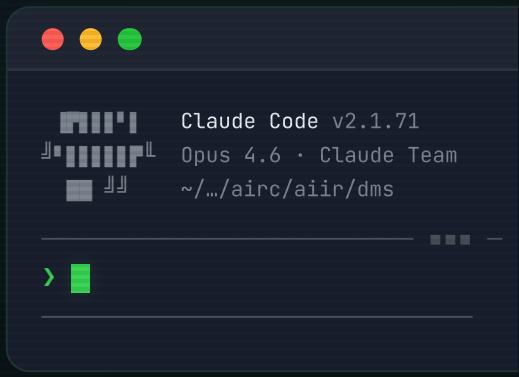
$ claude --help

CLAUDE CODE

From installation to common workflows - everything our team needs to get started

INTERNAL ENGINEERING ENABLEMENT



Agenda

1	Introduction & Interfaces	What CC is and where you run it
2	Core Toolbox & Configuration	CLAUDE.md, MCP, hooks, skills, permissions
3	Session & CLI Management	Starting, resuming, undoing mistakes
4	Context Management	The most critical constraint
5	Workflows	Explore > Plan > Code > Commit
6	The Mental Model	Orchestrating intelligence, not prompting a bot
7	Company Infrastructure	Our commands, skills, and MCP setup
8	Power User Topics	Hooks, plugins, subagents, worktrees, swarms

SECTION 1 -- INTRODUCTION

Not a chatbot. An agent.

- Lives directly in your workspace -- terminal, IDE, or browser
- Has direct file system access and executes commands in real-time
- Integrates natively with version control (git)
- Integrates well with the shell -- understands Linux, bash, etc.
- Operates in an agentic loop: explores > acts > verifies > repeats
- More like a junior developer sitting next to you than a chatbot

	Claude Code	Chatbot
Files	Edits directly	Copy-paste snippets
Commands	Runs tests	Describes what to run
Codebase	Full context	Paste-limited
Behavior	Agentic loop	Single turn
Git	Commit-ready	No VCS

SECTION 1 -- INTERFACES

Terminal (CLI) CORE

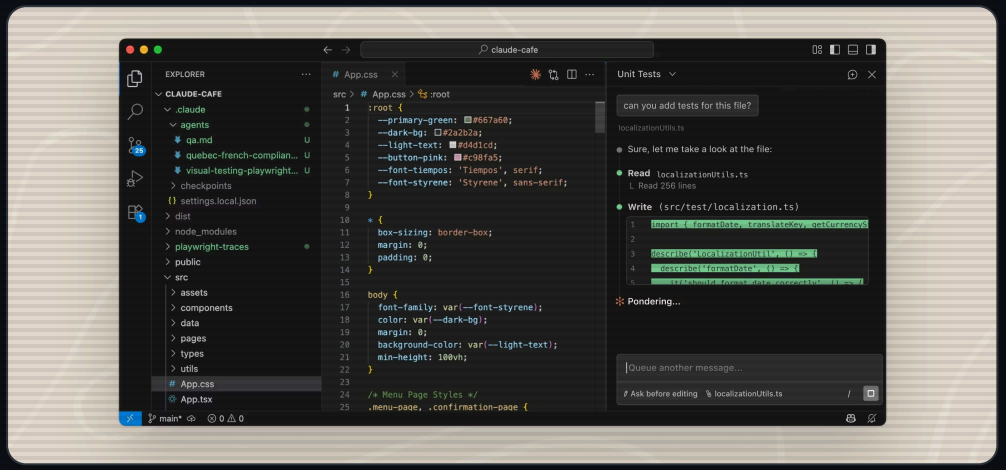
The full-featured environment. Every feature is available here first. Install via `bun` or `npm` and run natively alongside your existing tools. IDE aware.

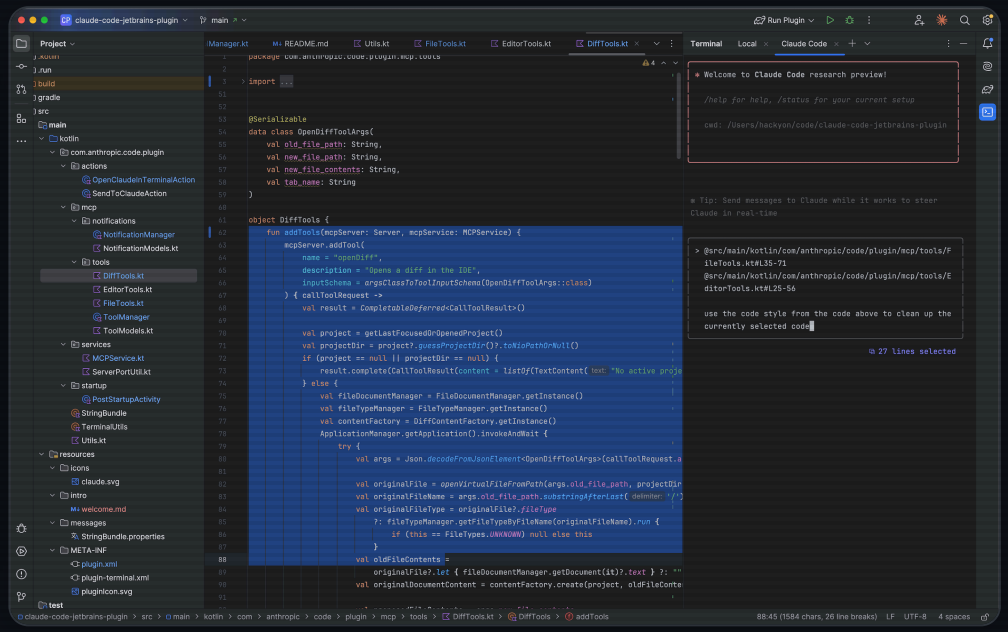
SECTION 1 -- INTERFACES

IDE Extensions

IDE

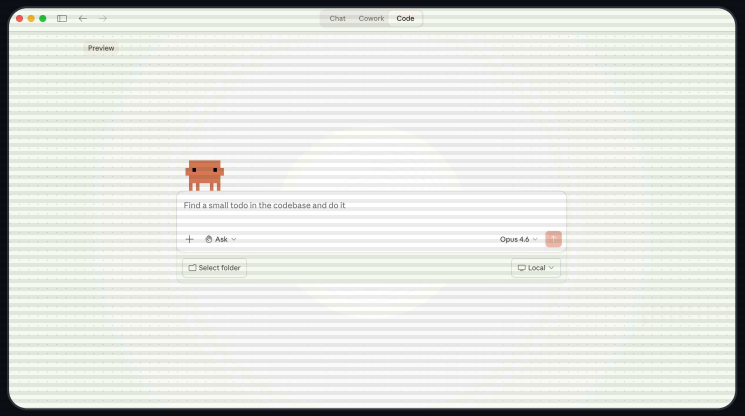
Inline diffs and shared context right inside VS Code and JetBrains.
See Claude's changes as diff hunks.



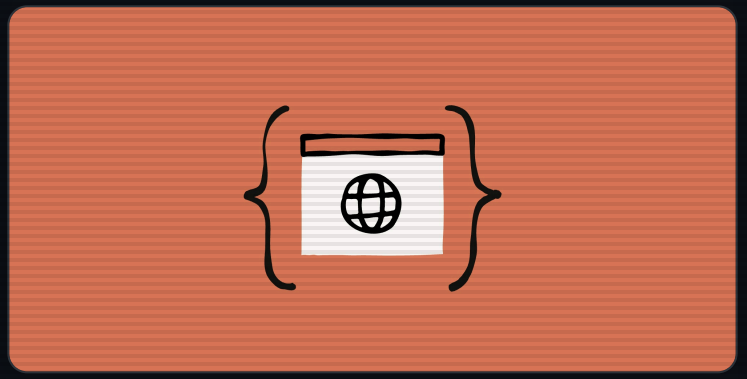


SECTION 1 -- INTERFACES

Desktop, Web & Mobile



STANDALONE Desktop



REMOTE Web

REMOTE Mobile

SECTION 2 -- CONFIGURATION

CLAUDE.md (Project Memory)

- A persistent markdown file loaded at the start of every session
- Defines project-wide coding styles, architecture choices, and common commands
- Lives in your repo root -- commit it so the whole team benefits
- Exists at three levels: global (`~/.claude/`), project root, or subdirectory
- Possible to use a hierarchy of `CLAUDE.md` files inside a project

Think of it as the onboarding doc you write for Claude. Everything you'd tell a new team member -- put it here.

SECTION 2 -- CONFIGURATION

Settings Hierarchy

User

`~/.claude/settings.js`

Your global defaults -- applies to all projects on your machine

Project

`.claude/settings.json`

Team standards, committed to repo -- shared across the team

Local

`.claude/settings.local.`

Your machine overrides, **.gitignored** -- personal customization

Later levels override earlier ones. Project settings are version-controlled -- everyone on the team gets the same baseline.

Common settings: allowed tools, default model, API keys, MCP server configs.

SECTION 2 -- CONFIGURATION

Permissions

- Claude asks before dangerous operations by default
- `/permissions` -- allowlist specific tools Claude can use without asking
- You can restrict: file write access, shell execution, network calls
- Team can set project-level permissions that apply to everyone

Level	Behavior
Allow	Auto-approve -- no prompt shown
Ask	Prompt before each use (default)
Deny	Block entirely -- tool unavailable

Granular control: allow `Bash(npm test)` but deny `Bash(rm -rf)`

Permissions can be updated on demand -- no need to predefine them.

SECTION 2 -- CONFIGURATION

MCP Servers (Model Context Protocol)

- Universal adapters connecting Claude to external tools and data sources
- Each MCP server exposes "tools" that Claude can call like native capabilities
- Add with: `claude mcp add <name> <command>`
- In many cases a CLI is better -- `glab` for example

Service	What it connects
GitHub / GitLab	PRs, issues, code search
Linear / Jira	Issue tracking, sprint data
Postgres / DBs	Schema queries, data lookups
Slack	Channel messages, notifications
Internal APIs	Custom business logic

Configure in `.mcp.json` (project) or `~/.claude/mcp.json` (global)

MCP servers are token-hungry -- watch your context fill rate when several are connected.

Commands & Skills

Slash Commands

- Markdown files in `.claude/commands/`
- Invoked explicitly with `/command-name`
- Accept arguments via `$ARGUMENTS`
- Examples: `/review`, `/deploy`, `/fix-lint`

Skills

- Markdown files in `.claude/skills/` providing passive expertise
- E.g. Python developer, database expert, coding standards
- Auto-discovered and applied by task context
- Force-invoke with `/skill-name` when needed

	Commands
Trigger	User-triggered, explicit
Timing	Manual -- you control when
Best for	On-demand actions
Location	`.claude/commands/`

	Skills
Trigger	Always-on, auto-discovered
Timing	Automatic -- applied by context
Best for	Passive expertise
Location	`.claude/skills/`

SECTION 3 -- SESSION MANAGEMENT

Starting & Resuming Sessions

Interactive

`claude`	Start a new session in current dir
`claude -c`	Continue the most recent conversation
`claude -r <name>`	Resume a named previous session
`claude -w [name]`	Start in a new git worktree

Sessions persist automatically. You can always come back to where you left off.

Non-interactive / CI

`claude -p "..."`	One-shot prompt, no interactive session
`cat logs | claude -p "explain"`	Pipe stdin into Claude
`-p --output-format json`	Machine-readable JSON output
`-p --allowedTools Bash Edit`	Restrict tools in CI mode

The `-p` flag transforms CC into a scriptable Unix tool. Use for CI/CD, cron jobs, log analysis, pre-commit checks.

SECTION 3 -- SESSION MANAGEMENT

Aborting & Undoing

`Esc`	Interrupt Claude mid-action to course-correct
`Esc Esc`	Open the checkpoint / rewind menu
`/rewind`	Undo mistakes -- restores both conversation and code state

Automatic Checkpoints

Claude snapshots state before every significant action. You can always roll back without touching git.

Plan Mode

Use Shift-Tab to toggle plan mode before large changes -- you'll need `/rewind` far less often.

SECTION 4 -- THE MOST CRITICAL CONSTRAINT

Context Management

Context window = every message + every file read + every command output. It fills fast. When too full: Claude loses instructions and quality degrades significantly.

/context

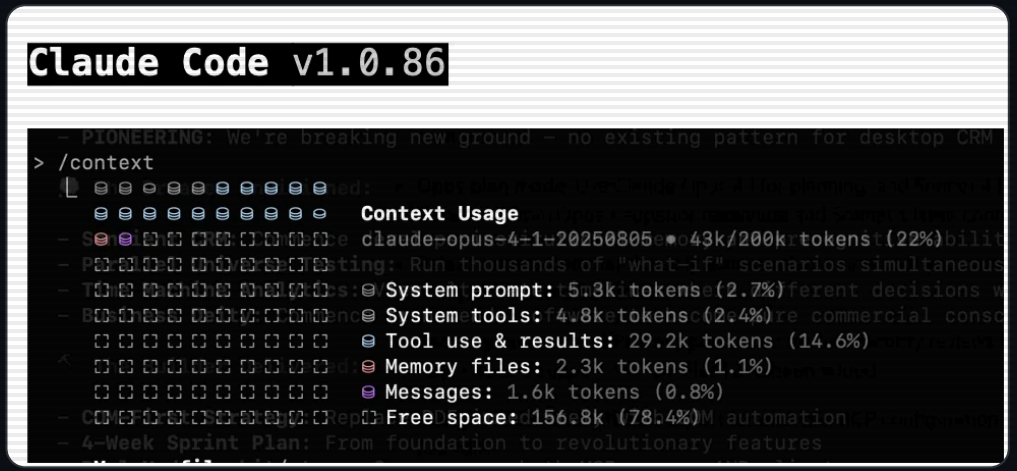
Visualize current token usage. Use proactively, not reactively.

/compact

Condenses history while preserving code patterns and key decisions. Use at 60-70% fill.

/clear

Completely resets context. Use between unrelated tasks. A fresh context is a sharp mind.



SECTION 5 -- WORKFLOWS

Explore > Plan > Code > Commit

1

Explore

Ask Claude to explain the codebase. Build shared understanding first.

→

2

Plan

Generate a written strategy with `/plan`. Review and approve.

→

3

Implement

Claude executes against the plan. Steer with short corrections.

Always give Claude a way to verify its own work. Provide test commands, expected outputs, or visual verification.

→

Prompting Effectively

Strategy	Instead of...	Try...
Scope the task	*"add tests for foo.py"*	*"write a test for foo.py covering the edge case where user is logged out. avoid mocks."*
Point to sources	*"why does ExecutionFactory have a weird api?"*	*"look through ExecutionFactory's git history and summarize how its api came to be"*
Reference patterns	*"add a calendar widget"*	*"look at how widgets are implemented on the home page. HotDogWidget.php is a good example. Follow the pattern."*
Describe symptoms	*"fix the login bug"*	*"login fails after session timeout. check auth flow in src/auth/, especially token refresh. write a failing test, then fix it"*

Let Claude Interview You

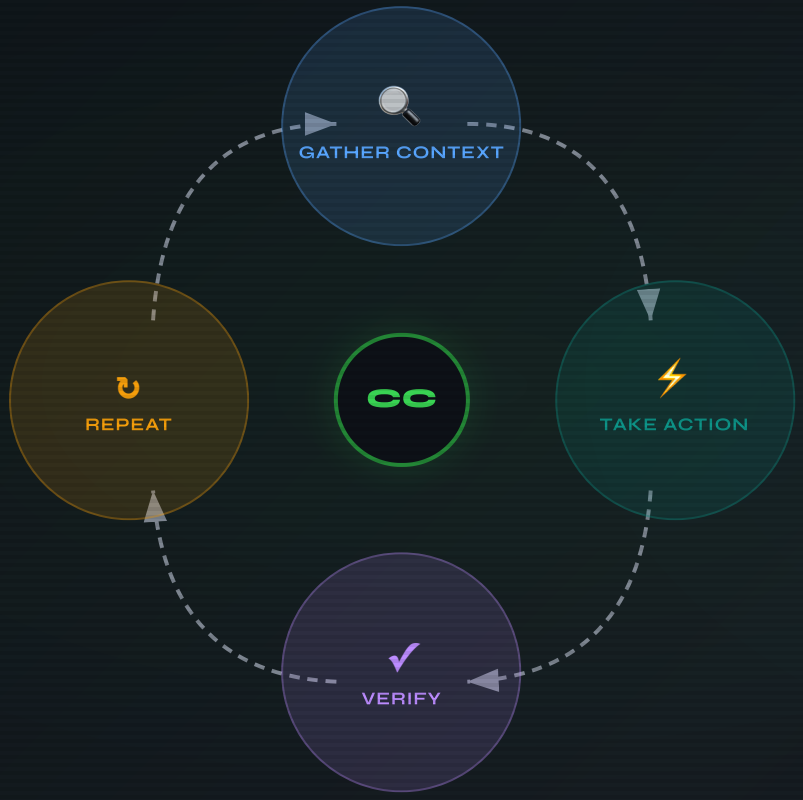
For larger features, ask Claude to **interview you** about requirements, edge cases, and tradeoffs before coding. Then write a spec and implement in a fresh session.

Common Failure Patterns

- **Kitchen sink session** -- mixing unrelated tasks. Fix: `/clear` between tasks
- **Correcting over and over** -- context polluted with failed attempts. Fix: `/clear` + better prompt
- **Over-specified CLAUDE.md** -- rules get lost in noise. Fix: prune ruthlessly
- **Infinite exploration** -- unscoped investigation fills context. Fix: use subagents

SECTION 6 -- MENTAL MODEL

The Agentic Loop



Claude Code is the **agentic harness** around Claude: it provides tools, context management, and execution that turn a language model into a coding agent.

Tool Category	What Claude can do
File ops	Read, edit, create, rename files
Search	Find files by pattern, regex content search
Execution	Shell commands, tests, git, builds
Web	Search web, fetch docs, look up errors
Code intel	Type errors, jump to def, find references

Example: "fix the failing tests"

Run tests → read errors → search source files → read & understand code → edit to fix → run tests again to verify

The loop adapts to your task. You can interrupt at any point to steer. Each tool use feeds information back, informing the next step.

SECTION 6 -- MENTAL MODEL

The Mental Model Shift

Don't think this	Think this instead
Prompt like a chatbot, wait for a reply	Describe goal + context + verification, let it run
Paste an error and ask what went wrong	Let Claude run the failing test itself and self-correct
Micromanage every file edit	Review at checkpoints, steer with high-level corrections
Start coding immediately	Explore first, then plan, then implement
I need to tell it every step	Give the end state and constraints, it figures out the steps

> *"Treat Claude Code like a developer you manage -- give clear requirements, provide context, establish how success is measured, then let them work."*

The more context and verification you provide upfront, the longer Claude can run autonomously without going off track.

SECTION 7 -- INTERNAL TOOLING

The code-agent Repo

Commands

`/review`	Local or remote code review
`/mr`	Create MR summary
`/smell`	Find code smells (globally, entire code base)
`/tidy`	Find and fix code smells (recent changes)
`/commit-message`	Generate a commit message
`/update-changelog`	Update docs/changelog.md
`/update-project-structure`	Update project-structure.md

Skills & Templates

- `git-commit` -- conventional commit automation
- `gitlab-ci-patterns` -- CI/CD pipeline patterns
- `python-standards` -- coding standards

Why AGENTS.md?

AGENTS.md is symlinked to CLAUDE.md but follows a **wider industry standard** that works across multiple AI coding tools, not just Claude Code. One file, multiple agents.

Reusable Base + Project Smarts

`code-agent` provides shared infrastructure (commands, skills, hooks). Projects add their own on top via `npx skills add` or `claude plugin add`. You get a common baseline without losing project-specific customization.

SECTION 8 -- POWER USER

Hooks

AUTOMATION

What Are Hooks?

- Automated, **deterministic** scripts that fire on Claude Code lifecycle events
- Not AI -- always predictable, guaranteed to run
- Unlike CLAUDE.md instructions which are advisory, hooks are enforced
- Configure via `/hooks` or edit `.claude/settings.json` directly

Lifecycle Events

`PreToolUse`, `PostToolUse`, `SessionStart`, `SessionEnd`

Example Uses

- Auto-run `prettier` after every file edit
- Post a Slack notification when a session completes
- Block writes to certain files (security hook)

SECTION 8 -- POWER USER

Plugins

BUNDLES

What Are Plugins?

- Shareable bundles that package commands + hooks + skills together
- Install with: `claude plugin add <name>`
- Browse the marketplace with `/plugin`
- One install, everyone gets the same tooling

Code Intelligence Plugins

For typed languages, install a code intelligence plugin to give Claude precise symbol navigation and automatic error detection after edits.

Publish & Share

Publish internally for your team or via the public ecosystem. Great for standardizing workflows across an organization.

SECTION 8 -- POWER USER

Subagents

ISOLATION

What Are Subagents?

- Specialized AI that run in their own **isolated context windows**
- Don't pollute your main session's context -- perfect for deep-dive research
- When done, they return a summary back to your main conversation
- Define in `.claude/agents/` with custom tools, model, and prompt

Use for Research

"Use subagents to investigate how our auth system handles token refresh" -- the subagent explores, reads files, and reports back without cluttering your context.

Use for Verification

"Use a subagent to review this code for edge cases" -- a fresh context won't be biased toward code it just wrote.

SECTION 8 -- POWER USER

Isolated Git Worktrees

PARALLEL

Run Multiple Sessions

- `claude -w [name]` creates a new git worktree and starts a session in it
- Pair with `--tmux` to open each worktree in its own pane
- Each instance works on a different branch -- no file conflicts
- Best for: parallel development, large multi-component projects

Fan Out Across Files

For large migrations, loop through tasks calling `claude -p` for each file. Use `--allowedTools` to scope permissions for batch operations.

Other Parallel Options

Desktop app manages multiple local sessions visually. `claude.ai/code` runs on Anthropic's cloud VMs. Agent teams coordinate multiple sessions with shared tasks.

SECTION 8 -- POWER USER

Writer/Reviewer & Agent Swarms

QUALITY

Writer / Reviewer Pattern

- One CC session writes the code
- A completely fresh CC session reviews it
- Zero shared context -- catches more bugs
- Best for any code going to production

SCALE

Custom Agents & Swarms

- Define agents with `--agents '{...}'`
- Combine with worktrees for parallel specialists
- List configured agents with `claude agents`
- High token cost -- budget accordingly

You're ready to start.

1. Start with a solid CLAUDE.md -- highest-leverage 5 minutes you'll invest
2. Get to know our internal commands -- they could be useful
3. Manage your context window intentionally: `/context`, `/compact`, `/clear`
4. Always give Claude a way to **verify** its own work
5. Think in workflows, not prompts: Explore > Plan > Implement > Commit

Docs: code.claude.com