AZADEV.

VERSION 1.0

MAY 2026

// CLAUDE CODE COMMANDS GUIDE

Claude Code Commands Guide

A practical reference for everyday use.

Every slash-command, skill, and core concept available inside Claude Code — built-in CLI commands, the Superpowers plugin, and the underlying ideas (memory, MCP, sub-agents) that make daily work faster. Written for daily reference and easy sharing.

AUTHOR

Piet Duijnstee
Founder, Azadev

CONTACT

piet@azadev.ai
azadev.ai

INTRODUCTION

How to read this guide

Claude Code is Anthropic's official CLI for Claude. You work with it the same way you'd work with a senior engineer next to you: you talk, it reads code, runs commands, and edits files. To save typing, you can call slash-commands and skills — pre-baked workflows that load specialised instructions on demand.

This guide covers everything available in a standard Claude Code session: built-in CLI commands, the Superpowers plugin (engineering workflows), and the core concepts behind it all — memory, MCP, sub-agents. Each entry gets a short paragraph in the same shape: what it does, when to reach for it, and a one-line example you can copy. Entries marked with a small tag (e.g. `[concept]`, `[feature]`) are not slash-commands — they're ideas or built-in features worth knowing about.

ANATOMY OF A COMMAND

```
/superpowers:brainstorming
```

`/`	The leading slash tells Claude this is a command, not a chat message.
`superpowers`	Optional namespace. Built-in commands omit this; plugin commands prefix it.
`:`	Separator between namespace and command name.
`brainstorming`	The command itself — the action you want Claude to take.

TIP

You don't need to memorise these. Type `/` in Claude Code and you'll see an auto-complete menu of every command currently installed. Use this guide to discover what's available and when to use it.

01

Getting Started

The first commands you'll use. These set up a new project, get help, manage the session, and adjust your environment. Master these four and you can navigate any Claude Code session.

`/init`

Scans the current codebase and writes a `CLAUDE.md` file with documentation that loads automatically in every future session. Captures architecture, tooling, conventions, and entry points.

When to use. Run once when you start working in a new repository so Claude has persistent context.

`/help`

Shows the built-in help screen with available commands, keyboard shortcuts, and links to documentation. Always reflects the commands actually loaded in your current session.

When to use. When you forget a command name or want to see what's installed.

`/clear`

Clears the conversation history and starts a fresh context window. The codebase, settings, and skills stay loaded — only the chat resets.

`/config`

Opens the settings UI where you can change model, theme, notification preferences, and other session-level options without editing JSON.

When to use. Quick tweaks like switching from Sonnet to Opus or toggling dark mode.

02

Memory & Context

Claude Code remembers things between sessions through `CLAUDE.md` files and auto-memory. These commands manage what Claude knows about you, your projects, and your preferences across conversations.

`/memory`

Opens the memory file (`CLAUDE.md` or user memory) in your editor so you can review or update persistent context. Anything saved here loads automatically at the start of every session.

When to use. When you want to add a preference, project rule, or fact Claude should always remember.

`CLAUDE.md` concept

A plain Markdown file at the project root. Claude reads it at session start and treats its contents as instructions. Use it for tech stack, coding conventions, glossary, or project-specific rules.

When to use. Drop a `CLAUDE.md` in any repo to give Claude permanent guidance for that codebase.

`Auto-memory` feature

A user-level memory system that persists across all projects. Lives in `~/.claude/projects/...` and surfaces relevant memories automatically based on conversation context. Useful for personal preferences and cross-project knowledge.

When to use. Save reusable facts like “I prefer Tailwind over Bootstrap” or “I always deploy via Vercel” once.

03

Skills & Agents

Skills are reusable sets of instructions that Claude follows on demand — each one teaches Claude how to handle a specific domain (debugging, copywriting, PR review). You can install skills via plugins (e.g. the Superpowers marketplace), bundle them with a project in `.claude/skills/`, or write your own from scratch. Agents are isolated sub-sessions that run focused tasks in parallel without polluting your main thread.

`/skills`

Lists every skill currently available in the session, grouped by source (built-in, plugins, user-installed). Useful to see exactly what Claude can reach for.

When to use. To browse what Claude can do beyond plain chat.

`/agents`

Shows installed sub-agents and how to invoke them. An agent is a separate Claude instance with its own context, used for parallel work or isolated tasks.

When to use. When you want to delegate a long-running task without burning your main context window.

`Skill invocation` `feature`

You can call any skill by name with `/skill-name`, or let Claude load it automatically when relevant. Skills come from built-ins, plugins, or your own `~/.claude/skills/` folder.

When to use. Whenever a task matches a known skill — Claude will often suggest one automatically.

04

Superpowers

The Superpowers plugin adds engineering-grade workflows — planning, TDD, debugging, code review, and parallel execution. These are battle-tested patterns from senior engineers, packaged as skills you can invoke on demand.

`/superpowers:brainstorming`

Walks through user intent, requirements, and design before any code gets written. Surfaces unknowns, edge cases, and constraints so you don't build the wrong thing.

When to use. Before starting any new feature, component, or behaviour change.

`/superpowers:writing-plans`

Turns a spec into a structured, step-by-step implementation plan with checkpoints. Plans become the source of truth for multi-step work.

When to use. Once you have requirements and need to break work into safe, reviewable chunks.

`/superpowers:executing-plans`

Runs a written plan in a separate session with built-in review checkpoints between steps. Keeps long implementations on track without losing context.

When to use. When you have a plan ready and want disciplined execution rather than freeform coding.

`/superpowers:test-driven-development`

Enforces the TDD loop: write a failing test, write the minimum code to pass, refactor. Prevents writing code that has no safety net.

When to use. Before implementing any feature or bugfix where correctness matters.

`/superpowers:using-git-worktrees`

Creates isolated git worktrees so feature work doesn't disturb your current workspace. Includes safety checks and smart directory selection.

When to use. Before starting parallel feature work or executing a plan that touches many files.

`/superpowers:subagent-driven-development`

Executes implementation plans by dispatching independent tasks to sub-agents in the current session. Speeds up work without context-switching.

When to use. When a plan has parallelisable tasks that don't depend on each other.

/superpowers:systematic-debugging

A structured debugging method — reproduce, isolate, hypothesise, verify — rather than guessing fixes. Prevents wasted hours on "it should work" bugs.

When to use. Whenever a test fails, a bug appears, or behaviour surprises you.

/superpowers:using-superpowers

The meta-skill that tells Claude how to find and invoke other Superpowers skills. Loaded at the start of any Superpowers-enabled session.

When to use. Automatic — runs at session start. You rarely call it directly.

/superpowers:dispatching-parallel-agents

Identifies independent tasks and dispatches each to a sub-agent simultaneously. Cuts wall-clock time when work can be parallelised.

When to use. When you have 2+ independent tasks with no shared state or sequential dependencies.

/superpowers:requesting-code-review

Asks Claude to perform a structured code review against requirements, conventions, and common pitfalls. Verifies work before merging.

When to use. After completing a task or major feature, before opening a PR.

/superpowers:receiving-code-review

Helps you respond to code-review feedback with technical rigour — verifying each point, pushing back on weak suggestions, fixing real issues.

When to use. When you get review comments and want to triage them properly instead of agreeing to everything.

/superpowers:verification-before-completion

Forces a verification step — running tests, checking output, confirming behaviour — before claiming work is done. Prevents "I think it works" mistakes.

When to use. Right before committing, merging, or telling the user you're finished.

/superpowers:finishing-a-development-branch

Guides the end-of-branch decision: merge, open a PR, or clean up. Presents structured options so nothing slips through.

When to use. When implementation is complete and tests pass — to choose how to integrate the work.

/superpowers:writing-skills

Helps you build, edit, and verify new skills. Covers structure, triggers, and testing so the skill actually fires when you expect it to.

When to use. When you want to package a repeatable workflow as a reusable skill.

05

Tasks & Workflow

Claude Code can track multi-step work with internal task lists so nothing gets dropped. These tools shine when a request spans several files or sessions.

`TodoWrite` feature

The built-in todo list tool. Claude uses it to plan and track multi-step work, marking items in-progress and completed as it goes. You'll see the list update live.

When to use. Automatic on complex tasks — Claude calls it without being asked. You can also ask explicitly.

`Background tasks` feature

Long-running shell commands can be launched in the background and monitored without blocking the conversation. Useful for builds, test suites, dev servers, and deployments.

When to use. When a command takes more than a few seconds and you want to keep working.

`Plan-then-execute` pattern

A pattern, not a single command: brainstorm, write a plan, then execute it across sessions. Keeps long projects coherent and reviewable.

When to use. For any task that touches multiple files, multiple sessions, or has unclear scope.

06

Scheduling & Automation

Claude Code can run on a schedule or loop on its own. These commands turn one-off tasks into recurring agents — useful for monitoring, reporting, and routine maintenance.

`/loop`

Runs a prompt or slash command on a recurring interval (e.g. every 5 minutes). Omit the interval to let the model self-pace.

When to use. For polling tasks like “check the deploy every 5 minutes” or babysitting PRs.

`/schedule`

Creates, lists, updates, or runs scheduled remote agents — called “routines” — on a cron schedule. Also supports one-time scheduled runs. There is no separate `/routine` command; manage routines through `/schedule`.

When to use. For recurring background work — daily reports, weekly summaries, scheduled reminders.

07

Code Quality & Review

Skills that help you ship safer code: review, simplify, and audit for security. Best used before merging, deploying, or handing work to a teammate.

`/review`

Reviews a pull request — checks correctness, conventions, edge cases, and obvious bugs. Outputs an actionable list of suggestions.

When to use. Before approving or merging a PR, especially when you wrote it with AI assistance.

`/security-review`

Performs a focused security audit on the pending changes in the current branch. Looks for injection risks, leaked secrets, unsafe dependencies, and common CVE patterns.

When to use. Before merging anything that touches auth, payments, user input, or third-party APIs.

`/simplify`

Reviews changed code for reuse opportunities, quality issues, and inefficiencies — then applies the fixes. Removes duplication and dead code.

When to use. After a long implementation session, to clean up before committing.

08

Settings & Configuration

Personalise Claude Code itself — permissions, hooks, environment variables, keybindings, and the status line. These commands tune the harness so daily work has less friction.

`/update-config`

Configures the Claude Code harness via `settings.json`. Handles permissions, env vars, automated behaviours via hooks, and troubleshooting.

When to use. When you want "from now on, always do X" — that needs a hook, not just a preference.

`/fewer-permission-prompts`

Scans your recent transcripts for common read-only Bash and MCP calls, then adds a prioritised allowlist to project `.claude/settings.json` to cut prompt fatigue.

When to use. When you keep approving the same harmless commands over and over.

`/statusline-setup`

Sets up the Claude Code status line — the bar at the bottom of your terminal that shows context, tokens used, and active model. Personalise what it displays.

When to use. Once, when you set up Claude Code on a new machine.

`/keybindings-help`

Customises keyboard shortcuts: rebind keys, add chord bindings, modify `~/.claude/keybindings.json`. Useful for muscle-memory tweaks.

When to use. When the default shortcuts clash with your terminal or editor habits.

09

Developer Tools

Tools and concepts for working on Claude itself, connecting external data, and reaching specialised functionality — building API integrations, hooking up MCP servers, and pulling in deferred tools on demand.

`/claude-api`

Builds, debugs, and optimises Claude API and Anthropic SDK apps. Adds prompt caching, handles model migrations (4.5 → 4.6 → 4.7), and tunes features like thinking, tool use, and citations.

When to use. When working in a codebase that imports `anthropic` or `@anthropic-ai/sdk`.

`MCP integration` concept

Connects external tools and data sources via the Model Context Protocol (MCP). Lets Claude read your Gmail, Drive, Calendar, Linear, or any custom MCP server.

When to use. When you want Claude to act on real data outside the codebase.

`Tool Search` feature

Fetches schemas for deferred tools — tools whose definitions aren't loaded by default to save tokens. Once fetched, they're callable like any built-in tool.

When to use. When you need a specialised tool (e.g. a specific MCP function) that wasn't auto-loaded.

// THANK YOU FOR READING

Keep this guide handy.

Print it, bookmark it, or share it with a teammate. Claude Code changes fast — new skills and commands ship regularly. When in doubt, type / in your terminal and explore the auto-complete menu. The official docs at docs.anthropic.com/claude-code are always the source of truth.

code.claude.com/docs

Built by Azadev · azadev.ai

AZADEV. A practical Claude Code reference