Working With Claude Code

A field guide to the practical tips the people who built it actually use

> **About this guide.** This is a faithful, reorganized distillation of a workshop given by **Boris Cherny** — a member of technical staff at Anthropic and the creator of Claude Code. The original talk was free, with no registration and no paywall. We've kept it that way. Aria Code rebuilt it into a lesson you can *learn from*, not just watch — with the techniques laid out as a ladder you climb, hands-on exercises at every step, and a one-page cheat sheet to keep.
>
> Everything here traces to the source talk. Where we add framing, we say so.

How to read this guide

You don't have to read it front to back. Pick the door that fits you:

If you are…	Start here
New to Claude Code	Read *The One Big Idea*, then climb Rungs 1–3. Do every "Try it now."
Comfortable, want to level up	Jump to Rungs 4–7 (tools, feedback loops, context, power moves).
In a hurry	Read *The One Big Idea* and *The Cheat Sheet*. Ten minutes, done.
Teaching a team	Use the slide deck; assign Rung 1 (codebase Q&A) as everyone's day-one task.

There are four formats of this lesson — Markdown, PDF, Word, and PowerPoint — so you can read, print, edit, or present it however you digest things best.

The One Big Idea

Most people open Claude Code, see a single prompt bar, and freeze. *What do I type?*

Here is the whole talk in one sentence:

> **You don't configure Claude Code with special syntax. You talk to it like you'd talk to another engineer — and the model is good enough to figure out the rest.**

Boris said it over and over, about git, about planning, about tools: *"We're not system-prompting it to do this. It just knows how to do this. The model is good."* Ask it to look through git history — it knows how to use git. Ask it to make a plan before coding — it makes a plan. Ask it to commit and open a PR — it reads your repo's commit style and does it your way.

That reframes your job. You are not a configurator. You are:

1. **A delegator** — describe the outcome in plain language.
2. **A context provider** — the more it knows about your world, the smarter its decisions.
3. **A feedback-loop builder** — give it a way to check its own work and it gets dramatically better.

Those three roles are the spine of everything below.

The Three Mental Models

Everything in this guide is an instance of one of these three ideas. If you remember nothing else, remember these.

1. Just talk to it

Claude Code is **fully agentic** — it's built for whole features, whole files, whole bug-fixes, not line-at-a-time autocomplete. It ships with a deliberately *small* set of tools: edit a file, run a bash command, search files. You don't tell it *which* tool to use. You describe the task and it strings the tools together itself. No magic incantations. Plain English (or plain voice).

2. Feed it context

> "The more context, the smarter the decisions will be — because as an engineer working in a code base, you have a ton of context in your head."

Your job is to move the context that's in your head into a place Claude can read it. That's what `CLAUDE.md`, slash commands, and `@`-mentioned files are for. Context is the single biggest lever on quality.

3. Give it a mirror

When Claude has a way to **check its own work** — run the unit tests, screenshot the page with Puppeteer, render the iOS simulator — it will *iterate on its own* and the result gets near-perfect. A model that can see its mistakes fixes them.

> "Give it some way to see its result, and it'll iterate and get better."

The Ladder: Seven Rungs from First Prompt to Power User

The talk is secretly a maturity ladder. Climb it in order. Each rung is a real capability you unlock — and a "Try it now" you can do in your own repo today.

Rung 0 — Set up your environment (5 minutes, once)

Before anything else, a few one-time setup steps make the tool pleasant:

- `/terminal-setup` — enables **Shift+Enter** for newlines, so you stop typing backslashes to write multi-line prompts.
- `/theme` — light, dark, or colorblind-friendly (Daltonized) themes.
- `/install-github-app` — lets you `@mention` Claude on any GitHub issue or pull request.
- **Allowed tools** — pre-approve the tool calls you say "yes" to constantly, so you're not prompted every time. (More control over this in Rung 6.)
- **Voice (Mac):** turn on **Dictation** in *System Settings → Accessibility*, tap the dictation key twice, and *speak your prompt*. Boris does this for most of his prompts — "you can just talk to Claude code like you would another engineer."

> 🟢 **Try it now:** Run `/terminal-setup` and `/theme`. Then dictate a one-sentence prompt instead of typing it.

Rung 1 — Start with codebase Q&A (do NOT start by editing)

This is the **single most important** beginner tip in the whole talk.

> "The thing I recommend above everything else is starting with codebase Q&A — just asking questions about your code base. This is what we teach new hires at Anthropic on day one."

Don't open Claude Code and immediately ask it to build a feature. Open it and **ask questions about code you already have.** This teaches you the boundary of what it can do — what it one-shots, what needs a little hand-holding — without any risk.

Why it's powerful:

- **No setup, no indexing, no waiting.** There is no remote database of your code. Nothing is uploaded. Your code stays local, and **Anthropic does not train models on it.** You start it and it works immediately.
- It goes **deeper than search.** Ask *"how is this class instantiated?"* and it won't just `grep` — it finds real usage examples and explains them, like a wiki you didn't have to write.
- It reads **git history.** Ask *"why does this function have 15 arguments?"* and it traces the commits, who introduced them, and the linked issues — then summarizes.
- It reads **GitHub issues** (via web fetch) for extra context.

The payoff at Anthropic: technical onboarding dropped from **2–3 weeks to 2–3 days**, because new hires ask Claude instead of taxing the rest of the team.

A Boris ritual worth stealing: *every Monday standup*, he asks Claude **"what did I ship this week?"** — it reads the git log, knows his username, and writes the summary he pastes into the doc.

> 🟢 **Try it now:** In a repo you know, ask three questions:
>
> 1. *"How is* `<some class>` *used across this codebase? Show me real examples."*
> 2. *"Look through git history and tell me why* `<some messy function>` *is shaped the way it is."*
> 3. *"What did I ship this week?"*

Rung 2 — Let it edit code

Once Q&A feels natural, let it touch the code. Remember Mental Model #1: you don't prompt it tool-by-tool. You say *"do this thing,"* and it strings together search → read → edit → run on its own.

The skill you're building here is calibration: *what can be one-shotted? two-shotted? what needs interactive back-and-forth in the REPL?* You only learn that by doing it.

> 🟢 **Try it now:** Pick a small, low-stakes change (rename a thing, add a log line, fix a typo'd error message). Describe the outcome in one sentence and let it work.

Rung 3 — Make it plan before it writes

Here's the most common failure mode Boris sees, and the one-line fix:

> "People take Claude code and ask it: *implement this enormous 3,000-line feature.* Sometimes it nails it. Sometimes the thing it builds is not at all what you wanted. The easiest way to get the result you want is to **ask it to think first.**"

You do **not** need plan mode or any special tool. You just say it:

> *"Before you write any code, brainstorm a few approaches, make a plan, and run it by me for approval."*

And a second incantation Boris uses constantly — three words:

> *"Commit this."*

There's nothing special about it. Claude reads your repo, infers your commit-message format from the git log, makes a branch, commits, pushes, and opens the PR — the way *your* team does it. You don't explain any of that.

> 🟢 **Try it now:** Take a slightly bigger task. Prompt: *"Before writing code, make a plan and check it with me."* Approve or correct the plan. Then, when it's done, just type *"commit this."*

> ⚠️ **Common mistake:** Asking for a huge feature cold, with no plan step. Plan-first is the cheapest quality upgrade you'll ever make.

Rung 4 — Teach it your team's tools

This is where Claude Code "really shines." There are two kinds of tools you can hand it:

1. Command-line tools (CLIs). Got an internal CLI? Just tell Claude about it and tell it to run `--help` to learn the interface. It will. If you use it a lot, write it into `CLAUDE.md` (Rung 6) so Claude remembers across sessions.

2. MCP tools (Model Context Protocol). Same idea — tell Claude about the MCP tool and how to use it, and it starts using it on your behalf.

The big unlock: when you start on a new codebase, **give Claude all the tools your team already uses for that codebase.** Now it can operate them for you.

> 🟢 **Try it now:** Point Claude at one CLI or script your team uses. *"We use* `./scripts/deploy`*. Run* `--help` *to learn it, then use it to do X."*

Rung 5 — Give it a feedback loop (the quality multiplier)

This is Mental Model #3 in practice, and it's the difference between "pretty good" and "almost perfect."

> "If you give it a mock and say *build this web UI*, it'll get it pretty good. But if you let it iterate two or three times, often it gets it almost perfect. The trick is to give it some sort of tool it can use to check its work."

Whatever your domain, give it a mirror:

- **Web/UI:** a Puppeteer MCP server so it can screenshot the page and compare to a mock.
- **Apps:** the iOS simulator screenshot.
- **Logic:** unit tests or integration tests.

Then it runs the check, sees the gap, and fixes it — by itself, in a loop — until it matches. (Real example from Anthropic: their apps repo ships a Puppeteer MCP server checked in, so *every* engineer can drive end-to-end tests and screenshot-iterate without installing anything.)

> 🟢 **Try it now:** Next UI tweak, drag a screenshot/mock into the prompt and say: *"Implement this. Use Puppeteer to screenshot the result and iterate until it matches the mock."*

Rung 6 — Pour in context with `CLAUDE.md`

The simplest, highest-leverage way to feed Claude context (Mental Model #2).

What it is: a file named `CLAUDE.md`. Drop it in your project root — the directory you start Claude in — and it's **automatically read into context at the start of every session** (it rides along on your first message).

What to put in it:

- Common bash commands (build, test, lint)
- Common MCP tools
- Architectural decisions

- The handful of important/core files
- Your style guide
- Anything you'd have to explain to a new engineer to work in this repo

The cardinal rule: keep it short. If it gets long, it just burns context and stops being useful. Short and load-bearing beats long and complete.

Three flavors: | Flavor | Where | Checked into git? | Loaded | |---|---|---|---| | **Project** `CLAUDE.md` | project root | ✅ yes — share with the team | automatically, every session | | **Local** `CLAUDE.md` | project root | ❌ no — just for you | automatically, every session | | **Nested** `CLAUDE.md` | any child directory | optional | on demand, when Claude works in that directory |

There's also an **enterprise** root for a `CLAUDE.md` that's shared across *all* a company's repos.

> 🟢 **Try it now:** Create a 10–15 line `CLAUDE.md` in your repo root. Put your build/test commands, your two or three most important files, and one architectural note. Commit it. Notice how much less you have to re-explain.

> ⚠️ **Common mistake:** A bloated `CLAUDE.md`. It's a cheat sheet, not documentation.

Rung 7 — Master the context hierarchy (and the network effect)

As you go deeper, *everything* about Claude — not just `CLAUDE.md`, but slash commands, permissions, and MCP servers — can be configured at three levels:

- **Project** — specific to this git repo (check in to share, or keep personal).
- **Global** — across all your projects.
- **Enterprise** — a policy rolled out to every employee automatically.

What you can manage this way:

- **Slash commands** — reusable prompts in `.claude/commands/`, in your home dir or checked into the project. (Claude Code's own repo uses one to auto-label GitHub issues via a GitHub Action — the same workflow, run by Claude, so humans don't have to.)
- **Permissions** — allow-list a command so it's auto-approved for everyone (e.g. your standard test command), or **block-list** one so it can *never* run (e.g. a URL that must never be fetched — an employee can't override it). Convenient for both unblocking people and keeping the codebase safe.
- **MCP servers** — check an `.mcp.json` into the repo; anyone who runs Claude there is prompted to install the shared servers.

Not sure where to start? Start with shared project context. You write it once and everyone on the team benefits — *"someone does a little bit of work and everyone on the team benefits."* That network effect is the whole point.

Two built-in tools to manage all of it:

- `/memory` — see every memory file currently being pulled in (enterprise policy, your user memory, project `CLAUDE.md`, nested `CLAUDE.md`s) and edit any of them.
- `#` **(pound sign)** — type `#` then whatever you want remembered, and pick which memory file it lands in. Great for the moment Claude misuses a tool: correct it once with `#` and it remembers from then on.

> 🟢 **Try it now:** Run `/memory` to see what's loaded. Then, next time Claude does something slightly wrong, fix it with `#<the correction>` instead of re-typing the correction every session.

The Cheat Sheet

Keep this page open. These are the keybindings and incantations that make the difference.

One-time setup

Command	Does
`/terminal-setup`	Shift+Enter for newlines
`/theme`	light / dark / colorblind themes
`/install-github-app`	`@mention` Claude on GitHub issues & PRs

Keybindings (hard to discover in a terminal — that's why they're here)

Key	Action
Shift+Tab	Toggle **auto-accept edits** mode — edits apply without approval (you can always ask Claude to undo). Great when it's on the right track or iterating on tests.
`#`	Remember something — written into `CLAUDE.md` automatically; you choose which file.
`!`	Drop to **bash mode** — run a command locally; its output goes into the context window so Claude sees it next turn.
`@`	Mention a file or folder to pull it into context.
Esc	Stop whatever Claude is doing — safe, never corrupts the session. Then redirect it.
Esc Esc	Jump back in history.
Ctrl+R	Expand to the full output Claude sees in its context window.
`--resume`	Restart Claude and resume a previous session.
`--continue`	Continue the most recent session.

Incantations (plain-English prompts that just work)

- *"Before you write code, make a plan and run it by me."*
- *"Commit this."* → branch + commit (in your repo's style) + push + PR.
- *"Look through git history and explain why this is the way it is."*
- *"What did I ship this week?"*
- *"Build this. Use [Puppeteer / the tests / the simulator] to check your work and iterate until it matches."*
- *"Run `<tool> --help` to learn it, then use it to…"*

Power Moves (the advanced tier)

Boris calls himself "a Claude normie" — usually one session, a few terminal tabs. But here's what the power users do.

The SDK — Claude Code as a scriptable Unix utility

The `-p` (print) flag *is* the SDK. `claude -p "<prompt>"` runs Claude **headless** — you pass a prompt, allowed tools, and an output format (`--output-format json` or streaming JSON), and you get structured output back.

> "Think of it as a super-intelligent Unix utility. You give it a prompt, it gives you JSON. You can pipe into it and pipe out of it."

Real uses at Anthropic: **CI pipelines, incident response, automation.** Examples:

- `git status | claude -p "..."` then `jq` the result.
- Pipe a giant log from a GCP bucket in and ask *"what's interesting here?"*
- Pull data from the Sentry CLI, pipe it in, have Claude act on it.

> "We've barely scratched the surface of how to use this."

Running many Claudes in parallel

Power users (inside and outside Anthropic) run lots of sessions at once:

- **SSH + tmux** sessions into their Claude instances.
- **Multiple checkouts** of the same repo, a Claude in each.
- **Git worktrees** for isolation between parallel runs.

You can run as many sessions as you want — there's a lot you can get done in parallel.

Straight from the Q&A — the *why* behind the design

These answers tell you a lot about how to think about the tool.

Why is bash so carefully gated? The hardest part to build. Bash is powerful and can change system state unexpectedly — but approving *every* command is miserable. The answer: read-only commands are recognized, static analysis figures out which commands combine safely, and a **tiered permission system** lets you allow-list and block-list at different levels. (This is why the permissions in Rung 7 exist.)

Is it multimodal? Fully, from the start — it's just hard to *discover* in a terminal. **Drag-and-drop an image, give a file path, or paste an image** — all work. Boris drops in a UI mock, points Claude at a Puppeteer server, and lets it iterate automatically.

Why a CLI instead of an IDE? Two reasons. (1) Anthropic engineers use everything — VS Code, Zed, Xcode, Vim, Emacs — and the **terminal is the common denominator** that works with all of them, locally or over SSH/tmux. (2) The models are improving so fast that betting heavily on a custom UI layer may be wasted work; the CLI keeps them close to the model.

Do researchers use it for ML? Yes — heavily. About **80% of technical staff at Anthropic use Claude Code every day**, including researchers who use it to edit and run notebooks.

Your First Hour — a guided path

A concrete plan if you're starting today:

1. **(5 min)** `/terminal-setup`, `/theme`. Turn on dictation. → *Rung 0*
2. **(15 min)** Ask your codebase three questions. Don't edit anything yet. → *Rung 1*
3. **(10 min)** Make one tiny edit by describing the outcome. → *Rung 2*
4. **(10 min)** Take a real task. Say *"plan first, check with me."* Approve. Let it build. Say *"commit this."* → *Rung 3*
5. **(15 min)** Write a 12-line `CLAUDE.md` with your build/test commands and key files. Commit it. → *Rung 6*
6. **(5 min)** Run `/memory` to see what's loaded. → *Rung 7*

That's the whole foundation. Everything else is depth on top of it.

The takeaways, one more time

1. **Talk to it like an engineer.** No magic syntax — the model figures out tools, git, and planning on its own.
2. **Start with codebase Q&A**, not editing. It's the safest way to learn the tool's boundaries (and it cut Anthropic onboarding from weeks to days).
3. **Make it plan before it writes** — the cheapest quality upgrade there is.
4. **Give it a feedback loop** (tests, screenshots) and it iterates to near-perfect on its own.
5. **Feed it context with a short** `CLAUDE.md`, and share it so the whole team compounds.
6. **Go further with the SDK** (`claude -p` as a Unix utility) and **parallel sessions** when you're ready.

Hosted free by Aria Code. Distilled faithfully from Anthropic's Claude Code workshop by Boris Cherny. No registration. No paywall. Share it freely.