "ResultMessage":

        print(message.result)

```

**# Resume specific session**

```

async for message in query(

    prompt="Update the tests",

    options=ClaudeCodeOptions(

        resume="550e8400-e29b-41d4-a716-446655440000",

        max_turns=3

    )

):

    if type(message).__name__ == "ResultMessage":

        print(message.result)

```

**# Run the examples**

```

asyncio.run(multi_turn_conversation())

```

...

**## Custom system prompts**

**System prompts define your agent's role, expertise, and behavior:**

```

```python

```

```

import asyncio

```

```

from claude_code_sdk import ClaudeSDKClient, ClaudeCodeOptions

async def specialized_agents():

    # SRE incident response agent with streaming

    async with ClaudeSDKClient(

        options=ClaudeCodeOptions(

            system_prompt="You are an SRE expert. Diagnose issues systematically and provide actionable
solutions.",

            max_turns=3

        )

    ) as sre_agent:

        await sre_agent.query("API is down, investigate")


    # Stream the diagnostic process

    async for message in sre_agent.receive_response():

        if hasattr(message, 'content'):

            for block in message.content:

                if hasattr(block, 'text'):

                    print(block.text, end="", flush=True)


    # Legal review agent with custom prompt

    async with ClaudeSDKClient(

        options=ClaudeCodeOptions(

            append_system_prompt="Always include comprehensive error handling and unit tests.",

            max_turns=2

        )

    ) as dev_agent:

        await dev_agent.query("Refactor this function")

```

```

# Collect full response

full_response = []

async for message in dev_agent.receive_response():

    if type(message).__name__ == "ResultMessage":

        print(message.result)

asyncio.run(specialized_agents())
'''

```

## ## Custom tools via MCP

The Model Context Protocol (MCP) lets you give your agents custom tools and capabilities:

```

'''python

import asyncio

from claude_code_sdk import ClaudeSDKClient, ClaudeCodeOptions

async def mcp_enabled_agent():

    # Legal agent with document access and streaming

    # Note: Configure your MCP servers as needed

    mcp_servers = {

        # Example configuration - uncomment and configure as needed:

        # "docusign": {

        #     "command": "npx",

        #     "args": ["-y", "@modelcontextprotocol/server-docusign"],

        #     "env": {"API_KEY": "your-key"}

```

```

    # }

}

async with ClaudeSDKClient(

    options=ClaudeCodeOptions(

        mcp_servers=mcp_servers,

        allowed_tools=["mcp__docusign", "mcp__compliance_db"],

        system_prompt="You are a corporate lawyer specializing in contract review.",

        max_turns=4

    )

) as client:

    await client.query("Review this contract for compliance risks")


# Monitor tool usage and responses

async for message in client.receive_response():

    if hasattr(message, 'content'):

        for block in message.content:

            if hasattr(block, 'type'):

                if block.type == 'tool_use':

                    print(f"\n[Using tool: {block.name}]\n")

                elif hasattr(block, 'text'):

                    print(block.text, end="", flush=True)

            elif hasattr(block, 'text'):

                print(block.text, end="", flush=True)

    if type(message).__name__ == "ResultMessage":

        print(f"\n\nReview complete. Total cost: ${message.total_cost_usd:.4f}")

```

```
asyncio.run(mcp_enabled_agent())
```

```
...
```

```
## Custom permission prompt tool
```

Implement custom permission handling for tool calls:

```
```python
```

```
import asyncio
```

```
from claude_code_sdk import ClaudeSDKClient, ClaudeCodeOptions
```

```
async def use_permission_prompt():
```

```
    """Example using custom permission prompt tool"""
```

```
    # MCP server configuration
```

```
    mcp_servers = {
```

```
        # Example configuration - uncomment and configure as needed:
```

```
        # "security": {
```

```
            # "command": "npx",
```

```
            # "args": ["-y", "@modelcontextprotocol/server-security"],
```

```
            # "env": {"API_KEY": "your-key"}
```

```
        # }
```

```
    }
```

```
    async with ClaudeSDKClient(
```

```
        options=ClaudeCodeOptions(
```

```
    permission_prompt_tool_name="mcp__security__approval_prompt", # Changed from
permission_prompt_tool
```

```
    mcp_servers=mcp_servers,
```

```
    allowed_tools=["Read", "Grep"],
```

```
    disallowed_tools=["Bash(rm*)", "Write"],
```

```
    system_prompt="You are a security auditor"
```

```
)
```

```
) as client:
```

```
    await client.query("Analyze and fix the security issues")
```

```
# Monitor tool usage and permissions
```

```
async for message in client.receive_response():
```

```
    if hasattr(message, 'content'):
```

```
        for block in message.content:
```

```
            if hasattr(block, 'type'): # Added check for 'type' attribute
```

```
                if block.type == 'tool_use':
```

```
                    print(f"[Tool: {block.name}] ", end="")
```

```
            if hasattr(block, 'text'):
```

```
                print(block.text, end="", flush=True)
```

```
# Check for permission denials in error messages
```

```
if type(message).__name__ == "ErrorMessage":
```

```
    if hasattr(message, 'error') and "Permission denied" in str(message.error):
```

```
        print(f"\n ⚠ Permission denied: {message.error}")
```

```
# Example MCP server implementation (Python)
```

```
# This would be in your MCP server code
```

```
async def approval_prompt(tool_name: str, input: dict, tool_use_id: str = None):
```

```

"""Custom permission prompt handler"""

# Your custom logic here

if "allow" in str(input):

    return json.dumps({

        "behavior": "allow",

        "updatedInput": input

    })

else:

    return json.dumps({

        "behavior": "deny",

        "message": f"Permission denied for {tool_name}"

    })

asyncio.run(use_permission_prompt())

...

## Output formats

### Text output with streaming

```python

# Default text output with streaming

async with ClaudeSDKClient() as client:

    await client.query("Explain file src/components/Header.tsx")

# Stream text as it arrives

async for message in client.receive_response():

```



```

if hasattr(message, 'content'):

    for block in message.content:

        if hasattr(block, 'text'):

            print(block.text, end="", flush=True)

            # Output streams in real-time: This is a React component showing...

```

...

### JSON output with metadata

```
```python
```

```
# Collect all messages with metadata
```

```
async with ClaudeSDKClient() as client:
```

```
    await client.query("How does the data layer work?")
```

```
messages = []
```

```
result_data = None
```

```
async for message in client.receive_messages():
```

```
    messages.append(message)
```

```
# Capture result message with metadata
```

```
if type(message).__name__ == "ResultMessage":
```

```
    result_data = {
```

```
        "result": message.result,
```

```
        "cost": message.total_cost_usd,
```

```
        "duration": message.duration_ms,
```

```
        "num_turns": message.num_turns,
```

```

        "session_id": message.session_id
    }

    break

print(result_data)
'''

## Input formats

'''python

import asyncio

from claude_code_sdk import ClaudeSDKClient

async def process_inputs():

    async with ClaudeSDKClient() as client:

        # Text input

        await client.query("Explain this code")

        async for message in client.receive_response():

            # Process streaming response

            pass

        # Image input (Claude will use Read tool automatically)

        await client.query("What's in this diagram? screenshot.png")

        async for message in client.receive_response():

            # Process image analysis

            pass

```

```
# Multiple inputs with mixed content
```

```
inputs = [
```

```
    "Analyze the architecture in diagram.png",
```

```
    "Compare it with best practices",
```

```
    "Generate improved version"
```

```
]
```

```
for prompt in inputs:
```

```
    await client.query(prompt)
```

```
    async for message in client.receive_response():
```

```
        # Process each response
```

```
        pass
```

```
asyncio.run(process_inputs())
```

```
...
```

```
## Agent integration examples
```

```
### SRE incident response agent
```

```
```python
```

```
import asyncio
```

```
from claude_code_sdk import ClaudeSDKClient, ClaudeCodeOptions
```

```
async def investigate_incident(incident_description: str, severity: str = "medium"):
```

```
    """Automated incident response agent with real-time streaming"""
```

**# MCP server configuration for monitoring tools**

```
mcp_servers = {  
  
    # Example configuration - uncomment and configure as needed:  
  
    # "datadog": {  
  
        # "command": "npx",  
  
        # "args": ["-y", "@modelcontextprotocol/server-datadog"],  
  
        # "env": {"API_KEY": "your-datadog-key", "APP_KEY": "your-app-key"}  
  
        # }  
  
}
```

```
async with ClaudeSDKClient(  
  
    options=ClaudeCodeOptions(  
  
        system_prompt="You are an SRE expert. Diagnose issues systematically and provide actionable  
solutions.",
```

```
        max_turns=6,  
  
        allowed_tools=["Bash", "Read", "WebSearch", "mcp__datadog"],  
  
        mcp_servers=mcp_servers  
  
    )
```

) as client:

**# Send the incident details**

```
prompt = f"Incident: {incident_description} (Severity: {severity})"
```

```
print(f"🔍 Investigating: {prompt}\n")
```

```
await client.query(prompt)
```

**# Stream the investigation process**

```
investigation_log = []
```

```
async for message in client.receive_response():
```

```
    if hasattr(message, 'content'):
```

```

for block in message.content:

    if hasattr(block, 'type'):

        if block.type == 'tool_use':

            print(f"[{block.name}] ", end="")

    if hasattr(block, 'text'):

        text = block.text

        print(text, end="", flush=True)

        investigation_log.append(text)

```

**# Capture final result**

```

if type(message).__name__ == "ResultMessage":

    return {

        'analysis': ".join(investigation_log),

        'cost': message.total_cost_usd,

        'duration_ms': message.duration_ms

    }

```

**# Usage**

```

result = await investigate_incident("Payment API returning 500 errors", "high")

print(f"\n\nInvestigation complete. Cost: ${result['cost']:.4f}")

```

...

**### Automated security review**

```

```python

```

```

import subprocess

```

```

import asyncio

```

```

import json

from claude_code_sdk import ClaudeSDKClient, ClaudeCodeOptions

async def audit_pr(pr_number: int):

    """Security audit agent for pull requests with streaming feedback"""

    # Get PR diff

    pr_diff = subprocess.check_output(

        ["gh", "pr", "diff", str(pr_number)],

        text=True

    )


    async with ClaudeSDKClient(

        options=ClaudeCodeOptions(

            system_prompt="You are a security engineer. Review this PR for vulnerabilities, insecure patterns, and compliance issues.",

            max_turns=3,

            allowed_tools=["Read", "Grep", "WebSearch"]

        )

    ) as client:

        print(f"🔍 Auditing PR #{pr_number}\n")

        await client.query(pr_diff)


        findings = []

        async for message in client.receive_response():

            if hasattr(message, 'content'):

                for block in message.content:

                    if hasattr(block, 'text'):

                        # Stream findings as they're discovered

```

```

        print(block.text, end="", flush=True)

        findings.append(block.text)

    if type(message).__name__ == "ResultMessage":

        return {

            'pr_number': pr_number,

            'findings': ".join(findings),

            'metadata': {

                'cost': message.total_cost_usd,

                'duration': message.duration_ms,

                'severity': 'high' if 'vulnerability' in ".join(findings).lower() else 'medium'

            }

        }

```

## # Usage

```

report = await audit_pr(123)

print(f"\n\nAudit complete. Severity: {report['metadata']['severity']}")

print(json.dumps(report, indent=2))

...

```

## ### Multi-turn legal assistant

```

```python

import asyncio

from claude_code_sdk import ClaudeSDKClient, ClaudeCodeOptions

async def legal_review():

```

"""Legal document review with persistent session and streaming"""

```
async with ClaudeSDKClient(
    options=ClaudeCodeOptions(
        system_prompt="You are a corporate lawyer. Provide detailed legal analysis.",
        max_turns=2
    )
) as client:
    # Multi-step review in same session

    steps = [
        "Review contract.pdf for liability clauses",
        "Check compliance with GDPR requirements",
        "Generate executive summary of risks"
    ]

    review_results = []

    for step in steps:
        print(f"\n📋 {step}\n")
        await client.query(step)

        step_result = []

        async for message in client.receive_response():
            if hasattr(message, 'content'):
                for block in message.content:
                    if hasattr(block, 'text'):
                        text = block.text
```



```
print(text, end="", flush=True)
```

```
step_result.append(text)
```

```
if type(message).__name__ == "ResultMessage":
```

```
    review_results.append({
```

```
        'step': step,
```

```
        'analysis': ".join(step_result),
```

```
        'cost': message.total_cost_usd
```

```
    })
```

```
# Summary
```

```
total_cost = sum(r['cost'] for r in review_results)
```

```
print(f"\n\n✅ Legal review complete. Total cost: ${total_cost:.4f}")
```

```
return review_results
```

```
# Usage
```

```
results = await legal_review()
```

```
...
```

```
## Python-specific best practices
```

```
### Key patterns
```

```
```python
```

```
import asyncio
```

```
from claude_code_sdk import ClaudeSDKClient, ClaudeCodeOptions
```

**# Always use context managers**

**async with ClaudeSDKClient() as client:**

**await client.query("Analyze this code")**

**async for msg in client.receive\_response():**

**# Process streaming messages**

**pass**

**# Run multiple agents concurrently**

**async with ClaudeSDKClient() as reviewer, ClaudeSDKClient() as tester:**

**await asyncio.gather(**

**reviewer.query("Review main.py"),**

**tester.query("Write tests for main.py")**

**)**

**# Error handling**

**from claude\_code\_sdk import CLINotFoundError, ProcessError**

**try:**

**async with ClaudeSDKClient() as client:**

**# Your code here**

**pass**

**except CLINotFoundError:**

**print("Install CLI: npm install -g @anthropic-ai/claude-code")**

**except ProcessError as e:**

**print(f"Process error: {e}")**

**# Collect full response with metadata**

```

async def get_response(client, prompt):

    await client.query(prompt)

    text = []

    async for msg in client.receive_response():

        if hasattr(msg, 'content'):

            for block in msg.content:

                if hasattr(block, 'text'):

                    text.append(block.text)

            if type(msg).__name__ == "ResultMessage":

                return {'text': ''.join(text), 'cost': msg.total_cost_usd}

```

'''

### IPython/Jupyter tips

'''python

# In Jupyter, use await directly in cells

```
client = ClaudeSDKClient()
```

```
await client.connect()
```

```
await client.query("Analyze data.csv")
```

```
async for msg in client.receive_response():
```

```
    print(msg)
```

```
await client.disconnect()
```

# Create reusable helper functions

```
async def stream_print(client, prompt):
```

```
    await client.query(prompt)
```

```
    async for msg in client.receive_response():
```

```
    if hasattr(msg, 'content'):

        for block in msg.content:

            if hasattr(block, 'text'):

                print(block.text, end="", flush=True)

...

```

## ## Related resources

- \* [\[CLI usage and controls\]\(/en/docs/claude-code/cli-reference\)](#) - Complete CLI documentation
- \* [\[GitHub Actions integration\]\(/en/docs/claude-code/github-actions\)](#) - Automate your GitHub workflow with Claude
- \* [\[Common workflows\]\(/en/docs/claude-code/common-workflows\)](#) - Step-by-step guides for common use cases

## # Python

> Build custom AI agents with the Claude Code Python SDK

## ## Prerequisites

\* Python 3.10+

\* `claude-code-sdk`` from PyPI

\* Node.js 18+

\* `@anthropic-ai/claude-code`` from NPM

<Note>

To view the Python SDK source code, see the `[`claude-code-sdk`](https://github.com/anthropics/claude-code-sdk-python)` repo.

</Note>

<Tip>

For interactive development, use [IPython](https://ipython.org/): `pip install ipython`

</Tip>

## ## Installation

Install `claude-code-sdk` from PyPI and `@anthropic-ai/claude-code` from NPM:

```
```bash
pip install claude-code-sdk
npm install -g @anthropic-ai/claude-code # Required dependency
```
```

(Optional) Install IPython for interactive development:

```
```bash
pip install ipython
```
```

## ## Quick start

Create your first agent:

```
```python
# legal-agent.py
import asyncio
```

```

from claude_code_sdk import ClaudeSDKClient, ClaudeCodeOptions

async def main():

    async with ClaudeSDKClient(

        options=ClaudeCodeOptions(

            system_prompt="You are a legal assistant. Identify risks and suggest improvements.",

            max_turns=2

        )

    ) as client:

        # Send the query

        await client.query(

            "Review this contract clause for potential issues: 'The party agrees to unlimited liability...'"

        )

        # Stream the response

        async for message in client.receive_response():

            if hasattr(message, 'content'):

                # Print streaming content as it arrives

                for block in message.content:

                    if hasattr(block, 'text'):

                        print(block.text, end="", flush=True)

if __name__ == "__main__":

    asyncio.run(main())

...

```

Save the code above as `legal-agent.py`, then run:

```
```bash

python legal-agent.py

```
```

For [IPython](https://ipython.org/) Jupyter notebooks, you can run the code directly in a cell:

```
```python

await main()

```
```

<Note>

The Python examples on this page use `asyncio`, but you can also use `anyio`.

</Note>

## ## Basic usage

The Python SDK provides two primary interfaces:

### ### 1. The `ClaudeSDKClient` class (recommended)

Best for streaming responses, multi-turn conversations, and interactive applications:

```
```python

import asyncio

from claude_code_sdk import ClaudeSDKClient, ClaudeCodeOptions
```

```

async def main():

    async with ClaudeSDKClient(

        options=ClaudeCodeOptions(

            system_prompt="You are a performance engineer",

            allowed_tools=["Bash", "Read", "WebSearch"],

            max_turns=5

        )

    ) as client:

        await client.query("Analyze system performance")


# Stream responses

async for message in client.receive_response():

    if hasattr(message, 'content'):

        for block in message.content:

            if hasattr(block, 'text'):

                print(block.text, end="", flush=True)


# Run as script

asyncio.run(main())


# Or in IPython/Jupyter: await main()
...

```

**### 2. The `query` function**

**For simple, one-shot queries:**



```

python

from claude_code_sdk import query, ClaudeCodeOptions

async for message in query(
    prompt="Analyze system performance",
    options=ClaudeCodeOptions(system_prompt="You are a performance engineer")
):
    if type(message).__name__ == "ResultMessage":
        print(message.result)

```

### ## Configuration options

The Python SDK accepts all arguments supported by the [\[command line\]](/en/docs/claude-code/cli-reference) through the `ClaudeCodeOptions` class.

### ### ClaudeCodeOptions parameters

```

python

from claude_code_sdk import ClaudeCodeOptions

options = ClaudeCodeOptions(
    # Core configuration
    system_prompt="You are a helpful assistant",
    append_system_prompt="Additional system instructions",
    max_turns=5,
    model="claude-3-5-sonnet-20241022",
    max_thinking_tokens=8000,

```

#### # Tool management

allowed\_tools=["Bash", "Read", "Write"],

disallowed\_tools=["WebSearch"],

#### # Session management

continue\_conversation=False,

resume="session-uuid",

#### # Environment

cwd="/path/to/working/directory",

add\_dirs=["/additional/context/dir"],

settings="/path/to/settings.json",

#### # Permissions

permission\_mode="acceptEdits", # "default", "acceptEdits", "plan", "bypassPermissions"

permission\_prompt\_tool\_name="mcp\_\_approval\_tool",

#### # MCP integration

mcp\_servers={

    "my\_server": {

        "command": "npx",

        "args": ["-y", "@modelcontextprotocol/server-example"],

        "env": {"API\_KEY": "your-key"}

    }

},

**# Advanced**

```
extra_args={"--verbose": None, "--custom-flag": "value"}
```

```
)
```

```
...
```

#### #### Parameter details

**\* \*\*`system\_prompt`\*\*:** ``str | None`` - Custom system prompt defining the agent's role

**\* \*\*`append\_system\_prompt`\*\*:** ``str | None`` - Additional text appended to system prompt

**\* \*\*`max\_turns`\*\*:** ``int | None`` - Maximum conversation turns (unlimited if None)

**\* \*\*`model`\*\*:** ``str | None`` - Specific Claude model to use

**\* \*\*`max\_thinking\_tokens`\*\*:** ``int`` - Maximum tokens for Claude's thinking process (default: 8000)

**\* \*\*`allowed\_tools`\*\*:** ``list[str]`` - Tools specifically allowed for use

**\* \*\*`disallowed\_tools`\*\*:** ``list[str]`` - Tools that should not be used

**\* \*\*`continue\_conversation`\*\*:** ``bool`` - Continue most recent conversation (default: False)

**\* \*\*`resume`\*\*:** ``str | None`` - Session UUID to resume specific conversation

**\* \*\*`cwd`\*\*:** ``str | Path | None`` - Working directory for the session

**\* \*\*`add\_dirs`\*\*:** ``list[str | Path]`` - Additional directories to include in context

**\* \*\*`settings`\*\*:** ``str | None`` - Path to settings file or settings JSON string

**\* \*\*`permission\_mode`\*\*:** ``str | None`` - Permission handling mode

**\* \*\*`permission\_prompt\_tool\_name`\*\*:** ``str | None`` - Custom permission prompt tool name

**\* \*\*`mcp\_servers`\*\*:** ``dict | str | Path`` - MCP server configurations

**\* \*\*`extra\_args`\*\*:** ``dict[str, str | None]`` - Pass arbitrary CLI flags to underlying Claude Code CLI

#### #### Permission modes

**\* \*\*`default`\*\*:** CLI prompts for dangerous tools (default behavior)

**\* \*\*`"acceptEdits"`\*\*:** Automatically accept file edits without prompting

**\* \*\*`"plan"`\*\*:** Plan Mode - analyze without making changes

**\* \*\*`"bypassPermissions"`\*\*:** Allow all tools without prompting (use with caution)

### **### Advanced configuration example**

```
```python

import asyncio

from claude_code_sdk import ClaudeSDKClient, ClaudeCodeOptions

async def advanced_agent():

    """Example showcasing advanced configuration options"""

    async with ClaudeSDKClient(

        options=ClaudeCodeOptions(

            # Custom working directory and additional context

            cwd="/project/root",

            add_dirs=["/shared/libs", "/common/utils"],

            # Model and thinking configuration

            model="claude-3-5-sonnet-20241022",

            max_thinking_tokens=12000,

            # Advanced tool control

            allowed_tools=["Read", "Write", "Bash", "Grep"],

            disallowed_tools=["WebSearch", "Bash(rm*)"],

        )

    pass
```

```

# Custom settings and CLI args

settings={"editor": "vim", "theme": "dark"},

extra_args={

    "--verbose": None,

    "--timeout": "300"

}

)

) as client:

    await client.query("Analyze the codebase structure")


    async for message in client.receive_response():

        if hasattr(message, 'content'):

            for block in message.content:

                if hasattr(block, 'text'):

                    print(block.text, end="", flush=True)


asyncio.run(advanced_agent())
...

```

## ## Structured messages and image inputs

The SDK supports passing structured messages and image inputs:

```

```python

from claude_code_sdk import ClaudeSDKClient, ClaudeCodeOptions


async with ClaudeSDKClient() as client:

```

**# Text message**

```
await client.query("Analyze this code for security issues")
```

**# Message with image reference (image will be read by Claude's Read tool)**

```
await client.query("Explain what's shown in screenshot.png")
```

**# Multiple messages in sequence**

```
messages = [  
    "First, analyze the architecture diagram in diagram.png",  
    "Now suggest improvements based on the diagram",  
    "Finally, generate implementation code"  
]
```

```
for msg in messages:
```

```
    await client.query(msg)
```

```
    async for response in client.receive_response():
```

```
        # Process each response
```

```
        pass
```

**# The SDK handles image files through Claude's built-in Read tool**

**# Supported formats: PNG, JPG, PDF, and other common formats**

...

**## Multi-turn conversations**

**### Method 1: Using ClaudeSDKClient for persistent conversations**

```

python

import asyncio

from claude_code_sdk import ClaudeSDKClient, ClaudeCodeOptions, query

# Method 1: Using ClaudeSDKClient for persistent conversations

async def multi_turn_conversation():

    async with ClaudeSDKClient() as client:

        # First query

        await client.query("Let's refactor the payment module")

        async for msg in client.receive_response():

            # Process first response

            pass

        # Continue in same session

        await client.query("Now add comprehensive error handling")

        async for msg in client.receive_response():

            # Process continuation

            pass

    # The conversation context is maintained throughout

# Method 2: Using query function with session management

async def resume_session():

    # Continue most recent conversation

    async for message in query(

        prompt="Now refactor this for better performance",

        options=ClaudeCodeOptions(continue_conversation=True)

```

```

):
    if type(message).__name__ == "ResultMessage":
        print(message.result)

# Resume specific session

async for message in query(
    prompt="Update the tests",
    options=ClaudeCodeOptions(
        resume="550e8400-e29b-41d4-a716-446655440000",
        max_turns=3
    )
):
    if type(message).__name__ == "ResultMessage":
        print(message.result)

# Run the examples

asyncio.run(multi_turn_conversation())
...

```

## ## Custom system prompts

System prompts define your agent's role, expertise, and behavior:

```

```python
import asyncio

from claude_code_sdk import ClaudeSDKClient, ClaudeCodeOptions

```



```

async def specialized_agents():

    # SRE incident response agent with streaming

    async with ClaudeSDKClient(

        options=ClaudeCodeOptions(

            system_prompt="You are an SRE expert. Diagnose issues systematically and provide actionable
solutions.",

            max_turns=3

        )

    ) as sre_agent:

        await sre_agent.query("API is down, investigate")

    # Stream the diagnostic process

    async for message in sre_agent.receive_response():

        if hasattr(message, 'content'):

            for block in message.content:

                if hasattr(block, 'text'):

                    print(block.text, end="", flush=True)

    # Legal review agent with custom prompt

    async with ClaudeSDKClient(

        options=ClaudeCodeOptions(

            append_system_prompt="Always include comprehensive error handling and unit tests.",

            max_turns=2

        )

    ) as dev_agent:

        await dev_agent.query("Refactor this function")

    # Collect full response

```

```

full_response = []

async for message in dev_agent.receive_response():

    if type(message).__name__ == "ResultMessage":

        print(message.result)

asyncio.run(specialized_agents())

```

```

'''

```

### ## Custom tools via MCP

The Model Context Protocol (MCP) lets you give your agents custom tools and capabilities:

```

'''python

import asyncio

from claude_code_sdk import ClaudeSDKClient, ClaudeCodeOptions

async def mcp_enabled_agent():

    # Legal agent with document access and streaming

    # Note: Configure your MCP servers as needed

    mcp_servers = {

        # Example configuration - uncomment and configure as needed:

        # "docusign": {

        #     "command": "npx",

        #     "args": ["-y", "@modelcontextprotocol/server-docusign"],

        #     "env": {"API_KEY": "your-key"}

        # }

    }

```

```

async with ClaudeSDKClient(
    options=ClaudeCodeOptions(
        mcp_servers=mcp_servers,
        allowed_tools=["mcp__docusign", "mcp__compliance_db"],
        system_prompt="You are a corporate lawyer specializing in contract review.",
        max_turns=4
    )
) as client:

    await client.query("Review this contract for compliance risks")

# Monitor tool usage and responses
async for message in client.receive_response():
    if hasattr(message, 'content'):
        for block in message.content:
            if hasattr(block, 'type'):
                if block.type == 'tool_use':
                    print(f"\n[Using tool: {block.name}]\n")
                elif hasattr(block, 'text'):
                    print(block.text, end="", flush=True)
            elif hasattr(block, 'text'):
                print(block.text, end="", flush=True)

    if type(message).__name__ == "ResultMessage":
        print(f"\n\nReview complete. Total cost: ${message.total_cost_usd:.4f}")

asyncio.run(mcp_enabled_agent())

```

```
...
```

```
## Custom permission prompt tool
```

Implement custom permission handling for tool calls:

```
```python
```

```
import asyncio
```

```
from claude_code_sdk import ClaudeSDKClient, ClaudeCodeOptions
```

```
async def use_permission_prompt():
```

```
    """Example using custom permission prompt tool"""
```

```
    # MCP server configuration
```

```
    mcp_servers = {
```

```
        # Example configuration - uncomment and configure as needed:
```

```
        # "security": {
```

```
            # "command": "npx",
```

```
            # "args": ["-y", "@modelcontextprotocol/server-security"],
```

```
            # "env": {"API_KEY": "your-key"}
```

```
        # }
```

```
    }
```

```
    async with ClaudeSDKClient(
```

```
        options=ClaudeCodeOptions(
```

```
            permission_prompt_tool_name="mcp__security__approval_prompt", # Changed from  
            permission_prompt_tool
```

```
            mcp_servers=mcp_servers,
```

```

        allowed_tools=["Read", "Grep"],

        disallowed_tools=["Bash(rm*)", "Write"],

        system_prompt="You are a security auditor"

    )

) as client:

    await client.query("Analyze and fix the security issues")


# Monitor tool usage and permissions

async for message in client.receive_response():

    if hasattr(message, 'content'):

        for block in message.content:

            if hasattr(block, 'type'): # Added check for 'type' attribute

                if block.type == 'tool_use':

                    print(f"[Tool: {block.name}] ", end="")

                if hasattr(block, 'text'):

                    print(block.text, end="", flush=True)


# Check for permission denials in error messages

if type(message).__name__ == "ErrorMessage":

    if hasattr(message, 'error') and "Permission denied" in str(message.error):

        print(f"\n ⚠️ Permission denied: {message.error}")


# Example MCP server implementation (Python)

# This would be in your MCP server code

async def approval_prompt(tool_name: str, input: dict, tool_use_id: str = None):

    """Custom permission prompt handler"""

    # Your custom logic here

```

```

    if "allow" in str(input):

        return json.dumps({

            "behavior": "allow",

            "updatedInput": input

        })

    else:

        return json.dumps({

            "behavior": "deny",

            "message": f"Permission denied for {tool_name}"

        })

asyncio.run(use_permission_prompt())

...

## Output formats

### Text output with streaming

```python
# Default text output with streaming

async with ClaudeSDKClient() as client:

    await client.query("Explain file src/components/Header.tsx")

# Stream text as it arrives

async for message in client.receive_response():

    if hasattr(message, 'content'):

        for block in message.content:

```

```
        if hasattr(block, 'text'):

            print(block.text, end="", flush=True)

            # Output streams in real-time: This is a React component showing...

...

```

### JSON output with metadata

```
```python

# Collect all messages with metadata

async with ClaudeSDKClient() as client:

    await client.query("How does the data layer work?")

    messages = []

    result_data = None

    async for message in client.receive_messages():

        messages.append(message)

        # Capture result message with metadata

        if type(message).__name__ == "ResultMessage":

            result_data = {

                "result": message.result,

                "cost": message.total_cost_usd,

                "duration": message.duration_ms,

                "num_turns": message.num_turns,

                "session_id": message.session_id

            }

}

```

```
break
```

```
print(result_data)
```

```
...
```

```
## Input formats
```

```
```python
```

```
import asyncio
```

```
from claude_code_sdk import ClaudeSDKClient
```

```
async def process_inputs():
```

```
    async with ClaudeSDKClient() as client:
```

```
        # Text input
```

```
        await client.query("Explain this code")
```

```
        async for message in client.receive_response():
```

```
            # Process streaming response
```

```
            pass
```

```
        # Image input (Claude will use Read tool automatically)
```

```
        await client.query("What's in this diagram? screenshot.png")
```

```
        async for message in client.receive_response():
```

```
            # Process image analysis
```

```
            pass
```

```
        # Multiple inputs with mixed content
```

```
        inputs = [
```



```
    "Analyze the architecture in diagram.png",  
    "Compare it with best practices",  
    "Generate improved version"  
]
```

```
for prompt in inputs:  
    await client.query(prompt)  
    async for message in client.receive_response():  
        # Process each response  
        pass
```

```
asyncio.run(process_inputs())
```

```
...
```

```
## Agent integration examples
```

```
### SRE incident response agent
```

```
```python
```

```
import asyncio
```

```
from claude_code_sdk import ClaudeSDKClient, ClaudeCodeOptions
```

```
async def investigate_incident(incident_description: str, severity: str = "medium"):
```

```
    """Automated incident response agent with real-time streaming"""
```

```
# MCP server configuration for monitoring tools
```

```
mcp_servers = {
```

```

# Example configuration - uncomment and configure as needed:

# "datadog": {

#   "command": "npx",

#   "args": ["-y", "@modelcontextprotocol/server-datadog"],

#   "env": {"API_KEY": "your-datadog-key", "APP_KEY": "your-app-key"}

# }

}

```

```

async with ClaudeSDKClient(

    options=ClaudeCodeOptions(

        system_prompt="You are an SRE expert. Diagnose issues systematically and provide actionable
solutions.",

        max_turns=6,

        allowed_tools=["Bash", "Read", "WebSearch", "mcp__datadog"],

        mcp_servers=mcp_servers

    )

) as client:

    # Send the incident details

    prompt = f"Incident: {incident_description} (Severity: {severity})"

    print(f"🔍 Investigating: {prompt}\n")

    await client.query(prompt)


# Stream the investigation process

investigation_log = []

async for message in client.receive_response():

    if hasattr(message, 'content'):

        for block in message.content:

            if hasattr(block, 'type'):

```

```

        if block.type == 'tool_use':

            print(f"[{block.name}] ", end=")

    if hasattr(block, 'text'):

        text = block.text

        print(text, end=" ", flush=True)

        investigation_log.append(text)


# Capture final result

if type(message).__name__ == "ResultMessage":

    return {

        'analysis': ".join(investigation_log),

        'cost': message.total_cost_usd,

        'duration_ms': message.duration_ms

    }


# Usage

result = await investigate_incident("Payment API returning 500 errors", "high")

print(f"\n\nInvestigation complete. Cost: ${result['cost']:.4f}")

...


### Automated security review


```python

import subprocess

import asyncio

import json

from claude_code_sdk import ClaudeSDKClient, ClaudeCodeOptions

```

```

async def audit_pr(pr_number: int):

    """Security audit agent for pull requests with streaming feedback"""

    # Get PR diff

    pr_diff = subprocess.check_output(

        ["gh", "pr", "diff", str(pr_number)],

        text=True

    )


    async with ClaudeSDKClient(

        options=ClaudeCodeOptions(

            system_prompt="You are a security engineer. Review this PR for vulnerabilities, insecure patterns,
and compliance issues.",

            max_turns=3,

            allowed_tools=["Read", "Grep", "WebSearch"]

        )

    ) as client:

        print(f"🔍 Auditing PR #{pr_number}\n")

        await client.query(pr_diff)


        findings = []

        async for message in client.receive_response():

            if hasattr(message, 'content'):

                for block in message.content:

                    if hasattr(block, 'text'):

                        # Stream findings as they're discovered

                        print(block.text, end="", flush=True)

                        findings.append(block.text)

```

```

if type(message).__name__ == "ResultMessage":

    return {

        'pr_number': pr_number,

        'findings': ".join(findings),

        'metadata': {

            'cost': message.total_cost_usd,

            'duration': message.duration_ms,

            'severity': 'high' if 'vulnerability' in ".join(findings).lower() else 'medium'

        }

    }
}

```

#### # Usage

```

report = await audit_pr(123)

print(f"\n\nAudit complete. Severity: {report['metadata']['severity']}")

print(json.dumps(report, indent=2))

...

```

#### ### Multi-turn legal assistant

```

```python

import asyncio

from claude_code_sdk import ClaudeSDKClient, ClaudeCodeOptions

async def legal_review():

    """Legal document review with persistent session and streaming"""

```

```

async with ClaudeSDKClient(
    options=ClaudeCodeOptions(
        system_prompt="You are a corporate lawyer. Provide detailed legal analysis.",
        max_turns=2
    )
) as client:

    # Multi-step review in same session

    steps = [
        "Review contract.pdf for liability clauses",
        "Check compliance with GDPR requirements",
        "Generate executive summary of risks"
    ]

    review_results = []

    for step in steps:
        print(f"\n📋 {step}\n")

        await client.query(step)

        step_result = []

        async for message in client.receive_response():
            if hasattr(message, 'content'):
                for block in message.content:
                    if hasattr(block, 'text'):
                        text = block.text

                        print(text, end="", flush=True)

                        step_result.append(text)

```

```
if type(message).__name__ == "ResultMessage":
```

```
    review_results.append({
```

```
        'step': step,
```

```
        'analysis': ".join(step_result),
```

```
        'cost': message.total_cost_usd
```

```
    })
```

```
# Summary
```

```
total_cost = sum(r['cost'] for r in review_results)
```

```
print(f"\n\n✅ Legal review complete. Total cost: ${total_cost:.4f}")
```

```
return review_results
```

```
# Usage
```

```
results = await legal_review()
```

```
...
```

```
## Python-specific best practices
```

```
### Key patterns
```

```
```python
```

```
import asyncio
```

```
from claude_code_sdk import ClaudeSDKClient, ClaudeCodeOptions
```

```
# Always use context managers
```

```
async with ClaudeSDKClient() as client:
```

```

await client.query("Analyze this code")

async for msg in client.receive_response():

    # Process streaming messages

    pass


# Run multiple agents concurrently

async with ClaudeSDKClient() as reviewer, ClaudeSDKClient() as tester:

    await asyncio.gather(

        reviewer.query("Review main.py"),

        tester.query("Write tests for main.py")

    )


# Error handling

from claude_code_sdk import CLINotFoundError, ProcessError


try:

    async with ClaudeSDKClient() as client:

        # Your code here

        pass

except CLINotFoundError:

    print("Install CLI: npm install -g @anthropic-ai/claude-code")

except ProcessError as e:

    print(f"Process error: {e}")


# Collect full response with metadata

async def get_response(client, prompt):

    await client.query(prompt)

```



```

text = []

async for msg in client.receive_response():

    if hasattr(msg, 'content'):

        for block in msg.content:

            if hasattr(block, 'text'):

                text.append(block.text)

        if type(msg).__name__ == "ResultMessage":

            return {'text': ".join(text), 'cost': msg.total_cost_usd}

'''

```

### IPython/Jupyter tips

'''python

# In Jupyter, use await directly in cells

```
client = ClaudeSDKClient()
```

```
await client.connect()
```

```
await client.query("Analyze data.csv")
```

```
async for msg in client.receive_response():
```

```
    print(msg)
```

```
await client.disconnect()
```

# Create reusable helper functions

```
async def stream_print(client, prompt):
```

```
    await client.query(prompt)
```

```
    async for msg in client.receive_response():
```

```
        if hasattr(msg, 'content'):
```

```
            for block in msg.content:
```

```
if hasattr(block, 'text'):

    print(block.text, end="", flush=True)
```

```
...
```

## ## Related resources

\* [\[CLI usage and controls\]\(/en/docs/claude-code/cli-reference\)](#) - Complete CLI documentation

\* [\[GitHub Actions integration\]\(/en/docs/claude-code/github-actions\)](#) - Automate your GitHub workflow with Claude

\* [\[Common workflows\]\(/en/docs/claude-code/common-workflows\)](#) - Step-by-step guides for common use cases

## # Python

> Build custom AI agents with the Claude Code Python SDK

## ## Prerequisites

\* Python 3.10+

\* `claude-code-sdk` from PyPI`

\* Node.js 18+

\* `@anthropic-ai/claude-code` from NPM`

<Note>

To view the Python SDK source code, see the `[`claude-code-sdk`](https://github.com/anthropics/claude-code-sdk-python)` repo.

</Note>

<Tip>

For interactive development, use [IPython](https://ipython.org/): `pip install ipython``

</Tip>

## ## Installation

Install `claude-code-sdk`` from PyPI and `@anthropic-ai/claude-code`` from NPM:

```
```bash
pip install claude-code-sdk
npm install -g @anthropic-ai/claude-code # Required dependency
```
```

(Optional) Install IPython for interactive development:

```
```bash
pip install ipython
```
```

## ## Quick start

Create your first agent:

```
```python
# legal-agent.py
import asyncio

from claude_code_sdk import ClaudeSDKClient, ClaudeCodeOptions
```

```

async def main():

    async with ClaudeSDKClient(

        options=ClaudeCodeOptions(

            system_prompt="You are a legal assistant. Identify risks and suggest improvements.",

            max_turns=2

        )

    ) as client:

        # Send the query

        await client.query(

            "Review this contract clause for potential issues: 'The party agrees to unlimited liability...'"

        )


        # Stream the response

        async for message in client.receive_response():

            if hasattr(message, 'content'):

                # Print streaming content as it arrives

                for block in message.content:

                    if hasattr(block, 'text'):

                        print(block.text, end="", flush=True)


if __name__ == "__main__":

    asyncio.run(main())

...

```

Save the code above as `legal-agent.py`, then run:

```

```bash

```

```
python legal-agent.py
```

```
'''
```

For [IPython](https://ipython.org/) Jupyter notebooks, you can run the code directly in a cell:

```
'''python
```

```
await main()
```

```
'''
```

<Note>

The Python examples on this page use `asyncio`, but you can also use `anyio`.

</Note>

## ## Basic usage

The Python SDK provides two primary interfaces:

### ### 1. The `ClaudeSDKClient` class (recommended)

Best for streaming responses, multi-turn conversations, and interactive applications:

```
'''python
```

```
import asyncio
```

```
from claude_code_sdk import ClaudeSDKClient, ClaudeCodeOptions
```

```
async def main():
```

```
    async with ClaudeSDKClient(
```

```

options=ClaudeCodeOptions(
    system_prompt="You are a performance engineer",
    allowed_tools=["Bash", "Read", "WebSearch"],
    max_turns=5
)
) as client:

    await client.query("Analyze system performance")


# Stream responses

async for message in client.receive_response():

    if hasattr(message, 'content'):

        for block in message.content:

            if hasattr(block, 'text'):

                print(block.text, end="", flush=True)


# Run as script

asyncio.run(main())


# Or in IPython/Jupyter: await main()
...

```

### ### 2. The `query` function

For simple, one-shot queries:

```

```python
from claude_code_sdk import query, ClaudeCodeOptions

```

```

async for message in query(
    prompt="Analyze system performance",
    options=ClaudeCodeOptions(system_prompt="You are a performance engineer")
):
    if type(message).__name__ == "ResultMessage":
        print(message.result)
...

```

### ## Configuration options

The Python SDK accepts all arguments supported by the [\[command line\]](/en/docs/claude-code/cli-reference) through the `ClaudeCodeOptions` class.

### ### ClaudeCodeOptions parameters

```

```python
from claude_code_sdk import ClaudeCodeOptions

options = ClaudeCodeOptions(
    # Core configuration

    system_prompt="You are a helpful assistant",
    append_system_prompt="Additional system instructions",
    max_turns=5,
    model="claude-3-5-sonnet-20241022",
    max_thinking_tokens=8000,

    # Tool management

```

```
allowed_tools=["Bash", "Read", "Write"],
```

```
disallowed_tools=["WebSearch"],
```

#### **# Session management**

```
continue_conversation=False,
```

```
resume="session-uuid",
```

#### **# Environment**

```
cwd="/path/to/working/directory",
```

```
add_dirs=["/additional/context/dir"],
```

```
settings="/path/to/settings.json",
```

#### **# Permissions**

```
permission_mode="acceptEdits", # "default", "acceptEdits", "plan", "bypassPermissions"
```

```
permission_prompt_tool_name="mcp__approval_tool",
```

#### **# MCP integration**

```
mcp_servers={
```

```
    "my_server": {
```

```
        "command": "npx",
```

```
        "args": ["-y", "@modelcontextprotocol/server-example"],
```

```
        "env": {"API_KEY": "your-key"}
```

```
    }
```

```
},
```

#### **# Advanced**

```
extra_args={"--verbose": None, "--custom-flag": "value"}
```



)  
...

#### #### Parameter details

\* \*\*`system\_prompt`\*\*: `str | None` - Custom system prompt defining the agent's role

\* \*\*`append\_system\_prompt`\*\*: `str | None` - Additional text appended to system prompt

\* \*\*`max\_turns`\*\*: `int | None` - Maximum conversation turns (unlimited if None)

\* \*\*`model`\*\*: `str | None` - Specific Claude model to use

\* \*\*`max\_thinking\_tokens`\*\*: `int` - Maximum tokens for Claude's thinking process (default: 8000)

\* \*\*`allowed\_tools`\*\*: `list[str]` - Tools specifically allowed for use

\* \*\*`disallowed\_tools`\*\*: `list[str]` - Tools that should not be used

\* \*\*`continue\_conversation`\*\*: `bool` - Continue most recent conversation (default: False)

\* \*\*`resume`\*\*: `str | None` - Session UUID to resume specific conversation

\* \*\*`cwd`\*\*: `str | Path | None` - Working directory for the session

\* \*\*`add\_dirs`\*\*: `list[str | Path]` - Additional directories to include in context

\* \*\*`settings`\*\*: `str | None` - Path to settings file or settings JSON string

\* \*\*`permission\_mode`\*\*: `str | None` - Permission handling mode

\* \*\*`permission\_prompt\_tool\_name`\*\*: `str | None` - Custom permission prompt tool name

\* \*\*`mcp\_servers`\*\*: `dict | str | Path` - MCP server configurations

\* \*\*`extra\_args`\*\*: `dict[str, str | None]` - Pass arbitrary CLI flags to underlying Claude Code CLI

#### #### Permission modes

\* \*\*`default`\*\*: CLI prompts for dangerous tools (default behavior)

\* \*\*`acceptEdits`\*\*: Automatically accept file edits without prompting

\* \*\*`plan`\*\*: Plan Mode - analyze without making changes

**\* \*\*`"bypassPermissions"`\*: Allow all tools without prompting (use with caution)**

### **### Advanced configuration example**

```
```python

import asyncio

from claude_code_sdk import ClaudeSDKClient, ClaudeCodeOptions

async def advanced_agent():

    """Example showcasing advanced configuration options"""

    async with ClaudeSDKClient(

        options=ClaudeCodeOptions(

            # Custom working directory and additional context

            cwd="/project/root",

            add_dirs=["/shared/libs", "/common/utils"],

            # Model and thinking configuration

            model="claude-3-5-sonnet-20241022",

            max_thinking_tokens=12000,

            # Advanced tool control

            allowed_tools=["Read", "Write", "Bash", "Grep"],

            disallowed_tools=["WebSearch", "Bash(rm*)"],

            # Custom settings and CLI args

            settings={"editor": "vim", "theme": "dark"},
```

```

        extra_args={
            "--verbose": None,
            "--timeout": "300"
        }
    )
) as client:

    await client.query("Analyze the codebase structure")


    async for message in client.receive_response():

        if hasattr(message, 'content'):

            for block in message.content:

                if hasattr(block, 'text'):

                    print(block.text, end="", flush=True)


    asyncio.run(advanced_agent())
...

```

## ## Structured messages and image inputs

The SDK supports passing structured messages and image inputs:

```

```python
from claude_code_sdk import ClaudeSDKClient, ClaudeCodeOptions

async with ClaudeSDKClient() as client:

    # Text message

    await client.query("Analyze this code for security issues")

```

**# Message with image reference (image will be read by Claude's Read tool)**

```
await client.query("Explain what's shown in screenshot.png")
```

**# Multiple messages in sequence**

```
messages = [
```

```
    "First, analyze the architecture diagram in diagram.png",
```

```
    "Now suggest improvements based on the diagram",
```

```
    "Finally, generate implementation code"
```

```
]
```

```
for msg in messages:
```

```
    await client.query(msg)
```

```
    async for response in client.receive_response():
```

```
        # Process each response
```

```
        pass
```

**# The SDK handles image files through Claude's built-in Read tool**

**# Supported formats: PNG, JPG, PDF, and other common formats**

...

**## Multi-turn conversations**

**### Method 1: Using ClaudeSDKClient for persistent conversations**

```
```python
```

```
import asyncio
```

```
from claude_code_sdk import ClaudeSDKClient, ClaudeCodeOptions, query
```

```
# Method 1: Using ClaudeSDKClient for persistent conversations
```

```
async def multi_turn_conversation():
```

```
    async with ClaudeSDKClient() as client:
```

```
        # First query
```

```
        await client.query("Let's refactor the payment module")
```

```
        async for msg in client.receive_response():
```

```
            # Process first response
```

```
            pass
```

```
        # Continue in same session
```

```
        await client.query("Now add comprehensive error handling")
```

```
        async for msg in client.receive_response():
```

```
            # Process continuation
```

```
            pass
```

```
    # The conversation context is maintained throughout
```

```
# Method 2: Using query function with session management
```

```
async def resume_session():
```

```
    # Continue most recent conversation
```

```
    async for message in query(
```

```
        prompt="Now refactor this for better performance",
```

```
        options=ClaudeCodeOptions(continue_conversation=True)
```

```
    ):
```

```
        if type(message).__name__ == "ResultMessage":
```

```

        print(message.result)

    # Resume specific session

    async for message in query(
        prompt="Update the tests",
        options=ClaudeCodeOptions(
            resume="550e8400-e29b-41d4-a716-446655440000",
            max_turns=3
        )
    ):
        if type(message).__name__ == "ResultMessage":
            print(message.result)

# Run the examples

asyncio.run(multi_turn_conversation())
...

```

## ## Custom system prompts

System prompts define your agent's role, expertise, and behavior:

```

```python
import asyncio

from claude_code_sdk import ClaudeSDKClient, ClaudeCodeOptions

async def specialized_agents():

    # SRE incident response agent with streaming

```

```

async with ClaudeSDKClient(
    options=ClaudeCodeOptions(
        system_prompt="You are an SRE expert. Diagnose issues systematically and provide actionable
solutions.",
        max_turns=3
    )
) as sre_agent:

    await sre_agent.query("API is down, investigate")

# Stream the diagnostic process

async for message in sre_agent.receive_response():

    if hasattr(message, 'content'):

        for block in message.content:

            if hasattr(block, 'text'):

                print(block.text, end="", flush=True)

# Legal review agent with custom prompt

async with ClaudeSDKClient(
    options=ClaudeCodeOptions(
        append_system_prompt="Always include comprehensive error handling and unit tests.",
        max_turns=2
    )
) as dev_agent:

    await dev_agent.query("Refactor this function")

# Collect full response

full_response = []

async for message in dev_agent.receive_response():

```

```
        if type(message).__name__ == "ResultMessage":
            print(message.result)
```

```
    asyncio.run(specialized_agents())
```

```
'''
```

```
## Custom tools via MCP
```

The Model Context Protocol (MCP) lets you give your agents custom tools and capabilities:

```
'''python
```

```
import asyncio
```

```
from claude_code_sdk import ClaudeSDKClient, ClaudeCodeOptions
```

```
async def mcp_enabled_agent():
```

```
    # Legal agent with document access and streaming
```

```
    # Note: Configure your MCP servers as needed
```

```
    mcp_servers = {
```

```
        # Example configuration - uncomment and configure as needed:
```

```
        # "docusign": {
```

```
            # "command": "npx",
```

```
            # "args": ["-y", "@modelcontextprotocol/server-docusign"],
```

```
            # "env": {"API_KEY": "your-key"}
```

```
        # }
```

```
    }
```

```
    async with ClaudeSDKClient(
```



```

options=ClaudeCodeOptions(
    mcp_servers=mcp_servers,
    allowed_tools=["mcp__docusign", "mcp__compliance_db"],
    system_prompt="You are a corporate lawyer specializing in contract review.",
    max_turns=4
)

) as client:

    await client.query("Review this contract for compliance risks")


# Monitor tool usage and responses

async for message in client.receive_response():

    if hasattr(message, 'content'):

        for block in message.content:

            if hasattr(block, 'type'):

                if block.type == 'tool_use':

                    print(f"\n[Using tool: {block.name}]\n")

                elif hasattr(block, 'text'):

                    print(block.text, end="", flush=True)

            elif hasattr(block, 'text'):

                print(block.text, end="", flush=True)


    if type(message).__name__ == "ResultMessage":

        print(f"\n\nReview complete. Total cost: ${message.total_cost_usd:.4f}")


asyncio.run(mcp_enabled_agent())
...

```

**## Custom permission prompt tool**

**Implement custom permission handling for tool calls:**

```
```python

import asyncio

from claude_code_sdk import ClaudeSDKClient, ClaudeCodeOptions

async def use_permission_prompt():

    """Example using custom permission prompt tool"""

    # MCP server configuration

    mcp_servers = {

        # Example configuration - uncomment and configure as needed:

        # "security": {

        #     "command": "npx",

        #     "args": ["-y", "@modelcontextprotocol/server-security"],

        #     "env": {"API_KEY": "your-key"}

        # }

    }

    async with ClaudeSDKClient(

        options=ClaudeCodeOptions(

            permission_prompt_tool_name="mcp__security__approval_prompt", # Changed from
permission_prompt_tool

            mcp_servers=mcp_servers,

            allowed_tools=["Read", "Grep"],

            disallowed_tools=["Bash(rm*)", "Write"],
```

```

        system_prompt="You are a security auditor"

    )

) as client:

    await client.query("Analyze and fix the security issues")


# Monitor tool usage and permissions

async for message in client.receive_response():

    if hasattr(message, 'content'):

        for block in message.content:

            if hasattr(block, 'type'): # Added check for 'type' attribute

                if block.type == 'tool_use':

                    print(f"[Tool: {block.name}] ", end="")

                if hasattr(block, 'text'):

                    print(block.text, end="", flush=True)


# Check for permission denials in error messages

if type(message).__name__ == "ErrorMessage":

    if hasattr(message, 'error') and "Permission denied" in str(message.error):

        print(f"\n ⚠️ Permission denied: {message.error}")


# Example MCP server implementation (Python)

# This would be in your MCP server code

async def approval_prompt(tool_name: str, input: dict, tool_use_id: str = None):

    """Custom permission prompt handler"""

    # Your custom logic here

    if "allow" in str(input):

        return json.dumps({

```

```

        "behavior": "allow",

        "updatedInput": input

    })

else:

    return json.dumps({

        "behavior": "deny",

        "message": f"Permission denied for {tool_name}"

    })

asyncio.run(use_permission_prompt())

...

## Output formats

### Text output with streaming

```python
# Default text output with streaming

async with ClaudeSDKClient() as client:

    await client.query("Explain file src/components/Header.tsx")

# Stream text as it arrives

async for message in client.receive_response():

    if hasattr(message, 'content'):

        for block in message.content:

            if hasattr(block, 'text'):

                print(block.text, end="", flush=True)

```

**# Output streams in real-time: This is a React component showing...**

...

**### JSON output with metadata**

**```python**

**# Collect all messages with metadata**

**async with ClaudeSDKClient() as client:**

**await client.query("How does the data layer work?")**

**messages = []**

**result\_data = None**

**async for message in client.receive\_messages():**

**messages.append(message)**

**# Capture result message with metadata**

**if type(message).\_\_name\_\_ == "ResultMessage":**

**result\_data = {**

**"result": message.result,**

**"cost": message.total\_cost\_usd,**

**"duration": message.duration\_ms,**

**"num\_turns": message.num\_turns,**

**"session\_id": message.session\_id**

**}**

**break**

```
print(result_data)
```

```
...
```

```
## Input formats
```

```
```python
```

```
import asyncio
```

```
from claude_code_sdk import ClaudeSDKClient
```

```
async def process_inputs():
```

```
    async with ClaudeSDKClient() as client:
```

```
        # Text input
```

```
        await client.query("Explain this code")
```

```
        async for message in client.receive_response():
```

```
            # Process streaming response
```

```
            pass
```

```
        # Image input (Claude will use Read tool automatically)
```

```
        await client.query("What's in this diagram? screenshot.png")
```

```
        async for message in client.receive_response():
```

```
            # Process image analysis
```

```
            pass
```

```
    # Multiple inputs with mixed content
```

```
    inputs = [
```

```
        "Analyze the architecture in diagram.png",
```

```
        "Compare it with best practices",
```

```
"Generate improved version"
```

```
]
```

```
for prompt in inputs:
```

```
    await client.query(prompt)
```

```
    async for message in client.receive_response():
```

```
        # Process each response
```

```
        pass
```

```
asyncio.run(process_inputs())
```

```
...
```

```
## Agent integration examples
```

```
### SRE incident response agent
```

```
```python
```

```
import asyncio
```

```
from claude_code_sdk import ClaudeSDKClient, ClaudeCodeOptions
```

```
async def investigate_incident(incident_description: str, severity: str = "medium"):
```

```
    """Automated incident response agent with real-time streaming"""
```

```
    # MCP server configuration for monitoring tools
```

```
    mcp_servers = {
```

```
        # Example configuration - uncomment and configure as needed:
```

```
        # "datadog": {
```





```
    if hasattr(block, 'text'):

        text = block.text

        print(text, end="", flush=True)

        investigation_log.append(text)
```

```
# Capture final result
```

```
if type(message).__name__ == "ResultMessage":

    return {

        'analysis': ".join(investigation_log),

        'cost': message.total_cost_usd,

        'duration_ms': message.duration_ms

    }
```

```
# Usage
```

```
result = await investigate_incident("Payment API returning 500 errors", "high")

print(f"\n\nInvestigation complete. Cost: ${result['cost']:.4f}")

...
```

```
### Automated security review
```

```
```python

import subprocess

import asyncio

import json

from claude_code_sdk import ClaudeSDKClient, ClaudeCodeOptions

async def audit_pr(pr_number: int):
```

```

"""Security audit agent for pull requests with streaming feedback"""

# Get PR diff

pr_diff = subprocess.check_output(

    ["gh", "pr", "diff", str(pr_number)],

    text=True

)


async with ClaudeSDKClient(

    options=ClaudeCodeOptions(

        system_prompt="You are a security engineer. Review this PR for vulnerabilities, insecure patterns,
and compliance issues.",

        max_turns=3,

        allowed_tools=["Read", "Grep", "WebSearch"]

    )

) as client:

    print(f"🔍 Auditing PR #{pr_number}\n")

    await client.query(pr_diff)


findings = []

async for message in client.receive_response():

    if hasattr(message, 'content'):

        for block in message.content:

            if hasattr(block, 'text'):

                # Stream findings as they're discovered

                print(block.text, end="", flush=True)

                findings.append(block.text)


    if type(message).__name__ == "ResultMessage":

```

```

    return {

        'pr_number': pr_number,

        'findings': ".join(findings),

        'metadata': {

            'cost': message.total_cost_usd,

            'duration': message.duration_ms,

            'severity': 'high' if 'vulnerability' in ".join(findings).lower() else 'medium'

        }

    }
}

```

## # Usage

```

report = await audit_pr(123)

print(f"\n\nAudit complete. Severity: {report['metadata']['severity']}")

print(json.dumps(report, indent=2))

...

```

## ### Multi-turn legal assistant

```

```python

import asyncio

from claude_code_sdk import ClaudeSDKClient, ClaudeCodeOptions

async def legal_review():

    """Legal document review with persistent session and streaming"""

    async with ClaudeSDKClient(

        options=ClaudeCodeOptions(

```

```

    system_prompt="You are a corporate lawyer. Provide detailed legal analysis.",
    max_turns=2
)
) as client:

    # Multi-step review in same session

    steps = [

        "Review contract.pdf for liability clauses",

        "Check compliance with GDPR requirements",

        "Generate executive summary of risks"

    ]

    review_results = []

    for step in steps:

        print(f"\n📋 {step}\n")

        await client.query(step)

        step_result = []

        async for message in client.receive_response():

            if hasattr(message, 'content'):

                for block in message.content:

                    if hasattr(block, 'text'):

                        text = block.text

                        print(text, end="", flush=True)

                        step_result.append(text)

        if type(message).__name__ == "ResultMessage":

```

```
review_results.append({  
    'step': step,  
    'analysis': ".join(step_result),  
    'cost': message.total_cost_usd  
})
```

**# Summary**

```
total_cost = sum(r['cost'] for r in review_results)
```

```
print(f"\n\n✅ Legal review complete. Total cost: ${total_cost:.4f}")
```

```
return review_results
```

**# Usage**

```
results = await legal_review()
```

```
...
```

**## Python-specific best practices**

**### Key patterns**

```
```python
```

```
import asyncio
```

```
from claude_code_sdk import ClaudeSDKClient, ClaudeCodeOptions
```

**# Always use context managers**

```
async with ClaudeSDKClient() as client:
```

```
    await client.query("Analyze this code")
```

```
    async for msg in client.receive_response():
```

```

# Process streaming messages

pass

# Run multiple agents concurrently

async with ClaudeSDKClient() as reviewer, ClaudeSDKClient() as tester:

    await asyncio.gather(

        reviewer.query("Review main.py"),

        tester.query("Write tests for main.py")

    )

# Error handling

from claude_code_sdk import CLINotFoundError, ProcessError

try:

    async with ClaudeSDKClient() as client:

        # Your code here

        pass

except CLINotFoundError:

    print("Install CLI: npm install -g @anthropic-ai/claude-code")

except ProcessError as e:

    print(f"Process error: {e}")

# Collect full response with metadata

async def get_response(client, prompt):

    await client.query(prompt)

    text = []

    async for msg in client.receive_response():

```

...

### ### IPython/Jupyter tips

```
python
```

## # In Jupyter, use await directly in cells

```
client = ClaudeSDKClient()
```

```
await client.connect()
```

```
await client.query("Analyze data.csv")
```

**async for msg in client.receive\_response():**

```
print(msg)
```

```
await client.disconnect()
```

## # Create reusable helper functions

```
async def stream_print(client, prompt):
```

```
await client.query(prompt)
```

**async for msg in client.receive\_response():**

```
if hasattr(msg, 'content'):
```

**for block in msg.content:**

```
if hasattr(block, 'text'):
```

```
print(block.text, end="", flush=True)
```

...

## ## Related resources

- \* [\[CLI usage and controls\]\(/en/docs/claude-code/cli-reference\)](#) - Complete CLI documentation
- \* [\[GitHub Actions integration\]\(/en/docs/claude-code/github-actions\)](#) - Automate your GitHub workflow with Claude
- \* [\[Common workflows\]\(/en/docs/claude-code/common-workflows\)](#) - Step-by-step guides for common use cases

## # Claude Code on Amazon Bedrock

> Learn about configuring Claude Code through Amazon Bedrock, including setup, IAM configuration, and troubleshooting.

## ## Prerequisites

Before configuring Claude Code with Bedrock, ensure you have:

- \* An AWS account with Bedrock access enabled
- \* Access to desired Claude models (e.g., Claude Sonnet 4) in Bedrock
- \* AWS CLI installed and configured (optional - only needed if you don't have another mechanism for getting credentials)
- \* Appropriate IAM permissions

## ## Setup

### ### 1. Enable model access



First, ensure you have access to the required Claude models in your AWS account:

1. Navigate to the [Amazon Bedrock console](https://console.aws.amazon.com/bedrock/)
2. Go to **Model access** in the left navigation
3. Request access to desired Claude models (e.g., Claude Sonnet 4)
4. Wait for approval (usually instant for most regions)

### ### 2. Configure AWS credentials

Claude Code uses the default AWS SDK credential chain. Set up your credentials using one of these methods:

#### **\*\*Option A: AWS CLI configuration\*\***

```
```bash
aws configure
```
```

#### **\*\*Option B: Environment variables (access key)\*\***

```
```bash
export AWS_ACCESS_KEY_ID=your-access-key-id
export AWS_SECRET_ACCESS_KEY=your-secret-access-key
export AWS_SESSION_TOKEN=your-session-token
```
```

#### **\*\*Option C: Environment variables (SSO profile)\*\***

```
```bash
```

```
aws sso login --profile=<your-profile-name>
```

```
export AWS_PROFILE=your-profile-name
```

```
```
```

#### **\*\*Option D: Bedrock API keys\*\***

```
```bash
```

```
export AWS_BEARER_TOKEN_BEDROCK=your-bedrock-api-key
```

```
```
```

Bedrock API keys provide a simpler authentication method without needing full AWS credentials. [Learn more about Bedrock API keys](<https://aws.amazon.com/blogs/machine-learning/accelerate-ai-development-with-amazon-bedrock-api-keys/>).

#### **#### Advanced credential configuration**

Claude Code supports automatic credential refresh for AWS SSO and corporate identity providers. Add these settings to your Claude Code settings file (see [Settings](/en/docs/claude-code/settings) for file locations).

When Claude Code detects that your AWS credentials are expired (either locally based on their timestamp or when Bedrock returns a credential error), it will automatically run your configured `awsAuthRefresh` and/or `awsCredentialExport` commands to obtain new credentials before retrying the request.

#### **##### Example configuration**

```
```json
```

```
{
```

```
"awsAuthRefresh": "aws sso login --profile myprofile",

"env": {

  "AWS_PROFILE": "myprofile"

}

}

...
```

#### ##### Configuration settings explained

**`awsAuthRefresh`**: Use this for commands that modify the `.aws` directory (e.g., updating credentials, SSO cache, or config files). Output is shown to the user (but user input is not supported), making it suitable for browser-based authentication flows where the CLI displays a code to enter in the browser.

**`awsCredentialExport`**: Only use this if you cannot modify `.aws` and must directly return credentials. Output is captured silently (not shown to the user). The command must output JSON in this format:

```
```json

{

  "Credentials": {

    "AccessKeyId": "value",

    "SecretAccessKey": "value",

    "SessionToken": "value"

  }

}

...


```

### ### 3. Configure Claude Code

Set the following environment variables to enable Bedrock:

```

```bash

# Enable Bedrock integration

export CLAUDE_CODE_USE_BEDROCK=1

export AWS_REGION=us-east-1 # or your preferred region


# Optional: Override the region for the small/fast model (Haiku)

export ANTHROPIC_SMALL_FAST_MODEL_AWS_REGION=us-west-2
```

```

When enabling Bedrock for Claude Code, keep the following in mind:

\* `AWS_REGION` is a required environment variable. Claude Code does not read from the `.aws` config file for this setting.

\* When using Bedrock, the `/login` and `/logout` commands are disabled since authentication is handled through AWS credentials.

\* You can use settings files for environment variables like `AWS_PROFILE` that you don't want to leak to other processes. See [\[Settings\]\(/en/docs/claude-code/settings\)](#) for more information.

### ### 4. Model configuration

Claude Code uses these default models for Bedrock:

| Model type       | Default value                                             |
|------------------|-----------------------------------------------------------|
| Primary model    | <code>us.anthropic.claude-3-7-sonnet-20250219-v1:0</code> |
| Small/fast model | <code>us.anthropic.claude-3-5-haiku-20241022-v1:0</code>  |

To customize models, use one of these methods:

```
```bash
```

```
# Using inference profile ID
```

```
export ANTHROPIC_MODEL='us.anthropic.claude-opus-4-1-20250805-v1:0'
```

```
export ANTHROPIC_SMALL_FAST_MODEL='us.anthropic.claude-3-5-haiku-20241022-v1:0'
```

```
# Using application inference profile ARN
```

```
export
```

```
ANTHROPIC_MODEL='arn:aws:bedrock:us-east-2:your-account-id:application-inference-profile/your-model-id'
```

```
# Optional: Disable prompt caching if needed
```

```
export DISABLE_PROMPT_CACHING=1
```

```
```
```

<Note>

[Prompt caching](/en/docs/build-with-claude/prompt-caching) may not be available in all regions

</Note>

### ### 5. Output token configuration

When using Claude Code with Amazon Bedrock, we recommend the following token settings:

```
```bash
```

```
# Recommended output token settings for Bedrock
```

```
export CLAUDE_CODE_MAX_OUTPUT_TOKENS=4096
```

```
export MAX_THINKING_TOKENS=1024
```

```
```
```

**\*\*Why these values:\*\***

**\* \*\*`CLAUDE\_CODE\_MAX\_OUTPUT\_TOKENS=4096`\*\*:** Bedrock's burndown throttling logic sets a minimum of 4096 tokens as the max\\_token penalty. Setting this lower won't reduce costs but may cut off long tool uses, causing the Claude Code agent loop to fail persistently. Claude Code typically uses less than 4096 output tokens without extended thinking, but may need this headroom for tasks involving significant file creation or Write tool usage.

**\* \*\*`MAX\_THINKING\_TOKENS=1024`\*\*:** This provides space for extended thinking without cutting off tool use responses, while still maintaining focused reasoning chains. This balance helps prevent trajectory changes that aren't always helpful for coding tasks specifically.

## **## IAM configuration**

Create an IAM policy with the required permissions for Claude Code:

**```json**

```
{  
  
  "Version": "2012-10-17",  
  
  "Statement": [  
  
    {  
  
      "Effect": "Allow",  
  
      "Action": [  
  
        "bedrock:InvokeModel",  
  
        "bedrock:InvokeModelWithResponseStream",  
  
        "bedrock:ListInferenceProfiles"  
  
      ],  
  
      "Resource": [  
  
        "arn:aws:bedrock:*:*:inference-profile/*",
```

```

        "arn:aws:bedrock:*:*:application-inference-profile/*"
    ]
}
]
}
...

```

For more restrictive permissions, you can limit the Resource to specific inference profile ARNs.

For details, see [Bedrock IAM documentation](https://docs.aws.amazon.com/bedrock/latest/userguide/security-iam.html).

<Note>

We recommend creating a dedicated AWS account for Claude Code to simplify cost tracking and access control.

</Note>

## ## Troubleshooting

If you encounter region issues:

- \* Check model availability: ``aws bedrock list-inference-profiles --region your-region``
- \* Switch to a supported region: ``export AWS_REGION=us-east-1``
- \* Consider using inference profiles for cross-region access

If you receive an error "on-demand throughput isn't supported":

- \* Specify the model as an [inference profile](https://docs.aws.amazon.com/bedrock/latest/userguide/inference-profiles-support.html) ID

## ## Additional resources

- \* [Bedrock documentation](https://docs.aws.amazon.com/bedrock/)
- \* [Bedrock pricing](https://aws.amazon.com/bedrock/pricing/)
- \* [Bedrock inference profiles](https://docs.aws.amazon.com/bedrock/latest/userguide/inference-profiles-support.html)
- \* [Claude Code on Amazon Bedrock: Quick Setup Guide](https://community.aws/content/2tXkZKrZzlrU0KfH8gST5Dkppq/claude-code-on-amazon-bedrock-quick-setup-guide)

## # Claude Code on Amazon Bedrock

> Learn about configuring Claude Code through Amazon Bedrock, including setup, IAM configuration, and troubleshooting.

## ## Prerequisites

Before configuring Claude Code with Bedrock, ensure you have:

- \* An AWS account with Bedrock access enabled
- \* Access to desired Claude models (e.g., Claude Sonnet 4) in Bedrock
- \* AWS CLI installed and configured (optional - only needed if you don't have another mechanism for getting credentials)
- \* Appropriate IAM permissions

## ## Setup

### ### 1. Enable model access



First, ensure you have access to the required Claude models in your AWS account:

1. Navigate to the [Amazon Bedrock console](https://console.aws.amazon.com/bedrock/)
2. Go to **Model access** in the left navigation
3. Request access to desired Claude models (e.g., Claude Sonnet 4)
4. Wait for approval (usually instant for most regions)

### ### 2. Configure AWS credentials

Claude Code uses the default AWS SDK credential chain. Set up your credentials using one of these methods:

#### **\*\*Option A: AWS CLI configuration\*\***

```
```bash
aws configure
```
```

#### **\*\*Option B: Environment variables (access key)\*\***

```
```bash
export AWS_ACCESS_KEY_ID=your-access-key-id
export AWS_SECRET_ACCESS_KEY=your-secret-access-key
export AWS_SESSION_TOKEN=your-session-token
```
```

#### **\*\*Option C: Environment variables (SSO profile)\*\***

```
```bash
```

```
aws sso login --profile=<your-profile-name>
```

```
export AWS_PROFILE=your-profile-name
```

```
```
```

#### **\*\*Option D: Bedrock API keys\*\***

```
```bash
```

```
export AWS_BEARER_TOKEN_BEDROCK=your-bedrock-api-key
```

```
```
```

Bedrock API keys provide a simpler authentication method without needing full AWS credentials. [Learn more about Bedrock API keys](<https://aws.amazon.com/blogs/machine-learning/accelerate-ai-development-with-amazon-bedrock-api-keys/>).

#### **#### Advanced credential configuration**

Claude Code supports automatic credential refresh for AWS SSO and corporate identity providers. Add these settings to your Claude Code settings file (see [Settings](/en/docs/claude-code/settings) for file locations).

When Claude Code detects that your AWS credentials are expired (either locally based on their timestamp or when Bedrock returns a credential error), it will automatically run your configured `awsAuthRefresh` and/or `awsCredentialExport` commands to obtain new credentials before retrying the request.

#### **##### Example configuration**

```
```json
```

```
{
```

```

"awsAuthRefresh": "aws sso login --profile myprofile",

"env": {

  "AWS_PROFILE": "myprofile"

}

}

...

```

#### ##### Configuration settings explained

**`awsAuthRefresh`**: Use this for commands that modify the `.aws` directory (e.g., updating credentials, SSO cache, or config files). Output is shown to the user (but user input is not supported), making it suitable for browser-based authentication flows where the CLI displays a code to enter in the browser.

**`awsCredentialExport`**: Only use this if you cannot modify `.aws` and must directly return credentials. Output is captured silently (not shown to the user). The command must output JSON in this format:

```

```json
{
  "Credentials": {
    "AccessKeyId": "value",
    "SecretAccessKey": "value",
    "SessionToken": "value"
  }
}

...

```

### ### 3. Configure Claude Code

Set the following environment variables to enable Bedrock:

```

```bash

# Enable Bedrock integration

export CLAUDE_CODE_USE_BEDROCK=1

export AWS_REGION=us-east-1 # or your preferred region


# Optional: Override the region for the small/fast model (Haiku)

export ANTHROPIC_SMALL_FAST_MODEL_AWS_REGION=us-west-2
```

```

When enabling Bedrock for Claude Code, keep the following in mind:

\* `AWS_REGION` is a required environment variable. Claude Code does not read from the `.aws` config file for this setting.

\* When using Bedrock, the `/login` and `/logout` commands are disabled since authentication is handled through AWS credentials.

\* You can use settings files for environment variables like `AWS_PROFILE` that you don't want to leak to other processes. See [\[Settings\]\(/en/docs/claude-code/settings\)](#) for more information.

### ### 4. Model configuration

Claude Code uses these default models for Bedrock:

| Model type       | Default value                                             |
|------------------|-----------------------------------------------------------|
| Primary model    | <code>us.anthropic.claude-3-7-sonnet-20250219-v1:0</code> |
| Small/fast model | <code>us.anthropic.claude-3-5-haiku-20241022-v1:0</code>  |

To customize models, use one of these methods:

```
```bash
```

```
# Using inference profile ID
```

```
export ANTHROPIC_MODEL='us.anthropic.claude-opus-4-1-20250805-v1:0'
```

```
export ANTHROPIC_SMALL_FAST_MODEL='us.anthropic.claude-3-5-haiku-20241022-v1:0'
```

```
# Using application inference profile ARN
```

```
export
```

```
ANTHROPIC_MODEL='arn:aws:bedrock:us-east-2:your-account-id:application-inference-profile/your-model-id'
```

```
# Optional: Disable prompt caching if needed
```

```
export DISABLE_PROMPT_CACHING=1
```

```
```
```

<Note>

[Prompt caching](/en/docs/build-with-claude/prompt-caching) may not be available in all regions

</Note>

### ### 5. Output token configuration

When using Claude Code with Amazon Bedrock, we recommend the following token settings:

```
```bash
```

```
# Recommended output token settings for Bedrock
```

```
export CLAUDE_CODE_MAX_OUTPUT_TOKENS=4096
```

```
export MAX_THINKING_TOKENS=1024
```

```
```
```

**\*\*Why these values:\*\***

**\* \*\*`CLAUDE\_CODE\_MAX\_OUTPUT\_TOKENS=4096`\*\*:** Bedrock's burndown throttling logic sets a minimum of 4096 tokens as the max\\_token penalty. Setting this lower won't reduce costs but may cut off long tool uses, causing the Claude Code agent loop to fail persistently. Claude Code typically uses less than 4096 output tokens without extended thinking, but may need this headroom for tasks involving significant file creation or Write tool usage.

**\* \*\*`MAX\_THINKING\_TOKENS=1024`\*\*:** This provides space for extended thinking without cutting off tool use responses, while still maintaining focused reasoning chains. This balance helps prevent trajectory changes that aren't always helpful for coding tasks specifically.

## **## IAM configuration**

Create an IAM policy with the required permissions for Claude Code:

**```json**

```
{  
  
  "Version": "2012-10-17",  
  
  "Statement": [  
  
    {  
  
      "Effect": "Allow",  
  
      "Action": [  
  
        "bedrock:InvokeModel",  
  
        "bedrock:InvokeModelWithResponseStream",  
  
        "bedrock:ListInferenceProfiles"  
  
      ],  
  
      "Resource": [  
  
        "arn:aws:bedrock:*:*:inference-profile/*",
```

```

        "arn:aws:bedrock:*:*:application-inference-profile/*"
    ]
}
]
}
...

```

For more restrictive permissions, you can limit the Resource to specific inference profile ARNs.

For details, see [Bedrock IAM documentation](https://docs.aws.amazon.com/bedrock/latest/userguide/security-iam.html).

<Note>

We recommend creating a dedicated AWS account for Claude Code to simplify cost tracking and access control.

</Note>

## ## Troubleshooting

If you encounter region issues:

- \* Check model availability: ``aws bedrock list-inference-profiles --region your-region``
- \* Switch to a supported region: ``export AWS_REGION=us-east-1``
- \* Consider using inference profiles for cross-region access

If you receive an error "on-demand throughput isn't supported":

- \* Specify the model as an [inference profile](https://docs.aws.amazon.com/bedrock/latest/userguide/inference-profiles-support.html) ID

## ## Additional resources

- \* [Bedrock documentation](https://docs.aws.amazon.com/bedrock/)
- \* [Bedrock pricing](https://aws.amazon.com/bedrock/pricing/)
- \* [Bedrock inference profiles](https://docs.aws.amazon.com/bedrock/latest/userguide/inference-profiles-support.html)
- \* [Claude Code on Amazon Bedrock: Quick Setup Guide](https://community.aws/content/2tXkZKrZzlrU0KfH8gST5Dkppq/claude-code-on-amazon-bedrock-quick-setup-guide)

## # Claude Code on Amazon Bedrock

> Learn about configuring Claude Code through Amazon Bedrock, including setup, IAM configuration, and troubleshooting.

## ## Prerequisites

Before configuring Claude Code with Bedrock, ensure you have:

- \* An AWS account with Bedrock access enabled
- \* Access to desired Claude models (e.g., Claude Sonnet 4) in Bedrock
- \* AWS CLI installed and configured (optional - only needed if you don't have another mechanism for getting credentials)
- \* Appropriate IAM permissions

## ## Setup

### ### 1. Enable model access



First, ensure you have access to the required Claude models in your AWS account:

1. Navigate to the [Amazon Bedrock console](https://console.aws.amazon.com/bedrock/)
2. Go to **Model access** in the left navigation
3. Request access to desired Claude models (e.g., Claude Sonnet 4)
4. Wait for approval (usually instant for most regions)

### ### 2. Configure AWS credentials

Claude Code uses the default AWS SDK credential chain. Set up your credentials using one of these methods:

#### **\*\*Option A: AWS CLI configuration\*\***

```
```bash
aws configure
```
```

#### **\*\*Option B: Environment variables (access key)\*\***

```
```bash
export AWS_ACCESS_KEY_ID=your-access-key-id
export AWS_SECRET_ACCESS_KEY=your-secret-access-key
export AWS_SESSION_TOKEN=your-session-token
```
```

#### **\*\*Option C: Environment variables (SSO profile)\*\***

```
```bash
```

```
aws sso login --profile=<your-profile-name>
```

```
export AWS_PROFILE=your-profile-name
```

```
```
```

#### **\*\*Option D: Bedrock API keys\*\***

```
```bash
```

```
export AWS_BEARER_TOKEN_BEDROCK=your-bedrock-api-key
```

```
```
```

Bedrock API keys provide a simpler authentication method without needing full AWS credentials. [Learn more about Bedrock API keys](<https://aws.amazon.com/blogs/machine-learning/accelerate-ai-development-with-amazon-bedrock-api-keys/>).

#### **#### Advanced credential configuration**

Claude Code supports automatic credential refresh for AWS SSO and corporate identity providers. Add these settings to your Claude Code settings file (see [Settings](/en/docs/claude-code/settings) for file locations).

When Claude Code detects that your AWS credentials are expired (either locally based on their timestamp or when Bedrock returns a credential error), it will automatically run your configured `awsAuthRefresh` and/or `awsCredentialExport` commands to obtain new credentials before retrying the request.

#### **##### Example configuration**

```
```json
```

```
{
```

```
"awsAuthRefresh": "aws sso login --profile myprofile",

"env": {

  "AWS_PROFILE": "myprofile"

}

}

...
```

#### ##### Configuration settings explained

**`awsAuthRefresh`**: Use this for commands that modify the `.aws` directory (e.g., updating credentials, SSO cache, or config files). Output is shown to the user (but user input is not supported), making it suitable for browser-based authentication flows where the CLI displays a code to enter in the browser.

**`awsCredentialExport`**: Only use this if you cannot modify `.aws` and must directly return credentials. Output is captured silently (not shown to the user). The command must output JSON in this format:

```
```json

{

  "Credentials": {

    "AccessKeyId": "value",

    "SecretAccessKey": "value",

    "SessionToken": "value"

  }

}

...

```
```

### ### 3. Configure Claude Code

Set the following environment variables to enable Bedrock:

```

```bash

# Enable Bedrock integration

export CLAUDE_CODE_USE_BEDROCK=1

export AWS_REGION=us-east-1 # or your preferred region


# Optional: Override the region for the small/fast model (Haiku)

export ANTHROPIC_SMALL_FAST_MODEL_AWS_REGION=us-west-2

```

```

When enabling Bedrock for Claude Code, keep the following in mind:

\* `AWS_REGION` is a required environment variable. Claude Code does not read from the `.aws` config file for this setting.

\* When using Bedrock, the `/login` and `/logout` commands are disabled since authentication is handled through AWS credentials.

\* You can use settings files for environment variables like `AWS_PROFILE` that you don't want to leak to other processes. See [\[Settings\]\(/en/docs/claude-code/settings\)](#) for more information.

### ### 4. Model configuration

Claude Code uses these default models for Bedrock:

| Model type       | Default value                                             |
|------------------|-----------------------------------------------------------|
| Primary model    | <code>us.anthropic.claude-3-7-sonnet-20250219-v1:0</code> |
| Small/fast model | <code>us.anthropic.claude-3-5-haiku-20241022-v1:0</code>  |

To customize models, use one of these methods:

```
```bash
```

```
# Using inference profile ID
```

```
export ANTHROPIC_MODEL='us.anthropic.claude-opus-4-1-20250805-v1:0'
```

```
export ANTHROPIC_SMALL_FAST_MODEL='us.anthropic.claude-3-5-haiku-20241022-v1:0'
```

```
# Using application inference profile ARN
```

```
export
```

```
ANTHROPIC_MODEL='arn:aws:bedrock:us-east-2:your-account-id:application-inference-profile/your-model-id'
```

```
# Optional: Disable prompt caching if needed
```

```
export DISABLE_PROMPT_CACHING=1
```

```
```
```

<Note>

[Prompt caching](/en/docs/build-with-claude/prompt-caching) may not be available in all regions

</Note>

### ### 5. Output token configuration

When using Claude Code with Amazon Bedrock, we recommend the following token settings:

```
```bash
```

```
# Recommended output token settings for Bedrock
```

```
export CLAUDE_CODE_MAX_OUTPUT_TOKENS=4096
```

```
export MAX_THINKING_TOKENS=1024
```

```
```
```

**\*\*Why these values:\*\***

**\* \*\*`CLAUDE\_CODE\_MAX\_OUTPUT\_TOKENS=4096`\*\*:** Bedrock's burndown throttling logic sets a minimum of 4096 tokens as the max\\_token penalty. Setting this lower won't reduce costs but may cut off long tool uses, causing the Claude Code agent loop to fail persistently. Claude Code typically uses less than 4096 output tokens without extended thinking, but may need this headroom for tasks involving significant file creation or Write tool usage.

**\* \*\*`MAX\_THINKING\_TOKENS=1024`\*\*:** This provides space for extended thinking without cutting off tool use responses, while still maintaining focused reasoning chains. This balance helps prevent trajectory changes that aren't always helpful for coding tasks specifically.

## **## IAM configuration**

Create an IAM policy with the required permissions for Claude Code:

```
```json
```

```
{  
  
  "Version": "2012-10-17",  
  
  "Statement": [  
  
    {  
  
      "Effect": "Allow",  
  
      "Action": [  
  
        "bedrock:InvokeModel",  
  
        "bedrock:InvokeModelWithResponseStream",  
  
        "bedrock:ListInferenceProfiles"  
  
      ],  
  
      "Resource": [  
  
        "arn:aws:bedrock:*:*:inference-profile/*",
```

```

        "arn:aws:bedrock:*:*:application-inference-profile/*"
    ]
}
]
}
...

```

For more restrictive permissions, you can limit the Resource to specific inference profile ARNs.

For details, see [Bedrock IAM documentation](https://docs.aws.amazon.com/bedrock/latest/userguide/security-iam.html).

<Note>

We recommend creating a dedicated AWS account for Claude Code to simplify cost tracking and access control.

</Note>

## ## Troubleshooting

If you encounter region issues:

- \* Check model availability: ``aws bedrock list-inference-profiles --region your-region``
- \* Switch to a supported region: ``export AWS_REGION=us-east-1``
- \* Consider using inference profiles for cross-region access

If you receive an error "on-demand throughput isn't supported":

- \* Specify the model as an [inference profile](https://docs.aws.amazon.com/bedrock/latest/userguide/inference-profiles-support.html) ID

## ## Additional resources

- \* [Bedrock documentation](https://docs.aws.amazon.com/bedrock/)
- \* [Bedrock pricing](https://aws.amazon.com/bedrock/pricing/)
- \* [Bedrock inference profiles](https://docs.aws.amazon.com/bedrock/latest/userguide/inference-profiles-support.html)
- \* [Claude Code on Amazon Bedrock: Quick Setup Guide](https://community.aws/content/2tXkZKrZzlrU0KfH8gST5Dkppq/claude-code-on-amazon-bedrock-quick-setup-guide)

## # Claude Code on Amazon Bedrock

> Learn about configuring Claude Code through Amazon Bedrock, including setup, IAM configuration, and troubleshooting.

## ## Prerequisites

Before configuring Claude Code with Bedrock, ensure you have:

- \* An AWS account with Bedrock access enabled
- \* Access to desired Claude models (e.g., Claude Sonnet 4) in Bedrock
- \* AWS CLI installed and configured (optional - only needed if you don't have another mechanism for getting credentials)
- \* Appropriate IAM permissions

## ## Setup

### ### 1. Enable model access



First, ensure you have access to the required Claude models in your AWS account:

1. Navigate to the [Amazon Bedrock console](https://console.aws.amazon.com/bedrock/)
2. Go to **Model access** in the left navigation
3. Request access to desired Claude models (e.g., Claude Sonnet 4)
4. Wait for approval (usually instant for most regions)

### ### 2. Configure AWS credentials

Claude Code uses the default AWS SDK credential chain. Set up your credentials using one of these methods:

#### **\*\*Option A: AWS CLI configuration\*\***

```
```bash
aws configure
```
```

#### **\*\*Option B: Environment variables (access key)\*\***

```
```bash
export AWS_ACCESS_KEY_ID=your-access-key-id
export AWS_SECRET_ACCESS_KEY=your-secret-access-key
export AWS_SESSION_TOKEN=your-session-token
```
```

#### **\*\*Option C: Environment variables (SSO profile)\*\***

```
```bash
```

```
aws sso login --profile=<your-profile-name>
```

```
export AWS_PROFILE=your-profile-name
```

```
```
```

#### **\*\*Option D: Bedrock API keys\*\***

```
```bash
```

```
export AWS_BEARER_TOKEN_BEDROCK=your-bedrock-api-key
```

```
```
```

Bedrock API keys provide a simpler authentication method without needing full AWS credentials. [Learn more about Bedrock API keys](<https://aws.amazon.com/blogs/machine-learning/accelerate-ai-development-with-amazon-bedrock-api-keys/>).

#### **#### Advanced credential configuration**

Claude Code supports automatic credential refresh for AWS SSO and corporate identity providers. Add these settings to your Claude Code settings file (see [Settings](/en/docs/claude-code/settings) for file locations).

When Claude Code detects that your AWS credentials are expired (either locally based on their timestamp or when Bedrock returns a credential error), it will automatically run your configured `awsAuthRefresh` and/or `awsCredentialExport` commands to obtain new credentials before retrying the request.

#### **##### Example configuration**

```
```json
```

```
{
```

```
"awsAuthRefresh": "aws sso login --profile myprofile",

"env": {

  "AWS_PROFILE": "myprofile"

}

}

...
```

#### ##### Configuration settings explained

**`awsAuthRefresh`**: Use this for commands that modify the `.aws` directory (e.g., updating credentials, SSO cache, or config files). Output is shown to the user (but user input is not supported), making it suitable for browser-based authentication flows where the CLI displays a code to enter in the browser.

**`awsCredentialExport`**: Only use this if you cannot modify `.aws` and must directly return credentials. Output is captured silently (not shown to the user). The command must output JSON in this format:

```
```json

{

  "Credentials": {

    "AccessKeyId": "value",

    "SecretAccessKey": "value",

    "SessionToken": "value"

  }

}

...

```
```

### ### 3. Configure Claude Code

Set the following environment variables to enable Bedrock:

```

```bash

# Enable Bedrock integration

export CLAUDE_CODE_USE_BEDROCK=1

export AWS_REGION=us-east-1 # or your preferred region


# Optional: Override the region for the small/fast model (Haiku)

export ANTHROPIC_SMALL_FAST_MODEL_AWS_REGION=us-west-2
```

```

When enabling Bedrock for Claude Code, keep the following in mind:

\* `AWS_REGION` is a required environment variable. Claude Code does not read from the `.aws` config file for this setting.

\* When using Bedrock, the `/login` and `/logout` commands are disabled since authentication is handled through AWS credentials.

\* You can use settings files for environment variables like `AWS_PROFILE` that you don't want to leak to other processes. See [\[Settings\]\(/en/docs/claude-code/settings\)](#) for more information.

### ### 4. Model configuration

Claude Code uses these default models for Bedrock:

| Model type       | Default value                                             |
|------------------|-----------------------------------------------------------|
| Primary model    | <code>us.anthropic.claude-3-7-sonnet-20250219-v1:0</code> |
| Small/fast model | <code>us.anthropic.claude-3-5-haiku-20241022-v1:0</code>  |

To customize models, use one of these methods:

```
```bash
```

```
# Using inference profile ID
```

```
export ANTHROPIC_MODEL='us.anthropic.claude-opus-4-1-20250805-v1:0'
```

```
export ANTHROPIC_SMALL_FAST_MODEL='us.anthropic.claude-3-5-haiku-20241022-v1:0'
```

```
# Using application inference profile ARN
```

```
export
```

```
ANTHROPIC_MODEL='arn:aws:bedrock:us-east-2:your-account-id:application-inference-profile/your-model-id'
```

```
# Optional: Disable prompt caching if needed
```

```
export DISABLE_PROMPT_CACHING=1
```

```
```
```

<Note>

[Prompt caching](/en/docs/build-with-claude/prompt-caching) may not be available in all regions

</Note>

### ### 5. Output token configuration

When using Claude Code with Amazon Bedrock, we recommend the following token settings:

```
```bash
```

```
# Recommended output token settings for Bedrock
```

```
export CLAUDE_CODE_MAX_OUTPUT_TOKENS=4096
```

```
export MAX_THINKING_TOKENS=1024
```

```
```
```

**\*\*Why these values:\*\***

**\* \*\*`CLAUDE\_CODE\_MAX\_OUTPUT\_TOKENS=4096`\*\*:** Bedrock's burndown throttling logic sets a minimum of 4096 tokens as the max\\_token penalty. Setting this lower won't reduce costs but may cut off long tool uses, causing the Claude Code agent loop to fail persistently. Claude Code typically uses less than 4096 output tokens without extended thinking, but may need this headroom for tasks involving significant file creation or Write tool usage.

**\* \*\*`MAX\_THINKING\_TOKENS=1024`\*\*:** This provides space for extended thinking without cutting off tool use responses, while still maintaining focused reasoning chains. This balance helps prevent trajectory changes that aren't always helpful for coding tasks specifically.

## **## IAM configuration**

Create an IAM policy with the required permissions for Claude Code:

```
```json
```

```
{  
  
  "Version": "2012-10-17",  
  
  "Statement": [  
  
    {  
  
      "Effect": "Allow",  
  
      "Action": [  
  
        "bedrock:InvokeModel",  
  
        "bedrock:InvokeModelWithResponseStream",  
  
        "bedrock:ListInferenceProfiles"  
  
      ],  
  
      "Resource": [  
  
        "arn:aws:bedrock:*:*:inference-profile/*",
```

```

        "arn:aws:bedrock:*:*:application-inference-profile/*"
    ]
}
]
}
...

```

For more restrictive permissions, you can limit the Resource to specific inference profile ARNs.

For details, see [Bedrock IAM documentation](https://docs.aws.amazon.com/bedrock/latest/userguide/security-iam.html).

<Note>

We recommend creating a dedicated AWS account for Claude Code to simplify cost tracking and access control.

</Note>

## ## Troubleshooting

If you encounter region issues:

- \* Check model availability: ``aws bedrock list-inference-profiles --region your-region``
- \* Switch to a supported region: ``export AWS_REGION=us-east-1``
- \* Consider using inference profiles for cross-region access

If you receive an error "on-demand throughput isn't supported":

- \* Specify the model as an [inference profile](https://docs.aws.amazon.com/bedrock/latest/userguide/inference-profiles-support.html) ID

## ## Additional resources

- \* [Bedrock documentation](https://docs.aws.amazon.com/bedrock/)
- \* [Bedrock pricing](https://aws.amazon.com/bedrock/pricing/)
- \* [Bedrock inference profiles](https://docs.aws.amazon.com/bedrock/latest/userguide/inference-profiles-support.html)
- \* [Claude Code on Amazon Bedrock: Quick Setup Guide](https://community.aws/content/2tXkZKrZzlrU0KfH8gST5Dkppq/claude-code-on-amazon-bedrock-quick-setup-guide)

## # Claude Code on Amazon Bedrock

> Learn about configuring Claude Code through Amazon Bedrock, including setup, IAM configuration, and troubleshooting.

## ## Prerequisites

Before configuring Claude Code with Bedrock, ensure you have:

- \* An AWS account with Bedrock access enabled
- \* Access to desired Claude models (e.g., Claude Sonnet 4) in Bedrock
- \* AWS CLI installed and configured (optional - only needed if you don't have another mechanism for getting credentials)
- \* Appropriate IAM permissions

## ## Setup

### ### 1. Enable model access



First, ensure you have access to the required Claude models in your AWS account:

1. Navigate to the [Amazon Bedrock console](https://console.aws.amazon.com/bedrock/)
2. Go to **Model access** in the left navigation
3. Request access to desired Claude models (e.g., Claude Sonnet 4)
4. Wait for approval (usually instant for most regions)

### ### 2. Configure AWS credentials

Claude Code uses the default AWS SDK credential chain. Set up your credentials using one of these methods:

#### **\*\*Option A: AWS CLI configuration\*\***

```
```bash
aws configure
```
```

#### **\*\*Option B: Environment variables (access key)\*\***

```
```bash
export AWS_ACCESS_KEY_ID=your-access-key-id
export AWS_SECRET_ACCESS_KEY=your-secret-access-key
export AWS_SESSION_TOKEN=your-session-token
```
```

#### **\*\*Option C: Environment variables (SSO profile)\*\***

```
```bash
```

```
aws sso login --profile=<your-profile-name>
```

```
export AWS_PROFILE=your-profile-name
```

```
```
```

#### **\*\*Option D: Bedrock API keys\*\***

```
```bash
```

```
export AWS_BEARER_TOKEN_BEDROCK=your-bedrock-api-key
```

```
```
```

Bedrock API keys provide a simpler authentication method without needing full AWS credentials. [Learn more about Bedrock API keys](<https://aws.amazon.com/blogs/machine-learning/accelerate-ai-development-with-amazon-bedrock-api-keys/>).

#### **#### Advanced credential configuration**

Claude Code supports automatic credential refresh for AWS SSO and corporate identity providers. Add these settings to your Claude Code settings file (see [Settings](/en/docs/claude-code/settings) for file locations).

When Claude Code detects that your AWS credentials are expired (either locally based on their timestamp or when Bedrock returns a credential error), it will automatically run your configured `awsAuthRefresh` and/or `awsCredentialExport` commands to obtain new credentials before retrying the request.

#### **##### Example configuration**

```
```json
```

```
{
```

```
"awsAuthRefresh": "aws sso login --profile myprofile",

"env": {

  "AWS_PROFILE": "myprofile"

}

}

...
```

#### ##### Configuration settings explained

**`awsAuthRefresh`**: Use this for commands that modify the `.aws` directory (e.g., updating credentials, SSO cache, or config files). Output is shown to the user (but user input is not supported), making it suitable for browser-based authentication flows where the CLI displays a code to enter in the browser.

**`awsCredentialExport`**: Only use this if you cannot modify `.aws` and must directly return credentials. Output is captured silently (not shown to the user). The command must output JSON in this format:

```
```json

{

  "Credentials": {

    "AccessKeyId": "value",

    "SecretAccessKey": "value",

    "SessionToken": "value"

  }

}

...

```
```

### ### 3. Configure Claude Code

Set the following environment variables to enable Bedrock:

```

```bash

# Enable Bedrock integration

export CLAUDE_CODE_USE_BEDROCK=1

export AWS_REGION=us-east-1 # or your preferred region


# Optional: Override the region for the small/fast model (Haiku)

export ANTHROPIC_SMALL_FAST_MODEL_AWS_REGION=us-west-2
```

```

When enabling Bedrock for Claude Code, keep the following in mind:

\* `AWS_REGION` is a required environment variable. Claude Code does not read from the `.aws` config file for this setting.

\* When using Bedrock, the `/login` and `/logout` commands are disabled since authentication is handled through AWS credentials.

\* You can use settings files for environment variables like `AWS_PROFILE` that you don't want to leak to other processes. See [\[Settings\]\(/en/docs/claude-code/settings\)](#) for more information.

### ### 4. Model configuration

Claude Code uses these default models for Bedrock:

| Model type       | Default value                                             |
|------------------|-----------------------------------------------------------|
| Primary model    | <code>us.anthropic.claude-3-7-sonnet-20250219-v1:0</code> |
| Small/fast model | <code>us.anthropic.claude-3-5-haiku-20241022-v1:0</code>  |

To customize models, use one of these methods:

```
```bash
```

```
# Using inference profile ID
```

```
export ANTHROPIC_MODEL='us.anthropic.claude-opus-4-1-20250805-v1:0'
```

```
export ANTHROPIC_SMALL_FAST_MODEL='us.anthropic.claude-3-5-haiku-20241022-v1:0'
```

```
# Using application inference profile ARN
```

```
export
```

```
ANTHROPIC_MODEL='arn:aws:bedrock:us-east-2:your-account-id:application-inference-profile/your-model-id'
```

```
# Optional: Disable prompt caching if needed
```

```
export DISABLE_PROMPT_CACHING=1
```

```
```
```

<Note>

[Prompt caching](/en/docs/build-with-claude/prompt-caching) may not be available in all regions

</Note>

### ### 5. Output token configuration

When using Claude Code with Amazon Bedrock, we recommend the following token settings:

```
```bash
```

```
# Recommended output token settings for Bedrock
```

```
export CLAUDE_CODE_MAX_OUTPUT_TOKENS=4096
```

```
export MAX_THINKING_TOKENS=1024
```

```
```
```

**\*\*Why these values:\*\***

**\* \*\*`CLAUDE\_CODE\_MAX\_OUTPUT\_TOKENS=4096`\*\*:** Bedrock's burndown throttling logic sets a minimum of 4096 tokens as the max\\_token penalty. Setting this lower won't reduce costs but may cut off long tool uses, causing the Claude Code agent loop to fail persistently. Claude Code typically uses less than 4096 output tokens without extended thinking, but may need this headroom for tasks involving significant file creation or Write tool usage.

**\* \*\*`MAX\_THINKING\_TOKENS=1024`\*\*:** This provides space for extended thinking without cutting off tool use responses, while still maintaining focused reasoning chains. This balance helps prevent trajectory changes that aren't always helpful for coding tasks specifically.

## **## IAM configuration**

Create an IAM policy with the required permissions for Claude Code:

**```json**

```
{  
  
  "Version": "2012-10-17",  
  
  "Statement": [  
  
    {  
  
      "Effect": "Allow",  
  
      "Action": [  
  
        "bedrock:InvokeModel",  
  
        "bedrock:InvokeModelWithResponseStream",  
  
        "bedrock:ListInferenceProfiles"  
  
      ],  
  
      "Resource": [  
  
        "arn:aws:bedrock:*:*:inference-profile/*",
```

```

        "arn:aws:bedrock:*:*:application-inference-profile/*"
    ]
}
]
}
...

```

For more restrictive permissions, you can limit the Resource to specific inference profile ARNs.

For details, see [Bedrock IAM documentation](https://docs.aws.amazon.com/bedrock/latest/userguide/security-iam.html).

<Note>

We recommend creating a dedicated AWS account for Claude Code to simplify cost tracking and access control.

</Note>

## ## Troubleshooting

If you encounter region issues:

- \* Check model availability: ``aws bedrock list-inference-profiles --region your-region``
- \* Switch to a supported region: ``export AWS_REGION=us-east-1``
- \* Consider using inference profiles for cross-region access

If you receive an error "on-demand throughput isn't supported":

- \* Specify the model as an [inference profile](https://docs.aws.amazon.com/bedrock/latest/userguide/inference-profiles-support.html) ID

## ## Additional resources

- \* [Bedrock documentation](https://docs.aws.amazon.com/bedrock/)
- \* [Bedrock pricing](https://aws.amazon.com/bedrock/pricing/)
- \* [Bedrock inference profiles](https://docs.aws.amazon.com/bedrock/latest/userguide/inference-profiles-support.html)
- \* [Claude Code on Amazon Bedrock: Quick Setup Guide](https://community.aws/content/2tXkZKrZzlrU0KfH8gST5Dkppq/claude-code-on-amazon-bedrock-quick-setup-guide)

## # Claude Code on Amazon Bedrock

> Learn about configuring Claude Code through Amazon Bedrock, including setup, IAM configuration, and troubleshooting.

## ## Prerequisites

Before configuring Claude Code with Bedrock, ensure you have:

- \* An AWS account with Bedrock access enabled
- \* Access to desired Claude models (e.g., Claude Sonnet 4) in Bedrock
- \* AWS CLI installed and configured (optional - only needed if you don't have another mechanism for getting credentials)
- \* Appropriate IAM permissions

## ## Setup

### ### 1. Enable model access



First, ensure you have access to the required Claude models in your AWS account:

1. Navigate to the [Amazon Bedrock console](https://console.aws.amazon.com/bedrock/)
2. Go to **Model access** in the left navigation
3. Request access to desired Claude models (e.g., Claude Sonnet 4)
4. Wait for approval (usually instant for most regions)

### ### 2. Configure AWS credentials

Claude Code uses the default AWS SDK credential chain. Set up your credentials using one of these methods:

#### **\*\*Option A: AWS CLI configuration\*\***

```
```bash
aws configure
```
```

#### **\*\*Option B: Environment variables (access key)\*\***

```
```bash
export AWS_ACCESS_KEY_ID=your-access-key-id
export AWS_SECRET_ACCESS_KEY=your-secret-access-key
export AWS_SESSION_TOKEN=your-session-token
```
```

#### **\*\*Option C: Environment variables (SSO profile)\*\***

```
```bash
```

```
aws sso login --profile=<your-profile-name>
```

```
export AWS_PROFILE=your-profile-name
```

```
```
```

#### **\*\*Option D: Bedrock API keys\*\***

```
```bash
```

```
export AWS_BEARER_TOKEN_BEDROCK=your-bedrock-api-key
```

```
```
```

Bedrock API keys provide a simpler authentication method without needing full AWS credentials. [Learn more about Bedrock API keys](<https://aws.amazon.com/blogs/machine-learning/accelerate-ai-development-with-amazon-bedrock-api-keys/>).

#### **#### Advanced credential configuration**

Claude Code supports automatic credential refresh for AWS SSO and corporate identity providers. Add these settings to your Claude Code settings file (see [Settings](/en/docs/claude-code/settings) for file locations).

When Claude Code detects that your AWS credentials are expired (either locally based on their timestamp or when Bedrock returns a credential error), it will automatically run your configured `awsAuthRefresh` and/or `awsCredentialExport` commands to obtain new credentials before retrying the request.

#### **##### Example configuration**

```
```json
```

```
{
```

```
"awsAuthRefresh": "aws sso login --profile myprofile",

"env": {

  "AWS_PROFILE": "myprofile"

}

}

...
```

#### ##### Configuration settings explained

**`awsAuthRefresh`**: Use this for commands that modify the `.aws` directory (e.g., updating credentials, SSO cache, or config files). Output is shown to the user (but user input is not supported), making it suitable for browser-based authentication flows where the CLI displays a code to enter in the browser.

**`awsCredentialExport`**: Only use this if you cannot modify `.aws` and must directly return credentials. Output is captured silently (not shown to the user). The command must output JSON in this format:

```
```json

{

  "Credentials": {

    "AccessKeyId": "value",

    "SecretAccessKey": "value",

    "SessionToken": "value"

  }

}

...

```
```

### ### 3. Configure Claude Code

Set the following environment variables to enable Bedrock:

```

```bash

# Enable Bedrock integration

export CLAUDE_CODE_USE_BEDROCK=1

export AWS_REGION=us-east-1 # or your preferred region


# Optional: Override the region for the small/fast model (Haiku)

export ANTHROPIC_SMALL_FAST_MODEL_AWS_REGION=us-west-2
```

```

When enabling Bedrock for Claude Code, keep the following in mind:

\* `AWS_REGION` is a required environment variable. Claude Code does not read from the `.aws` config file for this setting.

\* When using Bedrock, the `/login` and `/logout` commands are disabled since authentication is handled through AWS credentials.

\* You can use settings files for environment variables like `AWS_PROFILE` that you don't want to leak to other processes. See [\[Settings\]\(/en/docs/claude-code/settings\)](#) for more information.

### ### 4. Model configuration

Claude Code uses these default models for Bedrock:

| Model type       | Default value                                             |
|------------------|-----------------------------------------------------------|
| Primary model    | <code>us.anthropic.claude-3-7-sonnet-20250219-v1:0</code> |
| Small/fast model | <code>us.anthropic.claude-3-5-haiku-20241022-v1:0</code>  |

To customize models, use one of these methods:

```
```bash
```

```
# Using inference profile ID
```

```
export ANTHROPIC_MODEL='us.anthropic.claude-opus-4-1-20250805-v1:0'
```

```
export ANTHROPIC_SMALL_FAST_MODEL='us.anthropic.claude-3-5-haiku-20241022-v1:0'
```

```
# Using application inference profile ARN
```

```
export
```

```
ANTHROPIC_MODEL='arn:aws:bedrock:us-east-2:your-account-id:application-inference-profile/your-model-id'
```

```
# Optional: Disable prompt caching if needed
```

```
export DISABLE_PROMPT_CACHING=1
```

```
```
```

<Note>

[Prompt caching](/en/docs/build-with-claude/prompt-caching) may not be available in all regions

</Note>

### ### 5. Output token configuration

When using Claude Code with Amazon Bedrock, we recommend the following token settings:

```
```bash
```

```
# Recommended output token settings for Bedrock
```

```
export CLAUDE_CODE_MAX_OUTPUT_TOKENS=4096
```

```
export MAX_THINKING_TOKENS=1024
```

```
```
```

**\*\*Why these values:\*\***

**\* \*\*`CLAUDE\_CODE\_MAX\_OUTPUT\_TOKENS=4096`\*\*:** Bedrock's burndown throttling logic sets a minimum of 4096 tokens as the max\\_token penalty. Setting this lower won't reduce costs but may cut off long tool uses, causing the Claude Code agent loop to fail persistently. Claude Code typically uses less than 4096 output tokens without extended thinking, but may need this headroom for tasks involving significant file creation or Write tool usage.

**\* \*\*`MAX\_THINKING\_TOKENS=1024`\*\*:** This provides space for extended thinking without cutting off tool use responses, while still maintaining focused reasoning chains. This balance helps prevent trajectory changes that aren't always helpful for coding tasks specifically.

## **## IAM configuration**

Create an IAM policy with the required permissions for Claude Code:

```
```json
```

```
{  
  
  "Version": "2012-10-17",  
  
  "Statement": [  
  
    {  
  
      "Effect": "Allow",  
  
      "Action": [  
  
        "bedrock:InvokeModel",  
  
        "bedrock:InvokeModelWithResponseStream",  
  
        "bedrock:ListInferenceProfiles"  
  
      ],  
  
      "Resource": [  
  
        "arn:aws:bedrock:*:*:inference-profile/*",
```

```

        "arn:aws:bedrock:*:*:application-inference-profile/*"
    ]
}
]
}
...

```

For more restrictive permissions, you can limit the Resource to specific inference profile ARNs.

For details, see [Bedrock IAM documentation](https://docs.aws.amazon.com/bedrock/latest/userguide/security-iam.html).

<Note>

We recommend creating a dedicated AWS account for Claude Code to simplify cost tracking and access control.

</Note>

## ## Troubleshooting

If you encounter region issues:

- \* Check model availability: ``aws bedrock list-inference-profiles --region your-region``
- \* Switch to a supported region: ``export AWS_REGION=us-east-1``
- \* Consider using inference profiles for cross-region access

If you receive an error "on-demand throughput isn't supported":

- \* Specify the model as an [inference profile](https://docs.aws.amazon.com/bedrock/latest/userguide/inference-profiles-support.html) ID

## ## Additional resources

\* [Bedrock documentation](https://docs.aws.amazon.com/bedrock/)

\* [Bedrock pricing](https://aws.amazon.com/bedrock/pricing/)

\* [Bedrock inference profiles](https://docs.aws.amazon.com/bedrock/latest/userguide/inference-profiles-support.html)

\* [Claude Code on Amazon Bedrock: Quick Setup Guide](https://community.aws/content/2tXkZKrZzlrU0KfH8gST5Dkppq/claude-code-on-amazon-bedrock-quick-setup-guide)

## # Identity and Access Management

> Learn how to configure user authentication, authorization, and access controls for Claude Code in your organization.

## ## Authentication methods

Setting up Claude Code requires access to Anthropic models. For teams, you can set up Claude Code access in one of three ways:

\* Anthropic API via the Anthropic Console

\* Amazon Bedrock

\* Google Vertex AI

### ### Anthropic API authentication

**\*\*To set up Claude Code access for your team via Anthropic API:\*\***

1. Use your existing Anthropic Console account or create a new Anthropic Console account



2. You can add users through either method below:

- \* Bulk invite users from within the Console (Console -> Settings -> Members -> Invite)

- \* [Set up SSO](https://support.anthropic.com/en/articles/10280258-setting-up-single-sign-on-on-the-api-console)

3. When inviting users, they need one of the following roles:

- \* "Claude Code" role means users can only create Claude Code API keys

- \* "Developer" role means users can create any kind of API key

4. Each invited user needs to complete these steps:

- \* Accept the Console invite

- \* [Check system requirements](/en/docs/claude-code/setup#system-requirements)

- \* [Install Claude Code](/en/docs/claude-code/setup#installation)

- \* Login with Console account credentials

### ### Cloud provider authentication

**\*\*To set up Claude Code access for your team via Bedrock or Vertex:\*\***

1. Follow the [Bedrock docs](/en/docs/claude-code/amazon-bedrock) or [Vertex docs](/en/docs/claude-code/google-vertex-ai)

2. Distribute the environment variables and instructions for generating cloud credentials to your users. Read more about how to [manage configuration here](/en/docs/claude-code/settings).

3. Users can [install Claude Code](/en/docs/claude-code/setup#installation)

### ## Access control and permissions

We support fine-grained permissions so that you're able to specify exactly what the agent is allowed to do (e.g. run tests, run linter) and what it is not allowed to do (e.g. update cloud infrastructure). These permission settings can be checked into version control and distributed to all developers in your organization, as well as customized by individual developers.

### ### Permission system

Claude Code uses a tiered permission system to balance power and safety:

Tool Type	Example	Approval Required	"Yes, don't ask again" Behavior
Read-only	File reads, LS, Grep	No	N/A
Bash Commands	Shell execution	Yes	Permanently per project directory and command
File Modification	Edit/write files	Yes	Until session end

### ### Configuring permissions

You can view & manage Claude Code's tool permissions with ``/permissions``. This UI lists all permission rules and the settings.json file they are sourced from.

**\*\*Allow\*\*** rules will allow Claude Code to use the specified tool without further manual approval.

**\*\*Ask\*\*** rules will ask the user for confirmation whenever Claude Code tries to use the specified tool. Ask rules take precedence over allow rules.

**\*\*Deny\*\*** rules will prevent Claude Code from using the specified tool. Deny rules take precedence over allow and ask rules.

**\*\*Additional directories\*\*** extend Claude's file access to directories beyond the initial working directory.

**\*\*Default mode\*\*** controls Claude's permission behavior when encountering new requests.

Permission rules use the format: ``Tool`` or ``Tool(optional-specifier)``

A rule that is just the tool name matches any use of that tool. For example, adding ``Bash`` to the list of allow rules would allow Claude Code to use the Bash tool without requiring user approval.

### #### Permission modes

Claude Code supports several permission modes that can be set as the `defaultMode` in [settings files](/en/docs/claude-code/settings#settings-files):

Mode	Description
-----	-----
<code>`default`</code>	Standard behavior - prompts for permission on first use of each tool
<code>`acceptEdits`</code>	Automatically accepts file edit permissions for the session
<code>`plan`</code>	Plan Mode - Claude can analyze but not modify files or execute commands
<code>`bypassPermissions`</code>	Skips all permission prompts (requires safe environment - see warning below)

#### #### Working directories

By default, Claude has access to files in the directory where it was launched. You can extend this access:

**\*\*During startup\*\***: Use `--add-dir <path>` CLI argument

**\*\*During session\*\***: Use `/add-dir` slash command

**\*\*Persistent configuration\*\***: Add to `additionalDirectories` in [settings files](/en/docs/claude-code/settings#settings-files)

Files in additional directories follow the same permission rules as the original working directory - they become readable without prompts, and file editing permissions follow the current permission mode.

#### #### Tool-specific permission rules

Some tools support more fine-grained permission controls:

**\*\*Bash\*\***

- \* ``Bash(npm run build)`` Matches the exact Bash command ``npm run build``
- \* ``Bash(npm run test:*)`` Matches Bash commands starting with ``npm run test``.

<Tip>

Claude Code is aware of shell operators (like ``&&``) so a prefix match rule like ``Bash(safe-cmd:*)`` won't give it permission to run the command ``safe-cmd && other-cmd``

</Tip>

**\*\*Read & Edit\*\***

``Edit`` rules apply to all built-in tools that edit files. Claude will make a best-effort attempt to apply ``Read`` rules to all built-in tools that read files like Grep, Glob, and LS.

Read & Edit rules both follow the [gitignore](https://git-scm.com/docs/gitignore) specification with four distinct pattern types:

Pattern	Meaning	Example	Matches
-----	-----	-----	-----
<code>`//path`</code> <code>`/Users/alice/secrets/**`</code>	<b>**Absolute**</b> path from filesystem root	<code>`Read(//Users/alice/secrets/**)`</code>	
<code>`~/path`</code> <code>`/Users/alice/Documents/*.pdf`</code>	Path from <b>**home**</b> directory	<code>`Read(~/Documents/*.pdf)`</code>	
<code>`/path`</code>	Path <b>**relative to settings file**</b>	<code>`Edit(/src/**/*ts)`</code>	<code>`&lt;settings file path&gt;/src/**/*ts`</code>
<code>`path`</code> or <code>`./path`</code>	Path <b>**relative to current directory**</b>	<code>`Read(*.env)`</code>	<code>`&lt;cwd&gt;/*.env`</code>

<Warning>

A pattern like ``/Users/alice/file`` is NOT an absolute path - it's relative to your settings file! Use ``//Users/alice/file`` for absolute paths.

</Warning>

\* ``Edit(/docs/**)`` - Edits in ``<project>/docs/`` (NOT ``/docs/``!)

\* ``Read(~/.zshrc)`` - Reads your home directory's ``.zshrc``

\* ``Edit(/tmp/scratch.txt)`` - Edits the absolute path ``/tmp/scratch.txt``

\* ``Read(src/**)`` - Reads from ``<current-directory>/src/``

## **\*\*WebFetch\*\***

\* ``WebFetch(domain:example.com)`` Matches fetch requests to example.com

## **\*\*MCP\*\***

\* ``mcp__puppeteer`` Matches any tool provided by the ``puppeteer`` server (name configured in Claude Code)


\* ``mcp__puppeteer__puppeteer_navigate`` Matches the ``puppeteer_navigate`` tool provided by the ``puppeteer`` server

## **<Warning>**

Unlike other permission types, MCP permissions do NOT support wildcards (``**``).

To approve all tools from an MCP server:

\*  Use: ``mcp__github`` (approves ALL GitHub tools)

\*  Don't use: ``mcp__github__*`` (wildcards are not supported)

To approve specific tools only, list each one:

\*  Use: ``mcp__github__get_issue``

\*  Use: ``mcp__github__list_issues``

## **</Warning>**

### ### Additional permission control with hooks

[Claude Code hooks](/en/docs/claude-code/hooks-guide) provide a way to register custom shell commands to perform permission evaluation at runtime. When Claude Code makes a tool call, PreToolUse hooks run before the permission system runs, and the hook output can determine whether to approve or deny the tool call in place of the permission system.

### ### Enterprise managed policy settings

For enterprise deployments of Claude Code, we support enterprise managed policy settings that take precedence over user and project settings. This allows system administrators to enforce security policies that users cannot override.

System administrators can deploy policies to:

- \* macOS: `~/Library/Application Support/ClaudeCode/managed-settings.json``

- \* Linux and WSL: `/etc/claude-code/managed-settings.json``

- \* Windows: `C:\ProgramData\ClaudeCode\managed-settings.json``

These policy files follow the same format as regular [settings files](/en/docs/claude-code/settings#settings-files) but cannot be overridden by user or project settings. This ensures consistent security policies across your organization.

### ### Settings precedence

When multiple settings sources exist, they are applied in the following order (highest to lowest precedence):

1. Enterprise policies
2. Command line arguments
3. Local project settings (`.claude/settings.local.json``)

4. Shared project settings (`.claude/settings.json`)

5. User settings (`~/claude/settings.json`)

This hierarchy ensures that organizational policies are always enforced while still allowing flexibility at the project and user levels where appropriate.

## ## Credential management

Claude Code securely manages your authentication credentials:

**Storage location**: On macOS, API keys, OAuth tokens, and other credentials are stored in the encrypted macOS Keychain.

**Supported authentication types**: Claude.ai credentials, Anthropic API credentials, Bedrock Auth, and Vertex Auth.

**Custom credential scripts**: The `[`apiKeyHelper`](/en/docs/claude-code/settings#available-settings)` setting can be configured to run a shell script that returns an API key.

**Refresh intervals**: By default, `apiKeyHelper` is called after 5 minutes or on HTTP 401 response. Set `CLAUDE_CODE_API_KEY_HELPER_TTL_MS` environment variable for custom refresh intervals.

## # Data usage

> Learn about Anthropic's data usage policies for Claude

## ## Data policies

### ### Data training policy

**Consumer users (Free, Pro, and Max plans)**:

Starting August 28, 2025, we're giving you the choice to allow your data to be used to improve future Claude models.

We will train new models using data from Free, Pro, and Max accounts when this setting is on (including when you use Claude Code from these accounts).

\* If you're a current user, you can select your preference now and your selection will immediately go into effect.

This setting will only apply to new or resumed chats and coding sessions on Claude. Previous chats with no additional activity will not be used for model training.

\* You have until September 28, 2025 to make your selection.

If you're a new user, you can pick your setting for model training during the signup process.

You can change your selection at any time in your Privacy Settings.

**\*\*Commercial users\*\***: (Team and Enterprise plans, API, 3rd-party platforms, and Claude Gov) maintain existing policies: Anthropic does not train generative models using code or prompts sent to Claude Code under commercial terms, unless the customer has chosen to provide their data to us for model improvement (e.g. [Developer Partner Program](<https://support.anthropic.com/en/articles/11174108-about-the-development-partner-program>)).

### ### Development Partner Program

If you explicitly opt in to methods to provide us with materials to train on, such as via the [Development Partner Program](<https://support.anthropic.com/en/articles/11174108-about-the-development-partner-program>), we may use those materials provided to train our models. An organization admin can expressly opt-in to the Development Partner Program for their organization. Note that this program is available only for Anthropic first-party API, and not for Bedrock or Vertex users.

### ### Feedback using the `/bug` command

If you choose to send us feedback about Claude Code using the `/bug` command, we may use your feedback to improve our products and services. Transcripts shared via `/bug` are retained for 30 days.

### ### Data retention



Anthropic retains Claude Code data based on your account type and preferences.

**\*\*Consumer users (Free, Pro, and Max plans)\*\*:**

- \* Users who allow data use for model improvement: 5-year retention period to support model development and safety improvements

- \* Users who don't allow data use for model improvement: 30-day retention period

- \* Privacy settings can be changed at any time at [\[claude.ai/settings/data-privacy-controls\]](https://claude.ai/settings/data-privacy-controls)([claude.ai/settings/data-privacy-controls](https://claude.ai/settings/data-privacy-controls)).

**\*\*Commercial users (Team, Enterprise, and API)\*\*:**

- \* Standard: 30-day retention period

- \* Zero data retention: Available with appropriately configured API keys - Claude Code will not retain chat transcripts on servers

- \* Local caching: Claude Code clients may store sessions locally for up to 30 days to enable session resumption (configurable)

Learn more about data retention practices in our [\[Privacy Center\]](https://privacy.anthropic.com/)(<https://privacy.anthropic.com/>).

For full details, please review our [\[Commercial Terms of Service\]](https://www.anthropic.com/legal/commercial-terms)(<https://www.anthropic.com/legal/commercial-terms>) (for Team, Enterprise, and API users) or [\[Consumer Terms\]](https://www.anthropic.com/legal/consumer-terms)(<https://www.anthropic.com/legal/consumer-terms>) (for Free, Pro, and Max users) and [\[Privacy Policy\]](https://www.anthropic.com/legal/privacy)(<https://www.anthropic.com/legal/privacy>).

## ## Data flow and dependencies


```

Claude Code is installed from [NPM](<https://www.npmjs.com/package/@anthropic-ai/claude-code>). Claude Code runs locally. In order to interact with the LLM, Claude Code sends data over the network. This data includes all user prompts and model outputs. The data is encrypted in transit via TLS and is not encrypted at rest. Claude Code is compatible with most popular VPNs and LLM proxies.

Claude Code is built on Anthropic's APIs. For details regarding our API's security controls, including our API logging procedures, please refer to compliance artifacts offered in the [Anthropic Trust Center](<https://trust.anthropic.com>).

## ## Telemetry services

Claude Code connects from users' machines to the Statsig service to log operational metrics such as latency, reliability, and usage patterns. This logging does not include any code or file paths. Data is encrypted in transit using TLS and at rest using 256-bit AES encryption. Read more in the [Statsig security documentation](<https://www.statsig.com/trust/security>). To opt out of Statsig telemetry, set the `DISABLE\_TELEMETRY` environment variable.

Claude Code connects from users' machines to Sentry for operational error logging. The data is encrypted in transit using TLS and at rest using 256-bit AES encryption. Read more in the [Sentry security documentation](<https://sentry.io/security/>). To opt out of error logging, set the `DISABLE\_ERROR\_REPORTING` environment variable.

When users run the `bug` command, a copy of their full conversation history including code is sent to Anthropic. The data is encrypted in transit and at rest. Optionally, a Github issue is created in our public repository. To opt out of bug reporting, set the `DISABLE\_BUG\_COMMAND` environment variable.

## ## Default behaviors by API provider

By default, we disable all non-essential traffic (including error reporting, telemetry, and bug reporting functionality) when using Bedrock or Vertex. You can also opt out of all of these at once by setting the `CLAUDE_CODE_DISABLE_NONESSENTIAL_TRAFFIC` environment variable. Here are the full default behaviors:

| Service                                                                                                                                                                                                       | Anthropic API | Vertex API |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------|------------|
| Bedrock API                                                                                                                                                                                                   |               |            |
| <hr/>                                                                                                                                                                                                         |               |            |
| <hr/>                                                                                                                                                                                                         |               |            |
| <b>**Statsig (Metrics)**</b>                                                                                                                                                                                  |               |            |
| Default on.<br /> <code>DISABLE_TELEMETRY=1</code> to disable.   Default off.<br /> <code>CLAUDE_CODE_USE_VERTEX</code> must be 1.   Default off.<br /> <code>CLAUDE_CODE_USE_BEDROCK</code> must be 1.       |               |            |
| <b>**Sentry (Errors)**</b>                                                                                                                                                                                    |               |            |
| Default on.<br /> <code>DISABLE_ERROR_REPORTING=1</code> to disable.   Default off.<br /> <code>CLAUDE_CODE_USE_VERTEX</code> must be 1.   Default off.<br /> <code>CLAUDE_CODE_USE_BEDROCK</code> must be 1. |               |            |
| <b>**Anthropic API (/bug reports)**</b>                                                                                                                                                                       |               |            |
| Default on.<br /> <code>DISABLE_BUG_COMMAND=1</code> to disable.   Default off.<br /> <code>CLAUDE_CODE_USE_VERTEX</code> must be 1.   Default off.<br /> <code>CLAUDE_CODE_USE_BEDROCK</code> must be 1.     |               |            |

All environment variables can be checked into `settings.json` ([read more](/en/docs/claude-code/settings)).

## # Data usage

> Learn about Anthropic's data usage policies for Claude

## ## Data policies

### ### Data training policy

**\*\*Consumer users (Free, Pro, and Max plans)\*\*:**

Starting August 28, 2025, we're giving you the choice to allow your data to be used to improve future Claude models.

We will train new models using data from Free, Pro, and Max accounts when this setting is on (including when you use Claude Code from these accounts).

\* If you're a current user, you can select your preference now and your selection will immediately go into effect.

This setting will only apply to new or resumed chats and coding sessions on Claude. Previous chats with no additional activity will not be used for model training.

\* You have until September 28, 2025 to make your selection.

If you're a new user, you can pick your setting for model training during the signup process.

You can change your selection at any time in your Privacy Settings.

**\*\*Commercial users\*\***: (Team and Enterprise plans, API, 3rd-party platforms, and Claude Gov) maintain existing policies: Anthropic does not train generative models using code or prompts sent to Claude Code under commercial terms, unless the customer has chosen to provide their data to us for model improvement (e.g. [Developer Partner Program](<https://support.anthropic.com/en/articles/11174108-about-the-development-partner-program>)).

### ### Development Partner Program

If you explicitly opt in to methods to provide us with materials to train on, such as via the [Development Partner Program](<https://support.anthropic.com/en/articles/11174108-about-the-development-partner-program>), we may use those materials provided to train our models. An organization admin can expressly opt-in to the Development Partner Program for their organization. Note that this program is available only for Anthropic first-party API, and not for Bedrock or Vertex users.

### ### Feedback using the `/bug` command

If you choose to send us feedback about Claude Code using the `/bug` command, we may use your feedback to improve our products and services. Transcripts shared via `/bug` are retained for 30 days.

### ### Data retention

Anthropic retains Claude Code data based on your account type and preferences.

**\*\*Consumer users (Free, Pro, and Max plans)\*\*:**

- \* Users who allow data use for model improvement: 5-year retention period to support model development and safety improvements
- \* Users who don't allow data use for model improvement: 30-day retention period
- \* Privacy settings can be changed at any time at [\[claude.ai/settings/data-privacy-controls\]](https://claude.ai/settings/data-privacy-controls)([claude.ai/settings/data-privacy-controls](https://claude.ai/settings/data-privacy-controls)).

**\*\*Commercial users (Team, Enterprise, and API)\*\*:**

- \* Standard: 30-day retention period
- \* Zero data retention: Available with appropriately configured API keys - Claude Code will not retain chat transcripts on servers
- \* Local caching: Claude Code clients may store sessions locally for up to 30 days to enable session resumption (configurable)

Learn more about data retention practices in our [\[Privacy Center\]](https://privacy.anthropic.com/)(<https://privacy.anthropic.com/>).

For full details, please review our [\[Commercial Terms of Service\]](https://www.anthropic.com/legal/commercial-terms)(<https://www.anthropic.com/legal/commercial-terms>) (for Team, Enterprise, and API users) or [\[Consumer Terms\]](https://www.anthropic.com/legal/consumer-terms)(<https://www.anthropic.com/legal/consumer-terms>) (for Free, Pro, and Max users) and [\[Privacy Policy\]](https://www.anthropic.com/legal/privacy)(<https://www.anthropic.com/legal/privacy>).

**## Data flow and dependencies**



```
o=format&n=PF_69UDRSEsLpN9D&q=85&s=a25ba8e1c632bb02de4cf68e96ac5a8c 840w,  
https://mintcdn.com/anthropic/PF_69UDRSEsLpN9D/images/claude-code-data-flow.png?w=1100&fit=max&au  
to=format&n=PF_69UDRSEsLpN9D&q=85&s=434fb120de78f63df663268636485646 1100w,  
https://mintcdn.com/anthropic/PF_69UDRSEsLpN9D/images/claude-code-data-flow.png?w=1650&fit=max&au  
to=format&n=PF_69UDRSEsLpN9D&q=85&s=9baeb74ab4c1c8255e510c2c8b521e32 1650w,  
https://mintcdn.com/anthropic/PF_69UDRSEsLpN9D/images/claude-code-data-flow.png?w=2500&fit=max&au  
to=format&n=PF_69UDRSEsLpN9D&q=85&s=f4314f4067f037b57fe851d063ac2b77 2500w"  
data-optimize="true" data-opv="2" />
```

Claude Code is installed from [NPM](<https://www.npmjs.com/package/@anthropic-ai/claude-code>). Claude Code runs locally. In order to interact with the LLM, Claude Code sends data over the network. This data includes all user prompts and model outputs. The data is encrypted in transit via TLS and is not encrypted at rest. Claude Code is compatible with most popular VPNs and LLM proxies.

Claude Code is built on Anthropic's APIs. For details regarding our API's security controls, including our API logging procedures, please refer to compliance artifacts offered in the [Anthropic Trust Center](<https://trust.anthropic.com>).

## ## Telemetry services

Claude Code connects from users' machines to the Statsig service to log operational metrics such as latency, reliability, and usage patterns. This logging does not include any code or file paths. Data is encrypted in transit using TLS and at rest using 256-bit AES encryption. Read more in the [Statsig security documentation](<https://www.statsig.com/trust/security>). To opt out of Statsig telemetry, set the `DISABLE\_TELEMETRY` environment variable.

Claude Code connects from users' machines to Sentry for operational error logging. The data is encrypted in transit using TLS and at rest using 256-bit AES encryption. Read more in the [Sentry security documentation](<https://sentry.io/security/>). To opt out of error logging, set the `DISABLE\_ERROR\_REPORTING` environment variable.

When users run the `/bug` command, a copy of their full conversation history including code is sent to Anthropic. The data is encrypted in transit and at rest. Optionally, a Github issue is created in our public repository. To opt out of bug reporting, set the `DISABLE\_BUG\_COMMAND` environment variable.

## ## Default behaviors by API provider

By default, we disable all non-essential traffic (including error reporting, telemetry, and bug reporting functionality) when using Bedrock or Vertex. You can also opt out of all of these at once by setting the `CLAUDE_CODE_DISABLE_NONESSENTIAL_TRAFFIC` environment variable. Here are the full default behaviors:

| Service                                                                                                                                                                                                       | Anthropic API | Vertex API |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------|------------|
| Bedrock API                                                                                                                                                                                                   |               |            |
| <hr/>                                                                                                                                                                                                         |               |            |
| <hr/>                                                                                                                                                                                                         |               |            |
| <b>**Statsig (Metrics)**</b>                                                                                                                                                                                  |               |            |
| Default on.<br /> <code>DISABLE_TELEMETRY=1</code> to disable.   Default off.<br /> <code>CLAUDE_CODE_USE_VERTEX</code> must be 1.   Default off.<br /> <code>CLAUDE_CODE_USE_BEDROCK</code> must be 1.       |               |            |
| <b>**Sentry (Errors)**</b>                                                                                                                                                                                    |               |            |
| Default on.<br /> <code>DISABLE_ERROR_REPORTING=1</code> to disable.   Default off.<br /> <code>CLAUDE_CODE_USE_VERTEX</code> must be 1.   Default off.<br /> <code>CLAUDE_CODE_USE_BEDROCK</code> must be 1. |               |            |
| <b>**Anthropic API (/bug reports)**</b>                                                                                                                                                                       |               |            |
| Default on.<br /> <code>DISABLE_BUG_COMMAND=1</code> to disable.   Default off.<br /> <code>CLAUDE_CODE_USE_VERTEX</code> must be 1.   Default off.<br /> <code>CLAUDE_CODE_USE_BEDROCK</code> must be 1.     |               |            |

All environment variables can be checked into `settings.json` ([read more](/en/docs/claude-code/settings)).

## # Data usage

> Learn about Anthropic's data usage policies for Claude

## ## Data policies

### ### Data training policy

**\*\*Consumer users (Free, Pro, and Max plans)\*\*:**

Starting August 28, 2025, we're giving you the choice to allow your data to be used to improve future Claude models.

We will train new models using data from Free, Pro, and Max accounts when this setting is on (including when you use Claude Code from these accounts).

\* If you're a current user, you can select your preference now and your selection will immediately go into effect.

This setting will only apply to new or resumed chats and coding sessions on Claude. Previous chats with no additional activity will not be used for model training.

\* You have until September 28, 2025 to make your selection.

If you're a new user, you can pick your setting for model training during the signup process.

You can change your selection at any time in your Privacy Settings.

**\*\*Commercial users\*\***: (Team and Enterprise plans, API, 3rd-party platforms, and Claude Gov) maintain existing policies: Anthropic does not train generative models using code or prompts sent to Claude Code under commercial terms, unless the customer has chosen to provide their data to us for model improvement (e.g. [Developer Partner Program](<https://support.anthropic.com/en/articles/11174108-about-the-development-partner-program>)).

### ### Development Partner Program

If you explicitly opt in to methods to provide us with materials to train on, such as via the [Development Partner Program](<https://support.anthropic.com/en/articles/11174108-about-the-development-partner-program>), we may use those materials provided to train our models. An organization admin can expressly opt-in to the Development Partner Program for their organization. Note that this program is available only for Anthropic first-party API, and not for Bedrock or Vertex users.

### ### Feedback using the `/bug` command

If you choose to send us feedback about Claude Code using the `/bug` command, we may use your feedback to improve our products and services. Transcripts shared via `/bug` are retained for 30 days.

### ### Data retention

Anthropic retains Claude Code data based on your account type and preferences.



**\*\*Consumer users (Free, Pro, and Max plans)\*\*:**

- \* Users who allow data use for model improvement: 5-year retention period to support model development and safety improvements
- \* Users who don't allow data use for model improvement: 30-day retention period
- \* Privacy settings can be changed at any time at [\[claude.ai/settings/data-privacy-controls\]](https://claude.ai/settings/data-privacy-controls)([claude.ai/settings/data-privacy-controls](https://claude.ai/settings/data-privacy-controls)).

**\*\*Commercial users (Team, Enterprise, and API)\*\*:**

- \* Standard: 30-day retention period
- \* Zero data retention: Available with appropriately configured API keys - Claude Code will not retain chat transcripts on servers
- \* Local caching: Claude Code clients may store sessions locally for up to 30 days to enable session resumption (configurable)

Learn more about data retention practices in our [\[Privacy Center\]](https://privacy.anthropic.com/)(<https://privacy.anthropic.com/>).

For full details, please review our [\[Commercial Terms of Service\]](https://www.anthropic.com/legal/commercial-terms)(<https://www.anthropic.com/legal/commercial-terms>) (for Team, Enterprise, and API users) or [\[Consumer Terms\]](https://www.anthropic.com/legal/consumer-terms)(<https://www.anthropic.com/legal/consumer-terms>) (for Free, Pro, and Max users) and [\[Privacy Policy\]](https://www.anthropic.com/legal/privacy)(<https://www.anthropic.com/legal/privacy>).

**## Data flow and dependencies**



```
o=format&n=PF_69UDRSEsLpN9D&q=85&s=a25ba8e1c632bb02de4cf68e96ac5a8c 840w,  
https://mintcdn.com/anthropic/PF_69UDRSEsLpN9D/images/claude-code-data-flow.png?w=1100&fit=max&au  
to=format&n=PF_69UDRSEsLpN9D&q=85&s=434fb120de78f63df663268636485646 1100w,  
https://mintcdn.com/anthropic/PF_69UDRSEsLpN9D/images/claude-code-data-flow.png?w=1650&fit=max&au  
to=format&n=PF_69UDRSEsLpN9D&q=85&s=9baeb74ab4c1c8255e510c2c8b521e32 1650w,  
https://mintcdn.com/anthropic/PF_69UDRSEsLpN9D/images/claude-code-data-flow.png?w=2500&fit=max&au  
to=format&n=PF_69UDRSEsLpN9D&q=85&s=f4314f4067f037b57fe851d063ac2b77 2500w"  
data-optimize="true" data-opv="2" />
```

Claude Code is installed from [NPM](<https://www.npmjs.com/package/@anthropic-ai/claude-code>). Claude Code runs locally. In order to interact with the LLM, Claude Code sends data over the network. This data includes all user prompts and model outputs. The data is encrypted in transit via TLS and is not encrypted at rest. Claude Code is compatible with most popular VPNs and LLM proxies.

Claude Code is built on Anthropic's APIs. For details regarding our API's security controls, including our API logging procedures, please refer to compliance artifacts offered in the [Anthropic Trust Center](<https://trust.anthropic.com>).

## ## Telemetry services

Claude Code connects from users' machines to the Statsig service to log operational metrics such as latency, reliability, and usage patterns. This logging does not include any code or file paths. Data is encrypted in transit using TLS and at rest using 256-bit AES encryption. Read more in the [Statsig security documentation](<https://www.statsig.com/trust/security>). To opt out of Statsig telemetry, set the `DISABLE\_TELEMETRY` environment variable.

Claude Code connects from users' machines to Sentry for operational error logging. The data is encrypted in transit using TLS and at rest using 256-bit AES encryption. Read more in the [Sentry security documentation](<https://sentry.io/security/>). To opt out of error logging, set the `DISABLE\_ERROR\_REPORTING` environment variable.

When users run the `/bug` command, a copy of their full conversation history including code is sent to Anthropic. The data is encrypted in transit and at rest. Optionally, a Github issue is created in our public repository. To opt out of bug reporting, set the `DISABLE\_BUG\_COMMAND` environment variable.

## ## Default behaviors by API provider

By default, we disable all non-essential traffic (including error reporting, telemetry, and bug reporting functionality) when using Bedrock or Vertex. You can also opt out of all of these at once by setting the `CLAUDE_CODE_DISABLE_NONESSENTIAL_TRAFFIC` environment variable. Here are the full default behaviors:

| Service                                                                                                                                                                                                       | Anthropic API | Vertex API |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------|------------|
| Bedrock API                                                                                                                                                                                                   |               |            |
| <hr/>                                                                                                                                                                                                         |               |            |
| <hr/>                                                                                                                                                                                                         |               |            |
| <b>**Statsig (Metrics)**</b>                                                                                                                                                                                  |               |            |
| Default on.<br /> <code>DISABLE_TELEMETRY=1</code> to disable.   Default off.<br /> <code>CLAUDE_CODE_USE_VERTEX</code> must be 1.   Default off.<br /> <code>CLAUDE_CODE_USE_BEDROCK</code> must be 1.       |               |            |
| <b>**Sentry (Errors)**</b>                                                                                                                                                                                    |               |            |
| Default on.<br /> <code>DISABLE_ERROR_REPORTING=1</code> to disable.   Default off.<br /> <code>CLAUDE_CODE_USE_VERTEX</code> must be 1.   Default off.<br /> <code>CLAUDE_CODE_USE_BEDROCK</code> must be 1. |               |            |
| <b>**Anthropic API (/bug reports)**</b>                                                                                                                                                                       |               |            |
| Default on.<br /> <code>DISABLE_BUG_COMMAND=1</code> to disable.   Default off.<br /> <code>CLAUDE_CODE_USE_VERTEX</code> must be 1.   Default off.<br /> <code>CLAUDE_CODE_USE_BEDROCK</code> must be 1.     |               |            |

All environment variables can be checked into `settings.json` ([read more](#) `/en/docs/claude-code/settings`)).

## # Data usage

> Learn about Anthropic's data usage policies for Claude

## ## Data policies

### ### Data training policy

**\*\*Consumer users (Free, Pro, and Max plans)\*\*:**

Starting August 28, 2025, we're giving you the choice to allow your data to be used to improve future Claude models.

We will train new models using data from Free, Pro, and Max accounts when this setting is on (including when you use Claude Code from these accounts).

\* If you're a current user, you can select your preference now and your selection will immediately go into effect.

This setting will only apply to new or resumed chats and coding sessions on Claude. Previous chats with no additional activity will not be used for model training.

\* You have until September 28, 2025 to make your selection.

If you're a new user, you can pick your setting for model training during the signup process.

You can change your selection at any time in your Privacy Settings.

**\*\*Commercial users\*\***: (Team and Enterprise plans, API, 3rd-party platforms, and Claude Gov) maintain existing policies: Anthropic does not train generative models using code or prompts sent to Claude Code under commercial terms, unless the customer has chosen to provide their data to us for model improvement (e.g. [Developer Partner Program](<https://support.anthropic.com/en/articles/11174108-about-the-development-partner-program>)).

### ### Development Partner Program

If you explicitly opt in to methods to provide us with materials to train on, such as via the [Development Partner Program](<https://support.anthropic.com/en/articles/11174108-about-the-development-partner-program>), we may use those materials provided to train our models. An organization admin can expressly opt-in to the Development Partner Program for their organization. Note that this program is available only for Anthropic first-party API, and not for Bedrock or Vertex users.

### ### Feedback using the `/bug` command

If you choose to send us feedback about Claude Code using the `/bug` command, we may use your feedback to improve our products and services. Transcripts shared via `/bug` are retained for 30 days.

### ### Data retention

Anthropic retains Claude Code data based on your account type and preferences.

**\*\*Consumer users (Free, Pro, and Max plans)\*\*:**

- \* Users who allow data use for model improvement: 5-year retention period to support model development and safety improvements
- \* Users who don't allow data use for model improvement: 30-day retention period
- \* Privacy settings can be changed at any time at [\[claude.ai/settings/data-privacy-controls\]](https://claude.ai/settings/data-privacy-controls)([claude.ai/settings/data-privacy-controls](https://claude.ai/settings/data-privacy-controls)).

**\*\*Commercial users (Team, Enterprise, and API)\*\*:**

- \* Standard: 30-day retention period
- \* Zero data retention: Available with appropriately configured API keys - Claude Code will not retain chat transcripts on servers
- \* Local caching: Claude Code clients may store sessions locally for up to 30 days to enable session resumption (configurable)

Learn more about data retention practices in our [\[Privacy Center\]](https://privacy.anthropic.com/)(<https://privacy.anthropic.com/>).

For full details, please review our [\[Commercial Terms of Service\]](https://www.anthropic.com/legal/commercial-terms)(<https://www.anthropic.com/legal/commercial-terms>) (for Team, Enterprise, and API users) or [\[Consumer Terms\]](https://www.anthropic.com/legal/consumer-terms)(<https://www.anthropic.com/legal/consumer-terms>) (for Free, Pro, and Max users) and [\[Privacy Policy\]](https://www.anthropic.com/legal/privacy)(<https://www.anthropic.com/legal/privacy>).

**## Data flow and dependencies**



```
o=format&n=PF_69UDRSEsLpN9D&q=85&s=a25ba8e1c632bb02de4cf68e96ac5a8c 840w,  
https://mintcdn.com/anthropic/PF_69UDRSEsLpN9D/images/claude-code-data-flow.png?w=1100&fit=max&au  
to=format&n=PF_69UDRSEsLpN9D&q=85&s=434fb120de78f63df663268636485646 1100w,  
https://mintcdn.com/anthropic/PF_69UDRSEsLpN9D/images/claude-code-data-flow.png?w=1650&fit=max&au  
to=format&n=PF_69UDRSEsLpN9D&q=85&s=9baeb74ab4c1c8255e510c2c8b521e32 1650w,  
https://mintcdn.com/anthropic/PF_69UDRSEsLpN9D/images/claude-code-data-flow.png?w=2500&fit=max&au  
to=format&n=PF_69UDRSEsLpN9D&q=85&s=f4314f4067f037b57fe851d063ac2b77 2500w"  
data-optimize="true" data-opv="2" />
```

Claude Code is installed from [NPM](<https://www.npmjs.com/package/@anthropic-ai/claude-code>). Claude Code runs locally. In order to interact with the LLM, Claude Code sends data over the network. This data includes all user prompts and model outputs. The data is encrypted in transit via TLS and is not encrypted at rest. Claude Code is compatible with most popular VPNs and LLM proxies.

Claude Code is built on Anthropic's APIs. For details regarding our API's security controls, including our API logging procedures, please refer to compliance artifacts offered in the [Anthropic Trust Center](<https://trust.anthropic.com>).

## ## Telemetry services

Claude Code connects from users' machines to the Statsig service to log operational metrics such as latency, reliability, and usage patterns. This logging does not include any code or file paths. Data is encrypted in transit using TLS and at rest using 256-bit AES encryption. Read more in the [Statsig security documentation](<https://www.statsig.com/trust/security>). To opt out of Statsig telemetry, set the `DISABLE\_TELEMETRY` environment variable.

Claude Code connects from users' machines to Sentry for operational error logging. The data is encrypted in transit using TLS and at rest using 256-bit AES encryption. Read more in the [Sentry security documentation](<https://sentry.io/security/>). To opt out of error logging, set the `DISABLE\_ERROR\_REPORTING` environment variable.

When users run the `/bug` command, a copy of their full conversation history including code is sent to Anthropic. The data is encrypted in transit and at rest. Optionally, a Github issue is created in our public repository. To opt out of bug reporting, set the `DISABLE\_BUG\_COMMAND` environment variable.

## ## Default behaviors by API provider

By default, we disable all non-essential traffic (including error reporting, telemetry, and bug reporting functionality) when using Bedrock or Vertex. You can also opt out of all of these at once by setting the `CLAUDE_CODE_DISABLE_NONESSENTIAL_TRAFFIC` environment variable. Here are the full default behaviors:

| Service                                                                                                                                                                                                       | Anthropic API | Vertex API |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------|------------|
| Bedrock API                                                                                                                                                                                                   |               |            |
| <hr/>                                                                                                                                                                                                         |               |            |
| <hr/>                                                                                                                                                                                                         |               |            |
| <b>**Statsig (Metrics)**</b>                                                                                                                                                                                  |               |            |
| Default on.<br /> <code>DISABLE_TELEMETRY=1</code> to disable.   Default off.<br /> <code>CLAUDE_CODE_USE_VERTEX</code> must be 1.   Default off.<br /> <code>CLAUDE_CODE_USE_BEDROCK</code> must be 1.       |               |            |
| <b>**Sentry (Errors)**</b>                                                                                                                                                                                    |               |            |
| Default on.<br /> <code>DISABLE_ERROR_REPORTING=1</code> to disable.   Default off.<br /> <code>CLAUDE_CODE_USE_VERTEX</code> must be 1.   Default off.<br /> <code>CLAUDE_CODE_USE_BEDROCK</code> must be 1. |               |            |
| <b>**Anthropic API (/bug reports)**</b>                                                                                                                                                                       |               |            |
| Default on.<br /> <code>DISABLE_BUG_COMMAND=1</code> to disable.   Default off.<br /> <code>CLAUDE_CODE_USE_VERTEX</code> must be 1.   Default off.<br /> <code>CLAUDE_CODE_USE_BEDROCK</code> must be 1.     |               |            |

All environment variables can be checked into `settings.json` ([read more](#) `/en/docs/claude-code/settings`)).

## # Data usage

> Learn about Anthropic's data usage policies for Claude

## ## Data policies

### ### Data training policy

**\*\*Consumer users (Free, Pro, and Max plans)\*\*:**

Starting August 28, 2025, we're giving you the choice to allow your data to be used to improve future Claude models.

We will train new models using data from Free, Pro, and Max accounts when this setting is on (including when you use Claude Code from these accounts).

\* If you're a current user, you can select your preference now and your selection will immediately go into effect.

This setting will only apply to new or resumed chats and coding sessions on Claude. Previous chats with no additional activity will not be used for model training.

\* You have until September 28, 2025 to make your selection.

If you're a new user, you can pick your setting for model training during the signup process.

You can change your selection at any time in your Privacy Settings.

**\*\*Commercial users\*\***: (Team and Enterprise plans, API, 3rd-party platforms, and Claude Gov) maintain existing policies: Anthropic does not train generative models using code or prompts sent to Claude Code under commercial terms, unless the customer has chosen to provide their data to us for model improvement (e.g. [Developer Partner Program](<https://support.anthropic.com/en/articles/11174108-about-the-development-partner-program>)).

### ### Development Partner Program

If you explicitly opt in to methods to provide us with materials to train on, such as via the [Development Partner Program](<https://support.anthropic.com/en/articles/11174108-about-the-development-partner-program>), we may use those materials provided to train our models. An organization admin can expressly opt-in to the Development Partner Program for their organization. Note that this program is available only for Anthropic first-party API, and not for Bedrock or Vertex users.

### ### Feedback using the `/bug` command

If you choose to send us feedback about Claude Code using the `/bug` command, we may use your feedback to improve our products and services. Transcripts shared via `/bug` are retained for 30 days.

### ### Data retention



Anthropic retains Claude Code data based on your account type and preferences.

**\*\*Consumer users (Free, Pro, and Max plans)\*\*:**

- \* Users who allow data use for model improvement: 5-year retention period to support model development and safety improvements

- \* Users who don't allow data use for model improvement: 30-day retention period

- \* Privacy settings can be changed at any time at [\[claude.ai/settings/data-privacy-controls\]](https://claude.ai/settings/data-privacy-controls)([claude.ai/settings/data-privacy-controls](https://claude.ai/settings/data-privacy-controls)).

**\*\*Commercial users (Team, Enterprise, and API)\*\*:**

- \* Standard: 30-day retention period

- \* Zero data retention: Available with appropriately configured API keys - Claude Code will not retain chat transcripts on servers

- \* Local caching: Claude Code clients may store sessions locally for up to 30 days to enable session resumption (configurable)

Learn more about data retention practices in our [\[Privacy Center\]](https://privacy.anthropic.com/)(<https://privacy.anthropic.com/>).

For full details, please review our [\[Commercial Terms of Service\]](https://www.anthropic.com/legal/commercial-terms)(<https://www.anthropic.com/legal/commercial-terms>) (for Team, Enterprise, and API users) or [\[Consumer Terms\]](https://www.anthropic.com/legal/consumer-terms)(<https://www.anthropic.com/legal/consumer-terms>) (for Free, Pro, and Max users) and [\[Privacy Policy\]](https://www.anthropic.com/legal/privacy)(<https://www.anthropic.com/legal/privacy>).

## ## Data flow and dependencies


```

Claude Code is installed from [NPM](<https://www.npmjs.com/package/@anthropic-ai/claude-code>). Claude Code runs locally. In order to interact with the LLM, Claude Code sends data over the network. This data includes all user prompts and model outputs. The data is encrypted in transit via TLS and is not encrypted at rest. Claude Code is compatible with most popular VPNs and LLM proxies.

Claude Code is built on Anthropic's APIs. For details regarding our API's security controls, including our API logging procedures, please refer to compliance artifacts offered in the [Anthropic Trust Center](<https://trust.anthropic.com>).

## ## Telemetry services

Claude Code connects from users' machines to the Statsig service to log operational metrics such as latency, reliability, and usage patterns. This logging does not include any code or file paths. Data is encrypted in transit using TLS and at rest using 256-bit AES encryption. Read more in the [Statsig security documentation](<https://www.statsig.com/trust/security>). To opt out of Statsig telemetry, set the `DISABLE\_TELEMETRY` environment variable.

Claude Code connects from users' machines to Sentry for operational error logging. The data is encrypted in transit using TLS and at rest using 256-bit AES encryption. Read more in the [Sentry security documentation](<https://sentry.io/security/>). To opt out of error logging, set the `DISABLE\_ERROR\_REPORTING` environment variable.

When users run the `bug` command, a copy of their full conversation history including code is sent to Anthropic. The data is encrypted in transit and at rest. Optionally, a Github issue is created in our public repository. To opt out of bug reporting, set the `DISABLE\_BUG\_COMMAND` environment variable.

## ## Default behaviors by API provider

By default, we disable all non-essential traffic (including error reporting, telemetry, and bug reporting functionality) when using Bedrock or Vertex. You can also opt out of all of these at once by setting the `CLAUDE_CODE_DISABLE_NONESSENTIAL_TRAFFIC` environment variable. Here are the full default behaviors:

Service	Anthropic API	Vertex API
Bedrock API		
<hr/>		
<hr/>		
<b>**Statsig (Metrics)**</b>		
Default on.  <code>DISABLE_TELEMETRY=1</code> to disable.   Default off.  <code>CLAUDE_CODE_USE_VERTEX</code> must be 1.   Default off.  <code>CLAUDE_CODE_USE_BEDROCK</code> must be 1.		
<b>**Sentry (Errors)**</b>		
Default on.  <code>DISABLE_ERROR_REPORTING=1</code> to disable.   Default off.  <code>CLAUDE_CODE_USE_VERTEX</code> must be 1.   Default off.  <code>CLAUDE_CODE_USE_BEDROCK</code> must be 1.		
<b>**Anthropic API (/bug reports)**</b>		
Default on.  <code>DISABLE_BUG_COMMAND=1</code> to disable.   Default off.  <code>CLAUDE_CODE_USE_VERTEX</code> must be 1.   Default off.  <code>CLAUDE_CODE_USE_BEDROCK</code> must be 1.		

All environment variables can be checked into `settings.json` ([read more](#) `/en/docs/claude-code/settings`)).

## # Data usage

> Learn about Anthropic's data usage policies for Claude

## ## Data policies

### ### Data training policy

**\*\*Consumer users (Free, Pro, and Max plans)\*\*:**

Starting August 28, 2025, we're giving you the choice to allow your data to be used to improve future Claude models.

We will train new models using data from Free, Pro, and Max accounts when this setting is on (including when you use Claude Code from these accounts).

\* If you're a current user, you can select your preference now and your selection will immediately go into effect.

This setting will only apply to new or resumed chats and coding sessions on Claude. Previous chats with no additional activity will not be used for model training.

\* You have until September 28, 2025 to make your selection.

If you're a new user, you can pick your setting for model training during the signup process.

You can change your selection at any time in your Privacy Settings.

**\*\*Commercial users\*\***: (Team and Enterprise plans, API, 3rd-party platforms, and Claude Gov) maintain existing policies: Anthropic does not train generative models using code or prompts sent to Claude Code under commercial terms, unless the customer has chosen to provide their data to us for model improvement (e.g. [Developer Partner Program](<https://support.anthropic.com/en/articles/11174108-about-the-development-partner-program>)).

### ### Development Partner Program

If you explicitly opt in to methods to provide us with materials to train on, such as via the [Development Partner Program](<https://support.anthropic.com/en/articles/11174108-about-the-development-partner-program>), we may use those materials provided to train our models. An organization admin can expressly opt-in to the Development Partner Program for their organization. Note that this program is available only for Anthropic first-party API, and not for Bedrock or Vertex users.

### ### Feedback using the `/bug` command

If you choose to send us feedback about Claude Code using the `/bug` command, we may use your feedback to improve our products and services. Transcripts shared via `/bug` are retained for 30 days.

### ### Data retention

Anthropic retains Claude Code data based on your account type and preferences.

**\*\*Consumer users (Free, Pro, and Max plans)\*\*:**

- \* Users who allow data use for model improvement: 5-year retention period to support model development and safety improvements
- \* Users who don't allow data use for model improvement: 30-day retention period
- \* Privacy settings can be changed at any time at [\[claude.ai/settings/data-privacy-controls\]](https://claude.ai/settings/data-privacy-controls)([claude.ai/settings/data-privacy-controls](https://claude.ai/settings/data-privacy-controls)).

**\*\*Commercial users (Team, Enterprise, and API)\*\*:**

- \* Standard: 30-day retention period
- \* Zero data retention: Available with appropriately configured API keys - Claude Code will not retain chat transcripts on servers
- \* Local caching: Claude Code clients may store sessions locally for up to 30 days to enable session resumption (configurable)

Learn more about data retention practices in our [\[Privacy Center\]](https://privacy.anthropic.com/)(<https://privacy.anthropic.com/>).

For full details, please review our [\[Commercial Terms of Service\]](https://www.anthropic.com/legal/commercial-terms)(<https://www.anthropic.com/legal/commercial-terms>) (for Team, Enterprise, and API users) or [\[Consumer Terms\]](https://www.anthropic.com/legal/consumer-terms)(<https://www.anthropic.com/legal/consumer-terms>) (for Free, Pro, and Max users) and [\[Privacy Policy\]](https://www.anthropic.com/legal/privacy)(<https://www.anthropic.com/legal/privacy>).

**## Data flow and dependencies**



```
o=format&n=PF_69UDRSEsLpN9D&q=85&s=a25ba8e1c632bb02de4cf68e96ac5a8c 840w,  
https://mintcdn.com/anthropic/PF_69UDRSEsLpN9D/images/claude-code-data-flow.png?w=1100&fit=max&au  
to=format&n=PF_69UDRSEsLpN9D&q=85&s=434fb120de78f63df663268636485646 1100w,  
https://mintcdn.com/anthropic/PF_69UDRSEsLpN9D/images/claude-code-data-flow.png?w=1650&fit=max&au  
to=format&n=PF_69UDRSEsLpN9D&q=85&s=9baeb74ab4c1c8255e510c2c8b521e32 1650w,  
https://mintcdn.com/anthropic/PF_69UDRSEsLpN9D/images/claude-code-data-flow.png?w=2500&fit=max&au  
to=format&n=PF_69UDRSEsLpN9D&q=85&s=f4314f4067f037b57fe851d063ac2b77 2500w"  
data-optimize="true" data-opv="2" />
```

Claude Code is installed from [NPM](<https://www.npmjs.com/package/@anthropic-ai/claude-code>). Claude Code runs locally. In order to interact with the LLM, Claude Code sends data over the network. This data includes all user prompts and model outputs. The data is encrypted in transit via TLS and is not encrypted at rest. Claude Code is compatible with most popular VPNs and LLM proxies.

Claude Code is built on Anthropic's APIs. For details regarding our API's security controls, including our API logging procedures, please refer to compliance artifacts offered in the [Anthropic Trust Center](<https://trust.anthropic.com>).

## ## Telemetry services

Claude Code connects from users' machines to the Statsig service to log operational metrics such as latency, reliability, and usage patterns. This logging does not include any code or file paths. Data is encrypted in transit using TLS and at rest using 256-bit AES encryption. Read more in the [Statsig security documentation](<https://www.statsig.com/trust/security>). To opt out of Statsig telemetry, set the `DISABLE\_TELEMETRY` environment variable.

Claude Code connects from users' machines to Sentry for operational error logging. The data is encrypted in transit using TLS and at rest using 256-bit AES encryption. Read more in the [Sentry security documentation](<https://sentry.io/security/>). To opt out of error logging, set the `DISABLE\_ERROR\_REPORTING` environment variable.

When users run the `/bug` command, a copy of their full conversation history including code is sent to Anthropic. The data is encrypted in transit and at rest. Optionally, a Github issue is created in our public repository. To opt out of bug reporting, set the `DISABLE\_BUG\_COMMAND` environment variable.

## ## Default behaviors by API provider

By default, we disable all non-essential traffic (including error reporting, telemetry, and bug reporting functionality) when using Bedrock or Vertex. You can also opt out of all of these at once by setting the `CLAUDE_CODE_DISABLE_NONESSENTIAL_TRAFFIC` environment variable. Here are the full default behaviors:

Service	Anthropic API	Vertex API
Bedrock API		
<hr/>		
<hr/>		
<b>**Statsig (Metrics)**</b>		
Default on.  <code>DISABLE_TELEMETRY=1</code> to disable.   Default off.  <code>CLAUDE_CODE_USE_VERTEX</code> must be 1.   Default off.  <code>CLAUDE_CODE_USE_BEDROCK</code> must be 1.		
<b>**Sentry (Errors)**</b>		
Default on.  <code>DISABLE_ERROR_REPORTING=1</code> to disable.   Default off.  <code>CLAUDE_CODE_USE_VERTEX</code> must be 1.   Default off.  <code>CLAUDE_CODE_USE_BEDROCK</code> must be 1.		
<b>**Anthropic API (/bug reports)**</b>		
Default on.  <code>DISABLE_BUG_COMMAND=1</code> to disable.   Default off.  <code>CLAUDE_CODE_USE_VERTEX</code> must be 1.   Default off.  <code>CLAUDE_CODE_USE_BEDROCK</code> must be 1.		

All environment variables can be checked into `settings.json` ([read more](#) `/en/docs/claude-code/settings`)).

## # Data usage

> Learn about Anthropic's data usage policies for Claude

## ## Data policies

### ### Data training policy

**\*\*Consumer users (Free, Pro, and Max plans)\*\*:**

Starting August 28, 2025, we're giving you the choice to allow your data to be used to improve future Claude models.

We will train new models using data from Free, Pro, and Max accounts when this setting is on (including when you use Claude Code from these accounts).

\* If you're a current user, you can select your preference now and your selection will immediately go into effect.

This setting will only apply to new or resumed chats and coding sessions on Claude. Previous chats with no additional activity will not be used for model training.

\* You have until September 28, 2025 to make your selection.

If you're a new user, you can pick your setting for model training during the signup process.

You can change your selection at any time in your Privacy Settings.

**\*\*Commercial users\*\***: (Team and Enterprise plans, API, 3rd-party platforms, and Claude Gov) maintain existing policies: Anthropic does not train generative models using code or prompts sent to Claude Code under commercial terms, unless the customer has chosen to provide their data to us for model improvement (e.g. [Developer Partner Program](<https://support.anthropic.com/en/articles/11174108-about-the-development-partner-program>)).

### ### Development Partner Program

If you explicitly opt in to methods to provide us with materials to train on, such as via the [Development Partner Program](<https://support.anthropic.com/en/articles/11174108-about-the-development-partner-program>), we may use those materials provided to train our models. An organization admin can expressly opt-in to the Development Partner Program for their organization. Note that this program is available only for Anthropic first-party API, and not for Bedrock or Vertex users.

### ### Feedback using the `/bug` command

If you choose to send us feedback about Claude Code using the `/bug` command, we may use your feedback to improve our products and services. Transcripts shared via `/bug` are retained for 30 days.

### ### Data retention

Anthropic retains Claude Code data based on your account type and preferences.



**\*\*Consumer users (Free, Pro, and Max plans)\*\*:**

- \* Users who allow data use for model improvement: 5-year retention period to support model development and safety improvements
- \* Users who don't allow data use for model improvement: 30-day retention period
- \* Privacy settings can be changed at any time at [\[claude.ai/settings/data-privacy-controls\]](https://claude.ai/settings/data-privacy-controls)([claude.ai/settings/data-privacy-controls](https://claude.ai/settings/data-privacy-controls)).

**\*\*Commercial users (Team, Enterprise, and API)\*\*:**

- \* Standard: 30-day retention period
- \* Zero data retention: Available with appropriately configured API keys - Claude Code will not retain chat transcripts on servers
- \* Local caching: Claude Code clients may store sessions locally for up to 30 days to enable session resumption (configurable)

Learn more about data retention practices in our [\[Privacy Center\]](https://privacy.anthropic.com/)(<https://privacy.anthropic.com/>).

For full details, please review our [\[Commercial Terms of Service\]](https://www.anthropic.com/legal/commercial-terms)(<https://www.anthropic.com/legal/commercial-terms>) (for Team, Enterprise, and API users) or [\[Consumer Terms\]](https://www.anthropic.com/legal/consumer-terms)(<https://www.anthropic.com/legal/consumer-terms>) (for Free, Pro, and Max users) and [\[Privacy Policy\]](https://www.anthropic.com/legal/privacy)(<https://www.anthropic.com/legal/privacy>).

**## Data flow and dependencies**



```
o=format&n=PF_69UDRSEsLpN9D&q=85&s=a25ba8e1c632bb02de4cf68e96ac5a8c 840w,  
https://mintcdn.com/anthropic/PF_69UDRSEsLpN9D/images/claude-code-data-flow.png?w=1100&fit=max&au  
to=format&n=PF_69UDRSEsLpN9D&q=85&s=434fb120de78f63df663268636485646 1100w,  
https://mintcdn.com/anthropic/PF_69UDRSEsLpN9D/images/claude-code-data-flow.png?w=1650&fit=max&au  
to=format&n=PF_69UDRSEsLpN9D&q=85&s=9baeb74ab4c1c8255e510c2c8b521e32 1650w,  
https://mintcdn.com/anthropic/PF_69UDRSEsLpN9D/images/claude-code-data-flow.png?w=2500&fit=max&au  
to=format&n=PF_69UDRSEsLpN9D&q=85&s=f4314f4067f037b57fe851d063ac2b77 2500w"  
data-optimize="true" data-opv="2" />
```

Claude Code is installed from [NPM](<https://www.npmjs.com/package/@anthropic-ai/claude-code>). Claude Code runs locally. In order to interact with the LLM, Claude Code sends data over the network. This data includes all user prompts and model outputs. The data is encrypted in transit via TLS and is not encrypted at rest. Claude Code is compatible with most popular VPNs and LLM proxies.

Claude Code is built on Anthropic's APIs. For details regarding our API's security controls, including our API logging procedures, please refer to compliance artifacts offered in the [Anthropic Trust Center](<https://trust.anthropic.com>).

## ## Telemetry services

Claude Code connects from users' machines to the Statsig service to log operational metrics such as latency, reliability, and usage patterns. This logging does not include any code or file paths. Data is encrypted in transit using TLS and at rest using 256-bit AES encryption. Read more in the [Statsig security documentation](<https://www.statsig.com/trust/security>). To opt out of Statsig telemetry, set the `DISABLE\_TELEMETRY` environment variable.

Claude Code connects from users' machines to Sentry for operational error logging. The data is encrypted in transit using TLS and at rest using 256-bit AES encryption. Read more in the [Sentry security documentation](<https://sentry.io/security/>). To opt out of error logging, set the `DISABLE\_ERROR\_REPORTING` environment variable.

When users run the `/bug` command, a copy of their full conversation history including code is sent to Anthropic. The data is encrypted in transit and at rest. Optionally, a Github issue is created in our public repository. To opt out of bug reporting, set the `DISABLE\_BUG\_COMMAND` environment variable.

## ## Default behaviors by API provider

By default, we disable all non-essential traffic (including error reporting, telemetry, and bug reporting functionality) when using Bedrock or Vertex. You can also opt out of all of these at once by setting the ``CLAUDE_CODE_DISABLE_NONESSENTIAL_TRAFFIC`` environment variable. Here are the full default behaviors:

Service	Anthropic API	Vertex API
Bedrock API		
<hr/>		
<hr/>		
<b>**Statsig (Metrics)**</b>		
Default on.  <code>`DISABLE_TELEMETRY=1`</code> to disable.   Default off.  <code>`CLAUDE_CODE_USE_VERTEX`</code> must be 1.   Default off.  <code>`CLAUDE_CODE_USE_BEDROCK`</code> must be 1.		
<b>**Sentry (Errors)**</b>		
Default on.  <code>`DISABLE_ERROR_REPORTING=1`</code> to disable.   Default off.  <code>`CLAUDE_CODE_USE_VERTEX`</code> must be 1.   Default off.  <code>`CLAUDE_CODE_USE_BEDROCK`</code> must be 1.		
<b>**Anthropic API (`/bug` reports)**</b>		
Default on.  <code>`DISABLE_BUG_COMMAND=1`</code> to disable.   Default off.  <code>`CLAUDE_CODE_USE_VERTEX`</code> must be 1.   Default off.  <code>`CLAUDE_CODE_USE_BEDROCK`</code> must be 1.		

All environment variables can be checked into ``settings.json`` ([\[read more\]](#) `(/en/docs/claude-code/settings)`).

## # Data usage

> Learn about Anthropic's data usage policies for Claude

## ## Data policies

### ### Data training policy

**\*\*Consumer users (Free, Pro, and Max plans)\*\*:**

Starting August 28, 2025, we're giving you the choice to allow your data to be used to improve future Claude models.

We will train new models using data from Free, Pro, and Max accounts when this setting is on (including when you use Claude Code from these accounts).

\* If you're a current user, you can select your preference now and your selection will immediately go into effect.

This setting will only apply to new or resumed chats and coding sessions on Claude. Previous chats with no additional activity will not be used for model training.

\* You have until September 28, 2025 to make your selection.

If you're a new user, you can pick your setting for model training during the signup process.

You can change your selection at any time in your Privacy Settings.

**\*\*Commercial users\*\***: (Team and Enterprise plans, API, 3rd-party platforms, and Claude Gov) maintain existing policies: Anthropic does not train generative models using code or prompts sent to Claude Code under commercial terms, unless the customer has chosen to provide their data to us for model improvement (e.g. [Developer Partner Program](<https://support.anthropic.com/en/articles/11174108-about-the-development-partner-program>)).

### ### Development Partner Program

If you explicitly opt in to methods to provide us with materials to train on, such as via the [Development Partner Program](<https://support.anthropic.com/en/articles/11174108-about-the-development-partner-program>), we may use those materials provided to train our models. An organization admin can expressly opt-in to the Development Partner Program for their organization. Note that this program is available only for Anthropic first-party API, and not for Bedrock or Vertex users.

### ### Feedback using the `/bug` command

If you choose to send us feedback about Claude Code using the `/bug` command, we may use your feedback to improve our products and services. Transcripts shared via `/bug` are retained for 30 days.

### ### Data retention

Anthropic retains Claude Code data based on your account type and preferences.

**\*\*Consumer users (Free, Pro, and Max plans)\*\*:**

- \* Users who allow data use for model improvement: 5-year retention period to support model development and safety improvements
- \* Users who don't allow data use for model improvement: 30-day retention period
- \* Privacy settings can be changed at any time at [\[claude.ai/settings/data-privacy-controls\]](https://claude.ai/settings/data-privacy-controls)([claude.ai/settings/data-privacy-controls](https://claude.ai/settings/data-privacy-controls)).

**\*\*Commercial users (Team, Enterprise, and API)\*\*:**

- \* Standard: 30-day retention period
- \* Zero data retention: Available with appropriately configured API keys - Claude Code will not retain chat transcripts on servers
- \* Local caching: Claude Code clients may store sessions locally for up to 30 days to enable session resumption (configurable)

Learn more about data retention practices in our [\[Privacy Center\]](https://privacy.anthropic.com/)(<https://privacy.anthropic.com/>).

For full details, please review our [\[Commercial Terms of Service\]](https://www.anthropic.com/legal/commercial-terms)(<https://www.anthropic.com/legal/commercial-terms>) (for Team, Enterprise, and API users) or [\[Consumer Terms\]](https://www.anthropic.com/legal/consumer-terms)(<https://www.anthropic.com/legal/consumer-terms>) (for Free, Pro, and Max users) and [\[Privacy Policy\]](https://www.anthropic.com/legal/privacy)(<https://www.anthropic.com/legal/privacy>).

**## Data flow and dependencies**


```

Claude Code is installed from [NPM](<https://www.npmjs.com/package/@anthropic-ai/claude-code>). Claude Code runs locally. In order to interact with the LLM, Claude Code sends data over the network. This data includes all user prompts and model outputs. The data is encrypted in transit via TLS and is not encrypted at rest. Claude Code is compatible with most popular VPNs and LLM proxies.

Claude Code is built on Anthropic's APIs. For details regarding our API's security controls, including our API logging procedures, please refer to compliance artifacts offered in the [Anthropic Trust Center](<https://trust.anthropic.com>).

## ## Telemetry services

Claude Code connects from users' machines to the Statsig service to log operational metrics such as latency, reliability, and usage patterns. This logging does not include any code or file paths. Data is encrypted in transit using TLS and at rest using 256-bit AES encryption. Read more in the [Statsig security documentation](<https://www.statsig.com/trust/security>). To opt out of Statsig telemetry, set the `DISABLE_TELEMETRY` environment variable.

Claude Code connects from users' machines to Sentry for operational error logging. The data is encrypted in transit using TLS and at rest using 256-bit AES encryption. Read more in the [Sentry security documentation](<https://sentry.io/security/>). To opt out of error logging, set the `DISABLE_ERROR_REPORTING` environment variable.

When users run the `/bug` command, a copy of their full conversation history including code is sent to Anthropic. The data is encrypted in transit and at rest. Optionally, a Github issue is created in our public repository. To opt out of bug reporting, set the `DISABLE_BUG_COMMAND` environment variable.

## ## Default behaviors by API provider

By default, we disable all non-essential traffic (including error reporting, telemetry, and bug reporting functionality) when using Bedrock or Vertex. You can also opt out of all of these at once by setting the ``CLAUDE_CODE_DISABLE_NONESSENTIAL_TRAFFIC`` environment variable. Here are the full default behaviors:

| Service                                                                                                                                                                                                             | Anthropic API | Vertex API |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------|------------|
| Bedrock API                                                                                                                                                                                                         |               |            |
| <hr/>                                                                                                                                                                                                               |               |            |
| <hr/>                                                                                                                                                                                                               |               |            |
| <b>**Statsig (Metrics)**</b>                                                                                                                                                                                        |               |            |
| Default on.<br /> <code>`DISABLE_TELEMETRY=1`</code> to disable.   Default off.<br /> <code>`CLAUDE_CODE_USE_VERTEX`</code> must be 1.   Default off.<br /> <code>`CLAUDE_CODE_USE_BEDROCK`</code> must be 1.       |               |            |
| <b>**Sentry (Errors)**</b>                                                                                                                                                                                          |               |            |
| Default on.<br /> <code>`DISABLE_ERROR_REPORTING=1`</code> to disable.   Default off.<br /> <code>`CLAUDE_CODE_USE_VERTEX`</code> must be 1.   Default off.<br /> <code>`CLAUDE_CODE_USE_BEDROCK`</code> must be 1. |               |            |
| <b>**Anthropic API (`/bug` reports)**</b>                                                                                                                                                                           |               |            |
| Default on.<br /> <code>`DISABLE_BUG_COMMAND=1`</code> to disable.   Default off.<br /> <code>`CLAUDE_CODE_USE_VERTEX`</code> must be 1.   Default off.<br /> <code>`CLAUDE_CODE_USE_BEDROCK`</code> must be 1.     |               |            |

All environment variables can be checked into ``settings.json`` ([read more](/en/docs/claude-code/settings)).

## # Data usage

> Learn about Anthropic's data usage policies for Claude

## ## Data policies

### ### Data training policy

**\*\*Consumer users (Free, Pro, and Max plans)\*\*:**

Starting August 28, 2025, we're giving you the choice to allow your data to be used to improve future Claude models.

We will train new models using data from Free, Pro, and Max accounts when this setting is on (including when you use Claude Code from these accounts).

\* If you're a current user, you can select your preference now and your selection will immediately go into effect.

This setting will only apply to new or resumed chats and coding sessions on Claude. Previous chats with no additional activity will not be used for model training.

\* You have until September 28, 2025 to make your selection.

If you're a new user, you can pick your setting for model training during the signup process.

You can change your selection at any time in your Privacy Settings.

**\*\*Commercial users\*\***: (Team and Enterprise plans, API, 3rd-party platforms, and Claude Gov) maintain existing policies: Anthropic does not train generative models using code or prompts sent to Claude Code under commercial terms, unless the customer has chosen to provide their data to us for model improvement (e.g. [Developer Partner Program](<https://support.anthropic.com/en/articles/11174108-about-the-development-partner-program>)).

### ### Development Partner Program

If you explicitly opt in to methods to provide us with materials to train on, such as via the [Development Partner Program](<https://support.anthropic.com/en/articles/11174108-about-the-development-partner-program>), we may use those materials provided to train our models. An organization admin can expressly opt-in to the Development Partner Program for their organization. Note that this program is available only for Anthropic first-party API, and not for Bedrock or Vertex users.

### ### Feedback using the `/bug` command

If you choose to send us feedback about Claude Code using the `/bug` command, we may use your feedback to improve our products and services. Transcripts shared via `/bug` are retained for 30 days.

### ### Data retention

Anthropic retains Claude Code data based on your account type and preferences.



**\*\*Consumer users (Free, Pro, and Max plans)\*\*:**

- \* Users who allow data use for model improvement: 5-year retention period to support model development and safety improvements
- \* Users who don't allow data use for model improvement: 30-day retention period
- \* Privacy settings can be changed at any time at [\[claude.ai/settings/data-privacy-controls\]](https://claude.ai/settings/data-privacy-controls)([claude.ai/settings/data-privacy-controls](https://claude.ai/settings/data-privacy-controls)).

**\*\*Commercial users (Team, Enterprise, and API)\*\*:**

- \* Standard: 30-day retention period
- \* Zero data retention: Available with appropriately configured API keys - Claude Code will not retain chat transcripts on servers
- \* Local caching: Claude Code clients may store sessions locally for up to 30 days to enable session resumption (configurable)

Learn more about data retention practices in our [\[Privacy Center\]](https://privacy.anthropic.com/)(<https://privacy.anthropic.com/>).

For full details, please review our [\[Commercial Terms of Service\]](https://www.anthropic.com/legal/commercial-terms)(<https://www.anthropic.com/legal/commercial-terms>) (for Team, Enterprise, and API users) or [\[Consumer Terms\]](https://www.anthropic.com/legal/consumer-terms)(<https://www.anthropic.com/legal/consumer-terms>) (for Free, Pro, and Max users) and [\[Privacy Policy\]](https://www.anthropic.com/legal/privacy)(<https://www.anthropic.com/legal/privacy>).

**## Data flow and dependencies**



```
o=format&n=PF_69UDRSEsLpN9D&q=85&s=a25ba8e1c632bb02de4cf68e96ac5a8c 840w,  
https://mintcdn.com/anthropic/PF_69UDRSEsLpN9D/images/claude-code-data-flow.png?w=1100&fit=max&au  
to=format&n=PF_69UDRSEsLpN9D&q=85&s=434fb120de78f63df663268636485646 1100w,  
https://mintcdn.com/anthropic/PF_69UDRSEsLpN9D/images/claude-code-data-flow.png?w=1650&fit=max&au  
to=format&n=PF_69UDRSEsLpN9D&q=85&s=9baeb74ab4c1c8255e510c2c8b521e32 1650w,  
https://mintcdn.com/anthropic/PF_69UDRSEsLpN9D/images/claude-code-data-flow.png?w=2500&fit=max&au  
to=format&n=PF_69UDRSEsLpN9D&q=85&s=f4314f4067f037b57fe851d063ac2b77 2500w"  
data-optimize="true" data-opv="2" />
```

Claude Code is installed from [NPM](<https://www.npmjs.com/package/@anthropic-ai/claude-code>). Claude Code runs locally. In order to interact with the LLM, Claude Code sends data over the network. This data includes all user prompts and model outputs. The data is encrypted in transit via TLS and is not encrypted at rest. Claude Code is compatible with most popular VPNs and LLM proxies.

Claude Code is built on Anthropic's APIs. For details regarding our API's security controls, including our API logging procedures, please refer to compliance artifacts offered in the [Anthropic Trust Center](<https://trust.anthropic.com>).

## ## Telemetry services

Claude Code connects from users' machines to the Statsig service to log operational metrics such as latency, reliability, and usage patterns. This logging does not include any code or file paths. Data is encrypted in transit using TLS and at rest using 256-bit AES encryption. Read more in the [Statsig security documentation](<https://www.statsig.com/trust/security>). To opt out of Statsig telemetry, set the `DISABLE_TELEMETRY` environment variable.

Claude Code connects from users' machines to Sentry for operational error logging. The data is encrypted in transit using TLS and at rest using 256-bit AES encryption. Read more in the [Sentry security documentation](<https://sentry.io/security/>). To opt out of error logging, set the `DISABLE_ERROR_REPORTING` environment variable.

When users run the `/bug` command, a copy of their full conversation history including code is sent to Anthropic. The data is encrypted in transit and at rest. Optionally, a Github issue is created in our public repository. To opt out of bug reporting, set the `DISABLE_BUG_COMMAND` environment variable.

## ## Default behaviors by API provider

By default, we disable all non-essential traffic (including error reporting, telemetry, and bug reporting functionality) when using Bedrock or Vertex. You can also opt out of all of these at once by setting the `CLAUDE_CODE_DISABLE_NONESSENTIAL_TRAFFIC` environment variable. Here are the full default behaviors:

| Service                                   | Anthropic API                                                                                                                                                       | Vertex API                                                                                                 |
|-------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------|
| Bedrock API                               |                                                                                                                                                                     |                                                                                                            |
| -----                                     |                                                                                                                                                                     |                                                                                                            |
| -----                                     |                                                                                                                                                                     |                                                                                                            |
| <b>**Statsig (Metrics)**</b>              | Default on.<br>`DISABLE_TELEMETRY=1` to disable.                                                                                                                    | Default off.<br>`CLAUDE_CODE_USE_VERTEX` must be 1.   Default off.<br>`CLAUDE_CODE_USE_BEDROCK` must be 1. |
| <b>**Sentry (Errors)**</b>                | Default on.<br>`DISABLE_ERROR_REPORTING=1` to disable.   Default off.<br>`CLAUDE_CODE_USE_VERTEX` must be 1.   Default off.<br>`CLAUDE_CODE_USE_BEDROCK` must be 1. |                                                                                                            |
| <b>**Anthropic API (`/bug` reports)**</b> | Default on.<br>`DISABLE_BUG_COMMAND=1` to disable.   Default off.<br>`CLAUDE_CODE_USE_VERTEX` must be 1.   Default off.<br>`CLAUDE_CODE_USE_BEDROCK` must be 1.     |                                                                                                            |

All environment variables can be checked into `settings.json` ([\[read more\]](/en/docs/claude-code/settings)).

## # Status line configuration

> Create a custom status line for Claude Code to display contextual information

Make Claude Code your own with a custom status line that displays at the bottom of the Claude Code interface, similar to how terminal prompts (PS1) work in shells like Oh-my-zsh.

## ## Create a custom status line

You can either:

\* Run ``/statusline`` to ask Claude Code to help you set up a custom status line. By default, it will try to reproduce your terminal's prompt, but you can provide additional instructions about the behavior you want to Claude Code, such as ``/statusline show the model name in orange``

\* Directly add a ``statusLine`` command to your ``.claude/settings.json``:

```
```json
{
  "statusLine": {
    "type": "command",
    "command": "~/claude/statusline.sh",
    "padding": 0 // Optional: set to 0 to let status line go to edge
  }
}
```
```

## ## How it Works

- \* The status line is updated when the conversation messages update
- \* Updates run at most every 300ms
- \* The first line of stdout from your command becomes the status line text
- \* ANSI color codes are supported for styling your status line
- \* Claude Code passes contextual information about the current session (model, directories, etc.) as JSON to your script via stdin

## ## JSON Input Structure

Your status line command receives structured data via stdin in JSON format:

```
```json
{
  "hook_event_name": "Status",
  "session_id": "abc123...",
  "transcript_path": "/path/to/transcript.json",
  "cwd": "/current/working/directory",
  "model": {
    "id": "claude-opus-4-1",
    "display_name": "Opus"
  },
  "workspace": {
    "current_dir": "/current/working/directory",
    "project_dir": "/original/project/directory"
  },
  "version": "1.0.80",
  "output_style": {
    "name": "default"
  },
  "cost": {
    "total_cost_usd": 0.01234,
    "total_duration_ms": 45000,
    "total_api_duration_ms": 2300,
    "total_lines_added": 156,
    "total_lines_removed": 23
  }
}
```
```

## ## Example Scripts

### ### Simple Status Line

```
```bash

#!/bin/bash

# Read JSON input from stdin

input=$(cat)


# Extract values using jq

MODEL_DISPLAY=$(echo "$input" | jq -r '.model.display_name')

CURRENT_DIR=$(echo "$input" | jq -r '.workspace.current_dir')


echo "[$MODEL_DISPLAY] 📁 ${CURRENT_DIR##*/}"

```
```

### ### Git-Aware Status Line

```
```bash

#!/bin/bash

# Read JSON input from stdin

input=$(cat)


# Extract values using jq

MODEL_DISPLAY=$(echo "$input" | jq -r '.model.display_name')

CURRENT_DIR=$(echo "$input" | jq -r '.workspace.current_dir')
```

```
# Show git branch if in a git repo
```

```
GIT_BRANCH=""
```

```
if git rev-parse --git-dir > /dev/null 2>&1; then
```

```
    BRANCH=$(git branch --show-current 2>/dev/null)
```

```
    if [ -n "$BRANCH" ]; then
```

```
        GIT_BRANCH=" | 🌿 $BRANCH"
```

```
    fi
```

```
fi
```

```
echo "[$MODEL_DISPLAY] 📁 ${CURRENT_DIR##*/}$GIT_BRANCH"
```

```
...
```

```
### Python Example
```

```
```python
```

```
#!/usr/bin/env python3
```

```
import json
```

```
import sys
```

```
import os
```

```
# Read JSON from stdin
```

```
data = json.load(sys.stdin)
```

```
# Extract values
```

```
model = data["model"]["display_name"]
```

```
current_dir = os.path.basename(data["workspace"]["current_dir"])
```

```

# Check for git branch

git_branch = ""

if os.path.exists('.git'):

    try:

        with open('.git/HEAD', 'r') as f:

            ref = f.read().strip()

            if ref.startswith('ref: refs/heads/'):

                git_branch = f" | 🌿 {ref.replace('ref: refs/heads/', '')}"

    except:

        pass

print(f"[{model}] 📁 {current_dir}{git_branch}")

```

```

...

```

### ### Node.js Example

```

```javascript

#!/usr/bin/env node

const fs = require('fs');

const path = require('path');

// Read JSON from stdin

let input = "";

process.stdin.on('data', chunk => input += chunk);

process.stdin.on('end', () => {

```



```

const data = JSON.parse(input);

// Extract values

const model = data.model.display_name;

const currentDir = path.basename(data.workspace.current_dir);


// Check for git branch

let gitBranch = "";

try {

    const headContent = fs.readFileSync('.git/HEAD', 'utf8').trim();

    if (headContent.startsWith('ref: refs/heads/')) {

        gitBranch = ` | 🌿 ${headContent.replace('ref: refs/heads/', '')}`;

    }

} catch (e) {

    // Not a git repo or can't read HEAD

}


console.log(`[${model}] 📁 ${currentDir}${gitBranch}`);

});

...

```

### ### Helper Function Approach

For more complex bash scripts, you can create helper functions:

```

```bash

#!/bin/bash

```

**# Read JSON input once**

**input=\$(cat)**

**# Helper functions for common extractions**

**get\_model\_name() { echo "\$input" | jq -r '.model.display\_name'; }**

**get\_current\_dir() { echo "\$input" | jq -r '.workspace.current\_dir'; }**

**get\_project\_dir() { echo "\$input" | jq -r '.workspace.project\_dir'; }**

**get\_version() { echo "\$input" | jq -r '.version'; }**

**get\_cost() { echo "\$input" | jq -r '.cost.total\_cost\_usd'; }**

**get\_duration() { echo "\$input" | jq -r '.cost.total\_duration\_ms'; }**

**get\_lines\_added() { echo "\$input" | jq -r '.cost.total\_lines\_added'; }**

**get\_lines\_removed() { echo "\$input" | jq -r '.cost.total\_lines\_removed'; }**

**# Use the helpers**

**MODEL=\$(get\_model\_name)**

**DIR=\$(get\_current\_dir)**

**echo "[\$MODEL] 📁 \${DIR##\*/}"**

**...**

**## Tips**

**\* Keep your status line concise - it should fit on one line**

**\* Use emojis (if your terminal supports them) and colors to make information scannable**

**\* Use `jq` for JSON parsing in Bash (see examples above)**

**\* Test your script by running it manually with mock JSON input: `echo  
'{"model":{"display\_name":"Test"},"workspace":{"current\_dir":"/test"}}' | ./statusline.sh`**

**\* Consider caching expensive operations (like git status) if needed**

## ## Troubleshooting

- \* If your status line doesn't appear, check that your script is executable (`chmod +x`)
- \* Ensure your script outputs to stdout (not stderr)

## # Status line configuration

> Create a custom status line for Claude Code to display contextual information

Make Claude Code your own with a custom status line that displays at the bottom of the Claude Code interface, similar to how terminal prompts (PS1) work in shells like Oh-my-zsh.

## ## Create a custom status line

You can either:

\* Run `/statusline` to ask Claude Code to help you set up a custom status line. By default, it will try to reproduce your terminal's prompt, but you can provide additional instructions about the behavior you want to Claude Code, such as `/statusline show the model name in orange`

\* Directly add a `statusLine` command to your `.claude/settings.json`:

```
```json
{
  "statusLine": {
    "type": "command",
    "command": "~/claude/statusline.sh",
    "padding": 0 // Optional: set to 0 to let status line go to edge
  }
}
```

```
}  
...
```

## ## How it Works

- \* The status line is updated when the conversation messages update
- \* Updates run at most every 300ms
- \* The first line of stdout from your command becomes the status line text
- \* ANSI color codes are supported for styling your status line
- \* Claude Code passes contextual information about the current session (model, directories, etc.) as JSON to your script via stdin

## ## JSON Input Structure

Your status line command receives structured data via stdin in JSON format:

```
```json  
{  
  "hook_event_name": "Status",  
  "session_id": "abc123...",  
  "transcript_path": "/path/to/transcript.json",  
  "cwd": "/current/working/directory",  
  "model": {  
    "id": "claude-opus-4-1",  
    "display_name": "Opus"  
  },  
  "workspace": {  
    "current_dir": "/current/working/directory",
```

```
    "project_dir": "/original/project/directory"
  },
  "version": "1.0.80",
  "output_style": {
    "name": "default"
  },
  "cost": {
    "total_cost_usd": 0.01234,
    "total_duration_ms": 45000,
    "total_api_duration_ms": 2300,
    "total_lines_added": 156,
    "total_lines_removed": 23
  }
}
...

```

## ## Example Scripts

### ### Simple Status Line

```
```bash

#!/bin/bash

# Read JSON input from stdin
input=$(cat)

# Extract values using jq
MODEL_DISPLAY=$(echo "$input" | jq -r '.model.display_name')

```

```
CURRENT_DIR=$(echo "$input" | jq -r '.workspace.current_dir')
```

```
echo "[$MODEL_DISPLAY] 📁 ${CURRENT_DIR##*/}"
```

```
...
```

```
### Git-Aware Status Line
```

```
```bash
```

```
#!/bin/bash
```

```
# Read JSON input from stdin
```

```
input=$(cat)
```

```
# Extract values using jq
```

```
MODEL_DISPLAY=$(echo "$input" | jq -r '.model.display_name')
```

```
CURRENT_DIR=$(echo "$input" | jq -r '.workspace.current_dir')
```

```
# Show git branch if in a git repo
```

```
GIT_BRANCH=""
```

```
if git rev-parse --git-dir > /dev/null 2>&1; then
```

```
    BRANCH=$(git branch --show-current 2>/dev/null)
```

```
    if [ -n "$BRANCH" ]; then
```

```
        GIT_BRANCH=" | 🌿 $BRANCH"
```

```
    fi
```

```
fi
```

```
echo "[$MODEL_DISPLAY] 📁 ${CURRENT_DIR##*/}$GIT_BRANCH"
```

```
...
```

### ### Python Example

```
```python

#!/usr/bin/env python3

import json

import sys

import os


# Read JSON from stdin

data = json.load(sys.stdin)


# Extract values

model = data['model']['display_name']

current_dir = os.path.basename(data['workspace']['current_dir'])


# Check for git branch

git_branch = ""

if os.path.exists('.git'):

    try:

        with open('.git/HEAD', 'r') as f:

            ref = f.read().strip()

            if ref.startswith('ref: refs/heads/'):

                git_branch = f" | 🌿 {ref.replace('ref: refs/heads/', '')}"

    except:

        pass

```
```

```
print(f"[{model}] 📁 {current_dir}{git_branch}")
```

```
...
```

### ### Node.js Example

```
```javascript
```

```
#!/usr/bin/env node
```

```
const fs = require('fs');
```

```
const path = require('path');
```

```
// Read JSON from stdin
```

```
let input = "";
```

```
process.stdin.on('data', chunk => input += chunk);
```

```
process.stdin.on('end', () => {
```

```
    const data = JSON.parse(input);
```

```
    // Extract values
```

```
    const model = data.model.display_name;
```

```
    const currentDir = path.basename(data.workspace.current_dir);
```

```
    // Check for git branch
```

```
    let gitBranch = "";
```

```
    try {
```

```
        const headContent = fs.readFileSync('.git/HEAD', 'utf8').trim();
```

```
        if (headContent.startsWith('ref: refs/heads/')) {
```

```
            gitBranch = ` | 🌿 ${headContent.replace('ref: refs/heads/', '')}`;
```



```

    }

    } catch (e) {

        // Not a git repo or can't read HEAD

    }

    console.log(`${model}] 📁 ${currentDir}${gitBranch}`);

});

...

```

### ### Helper Function Approach

For more complex bash scripts, you can create helper functions:

```

```bash

#!/bin/bash

# Read JSON input once

input=$(cat)

# Helper functions for common extractions

get_model_name() { echo "$input" | jq -r '.model.display_name'; }

get_current_dir() { echo "$input" | jq -r '.workspace.current_dir'; }

get_project_dir() { echo "$input" | jq -r '.workspace.project_dir'; }

get_version() { echo "$input" | jq -r '.version'; }

get_cost() { echo "$input" | jq -r '.cost.total_cost_usd'; }

get_duration() { echo "$input" | jq -r '.cost.total_duration_ms'; }

get_lines_added() { echo "$input" | jq -r '.cost.total_lines_added'; }

get_lines_removed() { echo "$input" | jq -r '.cost.total_lines_removed'; }

```

**# Use the helpers**

**MODEL=\$(get\_model\_name)**

**DIR=\$(get\_current\_dir)**

**echo "[\$MODEL] 📁 \${DIR##\*/}"**

**...**

## **## Tips**

- \* Keep your status line concise - it should fit on one line**
- \* Use emojis (if your terminal supports them) and colors to make information scannable**
- \* Use `jq` for JSON parsing in Bash (see examples above)**
- \* Test your script by running it manually with mock JSON input: `echo '{"model":{"display\_name":"Test"},"workspace":{"current\_dir":"/test"}}' | ./statusline.sh`**
- \* Consider caching expensive operations (like git status) if needed**

## **## Troubleshooting**

- \* If your status line doesn't appear, check that your script is executable (`chmod +x`)**
- \* Ensure your script outputs to stdout (not stderr)**

## **# Interactive mode**

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

## **## Keyboard shortcuts**

### General controls

Shortcut	Description	Context
⌘+C	Cancel current input or generation	Standard interrupt
⌘+D	Exit Claude Code session	EOF signal
⌘+L	Clear terminal screen	Keeps conversation history
↑/↓	Navigate command history	Recall previous inputs
Esc + Esc	Edit previous message	Double-escape to modify
⇧+Tab	Toggle permission modes   Switch between Auto-Accept Mode, Plan Mode, and normal mode	

### Multiline input

Method	Shortcut	Context
Quick escape	⌨ + Enter	Works in all terminals
macOS default	⌥+Enter	Default on macOS
Terminal setup	⇧+Enter	After /terminal-setup
Control sequence	⌘+J	Line feed character for multiline
Paste mode		Paste directly   For code blocks, logs

<Tip>

Configure your preferred line break behavior in terminal settings. Run `/terminal-setup` to install Shift+Enter binding for iTerm2 and VS Code terminals.

</Tip>

### Quick commands

Shortcut	Description	Notes
<code>`#`</code> at start	Memory shortcut - add to CLAUDE.md	Prompts for file selection
<code>`/`</code> at start	Slash command	See [slash commands](/en/docs/claude-code/slash-commands)
<code>`!`</code> at start	Bash mode	Run commands directly and add execution output to the session

## Vim editor mode

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

### Mode switching

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

### Navigation (NORMAL mode)

Command	Action
<code>:</code>	

| `h j k l` | Move left/down/up/right |

| `w` | Next word |

| `e` | End of word |

| `b` | Previous word |

| `0` | Beginning of line |

| `$` | End of line |

| `^` | First non-blank character |

| `gg` | Beginning of input |

| `G` | End of input |

### ### Editing (NORMAL mode)

| Command | Action |

| :----- | :----- |

| `x` | Delete character |

| `dd` | Delete line |

| `D` | Delete to end of line |

| `dw / de / db` | Delete word/to end/back |

| `cc` | Change line |

| `C` | Change to end of line |

| `cw / ce / cb` | Change word/to end/back |

| `.` | Repeat last change |

### ## Command history

Claude Code maintains command history for the current session:

- \* History is stored per working directory
- \* Cleared with ``/clear`` command
- \* Use Up/Down arrows to navigate (see keyboard shortcuts above)
- \* **Ctrl+R**: Reverse search through history (if supported by terminal)
- \* **Note**: History expansion (``!``) is disabled by default

## See also

- \* [\[Slash commands\]\(/en/docs/claude-code/slash-commands\)](#) - Interactive session commands
- \* [\[CLI reference\]\(/en/docs/claude-code/cli-reference\)](#) - Command-line flags and options
- \* [\[Settings\]\(/en/docs/claude-code/settings\)](#) - Configuration options
- \* [\[Memory management\]\(/en/docs/claude-code/memory\)](#) - Managing CLAUDE.md files

# Interactive mode

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

## Keyboard shortcuts

### General controls

Shortcut	Description	Context	
:-----	:-----	:-----	
<code>`Ctrl+C`</code>	Cancel current input or generation	Standard interrupt	
<code>`Ctrl+D`</code>	Exit Claude Code session	EOF signal	
<code>`Ctrl+L`</code>	Clear terminal screen	Keeps conversation history	
<code>`Up/Down arrows`</code>	Navigate command history	Recall previous inputs	

<code>`Esc` + `Esc`</code>	Edit previous message	Double-escape to modify	
<code>`Shift+Tab`</code>	Toggle permission modes	Switch between Auto-Accept Mode, Plan Mode, and normal mode	

### Multiline input

Method	Shortcut	Context	
:-----	:-----	:-----	
Quick escape	<code>`\` + `Enter`</code>	Works in all terminals	
macOS default	<code>`Option+Enter`</code>	Default on macOS	
Terminal setup	<code>`Shift+Enter`</code>	After <code>`/terminal-setup`</code>	
Control sequence	<code>`Ctrl+J`</code>	Line feed character for multiline	
Paste mode	Paste directly	For code blocks, logs	

<Tip>

Configure your preferred line break behavior in terminal settings. Run ``/terminal-setup`` to install `Shift+Enter` binding for iTerm2 and VS Code terminals.

</Tip>

### Quick commands

Shortcut	Description	Notes	
:-----	:-----	:-----	
<code>`#`</code> at start	Memory shortcut - add to CLAUDE.md	Prompts for file selection	
<code>`/`</code> at start	Slash command	See [slash commands](/en/docs/claude-code/slash-commands)	
<code>`!`</code> at start	Bash mode	Run commands directly and add execution output to the session	

## Vim editor mode

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

### ### Mode switching

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

### ### Navigation (NORMAL mode)

Command	Action
<code>`h`/`j`/`k`/`l`</code>	Move left/down/up/right
<code>`w`</code>	Next word
<code>`e`</code>	End of word
<code>`b`</code>	Previous word
<code>`0`</code>	Beginning of line
<code>`\$`</code>	End of line
<code>`^`</code>	First non-blank character
<code>`gg`</code>	Beginning of input



`G`	End of input	
-----	--------------	--

### ### Editing (NORMAL mode)

Command	Action	
---------	--------	--

:-----	:-----	
--------	--------	--

`x`	Delete character	
-----	------------------	--

`dd`	Delete line	
------	-------------	--

`D`	Delete to end of line	
-----	-----------------------	--

`dw`/`de`/`db`	Delete word/to end/back	
----------------	-------------------------	--

`cc`	Change line	
------	-------------	--

`C`	Change to end of line	
-----	-----------------------	--

`cw`/`ce`/`cb`	Change word/to end/back	
----------------	-------------------------	--

``	Repeat last change	
----	--------------------	--

### ## Command history

Claude Code maintains command history for the current session:

\* History is stored per working directory

\* Cleared with `/clear` command

\* Use Up/Down arrows to navigate (see keyboard shortcuts above)

\* \*\*Ctrl+R\*\*: Reverse search through history (if supported by terminal)

\* \*\*Note\*\*: History expansion (!) is disabled by default

## See also

- \* [\[Slash commands\]](/en/docs/claude-code/slash-commands) - Interactive session commands
- \* [\[CLI reference\]](/en/docs/claude-code/cli-reference) - Command-line flags and options
- \* [\[Settings\]](/en/docs/claude-code/settings) - Configuration options
- \* [\[Memory management\]](/en/docs/claude-code/memory) - Managing CLAUDE.md files

# Interactive mode

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

## Keyboard shortcuts

### General controls

Shortcut	Description	Context
⌘+C	Cancel current input or generation	Standard interrupt
⌘+D	Exit Claude Code session	EOF signal
⌘+L	Clear terminal screen	Keeps conversation history
↑/↓	Navigate command history	Recall previous inputs
Esc + Esc	Edit previous message	Double-escape to modify
Shift+Tab	Toggle permission modes	Switch between Auto-Accept Mode, Plan Mode, and normal mode

### Multiline input

Method	Shortcut	Context

Quick escape	`` + `Enter`	Works in all terminals	
macOS default	`Option+Enter`	Default on macOS	
Terminal setup	`Shift+Enter`	After `/terminal-setup`	
Control sequence	`Ctrl+J`	Line feed character for multiline	
Paste mode	Paste directly	For code blocks, logs	

<Tip>

Configure your preferred line break behavior in terminal settings. Run `/terminal-setup` to install Shift+Enter binding for iTerm2 and VS Code terminals.

</Tip>

### ### Quick commands

Shortcut	Description	Notes	
:-----	:-----	:-----	
`#` at start	Memory shortcut - add to CLAUDE.md	Prompts for file selection	
`/` at start	Slash command	See [slash commands](/en/docs/claude-code/slash-commands)	
`!` at start	Bash mode	Run commands directly and add execution output to the session	

### ## Vim editor mode

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

### ### Mode switching

Command	Action	From mode	
:-----	:-----	:-----	

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

### ### Navigation (NORMAL mode)

Command	Action	
:-----	:-----	
`h`/`j`/`k`/`l`	Move left/down/up/right	
`w`	Next word	
`e`	End of word	
`b`	Previous word	
`0`	Beginning of line	
`\$`	End of line	
`^`	First non-blank character	
`gg`	Beginning of input	
`G`	End of input	

### ### Editing (NORMAL mode)

Command	Action	
:-----	:-----	
`x`	Delete character	

<code>`dd`</code>	Delete line	
<code>`D`</code>	Delete to end of line	
<code>`dw`/`de`/`db`</code>	Delete word/to end/back	
<code>`cc`</code>	Change line	
<code>`C`</code>	Change to end of line	
<code>`cw`/`ce`/`cb`</code>	Change word/to end/back	
<code>`.`</code>	Repeat last change	

## ## Command history

Claude Code maintains command history for the current session:

- \* History is stored per working directory
- \* Cleared with ``/clear`` command
- \* Use Up/Down arrows to navigate (see keyboard shortcuts above)
- \* **Ctrl+R**: Reverse search through history (if supported by terminal)
- \* **Note**: History expansion (``!``) is disabled by default

## ## See also

- \* [\[Slash commands\]](/en/docs/claude-code/slash-commands) - Interactive session commands
- \* [\[CLI reference\]](/en/docs/claude-code/cli-reference) - Command-line flags and options
- \* [\[Settings\]](/en/docs/claude-code/settings) - Configuration options
- \* [\[Memory management\]](/en/docs/claude-code/memory) - Managing CLAUDE.md files

## # Interactive mode

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

## Keyboard shortcuts

### General controls

Shortcut	Description	Context
⌘+C	Cancel current input or generation	Standard interrupt
⌘+D	Exit Claude Code session	EOF signal
⌘+L	Clear terminal screen	Keeps conversation history
↑/↓	Navigate command history	Recall previous inputs
Esc + Esc	Edit previous message	Double-escape to modify
Shift+Tab	Toggle permission modes	Switch between Auto-Accept Mode, Plan Mode, and normal mode

### Multiline input

Method	Shortcut	Context
Quick escape	⌨ + Enter	Works in all terminals
macOS default	Option+Enter	Default on macOS
Terminal setup	Shift+Enter	After /terminal-setup
Control sequence	Ctrl+J	Line feed character for multiline
Paste mode	Paste directly	For code blocks, logs

<Tip>

Configure your preferred line break behavior in terminal settings. Run ``/terminal-setup`` to install Shift+Enter binding for iTerm2 and VS Code terminals.

</Tip>

### Quick commands

Shortcut	Description	Notes
:-----   :-----   :-----		
<code>`#`</code> at start	Memory shortcut - add to CLAUDE.md	Prompts for file selection
<code>`/`</code> at start	Slash command	See [slash commands](/en/docs/claude-code/slash-commands)
<code>`!`</code> at start	Bash mode	Run commands directly and add execution output to the session

## Vim editor mode

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

### Mode switching

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

### ### Navigation (NORMAL mode)

Command	Action
:	:
h/j/k/l	Move left/down/up/right
w	Next word
e	End of word
b	Previous word
0	Beginning of line
\$	End of line
^	First non-blank character
gg	Beginning of input
G	End of input

### ### Editing (NORMAL mode)

Command	Action
:	:
x	Delete character
dd	Delete line
D	Delete to end of line
dw/de/db	Delete word/to end/back
cc	Change line
C	Change to end of line
cw/ce/cb	Change word/to end/back
.	Repeat last change



## ## Command history

Claude Code maintains command history for the current session:

- \* History is stored per working directory
- \* Cleared with ``/clear`` command
- \* Use Up/Down arrows to navigate (see keyboard shortcuts above)
- \* **Ctrl+R**: Reverse search through history (if supported by terminal)
- \* **Note**: History expansion (``!``) is disabled by default

## ## See also

- \* [\[Slash commands\]\(/en/docs/claude-code/slash-commands\)](/en/docs/claude-code/slash-commands) - Interactive session commands
- \* [\[CLI reference\]\(/en/docs/claude-code/cli-reference\)](/en/docs/claude-code/cli-reference) - Command-line flags and options
- \* [\[Settings\]\(/en/docs/claude-code/settings\)](/en/docs/claude-code/settings) - Configuration options
- \* [\[Memory management\]\(/en/docs/claude-code/memory\)](/en/docs/claude-code/memory) - Managing CLAUDE.md files