APRIL 2026 · 2ND EDITION

Claude Code
The Complete Guide

The Definitive AI Programming Guide for Engineers and Product Managers

Orange Book Series by HuaShu

Compatible with: Claude Code v2.1.88+

Models: Opus 4.6 / Sonnet 4.6 / Haiku 4.5

New: Computer Use • Voice Mode • Deep Architecture Analysis

HuaShu

AI Native Coder

> From the Knowledge Planet "AI Programming: From Beginner to Expert"

This guide is based on Anthropic's official documentation, public talks by Boris Cherny (creator of Claude Code), DeepLearning.AI official courses, and Claude Code v2.1.88 source code analysis. All operational details reflect the latest information available as of March-April 2026. AI tools evolve rapidly; please verify against official documentation.

This guide is continuously updated. Get the latest version at: huasheng.ai/orange-books

Table of Contents

CONTENTS

Preface From Zero Coding Skills to a National TV Interview

Preface

In August 2024, I posted a series of videos on Bilibili about how I — someone who had absolutely no programming background — was using AI to build real products. The comments were blunt:

"If you don't know how to code, how do you debug?"

"If you don't know how to code, how did you get an app on the App Store?"

"You call that a product? That's barely a demo!"

I couldn't argue. At the time, I was still figuring things out. AI coding tools were still in their early days — Cursor had just introduced Agent mode, and Claude Code hadn't even launched yet. My capabilities were limited, and what I built reflected that.

Four months later, near the end of 2024, my girlfriend offhandedly suggested: "Instead of a flashlight app, why not make a fill-light card?" I searched Xiaohongshu (China's Instagram) and found countless women sharing solid-color images to use as selfie fill lights — several posts with hundreds of thousands of likes. The demand was clearly there. Nobody had just turned it into a proper product.

That evening, I spent about an hour in Cursor and built the first version of Kitty Light. Pure development cost: a few dollars in API calls.

Two days after launch, Kitty Light hit number one on the App Store paid chart. Thousands of comments flooded in on Xiaohongshu — and because the App Store displays the developer's real name, users started calling me "Chen Yunfei" instead of my online handle. That's how I accidentally lost my internet alias on Xiaohongshu.

Then things started happening fast. People's Daily featured me as a representative of the "solo maker economy." CCTV News reached out for a segment on AI-powered one-person companies. During the interview, I built a small product live on camera in 10 minutes using Claude Code — likely the first time Claude Code's interface appeared on Chinese national television.

I'm not saying this to brag. I'm saying it to establish a fact: **I have never written a single line of code.**

Not "I used to, but stopped." Never. Not once.

Kitty Light was written by AI. So were all the follow-ups — the photo album app, the HarmonyOS version, the mini-program. My WeChat Official Account with 300,000 followers, four published Orange Books, my

Bilibili and YouTube content — more than 70% of the execution work in all of these is done by AI. I left a major tech company in March 2023, and for nearly three years since, I've been building products, writing content, making videos, teaching courses, and writing books — all solo. That would have taken a team of a dozen people not long ago.

Claude Code is my primary tool.

I'm not writing this book because I'm a Claude Code expert. I'm not a programmer. I don't know TypeScript. I can't read Claude Code's source code. I'm writing it because I'm probably the person in the Chinese-speaking world who has used Claude Code for the most **non-programming** purposes. I use it to write books, conduct research, manage files, generate illustrations, proofread articles, publish to Feishu, and distribute content across social platforms. I've installed over 60 Skills, built a complete Harness toolchain, and do most of my daily work inside the terminal.

The reader I'm writing for is who I was a year and a half ago: smart, full of ideas, can't write code, and not sure whether AI-assisted building actually works in practice.

I can tell you with complete confidence: it does. Not just "works" — works well enough to build things that real people use, pay for, top the charts, and get covered on national TV.

But I also want to say something less exciting — something I actually said during that CCTV interview: **the vast majority of solo-built products will quietly disappear.** What you see are the few that made the charts; behind them are thousands of projects nobody ever heard of. Getting from zero to one is easier now, but getting from one to a hundred still takes sustained effort. Building something isn't hard anymore. Building something people keep coming back to — that still takes real work.

Claude Code solves the "how to build" problem. The "what to build" and "whether it's any good" still depend entirely on your own judgment. Tools are just tools. Taste, insight into real needs, and the willingness to keep refining — AI can't give you those.

This book will walk you through everything from installation to building products independently with Claude Code. In each chapter, I'll share real scenarios and lessons learned the hard way — not just feature walkthroughs. By the final chapter, you should be able to do what I do: turn Claude Code from a coding tool into your entire workbench.

The door is open. You can choose to walk through it.

HuaShu (花叔) · Chen Yunfei
April 2026
Somewhere in an Airbnb

§01 Why Claude Code

Why Claude Code

AI coding tools have transformed three times in three years. Understanding this evolution is the key to grasping what makes Claude Code genuinely different.

Three Shifts in Three Years

In 2022, GitHub Copilot arrived. You write the first half of a line, it guesses the second. Like an intern sitting next to you — yes, you type faster, but nothing fundamentally changes: you're still the one writing code.

From 2023 to 2024, Cursor took off. You could use natural language to ask the editor to refactor a function or restructure a module — no need to spell out "how to write it," just say "what outcome I want." Cursor later added an Agent mode capable of cross-file operations and running commands automatically. But it still lived inside the IDE, an extension of the editor.

In 2025, Claude Code arrived. It doesn't live in any editor — it runs directly in the terminal. You describe a goal, and it plans the steps, reads the code, writes the code, runs the tests, and handles git — the entire loop, automated. Your role shifts from "the person writing code" to "the person giving direction."

Across these three steps, what changed wasn't just how advanced the technology got — it was the relationship between you and AI. Copilot is your smart autocomplete. Cursor is your pair programming partner. Claude Code is your independent engineering team.

How Is It Different from Cursor, Really

This is the most common question: "Cursor has an Agent mode too — aren't they both just AI writing code for me?"

Fair point. Post-2024 Cursor is genuinely powerful — cross-file operations, full project understanding, automatic command execution. The difference isn't about what's possible; it's about how far each one goes.

Dimension	IDE Agent (Cursor, etc.)	Terminal Agent (Claude Code)
Runtime Environment	Embedded in editor, depends on IDE framework	Terminal-native, operates directly on the OS
Autonomy	Usually requires your confirmation nearby	Can run fully unattended
System Integration	Bridges git/CLI via plugins	Direct access to git, shell, and MCP
Memory System	Implicit project index	Explicit CLAUDE.md memory file
Parallel Execution	Primarily single-instance	Native multi-instance parallelism

Pay close attention to the last two rows. CLAUDE.md lets you encode project knowledge, coding conventions, and architectural decisions into a file that Claude Code reads on every startup — giving the AI a persistent memory of your project. Multi-instance parallelism means you can have several Claude Code instances each handling a different module simultaneously, like a small team.

Here's an analogy: Cursor is like a pair programming partner sitting inside your IDE, the two of you staring at the same screen together. Claude Code is more like an independent engineer you hand a brief to — they pull the repo, write the code, run the tests, and commit, while you go grab a coffee and come back to review the results.

Boris Cherny, the creator of Claude Code, says he hasn't hand-written a single line of code since switching to Opus 4.5. He used it 46 out of 47 consecutive days, with his longest single session running 1 day, 18 hours, and 50 minutes. This isn't marketing copy — it's a reality already unfolding at scale.

> **HuaShu's (花叔) Experience:** I have never written code by hand. Every product I've shipped — including Kitty Light, which hit #1 on the App Store paid charts — was built entirely with AI. Claude Code has turned "build real products without knowing how to code" from a rare experiment into something anyone can do.

It's Not Really Helping You Write Code

This sounds strange, but after six months of daily use, it's the most accurate way I can describe it: Claude Code isn't helping you write code — it's helping you build products.

Traditional AI coding tools solve the problem of code production speed: how do you write this function or this component faster? Claude Code solves the problem of product development speed: how do you get from an idea to something that actually runs, faster?

Compare the two scenarios:

推荐

With Claude Code:
"Build me a Markdown-based blog system using Next.js, deployed to Vercel, with dark mode and RSS support."

It will: analyze the requirements → choose a technical approach → scaffold the project → implement step by step → run tests → fix bugs → ship. All you do is confirm direction and make decisions.

不推荐

With an IDE-based Agent:
The experience isn't bad, but you'll likely spend your time: watching the IDE to see what it changed → manually reverting when something breaks → waiting for your approval before it runs commands → never really stepping away. You're the supervisor throughout.

In the first scenario, you're making product decisions. In the second, you're overseeing a process. As AI capability keeps improving, "watching AI work" will become less and less valuable — and the ability to make product decisions will become more and more valuable.

Claude Code's rapid growth is fundamentally a product-market fit story: a huge number of smart people want to build things, and what they lack isn't ideas — it's the ability to turn ideas into products.

Who This Book Is For

Engineers who want a 10x productivity boost. You already know how to code, but most of your day goes to boilerplate, debugging, writing tests, and dealing with CI/CD. Claude Code can take all of that off your plate, freeing you to focus on architecture decisions and product thinking.

Product managers who want to build their own MVPs. You have strong product instincts and user insight, but you're always blocked by engineering bandwidth. Claude Code lets you ship a working prototype over a weekend — no sprint planning, no PRD waiting for a developer to interpret.

Founders who want to run a one-person company. You want to validate business ideas without spending a lot of time or money on engineering. Claude Code gives a single person the development capacity of a small team — websites, apps, backend APIs, all within reach on your own.

> 核心建议
>
> Whichever category you fall into, this book assumes you're smart but may have never used an AI coding tool before. We start from zero — but we won't linger too long on the basics.

How Fast Is It Growing

A few numbers.

Claude Code launched in public preview in February 2025 and went GA in May. Within just six months of GA, it reached **$1 billion in annualized revenue** — a pace that's extremely rare in SaaS history.

Enterprise adoption has been equally fast. Netflix, Spotify, DoorDash, Notion, and Vercel are all running it at scale internally. Anthropic's data shows teams using Claude Code see an average productivity gain of 2–5x. Rakuten cut new feature release time from 24 days to 5; Zapier achieved 97% organization-wide AI adoption with over 800 agents deployed.

Anthropic's "Agentic Coding Trends Report" published in early 2026 contains a few numbers that tell the story clearly: 78% of Claude Code sessions now involve multi-file edits (up from 34% a year earlier); average session length has grown from 4 minutes to 23 minutes; and the average session involves 47 tool calls. This isn't "autocomplete" — this is "executing a project."

In developer surveys, Claude Code holds 44% market share for complex tasks (multi-file refactoring, architectural decisions, large-scale debugging) — surpassing GitHub Copilot and Cursor within eight months. Startup adoption stands at 75%. Developer "most loved" ratings: Claude Code 46%, Cursor 19%, GitHub Copilot 9%.

Claude Code currently runs on three models:

- **Opus 4.6** — Strongest reasoning, for complex tasks and architectural decisions
- **Sonnet 4.6** — Best value, the everyday workhorse for most coding
- **Haiku 4.5** — Fastest response, for simple queries and completions

The signal behind all these numbers is simple: agentic coding is no longer a hobbyist experiment — it's becoming the standard way software gets built.

Why Claude, and Not Something Else

There's no shortage of AI coding tools. GitHub Copilot runs on OpenAI's models. Cursor integrates multiple providers. There are open-source alternatives too. Claude Code's breakout success comes from executing on two fronts simultaneously: **model capability and engineering design**.

Start with the model.

Coding tasks make very different demands on a model than conversation does. Chat rewards sounding right. Code requires actually running right. A single wrong character in a function is a bug; one bad architectural call costs days of rework. This demands exceptional reasoning depth, long-context comprehension, and instruction-following precision.

Claude has consistently ranked in the top tier across these dimensions. Opus 4.6 leads SWE-bench (a real-world GitHub issue resolution benchmark) on an ongoing basis; Sonnet 4.6 has no real competitor at its price point. The 1M-token context window is particularly important — it means Claude can "see" nearly an entire large codebase at once, rather than just the file you happen to have open.

Now the engineering.

Most people underestimate how much the engineering layer shapes the experience. The same model, wrapped differently, produces wildly different results. Claude Code got several things right:

- **Terminal-native.** No IDE framework constraints. Direct access to the operating system — run any command, touch any file.
- **Memory system.** CLAUDE.md isn't a gimmick. It gives Claude Code persistent memory across sessions. Your project conventions, architectural preferences, and coding style: write them once, and they apply forever.
- **Tool ecosystem.** Skills make capabilities reusable. Hooks make workflows automatable. MCP makes external services connectable. Together, these three form a complete Harness — a tool chain around the model.
- **Multi-agent architecture.** From SubAgents to Agent Teams, Claude Code was designed from day one for "a group of AIs collaborating," not "one AI doing everything."

The model provides the intelligence; the engineering provides the leverage. A powerful model with poor engineering feels awkward to use. Great engineering built on a weak model hits a low ceiling fast. Claude Code is solid on both fronts — which is why it holds up in enterprise-scale deployments.

> **核心建议**
> If you're in China and have difficulty accessing the Anthropic API, don't worry. Claude Code supports third-party model configurations. Appendix B has a detailed guide to connecting domestic models, including Zhipu GLM, DeepSeek, Kimi, Qwen, and other major platforms.

More Than a Coding Tool

If you think Claude Code only writes code, you're underselling it.

I've used Claude Code to write four Orange Books (including the one you're reading now), run a WeChat Official Account with 300,000+ followers, and manage an entire content production workflow — from topic research and article writing to image generation, AI-flavor proofreading, typesetting and publishing, and multi-platform distribution. All of it from the terminal.

This isn't an extreme edge case — it's a trend already in motion: **more and more products and services are being designed for AI interfaces, not just human ones.**

A few signals you may have already noticed:

- **Enterprise platforms are shipping CLIs.** Feishu, DingTalk, and WeCom all provide command-line interfaces and APIs so AI agents can send messages, create documents, and manage calendars directly — no browser required.

- **New AI tools ship with Skills on day one.** Tools like LibTV (AI video generation) launched with a Claude Code Skill integration from the very first day. Users don't need to learn a new interface — just say "generate a 30-second product demo video" in the terminal.
- **API-first has become Agent-first.** Products used to be built for human UIs first, with APIs as an afterthought. Now it's the reverse: ensuring AI can use the product comes first, and the human interface is second priority.

Think about what this means. If 80% of software eventually has an agent interface, the way you interact with that software won't be opening an app, navigating a menu, and clicking buttons — it'll be telling your AI agent "export this data, turn it into a chart, and post it to the Feishu group."

Claude Code is one of your entry points to that future.

Learning Claude Code teaches you more than "how to get AI to write code for me." It teaches you "how to get AI to handle anything I need to do on a computer." The leverage from that skill is far greater than coding ability alone.

> **HuaShu's (花叔) Practice:** I currently have over 60 Skills installed in Claude Code, spanning content creation, video production, image generation, data analysis, and project management. Of my daily interactions with Claude Code, writing code probably accounts for about 30%. The other 70% is content creation, research, file management, and cross-platform collaboration. That's the real power of a terminal agent: it doesn't specialize. It just works.

How This Book Is Structured

The whole book follows a single thread: **how a smart person goes from zero to shipping an AI-built product in one week.**

Phase	Chapters	What You'll Learn
Day 1: Getting Started	§ 01–§ 03	Understand AI coding → Install and configure → Ship your first project
Days 2–3: Core Skills	§ 04–§ 06	Master the workflow → Set up the memory system → Communicate effectively
Days 4–5: Going Deeper	§ 07–§ 08	Expand capabilities (Skills/MCP) → Multi-agent collaboration
Days 6–7: Real-World Build	§ 09–§ 10	Build a complete product independently → Develop a lasting mental model

Every chapter has hands-on sections you can follow along with. You don't need to read it all in one sitting — read a chapter, do the exercises, then come back for the next one. That works perfectly.

In the next chapter, we'll get Claude Code installed in about 10 minutes.

§02 Get Up and Running in 10 Minutes

Get Started in 10 Minutes

Installation is simpler than you think. This chapter covers setup, choosing your environment, pricing plans, and your first conversation with Claude Code.

The first time I installed Claude Code, I typed one command in the terminal, it finished in 30 seconds, I typed `claude`, hit Enter. A cursor blinked on the screen. I typed "hello," it replied. The whole thing took under a minute.

My reaction: that's it?

Yes, that's it. No complicated IDE configuration, no plugin conflicts, no hours wrestling with environment variables. I've since helped quite a few friends get set up, and the most common obstacle wasn't technical — it was psychological. They just couldn't believe it was really that simple. So in this chapter I'll walk through every step clearly, including the small gotchas my friends and I have run into.

Three Ways to Install

macOS, Linux, and Windows are all supported. Which one you choose depends on your system:

Method	Command	Platform	Recommendation
Native Install	`curl -fsSL https://claude.ai/install.sh | bash`	macOS / Linux	⭐ Recommended
Homebrew	`brew install --cask claude-code`	macOS	For Homebrew users
WinGet	`winget install Anthropic.ClaudeCode`	Windows	First choice on Windows

> **核心建议**
>
> Not sure which to pick? Go with Native Install. One command, no extra dependencies.

Let's Install It

macOS / Linux

Open a terminal (Terminal or iTerm2 on macOS, your preferred terminal on Linux) and run:

```
curl -fsSL https://claude.ai/install.sh | bash
```

The script automatically detects your system, downloads the binary, and adds the `claude` command to your PATH.

Once it's done, just type `claude` to launch. Homebrew users can also run `brew install --cask claude-code` — same result.

Windows

Windows has one prerequisite: you'll need to install Git for Windows first.

1. **Install Git for Windows**

   Download it from `git-scm.com`, or use WinGet: `winget install Git.Git`. The default installation options are fine — it comes with a Git Bash terminal.

2. **Install Claude Code**

   Open PowerShell or Git Bash and run:

   ```
   winget install Anthropic.ClaudeCode
   ```

3. **Verify the Installation**

   Open a new terminal window and type `claude --version`. If you see a version number, you're good to go.

> **注意**
> Windows users take note: Claude Code depends on the Unix toolchain that comes with Git Bash. Running it directly in CMD may cause issues. Use PowerShell or Git Bash instead.

Platform-Specific Gotchas

The official line is "one command and you're done," but after helping dozens of people through the process, here are the most common issues by platform:

macOS

Xcode Command Line Tools not installed. Claude Code depends on git, and git on macOS requires the Xcode command line tools. If you've never developed on your Mac before, the first time you run `git` in the terminal, a dialog will pop up asking you to install them. Click Install — it takes about 5 minutes. Then re-run the Claude Code install command.

Apple Silicon vs. Intel. The install command is identical for M-series and Intel Macs — the script detects your architecture automatically. That said, if you're running an Intel terminal environment under Rosetta 2, you might end up with the Intel build. To verify, run `file $(which claude)`; the output should include `arm64`.

PATH environment variable. If you type `claude` after installing and get "command not found," it's a PATH issue 99% of the time. Check that your `~/.zshrc` (the default shell since macOS Catalina) contains something like:

```
# Make sure this line exists (or something similar)
export PATH="$HOME/.local/bin:$PATH"
```

After adding it, run `source ~/.zshrc` or open a new terminal window.

Windows

Choosing a terminal. Windows has far more terminal options than Mac, which can be confusing. Priority order: Windows Terminal > PowerShell > Git Bash > CMD. Avoid CMD — it's ancient and lacks many commands. Windows Terminal is Microsoft's modern option, available free from the Microsoft Store.

WSL users. If you're used to WSL (Windows Subsystem for Linux), you can install the Linux version of Claude Code directly inside WSL. It works exactly like native Linux and spares you any Windows environment headaches. The WSL install and the Windows native install are independent of each other.

Corporate antivirus software. Some enterprise security tools (360, Kingsoft, or Windows Defender in strict mode) may block curl downloads. If this happens, either temporarily disable the antivirus or use WinGet (`winget install Anthropic.ClaudeCode`), which goes through Microsoft's official channel and is typically not flagged.

Linux

glibc version. In rare cases, older Linux distributions (such as CentOS 7) may have a glibc version too low to run the binary. The fix is to upgrade your system, or run Claude Code inside Docker.

Headless servers. The first-time login requires opening a browser. On a headless server, Claude Code will display a URL and a verification code — open the URL on another machine with a browser, enter the code, and authentication is complete.

Corporate Networks

Some corporate networks use SSL man-in-the-middle certificates for traffic inspection, which can cause SSL verification failures. Symptoms include errors like `UNABLE_TO_VERIFY_LEAF_SIGNATURE` or `CERT_HAS_EXPIRED`. The fix is to point Node.js to your company's CA certificate via an environment variable:

```
export NODE_EXTRA_CA_CERTS=/path/to/company-ca.pem
```

If you don't know where your company's CA certificate lives, ask IT. This isn't specific to Claude Code — any Node.js program will hit this issue on a corporate network.

Five Ways to Use It — Which Is Right for You?

Once installed, you have five ways to use Claude Code. Each offers a different experience:

Environment	Description	Best for
Terminal CLI	The most native experience with full feature access — just type `claude` in your terminal	Primary daily driver for development
VS Code Extension	Runs in the VS Code sidebar with inline visibility of file changes	Developers already in VS Code
Desktop App	Standalone application — no terminal required	Users less comfortable with the terminal
Web	Access directly at `claude.ai/code` in your browser — nothing to install	Quick tryout or occasional use
JetBrains Plugin	Works inside IntelliJ IDEA, WebStorm, and other JetBrains IDEs	JetBrains users

Recommendation: All the examples in this book use the terminal CLI. Even if you normally work in VS Code or JetBrains, I'd suggest getting comfortable with Claude Code in the terminal first before switching to the IDE integration. The terminal is where its full capabilities live.

Accounts and Pricing

Claude Code requires an Anthropic account. On first launch, it will automatically open a browser window to log in or sign up.

There are three paid tiers:

Plan	Monthly Price	Usage	Best for
Pro	$20/month	Base quota, sufficient for everyday development	Individual developers, learners
Max 5x	$100/month	5× the Pro quota	Heavy users, full-time AI-assisted coding
Max 20x	$200/month	20× the Pro quota	Team use, commercial projects

How do you choose? A simple rule of thumb: if you're using Claude Code for more than 2 hours a day, you'll likely hit Pro's limits. Upgrading to Max 5x is worth it at that point. $100/month sounds like a lot, but compared to the time it saves, it's a reasonable trade.

> **核心建议**
> Start with Pro. After a few days of real use, you'll know whether it's enough. A common pattern: first week feels fine on Pro, second week you're hooked and upgrade to Max.

Enterprise users can also use the Anthropic API with token-based billing, which suits teams with custom integrations or compliance requirements. Version 2.1.88 added a `claude auth login --console` command that lets you log in directly with an Anthropic Console account (billed via API) without separately configuring an API key. For most readers of this book, though, a direct subscription is the simplest starting point.

Your First Conversation

Installed, logged in. Let's try it.

1. **Open a terminal and navigate to a project directory**

   Claude Code uses your current directory as its working directory. Create a test folder to start:

   ```
   mkdir ~/my-first-project && cd ~/my-first-project
   ```

2. **Launch Claude Code**

   ```
   claude
   ```

   The first time you launch, a browser window will open to log in to your Anthropic account. Once logged in, the terminal will show the Claude Code interactive interface.

3 **Send your first instruction**

Try this:

```
Create a simple HTML page that displays "Hello, Claude Code!" with some nice CSS styling.
```

You'll watch Claude Code get to work: it creates an HTML file in your directory and writes out the complete code. The whole thing takes about 10–30 seconds.

4 **Check the result**

Once Claude Code finishes, open the file in your browser:

```
open index.html    # macOS
xdg-open index.html # Linux
start index.html   # Windows
```

If you see a styled "Hello, Claude Code!" page, everything is working perfectly.

It looks simple, but notice what just happened: one sentence of natural language, and the AI completed an entire loop — understanding the request, creating the file, writing the code. Everything in the chapters ahead builds on exactly this foundation.

Verification Checklist

Run through this list:

Check	Command / Action	Expected Result
CLI works	`claude --version`	A version number is displayed
Account logged in	Launch `claude` — no login prompt	Goes straight to the conversation interface
Can create files	Ask Claude Code to create a test file	File appears in the current directory
Can read a project	Launch inside an existing project and ask "what does this project do?"	Claude Code describes the project accurately
Can run commands	Ask it to run `ls` or `git status`	Returns the command output

All passing? You're ready to go.

Troubleshooting

Can't connect or log in

Claude Code needs access to Anthropic's API servers. If you're getting connection timeouts or errors, run `curl -I https://api.anthropic.com` in your terminal to check connectivity. If you get back an HTTP status code (any number), the network is reachable. If it times out, you'll need to investigate your network environment.

Permission denied during installation

If you hit a permissions error on macOS or Linux, don't reach for `sudo`. Try this instead:

```
# Make sure the local bin directory exists and is writable
mkdir -p ~/.local/bin
# Re-run the install script
curl -fsSL https://claude.ai/install.sh | bash
```

If the problem persists, check whether `~/.local/bin` is on your PATH.

How to upgrade

Claude Code will notify you when a new version is available. To update manually, just re-run the install command:

```
# Native Install
curl -fsSL https://claude.ai/install.sh | bash

# Homebrew
brew upgrade --cask claude-code

# WinGet
winget upgrade Anthropic.ClaudeCode
```

Stay up to date. Claude Code ships new features at a rapid pace — nearly every week. Updates aren't just bug fixes; they often bring meaningful capability improvements. Aim to update at least every two weeks.

Installing the VS Code Extension

To use Claude Code inside VS Code:

1. Open the Extensions marketplace (Cmd+Shift+X or Ctrl+Shift+X)
2. Search for "Claude Code"
3. Install the official Anthropic extension
4. A Claude Code icon will appear in the sidebar — click it to start a conversation

The VS Code extension calls the same underlying CLI, so you don't need to configure a separate account — login state is shared.

Installing the Desktop App

If you prefer not to work in the terminal, the desktop app is available at `claude.ai/download`. Download, install, and you're done. It's essentially a graphical wrapper around the terminal CLI.

> **注意**
> Whichever method you choose, install the CLI first. It's the foundation — the VS Code extension and Desktop App both depend on it. If the CLI runs cleanly, the other environments will too.

You're installed, logged in, and have had your first conversation. Next chapter, we'll start building something real.

§03 Your First Project

Your First Project — Learning by Doing

Enough theory — let's build something. In this chapter, we'll create a real CLI tool from scratch. By the end, you'll truly understand what conversational programming feels like.

The day Kitty Light (小猫补光灯) launched, my girlfriend asked me: "How long did it take from idea to done?" I said about an hour. She didn't believe me. But it was true: 5 minutes to validate the idea (a quick Xiaohongshu search showed tons of girls using solid-color screens as selfie fill lights — posts with hundreds of thousands of likes, demand confirmed), then a few rounds of conversation with an AI coding tool, and it was done.

Kitty Light was built with Cursor, of course — Claude Code hadn't launched yet. But the core experience is identical: you describe what you want, and the AI builds it. No need to spend three months learning to code first. No need to understand SwiftUI or React Native. You just need to know what you want to make.

In this chapter, we'll use Claude Code to build something from zero. Smaller in scope than Kitty Light, but the process is exactly the same: first get clear on what you're making, then tell Claude to get to work.

Warm-Up: Build a Personal Homepage in 5 Minutes

Before diving into the main project, let's do a quick warm-up. The goal of this exercise is to experience one complete "say → build → see" cycle firsthand.

```
mkdir my-homepage && cd my-homepage
claude
```

Once inside Claude Code, say:

```
Build me a personal homepage. Single-page HTML, make it look good.
Content: my name is [your name], and I'm a [your role/identity].
Add a tagline, an about section, and placeholder social media links at the bottom.
Use a modern minimalist style with responsive layout.
```

Claude will generate a complete HTML file within 30 seconds. Open it to take a look:

```
open index.html      # macOS
xdg-open index.html  # Linux
start index.html     # Windows
```

Odds are you'll see a decent-looking page. Maybe not exactly what you imagined — the color scheme might feel generic, or the layout could be better. That's fine, just keep talking:

```
Change the color scheme — use a dark background with light text. Switch to a serif font.
Add an animated gradient background effect.
```

Claude will update the file, and you'll see the changes when you refresh the browser. Iterate a few more rounds until you're happy.

The whole thing takes under 5 minutes, but you've just completed one full conversational programming loop: describe the need → Claude implements → review the result → suggest changes → Claude adjusts. Every project that follows, no matter how complex, is just this loop running repeatedly.

> **核心建议**
>
> **Deploy this page online.** If you want others to see it, say: "Help me deploy this page to GitHub Pages." Claude will create a Git repo, push the code, and configure GitHub Pages. That's another 5 minutes. You'll have a real, live personal homepage — your first AI-built thing shipped to the world.

Warm-up done. Now let's tackle something a bit more involved.

What We're Building

A daily AI news aggregator — a CLI tool. The features are simple:

- Fetch the latest articles from a few RSS feeds (TechCrunch AI, The Verge AI, Hacker News, etc.)
- Use AI to summarize the key points of each article
- Output a neatly formatted Markdown digest

Why this? It's small enough to finish in an afternoon, yet complete enough to touch networking, data processing, AI API calls, and file output. One project, and you'll have experienced the full range of what Claude Code can do.

> **First, make a mental shift:** starting now, you're the product manager and Claude is your engineer. Your job is to clearly articulate what you want — not to write code. Even if you're a seasoned developer, try it this way first.

Step 1: Tell Claude What You Want Describe What You Want

Open a terminal, navigate to an empty folder, and start Claude Code:

```
mkdir ai-news-digest && cd ai-news-digest
claude
```

Then, in plain language, describe what you want:

```
# What you type into Claude Code:

Build me an AI news aggregator CLI tool. Here's what I need:

1. Fetch articles from the last 24 hours from these RSS feeds:
   - TechCrunch AI (https://techcrunch.com/category/artificial-intelligence/feed/)
   - The Verge AI (https://www.theverge.com/rss/ai-artificial-intelligence/index.xml)
   - Hacker News top 30 (https://hnrss.org/newest?q=AI&count=30)

2. Extract title, link, publish time, and source for each article

3. Sort by time descending, and output a Markdown digest to the output/ directory

4. Write it in TypeScript, run directly with tsx

Don't start coding yet — give me an implementation plan first.
```

Notice that last line: "Don't start coding yet — give me an implementation plan first." That's not just politeness — it's an important technique. Make Claude **think it through** before it acts.

> **核心建议**
> The quality of your requirements description directly determines the quality of the output. A few principles: be clear about what the final deliverable looks like (a Markdown digest), specify the tech stack (TypeScript + tsx), and give concrete data source URLs rather than vague references like "a few RSS feeds." The more specific, the better.

Step 2: Review the Plan Review the Plan

When Claude receives your requirements, it won't start coding right away — it'll give you a plan first. It'll look something like this:

```
Here's my plan for the AI news aggregator:

Project structure:
ai-news-digest/
├── src/
│   ├── index.ts          # Entry point
│   ├── fetcher.ts        # RSS fetching logic
│   ├── formatter.ts      # Markdown formatting
│   └── types.ts          # Type definitions
├── output/               # Digest output directory
├── package.json
└── tsconfig.json

Technical approach:
- RSS parsing via rss-parser
- HTTP requests via Node's built-in fetch
- Date handling via date-fns
- Run TypeScript directly with tsx — no compile step needed

Implementation flow:
1. Fetch all three RSS feeds in parallel
2. Normalize format, filter to articles within 24 hours
3. Sort by time
4. Generate Markdown digest
5. Write to output/YYYY-MM-DD.md

Ready to start implementing?
```

Now you're the product manager reviewing an engineering proposal. If it looks good, say OK. If you want changes, just say so. For example:

```
# Your feedback:
The plan looks good. Two additions:
1. For each article in the digest, include a one-sentence summary (first 100 chars from the a
2. Add a stats header at the top of the digest: "X articles collected from Y sources"
```

This back-and-forth is the core of conversational programming: Claude proposes a plan, you add details, Claude refines. No flowcharts, no technical specs — just talking in plain language.

Step 3: Watch Claude Work Watch Claude Work

Once you've confirmed the plan, Claude gets to work. You'll see a series of operations in the terminal:

1. **Initialize the project**

   Claude will run `npm init -y` and then install dependencies. You'll see a permission prompt:

   ```
   Claude wants to run: npm init -y
   Allow? (y/n)
   ```

   Press `y` to allow. It'll ask again for `rss-parser`, `date-fns`, `tsx`, and other packages.

2. **Create the source files**

   Claude will create TypeScript files one by one. You'll see the code being written, with diffs shown for each file. You don't need to read every line — just do a quick scan to check that the file structure looks reasonable.

3. **Run a test**

   Once the code is written, Claude will typically try running it to see if there are any errors. If there are, it'll read the error message, identify the problem, fix the code, and run it again — an automatic self-repair loop.

The whole process takes about 2–5 minutes. What should you do? **Just watch.** It's like handing a task to a new colleague — the first few times, you'll observe how they work. Once you're familiar with their style, you can comfortably let them run with things.

> **注意**
>
> Claude may ask for permission multiple times during execution. Early on, it's worth glancing at each command it wants to run. Once you're comfortable, you can pre-authorize common commands with `/permissions` (covered in § 04), or just switch to Auto mode.

Step 4: Verify the Output Verify the Output

Claude's done — let's run it and see:

```
npx tsx src/index.ts
```

If everything went well, you'll find a Markdown file in the `output/` directory that looks something like this:

```
# AI News Digest — 2026-03-28

> 23 articles collected from 3 sources

---

## TechCrunch AI

### OpenAI Releases GPT-5.4 with 2M Context Window
🔗 https://techcrunch.com/2026/03/28/openai-gpt-54...
📅 2026-03-28 14:30
> OpenAI today released GPT-5.4, with the biggest change being an expanded context window fro

### Anthropic Launches Claude Code Desktop App
🔗 https://techcrunch.com/2026/03/28/anthropic-desktop...
📅 2026-03-28 11:00
> Anthropic announced that Claude Code is now available as a desktop application, supporting

---

## The Verge AI
...
```

Open the file and check the formatting and content. In most cases, it'll work on the first try.

If there's an error? Just paste the error message to Claude:

```
# When the run fails, paste the error to Claude:
Got an error when running:
TypeError: Cannot read properties of undefined (reading 'map')
  at formatArticles (src/formatter.ts:15:23)
```

Claude will read the error, find the issue, fix the code, and run it again. This fix loop usually takes only 1–2 rounds.

Step 5: Iterate and Improve Iterate and Improve

It works — but you want to add things. Just keep talking:

```
# Improvement 1: add AI summaries
Right now each article's summary is just truncated from the description — pretty rough.
Change it to use AI: for each article's title + description, call the Claude API to generate
Read the API key from the environment variable ANTHROPIC_API_KEY.
```

Claude will update the code to add the API call. Once you've verified it, keep going:

```
# Improvement 2: add scheduled runs
Add a cron mode that automatically runs at 8am every day, using node-cron.
Add a CLI argument:
- `npx tsx src/index.ts` runs once immediately
- `npx tsx src/index.ts --cron` enables scheduled mode
```

```
# Improvement 3: add deduplication
Some articles appear across multiple sources. Add URL-based deduplication.
```

Each round, Claude updates the code, runs the tests, and confirms the result. You only ever do two things: say what you want, and verify that it's right.

At this point, your first project is complete. Look back at the whole process:

Regardless of project size — tiny utility or full product — the underlying pattern is always these five steps.

An Important Mental Shift The Mental Shift

After finishing this project, you should have an intuitive sense of something: **your value isn't in writing code — it's in defining what to build and judging whether it was built right.**

Many engineers, the first time they use Claude Code, instinctively want to read every line and understand every implementation detail. That's a natural response, but it will slow you down.

A more effective approach is to manage Claude like a team member:

Traditional Programming	Programming with Claude Code
Design the solution yourself, write the code yourself	Describe the need; Claude produces the plan and code
Debug line by line	Give Claude the error message; it debugs itself
Read docs, search Stack Overflow	Ask Claude directly: "how do I implement X?"
Code review done manually	Ask Claude to explain what it wrote
Refactoring requires understanding all the code first	Tell Claude: "refactor this into the X pattern"

This doesn't mean ignoring the code entirely. It means your attention should operate at a higher level: Are the requirements accurate? Is the approach sound? Does the result match expectations? That's where your time should go.

Common Questions from Beginners Common Questions from Beginners

"What if I can't understand the code?"

Just ask. "Explain the implementation logic in fetcher.ts" — Claude will explain it in plain language. You can follow up: "Why use Promise.allSettled instead of Promise.all?" It'll explain the reasoning behind the technical choice.

You don't need to be able to write this code yourself. You just need to understand what it's doing. Like you don't need to know how to rebuild an engine — you just need to know when the car is running normally.

"What if it gets something wrong?"

Just say what's wrong. You don't need to know how to fix it — describing the symptom is enough:

推荐

"After running, only TechCrunch articles show up — articles from the other two sources are missing. Check the fetching logic."

不推荐

"Your code has a bug on line 23." (Unless you actually know where the problem is)

Describing the symptom is more effective than pointing to a line number. Claude may find that the root cause is somewhere entirely different from where you'd expect.

"How much oversight should I have?"

Four words: **trust, but verify**. Let Claude run with things, but check the result at every step. During the planning phase, read carefully — make sure the direction is right. During coding, a quick scan of the file structure is enough. During execution, check whether the output matches your expectations. During iteration, test edge cases thoroughly: empty data, network timeouts, malformed input.

You'll develop a feel for this balance after a few projects.

"What if it goes off the rails?"

If it's slightly off, just correct it: "Stop — don't use the XXX library, switch to YYY." If it's badly off, press `Esc` to stop and re-describe the requirement from scratch. Pressing `Esc` twice opens the Rewind menu, where you can roll back the conversation, roll back file changes, or both.

One rule of thumb: if two corrections haven't fixed it, stop and start over. Patching on top of a wrong foundation just makes things messier.

The core of this chapter: describe requirements, review the plan, confirm execution, verify results, iterate. You own the what and the whether — Claude owns the how. Once this division of labor becomes second nature, your productivity will jump to a different level.

When Things Go Wrong — A Beginner's Troubleshooting Guide

Building your first project, you'll almost certainly run into a few snags. I've seen the same issues come up with dozens of people I've helped — and nearly all of them have standard fixes.

Claude created the files but running them throws an error

The most common situation. Usually it's a missing dependency or an unset environment variable. The fix is simple: paste the full error message to Claude.

```
Got an error when running:
[paste full error here]

Please fix it.
```

Claude will read the error, locate the problem, and fix the code. 90% of the time, one round is all it takes. If two rounds haven't resolved it, you might be dealing with an environment issue (Node.js version, system permissions, etc.). Describe your setup:

```
My environment: macOS 15.3, Node.js v22.5.0, npm 10.8.0
The error persists after two fix attempts — could it be an environment issue?
```

Claude seems to be "thinking" and nothing is happening

If there's no output for more than a minute, press `Esc` to interrupt, then resend your request. If it keeps stalling, it might be a network issue — check your connection.

It could also be that your request is too large. If you ask Claude to process a massive file (thousands of lines of code) all at once, it may take a very long time. Try breaking it up: "Just look at the first 100 lines — any issues?"

What Claude built looks nothing like what I wanted

This is almost always a requirements description problem. Review what you said, then compare it to what Claude built. Common pitfalls:

- **Too vague.** "Make it look nice" → Claude doesn't know what "nice" means to you. Try instead: "dark background, large headline, card layout, rounded corners"

- **Assuming Claude knows the context.** "Add that feature" → Claude doesn't know what "that" refers to. Every request should be self-contained
- **Too many things at once.** Ten requirements crammed into one message, and Claude may quietly drop a few. Break them into 2–3 separate messages

Claude modified files it shouldn't have touched

This happens occasionally. Press `Esc` twice to open the Rewind menu, where you can:

- **Roll back the conversation:** undo the last few turns and resume from an earlier point
- **Roll back files:** restore modified files to their previous state
- **Roll back both:** undo both conversation history and file changes

Rewind is your safety net. Knowing it's there is enough — no need to be anxious.

Dependency installation fails (npm/pip errors)

On networks in mainland China, npm install can be slow or fail entirely. Two common solutions:

```
# Option 1: use the Taobao mirror
npm config set registry https://registry.npmmirror.com

# Option 2: ask Claude for alternatives
Tell Claude: "I can't install this package — is there an alternative, or can we avoid this de
```

Python users facing pip issues can do the same with Tsinghua's mirror: `pip install -i https://pypi.tuna.tsinghua.edu.cn/simple package-name`.

Hit my token/usage limit

Pro users have a daily usage limit. If you see a "limit reached" message, you have three options: wait a few hours for the limit to reset, upgrade to Max 5x, or call it a day.

One tip to reduce token consumption: use the `/compact` command to compress your conversation history. Long conversations burn through tokens fast (the full history is sent with every message) — compressing it helps.

> **核心建议**
>
> **The single most important troubleshooting tip:** give Claude the full error message. Don't retype it by hand (typos happen) — copy and paste directly from the terminal. With the complete error in front of it, Claude can self-resolve 90% of problems.

§04 Core Workflows

Core Workflows — The Patterns That Matter

After running through your first project, you might think you've seen everything Claude Code has to offer — write code, confirm permissions, check results. But in practice, the real efficiency gap comes down to a handful of core working patterns. This chapter breaks them down.

I once had Claude Code fix a bug in an iOS project. I hit Enter, went to grab a glass of water, and came back to find it was already done — seven files changed, tests run, all passing. I scrolled through the operation log: it had read the error output, traced the root cause to a type conversion issue, fixed the core file, then tracked down six places that referenced that type, updated each one, and ran tests to confirm. The whole thing took about three minutes.

If I'd done it myself, just understanding the relationships between those seven files would have taken half an hour. But then the flip side hit me: what if it got it wrong? What if it misread the cause of the bug? I let it touch everything without even looking.

That moment made me realize: the core question with Claude Code isn't "can it do this?" — it's "should you talk it through first before letting it go?" That's what this chapter is about.

Plan Mode: Think Before You Act Plan Mode

Boris (the creator of Claude Code) once said: **"A good plan really matters."** He starts most of his own sessions in Plan Mode.

Plan Mode does one thing: it lets Claude plan without executing. It tells you what it intends to do, but it won't touch your code, install packages, or run commands. You discuss the approach back and forth, and only once you're satisfied do you let it run.

How to Enter Plan Mode

In the Claude Code input box, press `Shift+Tab` twice. You'll see the interface switch to Plan Mode. Claude's behavior changes immediately:

- It can read files to understand the codebase, but won't modify anything
- It gives you a detailed implementation plan — which files to change and how
- You can iterate on the plan as many times as you need

The Golden Workflow for Plan Mode

Boris's recommended full workflow looks like this:

1. **Describe your requirements in Plan Mode and iterate**

   Press `Shift+Tab` × 2 to enter Plan Mode and describe what you need. After Claude proposes a plan, you can say things like "change step three" or "use XXX library instead here," and keep refining.

2. **Write a detailed execution prompt in your editor**

   Once you're roughly satisfied with the plan, press `Ctrl+G` to open the input box in your default editor (set by the `$EDITOR` environment variable). Write a complete execution prompt there — include the confirmed plan details and any constraints. Save it, and the content comes back into Claude Code's input box as the next prompt to send.

3. **Switch to execution mode and enable Auto-accept**

   Once the plan is confirmed, press `Shift+Tab` to switch back to normal mode. Since you've already worked through the plan thoroughly, you can confidently let Claude run all the way through without stepping in at each point.

The key insight here is: **front-load all the deliberation into the Plan phase, then execute in one clean run.** Going back and forth mid-execution and reworking things repeatedly is the most token-wasteful way to use Claude Code.

When to Use Plan Mode

Use Plan Mode	Skip Plan, Just Do It
You're not sure about the implementation approach	Changing one line, fixing a typo
Changes span multiple files	Adding a console.log for debugging
You're not familiar with this part of the codebase	You know exactly what needs to change
Refactoring or architectural changes	Running tests, installing a package
First time touching a new project	Routine, repetitive operations

> **核心建议**
>
> A practical rule of thumb: if this is a task you'd need to explain to a colleague before they could do it, it's worth using Plan Mode. If you can describe it in one sentence, just do it directly.

Auto Mode: Safer Autopilot Auto Mode

After using Claude Code for a while, you've probably felt the frustration. Every file write needs confirmation, every command needs confirmation, even installing a dependency needs confirmation. By the 50th time you hit `y`, you've stopped reading what it's actually about to run.

Anthropic's internal data confirms this: **93% of permission requests are approved directly by users.** Approval fatigue renders the safety mechanism meaningless.

Auto Mode exists to fix this. The core idea: replace your judgment calls with an AI classifier. Safe operations pass through automatically; only risky ones get flagged.

How Auto Mode Works

Auto Mode has two layers of defense:

Input layer: A prompt injection detector scans everything Claude reads — files, web pages, command output. If content looks like it's trying to hijack Claude's behavior (say, a file that contains "ignore your previous instructions"), the detector appends a warning before passing that content to Claude.

Output layer: A transcript classifier evaluates the risk of each action before it executes. The classifier runs in two phases:

- Phase 1: Fast judgment (a single-token yes/no), erring on the side of caution, with an 8.5% false-positive rate
- Phase 2: Deep reasoning, applied only to actions flagged in Phase 1, bringing the false-positive rate down to 0.4%

Most safe operations pass in Phase 1. Only suspicious ones trigger Phase 2's deeper analysis.

What Auto Mode Blocks

Real examples distilled from Anthropic's internal incident logs:

- **Scope escalation** — You say "clean up old branches," Claude deletes remote branches too. Auto Mode blocks it, because "clean up" is too vague to constitute authorization for deleting remote branches
- **Credential probing** — Claude hits an auth error and starts searching environment variables for other API tokens. Blocked
- **Bypassing safety checks** — A deployment pre-check fails, Claude retries with `--skip-verify`. Blocked
- **Data exfiltration** — Claude wants to share code and creates a public GitHub Gist on its own. Blocked

How to Enable Auto Mode

```
# Specify at launch
claude --permission-mode auto

# Or cycle through modes with Shift+Tab during a session
# Default → Auto-accept edits → Plan → Auto → Default
# Auto Mode is currently a Research Preview feature on the Team plan
```

> **注意**
> Auto Mode isn't foolproof. Anthropic's published data shows the classifier still has a 17% miss rate for "overly proactive behavior." For operations involving production databases or cloud infrastructure, manual confirmation is still the safer call. Auto Mode is best suited for everyday development: writing code, running tests, Git operations.

Auto Mode vs. --dangerously-skip-permissions

You may have seen people in the community recommend `--dangerously-skip-permissions` to bypass all permission prompts. The difference matters:

	Auto Mode	--dangerously-skip-permissions
Safety	AI classifier evaluates every operation	No protection at all
Dangerous operations	Blocked; Claude is guided to try a different approach	Executed immediately, no warning
Prompt injection protection	Input-layer detector active	None
Best suited for	Everyday development	Fully isolated sandbox environments, CI/CD

Boris himself uses neither. He uses `/permissions` to pre-authorize safe commands (covered in the next section). But for most people, Auto Mode is a solid middle ground.

Permission Management: You Set the Rules Permission Management

Beyond Auto Mode, Claude Code offers more granular permission controls.

/permissions: Pre-authorize Safe Commands

Type `/permissions` to open the permission management interface. You can pre-approve specific operations so Claude never has to ask about them again.

Wildcards are supported:

```
# Allow all npm scripts
Bash(npm run *)

# Allow editing any file under docs/
Edit(/docs/**)

# Allow running tests
Bash(npx vitest *)
Bash(npx jest *)

# Allow Git operations
Bash(git add *)
Bash(git commit *)
Bash(git push)
```

These rules can be saved to `.claude/settings.json` and committed to Git, so the whole team shares the same permission configuration.

> **核心建议**
>
> Boris's approach: no Auto Mode, no skipping permissions — instead, carefully configure a whitelist with /permissions. The whitelist goes into git and is shared with the team. It's the most precise and most secure option, though it takes a bit of time to set up initially.

Three Tiers of Permission Control

Here's a summary of Claude Code's permission system, from most convenient to most granular:

Approach	Convenience	Safety	Best for
Auto Mode	High	Medium (AI classifier protection)	Most everyday developers
/permissions whitelist	Medium	High (precise control per command)	Team use, fine-grained control needed
Confirm each action (default)	Low	Highest	High-risk operations, early learning phase

When you're just getting started, stick with the default confirm-each-action mode. After a few projects — once you have a feel for what commands Claude typically runs — switch to Auto Mode or configure a whitelist.

Git Operations: Claude Just Gets It Git Operations

Claude Code's understanding of Git goes beyond just running `git` commands. It genuinely knows your project's current version control state — which files you've changed, which branch you're on.

One-Line Commits and PRs

The most common operations:

```
# Let Claude look at what changed and write a commit message
Commit the current changes with a meaningful commit message

# Or be more specific
Commit these changes — describe that we added RSS feed fetching

# Create a PR directly
Create a PR with a clear title and description explaining what this feature does
```

Claude will analyze your code changes, generate a descriptive commit message, and run `git add` + `git commit`. For PRs, it auto-generates the PR description too — what changed, and why.

Git Worktrees: Parallel Work Made Easy

This is Boris's top recommended technique. Git worktrees let you check out multiple branches simultaneously within the same repository, each with its own working directory.

```
# Create a worktree and start working in a new directory
claude --worktree
```

The `--worktree` flag tells Claude Code to automatically create a new git worktree and work inside an isolated directory. The benefits:

- Your current branch stays untouched
- You can open multiple worktrees at once, each handling a different task
- Each worktree has its own working directory — Claude instances don't interfere with each other

This is especially useful for parallel work. Say you're fixing a bug and you want Claude to build a new feature on a separate branch at the same time. With worktrees, both proceed independently.

How Boris works in parallel: He runs five Claude Code instances simultaneously in the terminal, each in a different worktree. Add in five to ten sessions on claude.ai/code, and he's pushing a dozen tasks forward at once — on his own. That's the power of agent-style work: you don't do everything yourself, you manage a team of agents that does it for you.

Computer Use: AI Gets Eyes and Hands Computer Use

Every workflow covered so far has Claude operating in the world of text — reading code, writing code, running terminal commands. That's genuinely where it excels. But the Figma window on your desktop, Photoshop, that legacy admin panel with a GUI and no API — those were off-limits.

Not anymore. Claude Code's Computer Use feature lets Claude see your actual screen via screenshots, then control your mouse and keyboard. Not simulated, not API calls — it literally looks at your screen, moves your cursor, and clicks your buttons.

How to Use It

Zero configuration. Available automatically for Pro and Max subscribers — no toggle to flip. During its work, if Claude determines it needs to interact with a GUI, it takes a screenshot to "see" the current state of the screen, then decides where to click and what to type next.

You can also proactively point it at your screen:

```
# Let Claude look at what's on screen
Look at this page on my screen and tell me what's wrong with the layout

# Have it operate a GUI app
Open System Preferences and turn off Dark Mode

# Test a web app you're building
Open localhost:3000 in the browser, walk through the signup flow, and look for bugs
```

Real Use Cases

From my own experience, the scenarios where Computer Use feels most natural:

Scenario	Why Computer Use helps
Testing web app UI	Claude doesn't just run test scripts — it clicks through pages like a real user, fills out forms, sees the rendered result, and catches visual issues
Operating desktop software with no API	Legacy admin panels, GUI-only tools — Claude was completely helpless before. Now it can operate them directly
Automating repetitive GUI tasks	Batch processing files, copying data between windows — let Claude handle the mechanical stuff
Debugging Chrome extensions	Extension popups and content script effects are only visible in the browser; Claude can screenshot them directly and pinpoint issues

What This Means

Computer Use looks like just another feature, but think deeper — it represents a directional shift for AI coding tools.

For the past year, every capability in AI programming has been built on text operations. Read files, write files, run commands, analyze logs. The entire interface was a terminal. Put another way: AI could only operate on things that could be expressed in text.

Computer Use breaks that boundary. AI gains the same GUI capabilities that humans have — it can see everything on the screen and react to it.

Why does this matter? Because **it directly expands who can use AI coding tools.** Before, you needed at least a basic understanding of the command line — you had to know what a terminal was — to collaborate with Claude Code. Now a product manager can tell Claude "change this button to blue in Figma," and an operations person can say "update these users' status to VIP in the backend." No technical concepts required.

In the long run, **the boundary of what AI can operate shifts from "wherever code can be written" to "anything visible on screen."** That's a qualitative change.

Current Limitations

Don't get too excited yet. Computer Use still has real weaknesses at this stage:

- **It's slow.** Every action requires screenshot → analysis → decision → execution. A click that takes a human half a second might take Claude several seconds
- **Precision operations are unreliable.** Dragging a slider to an exact position, selecting a specific cell in a dense table — these often come out off
- **Not suited for fast-response scenarios.** Animations, real-time interactions, game testing — Claude's reaction time can't keep up

> **核心建议**
>
> The best mental model for Computer Use right now: treat it like a patient but slow-handed tester. Give it tasks that follow a fixed, repeatable sequence and it performs well. Anything requiring flexible judgment or quick reactions, handle it yourself.

Voice Mode: Code by Talking Voice Mode

Hold Space to speak, release to send. That's it.

Type `/voice` in Claude Code to enter voice mode. It supports 20 languages, Chinese included. Hold the spacebar, say what you need, release — Claude transcribes the speech and processes it exactly like typed input.

When Voice Beats Typing

Voice isn't meant to replace the keyboard. It has its own natural fit:

- **When your hands are occupied.** A bug fix pops into your head while you're walking, or a feature idea strikes while you're cooking. Pull out your phone (if you're SSH'd into a server) or just speak to the computer — faster than hunting for a keyboard
- **When brainstorming.** Ideas are flowing faster than your fingers can type. Voice lets you dump a messy stream of thoughts to Claude all at once and let it structure them into concrete requirements
- **When describing spatial or visual concepts.** "I want a sidebar on the left, the right side split into two rows, chart on top and a table below" — saying that is faster than drawing ASCII diagrams

The Interaction Shift Matters More Than the Feature

I want to dwell on this for a moment.

Voice Mode's actual functionality is straightforward — it's speech-to-text. But shifts in interaction methods tend to have much larger reach than feature updates.

Keyboard → mouse → touchscreen → voice. Looking back, every time the interaction model changed, the circle of people who could use the tool got bigger. The mouse let people who couldn't type use a computer. The touchscreen let elderly people and children use smartphones. And voice?

Now put Voice Mode and Computer Use together: describe what you want by speaking, and Claude uses Computer Use to operate the screen and make it happen. **Voice expresses intent, Computer Use handles execution. A person can step away from the keyboard and code entirely — just talk, and AI builds things for you.**

We are closer to "speak to your computer and ship a product" than most people realize.

Current Limitations

Voice mode still has a few rough edges:

- Requires a reasonably quiet environment — accuracy drops in noisy backgrounds
- Long instructions are still more reliable typed. A 200-word technical specification spoken aloud may have transcription errors; typing is more precise
- It's best suited today for kicking off tasks and quick interactions: "run the tests," "rename this function to XXX," "look at this file for problems"

> **核心建议**
> A useful combination: voice to launch a task quickly ("help me do XXX"), then switch back to keyboard to enter precise details and constraints. Mixing both interaction modes is more efficient than committing to either one.

Session Management: Don't Let Context Become a Junk Drawer Session Management

Claude Code has context limits. The longer a conversation runs, the more scattered Claude's attention becomes. Session management matters far more than most people realize.

Core Command Reference

Action	Command / Shortcut	When to use it
Clear current session	`/clear`	Switching to a completely unrelated task
Compress context	`/compact`	Session is getting long, Claude is slowing down or forgetting
Stop current operation	`Esc`	Claude is doing something you don't want
Rewind	`Esc` × 2 or `/rewind`	Claude broke the code — opens the rewind menu to restore conversation, code, or both
Resume last session	`claude --continue`	Terminal accidentally closed; want to pick up where you left off
Resume a specific session	`claude --resume`	Want to return to a particular past session
Side-chain question	`/btw`	Want to ask something unrelated without polluting the current context

When to Use /clear

This command matters more than it seems. `/clear` wipes all conversation history for the current session and returns to a clean slate. CLAUDE.md and project files loaded at startup are unaffected.

When should you use it? **Any time you're starting a task that's fundamentally different from the previous one.**

Say you just finished debugging an API, and now you want Claude to build a new frontend component. If you don't clear, Claude's context is still full of information about that API bug, which will bleed into its understanding of the new task.

推荐

Finish fixing API bug → `/clear` → Start frontend component task

不推荐

Finish fixing API bug → just say "now help me build a frontend component" → Claude may blend the API context into the new task

The Clever Uses of /compact and /btw

`/compact` doesn't clear the conversation — it has Claude compress the current session into a summary. Use it mid-way through a long session: you and Claude have been working through a lot, the context length is hurting performance, but you don't want to lose the conclusions you've reached. `/compact` preserves the key information while freeing up context space.

`/btw` is an easy-to-overlook but highly practical command. It opens a "side-chain" conversation: you ask Claude something unrelated to the current task, get your answer, and the side chain closes — the main conversation context is untouched.

For example, while Claude is refactoring a piece of code, you suddenly wonder "how does TypeScript's `Record` type work again?" Use `/btw` to ask — it won't pollute the refactoring context.

Six Traps You're Likely to Fall Into Common Anti-Patterns

Let's talk about the most common mistakes when using Claude Code. I've made all of these. The official Best Practices doc flags them repeatedly. The community never stops talking about them.

Trap 1: Stuffing Everything Into One Session

Bug fixes, new features, refactoring, writing docs — all in a single session. The context fills up, and Claude's grasp of each task gets thinner and thinner.

Keep one session focused on one task. When it's done, `/clear`, or open a new terminal window.

Trap 2: Repeated Corrections That Spiral Sideways

Claude gets a step wrong, you correct it. It fixes one thing and breaks another, you correct again. Third try, still wrong. You've spent more time correcting than you would have spent doing it yourself.

If two corrections don't land, cut your losses with `/clear` and start over. Re-describe the requirement, but this time be more specific. Patching a conversation that's gone off the rails is almost always worse than starting fresh.

Trap 3: Accepting Output That Looks Right

Claude generates a pile of code. The output looks reasonable, you accept it, never actually run it. A few days later you find an edge-case bug.

Run the code after every round of changes. "Looks correct" and "is correct" can be very far apart. Boris's Tip #13 is exactly this: give Claude a way to verify its work. You should do the same.

Trap 4: Over-Managing Every Step

You inspect every file Claude writes, comment on every line changed. Result: both you and Claude move slowly, and you're essentially using Claude Code like traditional programming.

Focus on outcomes. Let Claude complete a full task, then evaluate whether the final output matches your expectations. Unless something is obviously going sideways mid-execution, stay out of it.

Trap 5: Vague Requirements, Then Blaming Claude for Not Getting It

"Help me optimize this code." "Make this page look better." Claude can only guess — and the direction it guesses is probably not what you had in mind.

Give specific, verifiable requirements: "Bring this API's response time from 2 seconds down to under 500ms. The bottleneck is the database query — consider adding caching or optimizing the SQL." The more specific you are, the closer the output lands.

Trap 6: Not Writing a CLAUDE.md

No CLAUDE.md in the project root, or there is one but it's never updated. Every new session means re-explaining the project background, code conventions, and tech choices from scratch.

This is important enough to deserve its own chapter. Jump to § 05.

The core of this chapter: Plan Mode to think before you act, Auto Mode to cut through approval fatigue, /permissions for granular control, Git integration for version management, session management to keep context clean. These five workflows cover 90% of everyday use. The remaining 10% — more advanced patterns — comes in the chapters ahead.

§05 CLAUDE.md: The Map You Draw for Your AI

CLAUDE.md — The Map You Draw for Your AI

Claude Code automatically reads CLAUDE.md at the start of every conversation. This file isn't a manual — it's more like a contract. Everything you and the AI have agreed on about how to work together lives in this one file.

I have one hard rule for my WeChat writing: no em dashes (——). They're a dead giveaway of AI-generated text. Once I added that rule to CLAUDE.md, Claude followed it. But during one long session that ran nearly an hour, the context got auto-compressed, and Claude started sprinkling em dashes everywhere again. That's when it clicked: things said in conversation can be compressed and forgotten, but CLAUDE.md is re-read fresh every single session.

After that, I developed a habit: any instruction I'd given Claude more than twice went straight into CLAUDE.md. Six months later, my writing project's CLAUDE.md had grown from a blank file into a full routing system — the root CLAUDE.md dispatches tasks to different subdirectories, each with its own rules file. Over 60 Skills, hundreds of rules, built up one line at a time.

Why It's the Most Important File

Working with Claude Code, you'll encounter many config files. `package.json`, `tsconfig.json`, `.eslintrc`... But one file outweighs all of them combined.

CLAUDE.md.

The reason is simple: **the very first thing Claude Code does when starting a new session is read this file.** Project structure, code style, test commands, common pitfalls — Claude learns all of it from here. Without it, Claude is like a new hire air-dropped into an unfamiliar codebase, having to figure everything out from scratch. With it, Claude walks in already knowing the rules.

Shrivu Shankar (VP of AI Strategy at Abnormal AI, whose team consumes tens of billions of tokens monthly for code generation) put it plainly:

> When it comes to using Claude Code effectively, the most important file in your codebase is the CLAUDE.md in your root directory. This file is the agent's "constitution" — the primary source of truth for how your specific codebase works.

He used the word "constitution." A constitution is short, principle-based, and doesn't get into details. The analogy is exact.

Guardrails, Not Manuals Guardrails, Not Manuals

The most common beginner mistake with CLAUDE.md: trying to write an encyclopedia. Cramming in every function's usage, every file's purpose, every API's parameters. After several thousand lines, Claude burns through a massive chunk of context just reading the file, leaving less room for actual work.

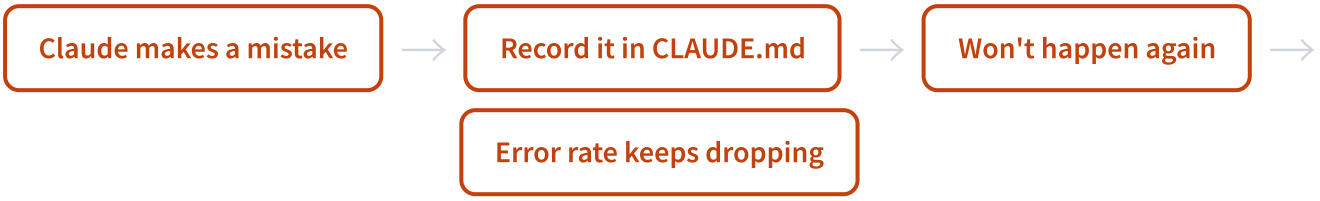
Boris's team (the creators of Claude Code) keeps their CLAUDE.md at roughly 2,500 tokens — about 100 lines. The core rules file for managing Claude Code itself is that short.

Shrivu shared an even more interesting approach:

> **核心建议**
> Your CLAUDE.md should start small and grow based on things Claude gets wrong. Instead of trying to write a complete manual upfront, **every time Claude makes a mistake, add a rule.** That's what "start with guardrails" means.

Why does this work so well? The rules file stays accurate because every entry corresponds to a real mistake that actually happened. The file naturally stays lean, because you only record things that have genuinely gone wrong.

Boris also described a flywheel effect in his tips:



During code reviews, he even @.claude on teammates' PRs to have it automatically add a rule to CLAUDE.md. The team shares a single CLAUDE.md checked into git, with contributions coming in every week. **This file is alive — it's not something you write once and forget.**

In Boris's own words: "Claude is very good at writing rules for itself." Tell it what it did wrong, and it can write a precise rule to prevent the same mistake next time.

What Actually Belongs in CLAUDE.md

This is probably the most practical part. There's one criterion: **if Claude can figure it out from the code, don't write it; if Claude can't guess it, you must write it.**

Write This	Don't Write This
Bash commands Claude can't guess (e.g., custom build scripts)	Things Claude can infer from reading the code (e.g., "this is a React project")
Code style preferences that differ from defaults	Standard language conventions (Claude already knows these)
Test commands and preferred testing frameworks	Detailed API docs (link to them, don't paste them in full)
Architectural decisions and their context	Frequently changing information (things you'd have to update constantly)
Dev environment quirks (e.g., special environment variables)	File-by-file descriptions (Claude will read the file tree itself)
Common pitfalls and how to fix them	Filler like "write clean code" or "follow best practices"

Shrivu also flagged a few common anti-patterns:

Don't `@`-reference large documents. When you `@` a long file in CLAUDE.md, it gets fully embedded at the start of every session, silently eating up context. The right approach: mention the path and tell Claude when to go read it. For example: "When encountering a FooBarError, see `docs/troubleshooting.md` for remediation steps."

Don't just say "never do X." When Claude thinks it must do X, it'll get stuck. Always provide an alternative: "Don't use the `--foo-bar` flag; use `--baz` instead."

Use CLAUDE.md as a forcing function to simplify your codebase. If a CLI command is so complex it takes several paragraphs to explain in CLAUDE.md, the command itself needs simplifying. Write a bash wrapper with a clean API, then document just that wrapper in CLAUDE.md.

Hierarchy Hierarchy

CLAUDE.md isn't just one file — it's a layered system. Claude Code automatically reads CLAUDE.md files from multiple locations in order:

```
~/.claude/CLAUDE.md ← Global: preferences shared across all projects
./CLAUDE.md ← Project: checked into git, shared with your team
./src/CLAUDE.md ← Subdirectory: rules for a specific module in a monorepo
./src/api/CLAUDE.md ← Deeper subdirectories
```

1. **Global `~/.claude/CLAUDE.md`**

   Put your personal, universal preferences here. Things like: prefer TypeScript, use Jest for testing, write commit messages in English. These rules apply across all projects — no need to repeat them everywhere.

2. **Project-level `./CLAUDE.md`**

   Put project-specific rules here. This file should be checked into git and shared with the team — which is exactly what Boris's team does. Code style, architectural constraints, test commands, common pitfalls: all of it lives here.

3. **Subdirectory-level**

   Especially useful in monorepos. Frontend rules go in the frontend directory, backend rules in the backend directory, each keeping to itself. When Claude enters a directory, it automatically loads the corresponding CLAUDE.md.

There's also an `@` import syntax for pulling other files into CLAUDE.md:

```
# CLAUDE.md
@docs/coding-standards.md
@docs/api-conventions.md
```

But remember what was said earlier: files referenced with `@` get fully embedded into context. Only import short files that are genuinely needed every single session.

What a Good CLAUDE.md Actually Looks Like

Here's a concise, well-crafted example of a project-level CLAUDE.md. Notice how short it is:

```
# MyApp

## Architecture
- Next.js 15 + TypeScript + Tailwind CSS
- Database: PostgreSQL + Drizzle ORM
- Auth: Better Auth
- State management: Zustand (do not use Redux)

## Dev Commands
- Start dev server: pnpm dev
- Run tests: pnpm test (Jest + React Testing Library)
- Type check: pnpm typecheck
- Lint: pnpm lint

## Code Style
- Use functional components, not class components
- Use Tailwind for styling, no CSS files
- Use server components for data fetching, not useEffect
- Use error.tsx boundaries for error handling, not try-catch around components

## Common Pitfalls
- After Drizzle migrations, always run pnpm db:generate or types will be out of sync
- Restart the dev server after changing environment variables
- better-auth session checks live in middleware — don't duplicate them in page components

## Don't
- Don't install new dependencies without my explicit approval
- Don't modify drizzle.config.ts
- Don't call the database directly from client components
```

The whole file is under 300 words. But every line earns its place: either a command Claude couldn't guess, or a hard-won lesson from a real mistake. Not a word wasted.

> **注意**
> Don't copy this example directly. A good CLAUDE.md grows out of your own project. Start with an empty file, add one rule each time Claude makes a mistake, and in three months that file will be your customized guardrail system.

Auto Memory: What Claude Remembers on Its Own Automatic Memory

Beyond the CLAUDE.md you write by hand, Claude Code also has an automatic memory system.

When you correct Claude's behavior mid-conversation — things like "use English for all commit messages going forward" or "put test files in the `__tests__` directory" — Claude automatically saves these preferences. Next session, it already knows. You don't need to repeat yourself or manually add anything to CLAUDE.md.

These memories are stored in `~/.claude/projects/<project>/memory/`, with MEMORY.md as the entry point, running in parallel with CLAUDE.md. The difference:

Hand-written CLAUDE.md	Auto Memory
Suited for team-shared rules	Suited for personal preferences
Checked into git	Stored locally
You maintain it actively	Claude maintains it automatically
Structured and organized	Informal, accumulated over time

The two work best together. Team rules go in the project CLAUDE.md; personal habits get handled automatically by Auto Memory.

The Iterative Flywheel: A System That Gets Better With Use The Iterative Flywheel

Back to the flywheel from earlier. This isn't just a metaphor — it's the actual experience curve for Claude Code users.

Mitchell Hashimoto (co-founder of HashiCorp, creator of Terraform) described the exact same process. When he built his AI workflow for Ghostty, every line in the config file mapped to a mistake the agent had made before. **The file is alive. It keeps growing.**

Here's how the progression looks:

1. **Week One: Empty File**

   You've written only the basic architecture and dev commands. Claude makes a lot of mistakes.

2. **Week Two: First Guardrails**

   You start logging Claude's mistakes one by one. "Don't use relative paths in this file." "Regenerate types after running migrations." The error rate starts falling.

3. **Month One: Flywheel Kicks In**

   CLAUDE.md has 20–30 rules, all from real mistakes. Output quality has visibly improved, and you're correcting Claude less and less often.

4. **Beyond: Continuous Iteration**

   You add new rules occasionally, prune outdated ones. The file stays lean but highly customized. You carry the same approach to new projects and get up to speed faster every time.

This is why CLAUDE.md is the most important file. Every rule represents a real mistake that was actually made. Every iteration makes Claude understand your project a little better.

The one-sentence summary: Start CLAUDE.md as an empty file, add one rule each time something goes wrong, keep it lean (Boris's team uses roughly 2,500 tokens), and check it into git to share with your team. The file you grow over three months is your most valuable AI asset.

§06 Advanced Prompting Techniques

Advanced Prompting & Context Engineering

Claude Code is not a search engine — you don't need to carefully craft keywords. But how you communicate with it does affect output quality. This chapter covers prompting strategies that actually work in practice. No theory.

Last year I told Claude Code to "help me optimize the styles on this page." What I meant was: tweak a few spacing values and font sizes. Instead, it refactored the entire CSS file and — while it was at it — added a bunch of "improvements" I never asked for, completely destroying a layout I'd spent an afternoon perfecting. It took me half an hour to roll everything back.

I learned my lesson. Same request, rephrased: "Change the heading font-size from 16px to 18px, and paragraph spacing from 10px to 14px. Only these two changes — don't touch anything else." Claude finished in 30 seconds. Perfect.

What changed? Claude didn't get smarter — I got clearer. This chapter is about exactly that: not teaching you fancy prompts, but teaching you how to speak plainly so Claude executes accurately.

How to Talk So Claude Understands Describing What You Want

Many people's first Claude Code prompt is something like "build me a user management system." Claude will do it — but what comes out probably isn't what you had in mind. Too little information, so it can only guess.

The official Best Practices distills this into three principles that I've found genuinely useful:

1. **Be specific: name files, context, and preferences**
   Don't say "add a login feature." Say "add Google OAuth login under `src/auth/` using the Better Auth library, following the pattern of the existing GitHub login implementation." File paths, tech choices, reference patterns — the more specific you are, the clearer Claude's direction.

2. **Point to existing patterns: "Do it like this"**
   Already have a well-written UserWidget in your project? Just tell Claude: "Look at how `src/components/UserWidget.tsx` is implemented, then build a CalendarWidget the same way." Claude's ability to read code is exceptional — giving it a reference example beats writing ten lines of description every time.

3 **Describe symptoms, don't diagnose causes**

When you hit a bug, don't say "there's a problem with the token refresh logic" (unless you've confirmed it). Say "users fail to log in after session timeout — please check the token refresh flow under `src/auth/`." Claude can see all the code. Let it locate the root cause — its guess will be more reliable than yours.

A few Before/After comparisons make this concrete:

不推荐

```
Add a search feature
```

推荐

```
Add a search box to the navbar in
src/components/Header.tsx, using Fuse.js
for fuzzy search over the posts array,
styled to match the existing
FilterDropdown component
```

不推荐

```
The API is throwing an error, take a look
```

推荐

```
POST /api/orders returns 500 when
quantity > 100 — check the input
validation and database write logic
in src/api/orders.ts
```

不推荐

```
Optimize performance
```

推荐

```
The homepage takes 4 seconds to load.
The main bottleneck is the Dashboard
component — it fetches all user data
at once. Switch to pagination, 20
records per page
```

Context Engineering: More Information Isn't Always Better Context Engineering

Chapter Ten covers the three-layer architecture of Harness Engineering in detail: Prompt, Context, and Harness. For now, let's focus on Context.

Context is more than just the message you type. It includes everything in CLAUDE.md, files Claude has read, screenshots you've pasted, and the full conversation history — all of it counts.

Your intuition might be: the more information I give Claude, the better, right?

Actually, the opposite is true.

Anthropic's engineering team found that **too much context degrades model performance.** It gets lost in the noise and starts making incoherent decisions. Shrivu recommends regularly using the `/context` command to check context window usage. In his monorepo tests, just loading the base configuration in a fresh session consumed roughly 20k tokens — leaving 180k for actual work.

> **核心建议**
>
> The core principle of context management: **don't give all the information — give the right information.** Show Claude what it needs to solve the current problem, not an encyclopedia of your entire project.

Here are a few ways to actively manage context:

- **`@` file references**: Use `@src/utils/auth.ts` to point Claude to a specific file
- **Paste screenshots**: For UI issues, a screenshot is ten times more accurate than a text description
- **Pipe data**: `cat error.log | claude` — feed logs directly to Claude
- **Share URLs**: Claude can fetch web content — linking to API docs is better than copy-pasting them

Let Claude Interview You

Let Claude Interview You

When you're about to build something substantial — say, a payment system from scratch — don't start by writing a requirements document. Instead, tell Claude:

```
I want to build a payment feature. Before you start,
interview me and ask everything you need to know.
```

Claude will ask you a series of questions: Which payment methods do you need to support? Do you need to handle refunds? What's your expected concurrency? Do you need webhook callbacks? What currencies?

At least half of those questions will be things you hadn't considered. Claude just did the work of a requirements analyst for you.

Once the interview is done, ask Claude to compile the answers into a Spec. Then — here's the key move: **open a fresh session**, feed the Spec to a new Claude instance, and let it execute.

Why a fresh session? Because the interview conversation has already accumulated a lot of history, consuming a large chunk of context. A new session starting from a clean Spec lets Claude focus entirely on execution — without the noise of everything discussed along the way.

Treat Claude Like a Senior Engineer Claude as Your Senior Engineer

Many people only use Claude Code as a code-writing tool. But it's equally valuable as a codebase navigator.

You can ask it directly:

- "How does logging work in this project?"
- "What's the process for adding a new API endpoint?"
- "What's the call chain for this `useAuth` hook?"
- "What's the difference between `src/lib/db.ts` and `src/utils/database.ts`? Why do both exist?"

Claude will read the relevant code and give you a structured explanation. Faster than reading documentation, more convenient than asking a colleague — especially when you're onboarding to a new project.

That's exactly how Boris's team uses it. New members don't start by reading a pile of wikis — they just ask Claude Code. Its understanding of the codebase is often more accurate than the outdated docs.

> **Onboarding accelerator:** When joining a new project, spend 10 minutes asking Claude Code: "What's the architecture of this project? What are the core modules? How does data flow through the system?" You'll save at least half a day of digging through documentation.

Multi-turn Conversation Strategy Multi-turn Conversation Strategy

Conversations with Claude Code are rarely one-shot. You'll often need to work through a task across multiple turns. Here are a few battle-tested strategies:

Tight feedback loops

Don't wait for Claude to write 500 lines of code before checking. **If you see it going sideways, correct it immediately.** The earlier you correct, the lower the cost. If you say "that's not right, try a different approach" after 10 lines, the cost is nearly zero. Waiting until a full feature is built before tearing it down wastes tokens and time.

Two corrections and still off? Start over

Corrected Claude twice and it's still not doing what you want? Stop correcting. Use `/clear` to wipe the context and start fresh with a better initial prompt. Trying to untangle a conversation that's already gone off the rails usually makes things worse.

Switch tasks, clear context

Finished a component and now moving on to a database schema change? Use `/clear`. Different tasks have different context needs — carrying over conversation history from the previous task just adds noise. Shrivu's recommended approach: after `/clear`, run a custom `/catchup` command that tells Claude to read the changes in the current git branch to restore relevant context.

Use subagents for research

Sometimes you need Claude to research before it acts: "figure out how this library works," "analyze how competitors implement this." These research tasks can be delegated to a subagent — the findings return to the main session without the intermediate reasoning polluting your main context.

> **注意**
>
> Shrivu specifically warns: don't rely on `/compact` (auto-compression). It's opaque and error-prone. **When you need to reset, use `/clear` — not `/compact`.**

Effort Level: Don't Skimp Here Effort Level

Claude Code has four effort levels: Low, Medium, High, and Max. They control how much reasoning the model invests in a task.

Level	Best For	Characteristics
Low	Simple formatting, renaming	Fast, but prone to careless mistakes
Medium	Routine development tasks	Lighter than the default
High	Complex features, debugging	Default level — what Boris uses
Max	Extremely complex architectural decisions	Uncapped reasoning tokens — slowest and deepest

High is already the default, and Boris never dials it down. His reasoning mirrors why he sticks with Opus: Claude thinks more deeply, needs fewer revisions, and ends up being more efficient overall.

Many people think "this task is simple, I'll drop it to Low to save time." But if Low makes a mistake, the time you spend fixing it will likely exceed what High would have taken to get it right the first time.

> **核心建议**
>
> If you're on the Max plan, High is already your default — no adjustment needed. **Don't lower the effort level to save a few seconds. Fixing low-effort mistakes always costs more than those few seconds.**

Three Principles for Better Prompts The Art of Asking

Working with Claude Code really comes down to three things.

Be specific. File names, line numbers, function names, expected behavior — share whatever you have. The more specific the instruction, the more precise the output.

Point to examples. Your codebase definitely has parts that are well-written. Use them as reference models. "Do it like that" is 100 times more effective than "make it nice."

Stay focused. One thing at a time. For larger tasks, work in steps — confirm each result before moving on. If you pack three unrelated requests into one message, Claude will likely nail only one of them.

The mental model I find most useful: treat Claude like a brilliant new hire. Highly capable, but unfamiliar with your project's history and conventions. The more precisely you provide context, the closer the output will be to what you actually want.

The core of this chapter: Good conversations don't depend on elaborate prompts — they depend on precise context. Specify your needs clearly, let Claude interview you to fill in your blind spots, keep things clean with `/clear`, and don't lower the effort level. Ultimately, the most effective advancement comes from building intuition through practice — not from learning more prompt tricks.

§07 Extending Claude Code: Skills, Hooks & MCP

Extensions: Skills, Hooks & MCP

As you use Claude Code more, you'll realize its true value isn't how capable it is out of the box — it's how much you can plug into it. Skills, Hooks, and MCP are three extension mechanisms that transform it from a terminal tool into an infinitely expandable workbench.

Why Extensions Matter

When I first installed Claude Code, I figured that was all there was to it. Then I noticed I kept repeating myself: reminding it to "run lint first" before every commit, re-explaining project conventions every time I created a new component, manually copying SQL results to paste into Claude every time I needed to query data.

Once you catch yourself doing the same thing more than three times, it's time to automate it. Claude Code provides three extension mechanisms, each solving a different layer of the problem:

Mechanism	Nature	Certainty	Best For
Skills	Markdown instruction packages	High but not 100% (advisory)	Domain knowledge, reusable workflows
Hooks	Shell script triggers	100% guaranteed execution	Formatting, lint, security checks
MCP	External tool connectors	100%	Databases, APIs, third-party services

Think of them this way: **Skills teach Claude how to do things**, **Hooks enforce checks at critical moments**, **MCP connects Claude to the outside world**. Personally, I use Skills the most — I'm adding new ones almost every day.

Skills: Where I'd Suggest Starting

Skills are the easiest extension to get started with. The idea is almost deceptively simple: create a folder inside `.claude/skills/`, put a `SKILL.md` file in it, and Claude will automatically load its instructions based on context.

```
.claude/skills/ ├── react-component/ | └── SKILL.md # Standards and steps for creating
React components ├── fix-issue/ | └── SKILL.md # Standard workflow for fixing bugs └──
deploy-preview/ └── SKILL.md # Steps for deploying a preview environment
```

You can manually invoke a skill with `/skill-name`, or Claude will automatically decide whether to load one based on the conversation. If you say "help me create a new React component," Claude will automatically load the conventions from the `react-component` skill.

Two Types of Skills

Knowledge-based: These tell Claude "here's how things work in this project" — API conventions, coding style, project agreements. This type of skill reads more like documentation; Claude absorbs it and follows the rules.

Workflow-based: These tell Claude "here's the exact steps to take for this kind of task" — like `/fix-issue` (standard bug-fix workflow) or `/review-pr` (code review process). These are more like SOPs, with clear steps and checkpoints.

Boris has a practical rule of thumb: **if you do something more than once a day, make it a skill or command.** I'm even more aggressive — if something comes up twice, I write a skill for it.

> **Real Example: Creating a /techdebt Command**
> Write the entire "spot technical debt → assess impact → create issue → link to sprint" workflow as a skill. Whenever you find tech debt, just type `/techdebt` and Claude automatically walks through the whole process — evaluating priority, creating a GitHub issue, and applying the right labels.

Key Configuration for Workflow Skills

Workflow skills often perform side-effectful operations — creating issues, sending messages, triggering deployments. To prevent Claude from auto-triggering them at the wrong moment, add this line to the front matter of your `SKILL.md`:

```
---
disable-model-invocation: true
---
```

With this setting, the skill can only be invoked manually via `/skill-name` — Claude won't trigger it on its own initiative.

Installing Other People's Skills

Skills are shareable. Boris has compiled a set of his most-used skills that you can install with a single command:

```
mkdir -p ~/.claude/skills/boris && \
curl -L -o ~/.claude/skills/boris/SKILL.md \
  https://howborisusesclaudecode.com/api/install
```

After installing, you get Boris's daily workflows — his commit conventions, PR templates, code review standards, and more. The community is also constantly contributing new skills, and you can browse the marketplace inside Claude Code with `/plugin`.

> **核心建议**
> Best practice for writing skills: start with the sentence you say to Claude most often. If you always say "run the tests, format the code, then commit" before every push, that's a skill waiting to be written. Put those steps in a SKILL.md, and next time one slash command does it all.

Hooks: Not a Suggestion — Enforced

Skills have a natural limitation: they're fundamentally just "advice" to Claude. Claude will try to follow them, but compliance isn't 100% — especially deep into a long conversation, when it may simply forget. That's fine for most situations, but sometimes you need absolute certainty.

I learned this the hard way: I wrote "run eslint formatting after every file edit" in CLAUDE.md. The first few rounds of conversation went fine, but once the context got compressed further into the session, that rule disappeared.

Hooks exist to solve exactly this problem.

Hooks vs. CLAUDE.md: The Fundamental Difference

A lot of people write rules in CLAUDE.md like: "please run `npx eslint --fix` after modifying any file." This works most of the time, but Claude occasionally forgets — especially in long conversations after context compression.

CLAUDE.md is advisory. Hooks are enforced. CLAUDE.md influences Claude's behavior through natural language; Hooks are a platform-level mechanism in Claude Code that trigger shell scripts at specific lifecycle points. Claude cannot skip or ignore them.

Lifecycle Hooks

Hooks support multiple trigger points:

Hook	When It Fires	Typical Use
PreToolUse	Before Claude calls a tool	Intercept dangerous operations
PostToolUse	After Claude calls a tool	Auto-format, auto-test
PermissionRequest	When user authorization is needed	Auto-approve low-risk operations
Stop	When Claude finishes a turn	Push Claude to continue executing
PostCompact	After context compression	Re-inject critical instructions to prevent amnesia
PermissionDenied	After auto-mode classifier rejects an operation	Log rejected operations, notify user, trigger fallbacks

Practical Examples

Example 1: Auto-formatting. Run eslint automatically after every file Claude edits — no relying on Claude to "remember" to format.

```
// .claude/settings.json
{
  "hooks": {
    "PostToolUse": [
      {
        "matcher": "Edit|Write",
        "command": "npx eslint --fix $CLAUDE_FILE_PATH"
      }
    ]
  }
}
```

Example 2: Auto-approve low-risk operations. Use a PermissionRequest hook to route permission requests to a script. The script evaluates the operation type — low-risk ones (reading files, running tests) are auto-approved; high-risk ones (deleting files, pushing code) still prompt for confirmation.

Example 3: Re-inject critical instructions after context compression. In long conversations, Claude compresses context to save tokens. After compression, important early instructions can get lost. A PostCompact hook can automatically re-inject critical rules after compression happens, ensuring Claude doesn't "forget."

Example 4: Push Claude to keep going. Sometimes Claude will pause mid-task and ask "should I continue?" A Stop hook can detect this and automatically prompt Claude to keep executing — ideal for unattended batch processing.

> **核心建议**
> You don't need to write hooks from scratch. Just tell Claude: "Write a hook that runs eslint after every file edit" — it will generate the configuration and write it to `.claude/settings.json` for you.

MCP: Giving Claude a Window to the Outside World

Skills give Claude knowledge, and Hooks guarantee execution certainty — but both operate within Claude Code's internal world. If you need Claude to directly query a database, call an API, or read a design file, you need MCP.

MCP (Model Context Protocol) is an open standard from Anthropic that lets AI tools connect to external data sources and services. Think of it as Claude Code's USB port: plug in different MCP servers, and Claude gains the corresponding capabilities.

Adding an MCP Server

```
# Add an MCP server
claude mcp add slack -- npx -y @modelcontextprotocol/server-slack

# List installed MCPs
claude mcp list
```

Once added, the MCP server's capabilities are exposed to Claude as "tools." After installing the Slack MCP, for instance, Claude can search Slack messages, send messages, and create channels.

Recommended MCPs to Start With

MCP	Capability	Best For
Slack MCP	Search/send messages	Auto-sync progress, reply to questions
Database MCP	Direct database queries	No more manually copying SQL results
Figma MCP	Read design files	Turn designs directly into code
Sentry MCP	Fetch error logs	Claude auto-locates production bugs
GitHub MCP	Manage repos/Issues/PRs	Automate project management

Boris has a classic setup: he connected Claude Code to the Slack MCP so that when someone reports a bug in Slack, Claude automatically reads the bug description, finds the relevant code, attempts a fix, submits a PR, and replies in Slack with "Fixed — PR link here." The entire process requires no human intervention.

MCP Configuration File

MCP configuration lives in `.mcp.json` at the project root. You can commit it to your Git repository, so team members who clone the project automatically get the same MCP setup.

```
// .mcp.json
{
  "mcpServers": {
    "slack": {
      "command": "npx",
      "args": ["-y", "@modelcontextprotocol/server-slack"],
      "env": {
        "SLACK_TOKEN": "${SLACK_TOKEN}"
      }
    },
    "postgres": {
      "command": "npx",
      "args": ["-y", "@modelcontextprotocol/server-postgres", "${DATABASE_URL}"]
    }
  }
}
```

> **注意**
>
> MCP servers gain access to external services. Before adding a new MCP, make sure you understand exactly what data it can access. Never hardcode sensitive tokens in `.mcp.json` — reference them via environment variables.

Plugins: Pre-Packaged Extensions

Skills, Hooks, and MCP are each useful on their own, but combining them is where things get really powerful. Plugins are exactly that — a packaged combination of all three.

Type `/plugin` in Claude Code to browse a growing plugin marketplace. Each plugin may include any combination of skills, hooks, agents, and MCP configuration — one install sets everything up.

For example, a "Code Intelligence" plugin might include:

- A skill: teaches Claude how to use symbol navigation to understand code structure
- A hook: automatically runs type-checking after edits
- An MCP: connects to a language server for precise symbol information

Together, these make Claude significantly more accurate when reading and modifying code — and you only need to install it once.

Slash Commands: Shortcuts with Pre-Computation

Beyond using `/skill-name` to invoke skills, there's a more flexible option: Slash Commands.

Commands live in the `.claude/commands/` directory. Unlike skills, commands can include inline Bash scripts to pre-compute information. Before Claude even reads the prompt, the command runs some shell commands and embeds the results into it.

```
# .claude/commands/commit-push-pr.md

Help me complete the following:

1. Review the current git diff:
```bash
git diff --stat
```

2. Generate a commit message and commit
3. Push to the remote branch
4. Create a Pull Request with a title based on the commit content

Note: The PR description should include a summary of the changes.
```

Type `/commit-push-pr` and Claude executes the entire workflow automatically. Because the command file lives in `.claude/commands/`, it gets committed with Git and is available to every team member.

> **Skills vs. Commands: A Decision Guide**
> There's overlap, but the intent differs. Skills are more like "knowledge and capability" — Claude applies them based on context or on manual invocation. Commands are more like "macros" — they include pre-computation steps and emphasize execution flow. Rule of thumb: if you need Claude to "know something," use a skill; if you need Claude to "do a sequence of things," use a command.

How the Three Mechanisms Work Together

In real projects, all three mechanisms often work in concert. Here's a complete example:

Imagine your team has this workflow: receive bug report → locate the issue → fix it → run tests → submit PR → notify stakeholders.

- **MCP** (Slack) lets Claude receive bug reports and reply with fix results

- **Skill** (fix-issue) guides Claude through a standard process for locating and resolving the issue
- **Hook** (PostToolUse) ensures tests and formatting run automatically after every change

Each piece is valuable on its own. Combined, they form a fully automated bug-fix pipeline.

HuaShu's (花叔) Skills in Practice: From Zero to 60+

Enough about the mechanics — here's how I actually use this stuff. I currently have over 60 Skills installed in Claude Code, covering content creation, ebook publishing, video production, project management, and more. These didn't appear all at once; they accumulated over six months, each one squeezed out by a real need.

The First Skill: Three-Pass Proofreading

The moment I realized "I should write a Skill" was during article proofreading. After finishing a draft, I'd always dictate the same wall of instructions to Claude: "reduce the AI tone, no em-dashes, avoid filler phrases like 'to put it simply,' check for any fabricated data, go through every paragraph…" I'd recite this for every single article, and still miss a few points each time.

So I wrote my first Skill: `huashu-proofreading`.

```
# SKILL.md core structure (simplified)

Three-pass proofreading workflow:

Pass 1: Factual verification
- Are all data points, dates, and product names accurate?
- Is anything fabricated?

Pass 2: Reduce AI tone
- Check for 6 categories of AI-speak: excessive em-dashes, triple parallel structures, hypoth
- Banned phrases: "to put it simply," "in other words," "simply put"...

Pass 3: Style refinement
- 「」quotation mark conventions
- Bold key sentences (~10 per article)
- Paragraph density check
```

After writing this Skill, proofreading went from 15 minutes of verbal dictation to a single `/proofreading` command. More importantly, the rules no longer vanish when context gets compressed.

Expanding from Code to Content Creation

Once I tasted the benefits, I started Skill-ifying every repetitive workflow I had. Over six months, I built a Skills system that covers an entire content creation pipeline:

Stage	Skill	What It Does
Topic Generation	`/topic-gen`	Generates 3–4 topic directions, each with a title, outline, and effort estimate
Research	`/research`	Multi-round WebSearch + incremental saves to a research file to prevent session-cut data loss
Writing	`/article-edit`	Standardized editing flow: full read → list changes → incremental edits → change summary
Images	`/image-upload`	AI-generated images → upload to image host → insert Markdown links, fully automated
Proofreading	`/proofreading`	Three-pass review to reduce AI tone, bringing it from ~60% to below 30%
Distribution	`/article-to-x`	Condenses long articles into 200–500 word social media content
Publishing	`/feishu`	Write to Feishu doc + set permissions + send group notification

Every stage of an article's lifecycle — from topic selection to publishing — has a corresponding Skill. Each Skill encodes lessons learned the hard way.

The Most Unexpected Use Case: Ebook Publishing

I have a Skill called `huashu-book-pdf` that handles the full Orange Book pipeline: research → chapter planning → multi-Agent parallel writing → HTML fragment assembly → EPUB generation → WeRead upload. This Skill compresses an entire ebook production process into a matter of hours.

The video side is even more interesting. `huashu-video-director` starts from a creative concept, works through character setup, storyboarding, and prompt generation, then calls AI video generation models to produce a multi-shot video. `huashu-script-polish` goes even deeper: it rewrites written scripts into natural spoken language, running three rounds of review to ensure it "sounds like a real person talking."

Skills Aren't for You — They're for the AI

There's a mindset shift worth making explicit here.

A lot of people's first reaction is: "A Skill is just an SOP document — I could follow it myself and get the same result." Technically true, but it misses the point entirely.

The audience for a Skill isn't you — it's the AI. The phrasing, structure, and checkpoints are all optimized for Claude to execute, not for humans to skim. Humans read documentation selectively; AI executes it

word by word. Humans forget edge cases; AI doesn't. Humans need willpower to follow an SOP; AI doesn't.

Some of my Skills total over 2,000 lines of instructions. If I had to execute those steps myself, each article would cost 3–4 hours of process work alone. With Skills, I spend that time on the things that actually require judgment: Is this topic worth writing about? Does the argument hold up? Will readers be able to act on this after reading it?

A Growing Trend: More and more third-party tools are shipping Claude Code Skills on their launch day. LibTV (an AI video generation platform), for example, published a corresponding Skill the day it launched — users don't need to learn a new interface, they just generate videos straight from the terminal. Enterprise platforms like Feishu and DingTalk are also rolling out CLI integrations. This means the Skills ecosystem will only get richer. Before long, a single `/xxx` command may accomplish what currently takes five separate apps to do.

核心建议

Don't try to build a comprehensive extension system all at once. Start with your single most painful repetitive task: always dictating the same rules? Write a skill. Keep forgetting to run lint? Add a hook. Constantly shuttling data around manually? Connect an MCP. Add them one at a time, and gradually your Claude Code becomes a workbench built exactly for you.

§08 Multi-Agent Collaboration

Multi-Agent Collaboration

The most underrated capability of Claude Code isn't how fast it writes code — it's how many instances you can run at once. Once you master parallelism, your workflow shifts from "one person, one AI" to "one person commanding an AI team."

While writing this Orange Book, I ran 6 Claude Code processes simultaneously, each responsible for a different chapter. Agent A wrote § 01 and § 02, Agent B wrote § 03 and § 04, and so on. Six processes ran in parallel for about 2 hours, and all 10 chapter drafts were done. Running them sequentially would have taken at least a full day.

But there's a real cost. Six agents means six separate contexts — token consumption is 6x that of a single agent. That afternoon cost around $50. And the writing style across the six agents wasn't perfectly consistent; I still needed a full pass to unify the voice. So more isn't always better. The real question is: can this task be split into independent pieces? And how expensive is the merge afterward?

Why You'd Want That Many Windows Open

Here's how Boris Cherny works day-to-day: 5 Claude Code instances running locally (each in its own git checkout), plus 5 to 10 claude.ai/code browser sessions in the cloud, each handling a different task. One writing a new feature, one fixing bugs, one writing tests, one refactoring, one doing code review. All at the same time.

My first reaction was: that's excessive. Then I tried it and understood — this isn't showing off. It's the first productivity tip Boris gives his team: **do more work in parallel.**

The reason is simple: Claude Code's work pattern is "you give a task → Claude spends a few minutes on it → you review the result → you give the next task." There's a lot of waiting in between. With one session, you spend most of your time waiting for Claude to finish. With five sessions, you're reviewing the first one while the other four are still running — wait time drops to nearly zero.

The key prerequisite: **each session needs to run in its own isolated code environment**, otherwise they overwrite each other's files and create conflicts. That's exactly what Git Worktrees are designed to solve.

Git Worktrees: The Infrastructure for Parallelism

Git Worktrees let you create multiple working directories from the same repository, each on a different branch, with completely isolated filesystems. Claude Code has native worktree support:

```
# Start a Claude session running in an isolated worktree
claude --worktree

# Start inside a Tmux session (can run in the background)
claude --worktree --tmux
```

Each time you run `claude --worktree`, Claude Code automatically creates a new worktree, checks out a new branch, and works in that isolated environment. When it's done, you can merge the branch back into main.

Tmux Integration

With the `--tmux` flag, the session launches inside a Tmux window, and you can switch between sessions using keyboard shortcuts. Boris uses shell aliases for quick navigation:

```
# Add to ~/.zshrc
alias za="tmux select-window -t claude:0"
alias zb="tmux select-window -t claude:1"
alias zc="tmux select-window -t claude:2"
```

`za` jumps to the first session, `zb` to the second, and so on. If you're using the Desktop App, there's a worktree checkbox in the UI — just tick it, no Tmux configuration needed.

Subagents: Calling in a Specialist

Parallel sessions are great for independent tasks that have nothing to do with each other. But sometimes you don't want a new window — you want to call in an "expert" to handle a specific step within your current task, like having a security reviewer look at the authentication code you just wrote. That's what subagents do.

Drop a `.md` file in the `.claude/agents/` directory, and you've defined a subagent:

```
.claude/agents/ ├── security-reviewer.md # Security review specialist ├── code-
simplifier.md # Code simplification specialist ├── verify-app.md # Application
verification specialist └── code-architect.md # Architecture design specialist
```

Each agent file can define a custom name, tool permissions, permission mode, and even which model to use. A security review agent, for example, can be configured with read-only access (no code modifications) and set to use a model with stronger reasoning capabilities.

The Core Value of Subagents

The most important feature of subagents isn't "specialized division of labor" — it's **independent context**.

Each subagent runs in its own context window, without consuming the main session's context space. When the main session has grown long and is approaching its context limit, delegating a subtask to a subagent opens up a fresh "thinking space" without squeezing the main session's capacity.

You can even add "use subagents" to your prompt, letting Claude decide when to delegate subtasks. This causes Claude to invest more compute into completing complex tasks.

> **核心建议**
> A practical subagent combo: **security-reviewer** (automatically invoked whenever authentication, permissions, or data storage are involved) + **verify-app** (automatically starts the app and verifies functionality after changes are made). Together they cover the two most common verification needs: "is it written correctly?" and "does it actually run?"

Agent Teams: Let Them Coordinate Themselves

Worktrees let you manually manage multiple parallel sessions. Subagents let the main session call in a specialist. Agent Teams go further: multiple sessions can communicate with each other and coordinate their own division of work.

Agent Teams shipped in February 2026, and it's currently Claude Code's most powerful collaboration mode. The core idea is simple: instead of you coordinating multiple agents, the agents coordinate themselves. I wrote about a real test earlier where 3 AI teammates built a retro arcade game in 45 minutes — that was using this feature.

Writer/Reviewer Pattern

The most classic use case is one agent writing code and another reviewing it:

1. **Writer Agent writes code**
   Responsible for implementing features — writes code and runs tests according to requirements

2. **Reviewer Agent reviews code**
   Reviews the Writer's output, identifies issues, and suggests improvements

3. **Writer revises based on feedback**
   Improves the code after receiving review comments, forming an iterative loop

This pattern produces noticeably better results than a single agent writing alone. The reason is the same as with human teams: the person writing code tends to get locked into their own thinking, while the reviewer catches problems from a different angle. Two agents keeping tabs on each other visibly raises the quality of the output.

Test-Driven Pattern

Another highly effective pattern: one agent writes tests, the other writes the implementation. The test-writing agent first defines "what correct behavior looks like" based on the requirements; the implementation agent then satisfies those tests. This is TDD (Test-Driven Development), AI edition.

Agent Teams automatically share task state and messages — you don't need to manually copy information between agents. There's a team lead role that coordinates task assignment and progress.

Coordinator Mode: Four-Phase Coordination

Under the hood, Agent Teams have a more refined coordination mechanism. Complex tasks automatically go through four phases: multiple workers investigate the codebase in parallel (Research), then the coordinator synthesizes the findings into a specification (Synthesis), workers make targeted changes according to the spec (Implementation), and finally the results are verified (Verification). You don't need to configure this manually — Agent Teams automatically decides whether to run the full four-phase process based on task complexity.

Fan-out Batch Processing: AI-Scale Parallelism

Everything so far has been about a few agents collaborating on one task. Fan-out mode solves a different problem: **the same operation needs to be performed repeatedly across a large number of files**.

Non-Interactive Mode

Claude Code supports non-interactive mode — pass a prompt via the `-p` flag, which makes it easy to call from scripts:

```
# Execute a single task in non-interactive mode
claude -p "Migrate this file from JavaScript to TypeScript"
```

Combine this with a shell loop for batch processing:

```
# Batch migrate a set of files
for file in $(cat files-to-migrate.txt); do
  claude -p "Migrate $file from JS to TS" \
    --allowedTools "Edit,Bash(git commit *)" &
done
```

Notice the trailing `&`: this runs each Claude instance in the background in parallel. If you have 50 files to migrate, 50 Claude instances run simultaneously — work that would take a full day might finish in a few minutes.

The /batch Command

If you'd rather not write your own shell scripts, Claude Code provides the `/batch` command to simplify this process:

1. **Interactive planning**

   Tell Claude what you want to do (e.g., "migrate all React class components to function components"), and Claude analyzes the project and lists every file that needs to be touched

2. **Confirm and execute**

   You review the plan, and once confirmed, Claude launches dozens of agents to execute in parallel

3. **Aggregate results**

   Once all agents finish, Claude summarizes successes and failures — you only need to handle the handful of cases that didn't go through

This pattern is especially well-suited for large-scale refactors, code migrations, and bulk fixes. One person with Claude can match what an engineering team would spend a week doing on a migration.

You Don't Have to Watch the Screen

Everything above assumes you're working at a local terminal. But Claude Code also supports remote and asynchronous execution — you don't have to sit at your terminal the whole time.

Remote Control

Via the Remote Control feature, you can generate a connection link. Open it on your phone and you can remotely create and manage local Claude sessions. Perfect for starting a task during your commute or kicking something off before you head out. Boris mentioned he uses his iPhone with the Claude mobile app to start sessions in the morning, then continues on his desktop.

Claude Code on Web

If your development environment is in the cloud (or you just want to work from a browser), you can run Claude Code directly at claude.ai/code without installing anything locally.

/schedule: Cloud-Based Scheduled Tasks

```
# Set up a cloud-based scheduled task
/schedule "Check for outdated dependencies and create PRs"
```

`/schedule` sets up a Claude task that triggers on a schedule and runs in the cloud. The task runs on time even when your computer is off. Great for routine maintenance: dependency updates, security scans, daily report generation.

/loop: Long-Running Local Tasks

Some tasks need to run for a long time — monitoring CI status, running continuous integration tests. `/loop` lets Claude run unattended locally for up to 3 days. You go do other things; it keeps running in the background.

> **The mindset shift for async work**
> Traditional development is synchronous: you write code, run tests, wait for results. In async mode, you kick off a batch of tasks before bed and review the results in the morning. Think of AI as a "night shift team" — you set direction and make decisions during the day, and it executes overnight.

How Anthropic Uses It Internally

Anthropic published a white paper, "How Anthropic Teams Use Claude Code," documenting real usage patterns across internal teams. A few highlights worth sharing:

The **data infrastructure team** uses Claude Code to debug Kubernetes clusters. When a pod has issues, they let Claude read the logs, analyze the error stack, identify the root cause, and suggest fixes. Problems that used to require a senior engineer spending hours on, Claude can often point in the right direction within minutes.

The **security team** uses Claude Code to trace complex control flows. Security audits require tracking a request's full path from entry point to database — doing this manually is extremely time-consuming. They have Claude automatically trace and generate call flow diagrams.

The **marketing team** uses Claude Code to generate ad variations. A single campaign can need dozens of copy and asset combinations, which used to take designers and copywriters several days to iterate through. Now Claude batch-generates them, and marketing only handles selection and fine-tuning.

But the case that surprised me most was the **legal team**: a lawyer used Claude Code to build a phone tree system (automatically routing incoming calls to the appropriate legal counsel). This person was not an engineer and didn't know how to write code — but they built and shipped this system from scratch. That example made me realize Claude Code's audience has already expanded well beyond engineers.

A Few Things I've Learned Along the Way

Start with just 2 sessions. No need to open 10 from day one. First get comfortable switching between two: one for the main task, one for support work (writing tests, doing code review). Add more once that feels natural.

Give each session a clear role. Don't let every session do "whatever comes up." Assign responsibilities: this one handles the frontend, that one handles the backend, that one runs tests exclusively. The clearer the roles, the easier they are to manage.

Use git branches to isolate everything. Each session works on its own branch and merges via PR. Never let multiple sessions work on the same branch — I tried it once, and resolving the conflicts was a genuine nightmare.

Check in periodically — don't just let them run wild. Parallel doesn't mean hands-off. Glance at each session's progress every 15–20 minutes and course-correct early. Stopping a session that's gone off track is far cheaper than letting it finish and having to redo everything.

> **核心建议**
>
> Boris's summary resonates with me: think of parallel work like managing a remote team. You don't need to watch every person write every line of code, but you do need to know what everyone is working on, how far along they are, and whether anyone is stuck. Your job shifts from writing code to doing project management.

§09 Build a Complete Product from Scratch

Build a Complete Product from Scratch

The first eight chapters covered the individual parts. This chapter puts them all together. By following a real project from start to finish, you'll discover that Claude Code's true power isn't in any single feature — it's in the chain reaction that happens when those features work together.

The day CCTV came to interview me, the reporter asked if I could build an app live on camera. I opened the terminal, launched Claude Code, and had a working mini-product running in 10 minutes. The camera rolled the whole time.

Later, someone left a comment on Bilibili: "Can something built in 10 minutes even count as a product?" Honestly? No. What you get in 10 minutes is a working prototype. The gap between prototype and product is filled with a lot of work: handling user feedback, covering edge cases, performance tuning, app store review. Kitty Light took an hour to build the first version, but it went through months of iteration — a Pro version, a mini-program version, a HarmonyOS version.

That gap — from prototype to product — is exactly what this chapter is about. Not a 10-minute demo. A real product you can ship, people can use, and you can keep improving.

Why "AI Weekly Report Assistant" as the Example

I thought about this choice for a while. A good teaching project needs to check a few boxes: genuinely useful (not a to-do list you'll open twice and forget), appropriately complex (doable in half a day to a full day, but touching frontend, backend, API calls, and AI), and technically well-matched (Next.js + Tailwind is exactly where Claude Code shines).

The AI Weekly Report Assistant does something straightforward: connect to your GitHub, pull all this week's commits, use AI to summarize them into a readable report page, and share it with your team in one click. Every engineer does this manually every Friday — it takes about 30 minutes. We're going to cut that down to 10 seconds.

This project draws on almost every chapter that came before. You don't need to have memorized any of it — just follow along.

Phase 0: Don't Rush to Write Code

Early in my product-building days, I made a classic mistake: the moment an idea hit me, I'd immediately tell Claude to start coding. Halfway through, I'd realize I hadn't thought the requirements through, and I'd

have to tear everything down and start over.

Since then, I've built a habit: let Claude interview me first. It's actually very good at this.

```
claude "I want to build a weekly report tool. Before writing any code,
interview me to understand the requirements. Ask me questions one at a
time about target users, core features, and technical constraints."
```

Claude will start asking you questions like a product manager:

- Who are the target users? (Individual developers? Or a team?)
- What are the core features? (Just generate the report, or share it too?)
- Technical preferences? (Where will it be deployed? What framework?)
- Any design references? (What should the report look like?)

Once you've answered those questions, have Claude consolidate everything into a spec document:

```
claude "Based on our discussion, create a SPEC.md file that captures
all requirements, user stories, and technical decisions."
```

This SPEC.md becomes the anchor for the entire project. Everything that follows is built around it.

Why start a new session for development: Requirements gathering is a conversation-heavy process that burns through a lot of your context window. Once SPEC.md is locked in, open a fresh session to begin development. The new session will automatically load SPEC.md and CLAUDE.md, giving you a clean context for writing code. This is the "when to start a new session" principle from § 06 — applied in practice.

Phase 1: Project Initialization

New session, clean slate. One prompt to scaffold the project:

```
claude "Create a Next.js project called weekly-report with Tailwind CSS.
Set up the basic folder structure following SPEC.md requirements.
Include TypeScript, and configure ESLint."
```

Claude will execute a sequence of operations: run `npx create-next-app`, install dependencies, configure Tailwind, create the base directory structure. In your terminal, you'll see every command it runs and every file it creates.

When the project is ready, the file structure will look roughly like this:

```
weekly-report/
├── app/
│   ├── layout.tsx
│   ├── page.tsx
│   ├── api/
│   │   ├── github/route.ts
│   │   ├── summarize/route.ts
│   │   └── auth/[...nextauth]/route.ts
│   └── report/[id]/page.tsx
├── components/
├── lib/
├── CLAUDE.md
├── SPEC.md
├── tailwind.config.ts
└── package.json
```

Next comes a critical step: configuring CLAUDE.md. This is the core topic of § 05 — now put into practice:

```
# CLAUDE.md

## Project Overview
AI-powered weekly report generator. Connects to GitHub, summarizes
commits with AI, generates shareable report pages.

## Tech Stack
- Next.js 15 (App Router) + TypeScript
- Tailwind CSS for styling
- NextAuth.js for GitHub OAuth
- Claude API (via Anthropic SDK) for summarization

## Code Style
- Use server components by default, 'use client' only when needed
- API routes in app/api/, use Route Handlers
- Prefer named exports
- Error handling: always use try-catch in API routes

## Testing
- Run `npm run lint` before committing
- Test API routes with curl before building UI
```

With this CLAUDE.md in place, every new session or conversation going forward will already know the project's technical decisions and code conventions — no need to re-explain them each time.

Phase 2: The Build (The Most Time-Consuming — and Most Satisfying — Part)

Now we actually write code. I usually start with a round of architecture discussion in Plan mode before touching a single file. Type `/plan` in the terminal to switch:

```
"I need to implement the core flow: GitHub OAuth login → fetch this
week's commits → send to Claude API for summarization → display
the report. Let's discuss the architecture before coding."
```

Claude will lay out a complete technical proposal: how to design the API routes, how data flows through the system, how to break down the components. This is the stage where you raise questions and adjust the plan — like working through a design on a whiteboard with a senior engineer.

Once you're satisfied with the plan, exit Plan mode and let Claude start building:

```
"Plan looks good. Now implement it step by step. Start with GitHub
OAuth, then the commit fetching API, then the AI summarization."
```

Claude will follow the agreed plan, progressively creating files, writing code, and installing necessary packages. You'll watch it create `app/api/auth/[...nextauth]/route.ts` to configure NextAuth, then `lib/github.ts` to wrap the GitHub API calls, then `app/api/summarize/route.ts` to integrate the Claude API.

After each module, verify before moving on:

1. **GitHub OAuth**

   Have Claude run `npm run dev`, open the browser, click the login button, and confirm it redirects to GitHub's authorization page. If there's an error, paste it into Claude — it will fix it.

2. **Commit Data Fetching**

   After a successful login, test the API route with curl: `curl localhost:3000/api/github` — check that the returned commit data looks correct.

3. **AI Summary Generation**

   Feed real commit data into the summarization endpoint, and check whether the AI-generated report reads well and is free of hallucinations.

> **注意**
>
> Verify after every step. This is the single most important habit I've built from all the mistakes I've made. Don't let Claude write all the code at once before you test. The earlier you catch a problem, the easier it is to fix. I once let it write eight files in a row, then discovered the API route in the second file was wrong — everything after it was garbage.

Phase 3: Make It Look Good

The functionality works, but the UI is probably ugly. That's normal. I always get the features running first before caring about aesthetics.

Claude Code accepts image input. The most direct approach: take a screenshot of the current page and paste it in:

```
claude "Here's a screenshot of the current report page.
[paste screenshot]
Issues: 1) The header is too cramped 2) The commit list needs
better spacing 3) Add a share button in the top right"
```

Claude will look at your screenshot, understand the visual problems, and make precise changes to the relevant CSS and component code.

This "screenshot → feedback → fix" loop is incredibly efficient. In traditional development, you'd constantly flip between design mockups and code. Now you just screenshot, describe the problem, and wait for the fix.

After a few rounds, work on responsive behavior:

```
"Test the report page on mobile viewport (375px width). Fix any
layout issues. The share button should be full-width on mobile."
```

Claude will resize the browser viewport (if you're using VS Code integration) or directly adjust Tailwind's responsive class names.

> **核心建议**
>
> The most efficient approach during UI polish: list all your issues at once and let Claude handle them in bulk, rather than fixing one thing and reviewing. Batched feedback helps Claude better understand the overall design intent.

Phase 4: Extending the Stack

The core product works. Now it's time to put § 07 into practice: adding Skills, MCP, and Hooks to the project.

Create a Skill

Every time you want to generate a report, you have to open the project and run through a series of steps. Can you collapse that into a single command? Yes — with a Skill.

Create `.claude/skills/generate-report/SKILL.md` in the project root:

```
# /weekly-report

Generate this week's report.

## Steps
1. Run the dev server if not running
2. Call /api/github to fetch commits since last Monday
3. Call /api/summarize to generate the report
4. Open the report page in browser
5. Show the shareable URL
```

Once configured, type `/weekly-report` in any Claude Code session and it will execute all of the above automatically. One command, report in 10 seconds. That's the power of Skills: turning a repetitive workflow into a single keystroke.

Add MCP: Connect Slack

The report is generated — but manually posting it to a Slack channel is a hassle. Use MCP to connect them:

```
claude "Add a Slack MCP server so the generated report can be
automatically posted to #team-updates channel. Use the Slack
Web API with a bot token."
```

Claude will configure the MCP server and register the Slack connection in `.claude/mcp.json`. Once set up, your `/weekly-report` skill can include a final step: posting the report to Slack.

Set Up a Hook: Auto-Lint

One last extension: automatically run lint checks every time Claude commits code.

```
claude "Set up a pre-commit hook that runs ESLint and TypeScript
type checking. If there are errors, fix them before committing."
```

Now every time Claude Code runs `git commit`, it verifies code quality first. This is the classic Hook use case from § 07: injecting automatic checks at key checkpoints.

Phase 5: Deploy to Production

The product runs great locally. Time to make it accessible to the world.

```
claude "Deploy this project to Vercel. Set up the environment
variables for GitHub OAuth, Claude API key, and Slack bot token.
Also create a GitHub Actions workflow that runs lint and type
check on every PR."
```

Claude will run the deployment commands, configure environment variables, and create the CI/CD config files. You won't need to open the Vercel dashboard or hand-write a GitHub Actions YAML file.

When deployment is done, you'll get a live URL. Open it in the browser and run through the full flow: login → fetch commits → generate report → share. Confirm everything works.

If your team is also using Claude Code, you can add one more step:

```
claude "Add a Claude Code Action to the GitHub repo that
automatically reviews PRs for code quality and potential bugs."
```

Now every time someone opens a PR, Claude will automatically perform a code review and leave comments. The CI/CD loop is complete.

Looking Back: What We Used

Let's take stock of what we built in an afternoon. A complete web application, from idea to production, entirely from the terminal.

Phase	What We Did	Chapter Reference
Phase 0	Used Claude to interview ourselves, produced SPEC.md	§ 06 Conversation Techniques
Phase 1	Project scaffolding + CLAUDE.md configuration	§ 02 Installation + § 05 CLAUDE.md
Phase 2	Architecture discussion in Plan mode + implementation in Auto mode	§ 04 Core Workflow
Phase 3	Screenshot feedback + UI iteration	§ 03 Agent-Style Work
Phase 4	Skills + MCP + Hooks	§ 07 Extending Capabilities
Phase 5	Vercel deployment + CI/CD	§ 04 Git Operations

If everything goes smoothly, expect the whole process to take 5–8 hours. Phase 2 (core feature development) is the biggest time sink — OAuth setup and API debugging require repeated verification, and fiddly details like environment variables and callback URLs can each eat 10–15 minutes. If things don't go smoothly, a full day is perfectly normal.

What would the same project take if you wrote it by hand? A full-stack engineer who knows Next.js and OAuth well could probably do it in 2–3 days. Someone less familiar might need a week. The difference isn't just speed — more importantly, throughout the whole process you're making product decisions instead of writing implementation code. You're thinking about "what information should the weekly report include?" and "does the share page need a login?" — not hunting Stack Overflow for "how do I configure the NextAuth GitHub provider?"

Hard-Won Lessons

Let me be straight with you.

When Kitty Light hit #1 on the App Store paid charts, a lot of people asked if I had a development team. I didn't. From the first line of code to the app store approval, every line was written by AI. I've never written code by hand.

But that doesn't mean the process was easy. Far from it. I hit a lot of walls, and here are the most important lessons I took away:

1. Break Requirements Down — Give Claude One Step at a Time

The most common beginner mistake: excitedly dumping the entire product vision on Claude at once. "Build me a complete app with login, database, payments, push notifications, sharing..." Claude will start building, but what comes out is usually a mess.

My approach now is to break things into the smallest verifiable steps. Do login first, confirm it works. Then data storage, confirm it reads and writes. Then core business logic, then UI. Only move to the next step after the current one passes verification.

推荐

Give requirements like this:
"Implement GitHub OAuth login. After login, display the user's name and avatar on the page."

Once verified →
"Now add the commit-fetching API. Fetch all commits by the logged-in user across all repos from the last 7 days, and return them as JSON."

不推荐

Not like this:
"Build a weekly report tool with GitHub login, commit fetching, AI summarization, a nice UI, a share feature, Slack notifications, and deploy it to Vercel."

2. Get the Minimal Version Running First, Then Layer On

Related to the previous point, but different. Breaking things down is about how you give Claude instructions. This one is about product strategy.

When I built Kitty Light, the first version had exactly one feature: open the app, the screen goes white, brightness maxed out. That's it. I took a selfie with it, decided the lighting effect was decent, and only then started adding color temperature control, a brightness slider, and timed shooting.

The same applies with Claude Code. Build the simplest possible working version first. Use it yourself for a couple of days. Discover what you actually need before adding anything. The features you imagine in your head and the features you actually need in practice are often very different.

3. Verification Is More Important Than Development

I keep saying this. Claude Code writes code fast — fast enough that you might develop an illusion: it's writing so quickly, it must be right, right?

Not necessarily. AI-generated code needs to be verified, just like human-written code needs to be tested. The difference is scale: a person might write 200 lines in a day, which is manageable to verify. Claude Code can write 2,000 lines in an hour. If you don't verify as you go, problems compound and surface later at a much higher cost.

My rule: **the moment a feature module is done, open the browser or run the tests. Fix problems immediately. Don't let them pile up.**

4. Don't Tackle Too Many Unrelated Things in One Session

Every Claude Code session has a context window limit. If you're tweaking the frontend, debugging the backend, configuring deployment, and fixing bugs all in the same session, the context eventually becomes noise, and Claude's responses noticeably degrade in quality.

In practice, I usually split sessions like this:

- Session 1: Project scaffolding + foundation architecture
- Session 2: Core backend logic
- Session 3: Frontend pages and interactions
- Session 4: Testing and bug fixes
- Session 5: Deployment and CI/CD

Each session focuses on one category of work. Context passes between sessions through CLAUDE.md and the codebase itself. This is far more efficient than grinding through one endless session.

5. Product Sense Is Your Biggest Lever

This is the most important one.

Claude Code can write your code, polish your UI, configure your deployment, and fix your bugs. What it can't do is decide: what problem should this product actually solve? Who are the target users? Which features matter and which should be cut?

Those are product judgments. They come from your understanding of users, your read of the market, your honest assessment of your own capabilities. AI can multiply your execution speed by 10x — but if you're pointed in the wrong direction, you just arrive at the wrong destination 10 times faster.

Kitty Light's success had nothing to do with the quality of its code (it's genuinely unremarkable). It succeeded because it precisely solved a real problem: you want better selfies without downloading a complicated photo editing app. One simple feature, one simple problem solved.

> **The one-person company product rhythm:** Idea → MVP in one day → use it yourself for three days → test with 10 people → iterate on feedback → if it feels right, ship it → let the data speak. In this loop, Claude Code covers "MVP in one day" and "iterate on feedback." Everything else runs on your judgment.

Traps I Fell Into — So You Don't Have To

A few problems that come up constantly in real product development:

Trap	What It Looks Like	The Fix
Scope creep	Features keep getting added mid-build; it never feels done	Go back to SPEC.md. Anything not in scope goes in a todo list — not in the current session
Context pollution	Sessions grow too long; Claude starts forgetting the earlier code structure	Open a new session promptly; let CLAUDE.md and the codebase carry the context
Skipping verification	Let Claude write 10 files in a row, then discovered a bug in the second one	Verify every module before moving on. Slower is faster
Environment variable chaos	Works locally, breaks in production with undefined errors everywhere	List all environment variables in CLAUDE.md; use a checklist to verify before deploying
Over-delegating decisions to AI	Claude says "this is the best approach" and you adopt it without thinking	AI proposes, you decide. Architecture choices especially — always your call

Knowing these pitfalls in advance can save you a lot of rework.

At this point, you have everything you need to build products independently with Claude Code. The final chapter takes a step back and asks a few more fundamental questions.

§09b Pitfalls Guide: Where AI Coding Has Its Limits

Pitfalls & Boundaries

The previous chapters covered what Claude Code can do. This chapter is about what it does poorly, where you'll stumble, and how to avoid it. Everything here comes from real experience, not theory.

On "Forgetting": It Really Will Forget What You Said

This is the earliest and most painful pitfall I fell into.

Once, while writing a long article, I repeatedly told Claude across several rounds of conversation: "don't use em-dashes, don't use filler phrases." Claude complied for the first few rounds, but after about 40 minutes, it started using em-dashes again with abandon. Looking back, I realized my instructions had been swallowed up by context compression.

This isn't a fluke. Claude Code's context window is 1M tokens, but long conversations trigger automatic compaction — and compaction is lossy. What you said early on may only survive as a vague shadow by the end.

Lesson: Important rules always go in CLAUDE.md — never rely on saying them in conversation. Conversation gets compressed; CLAUDE.md is re-read fresh every session. This is exactly why I eventually turned all my writing rules into Skills rather than restating them each time.

On "Confident Wrongness": It Won't Say It Doesn't Know

Once I asked Claude Code to look up the latest pricing for an AI product. It gave me a set of numbers with complete confidence. I put them directly in my article. After publishing, a reader pointed out the pricing was wrong — that was data from three months ago.

Claude doesn't say "I'm not sure." It gives you a plausible-sounding answer based on its training data, even when that answer is already outdated. For AI tools specifically, three months is an entire product lifecycle.

Lesson: Any content involving specific data, dates, pricing, or product features must be verified with WebSearch. Don't use something just because Claude said it confidently. I now have this rule written into CLAUDE.md under "Absolute Rules" — it's number one.

On "Token Anxiety": It Can Get Very Expensive

Agent Teams is one of the most powerful features I've used, but also the most expensive. Each Teammate maintains its own context window, and token consumption is roughly 7× that of a normal session. Once I used Agent Teams to write 6 chapters of the Orange Book in parallel — a single afternoon cost around $50.

There's another easily overlooked cost: MCP servers. Every MCP server's tool definitions consume context space. At one point I had 8 MCP servers running simultaneously, and I later discovered that tool descriptions alone were eating 15% of my context — meaning every conversation had 15% less usable space.

Lessons:

- Use `/cost` and `/context` to check consumption regularly
- Shut down MCP servers you're not using
- Break large tasks into smaller sessions — it's cheaper and more stable than one massive session
- Use Agent Teams only when you genuinely need parallelism, not just to show off

On "AI Flavor": The Writing Looks Fake at a Glance

This is probably the problem I've spent the most time solving. Claude's Chinese writing has some very recognizable tells:

- Em-dashes (——) used constantly — normal writers rarely do this
- Triplet parallelism: "It can XXX, it can YYY, it can ZZZ"
- Hypothetical openings: "Imagine that…" or "If you've ever…"
- Filler phrase clusters: "Simply put," "In other words," "To put it plainly"
- Mandatory uplifting endings: "Let's embrace this future full of possibility together"

The first time I published an article Claude wrote directly, reader feedback came down to two words: AI vibes. It took me several months to develop a three-pass proofreading workflow aimed at bringing the AI detection rate from above 60% down to below 30%. That workflow is now a Skill (`/proofreading`) that every article must go through.

Lesson: AI writing a first draft is fine — publishing it directly is not. Run it through at least one round of AI-flavor reduction. If you're writing content for human readers (not code), this step matters more than the draft itself.

On "Going Off the Rails": It Wanders If You're Not Watching

Claude Code has a tendency: give it a vague request and it will "improvise." That sounds like a feature, but it regularly turns into a pitfall.

Once I asked it to "clean up the styling on this page," meaning I wanted a few spacing and font-size tweaks. It interpreted this as refactoring the entire CSS file, overwriting the layout I'd carefully tuned, and helpfully adding a pile of "improvements." It took me half an hour to roll back to where I'd started.

Another time I asked it to "fix a small bug." It fixed the bug — and also "took the opportunity" to refactor the surrounding code, introducing two new bugs in the process.

Lessons:

- Be specific: "change the heading font size from 16px to 18px" beats "clean up the heading" by a hundred times
- Give clear stopping conditions: "stop when tests pass," "only touch this one file"
- For complex changes, use Plan mode to discuss the approach first, then execute once confirmed
- Git is your safety net: always make sure you have a clean commit to roll back to before any major change

On "Dependency": You May Lose Your Own Judgment

This pitfall is more subtle than the ones above.

After six months with Claude Code, I noticed I was sometimes accepting its proposals without thinking. It says use approach A — I use approach A. Why? Because most of the time its suggestions were right, so I got lazy and stopped asking "why A and not B?"

Then once it recommended a tech stack I knew nothing about for a project. I built it out, only to find the performance was terrible — but I'd already put two days of work in. If I'd spent ten minutes thinking through the technical choices myself at the start, I could have avoided the whole detour entirely.

Lesson: Claude Code is an engineer, not a product manager. On execution, trust it fully. But directional decisions — what to build, which technology to use, who it's for — those have to come from you. Anthropic's own trend reports put it plainly: engineers use AI for 60% of their work, but only 0–20% of tasks can be fully delegated. That 80% requiring human judgment is precisely where the most value lies.

On "No Silver Bullet": Things It Genuinely Can't Do Well

After all that talk of pitfalls, here's a list of things I've found Claude Code genuinely struggles with:

Area	What Happens	My Workaround
Fine-grained UI tweaks	Pixel-level visual adjustments are hard to convey accurately in words	Show it a screenshot, or use Computer Use so it can see the screen directly
Long-term consistency	Style and conventions easily drift across multiple sessions	Write everything in CLAUDE.md; lock rules into Skills
Taste and aesthetics	It can execute a design, but it can't judge whether it's a good one	You set the direction; it does the execution
Current information	Training data has a cutoff; anything from 3+ months ago may be outdated	Verify all time-sensitive information with WebSearch
Collaboration scenarios	It doesn't understand team dynamics, interpersonal relationships, or org politics	Delegate technical decisions; keep human decisions for yourself
Extreme performance optimization	Routine optimization is fine, but peak performance tuning requires deep understanding	Let it do the first pass; review critical paths yourself

A Mindset Suggestion: Treat It Like a Junior Engineer

After more than a year with Claude Code, I think the healthiest mindset is: **treat it like an incredibly hardworking but occasionally careless junior engineer.**

It types fast, never complains about overtime, and can handle multiple tasks at once. But it occasionally makes basic mistakes that need checking. It won't proactively tell you "this direction might be a problem" — you have to judge that yourself. Its execution is exceptional but its big-picture vision is limited; you have to give it direction.

If you go in with that expectation, you'll find it exceeds expectations far more often than it disappoints.

A quote from something I said in a CCTV interview: The essence of the solo-creator economy isn't that everyone needs to build an app or become a one-person digital business. Its meaning is simpler: the door is open. You can choose to walk through it, or not — but at least this door, which was closed before, is open now. Using Claude Code well is the same: knowing where its capabilities end is what lets you truly use the capabilities it has.

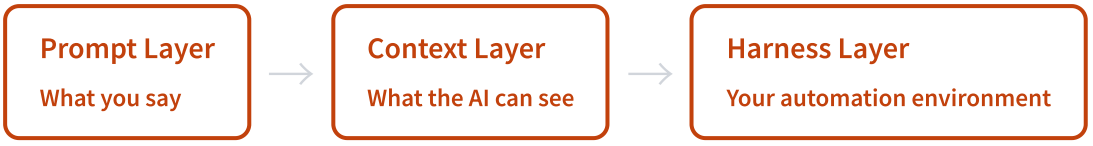
§10 Mental Models & Continuous Evolution

Mental Models & Continuous Evolution

Tools go obsolete, features get updated, but good ways of thinking never lose their value. This final chapter steps back to discuss a few things I consider more important than any specific technique.

The Three-Layer Model: Where Should Your Time Go

After covering so many specific operations, it's worth stepping back for a bird's-eye view. All of Claude Code's capabilities can be organized into three layers:



The **Prompt Layer** is every line you type into the terminal. "Add a login page." "Fix this bug." Most beginners never go beyond this layer. It works, but it requires you to speak up every time, starting from scratch each session.

The **Context Layer** is everything Claude has already "seen" before it responds to you. Your CLAUDE.md file, the project's file structure, git history, the dependency list in package.json. You don't have to repeat this information every time — Claude reads it automatically. The CLAUDE.md work we covered in § 05 is fundamentally about optimizing this layer.

The **Harness Layer** is the automation environment you build. Skills let you package common workflows into reusable commands; Hooks trigger actions automatically on specific events; MCP connects external services; Agent Teams let multiple Claude instances work in parallel. The defining characteristic of this layer: once it's built, it keeps working — no manual triggering required.

> **An analogy:** The Prompt is you speaking. The Context is the presentation you prepared in advance. The Harness is the entire stage you've built. The audience (Claude) performs according to the combined quality of all three.

Beginners pour all their energy into the Prompt Layer — obsessing over phrasing, studying prompting techniques. That's not wrong, but the ceiling is low. Experts do something different: **sink information into the Context Layer, hand off repetitive work to the Harness Layer, and use the Prompt Layer only for decisions that genuinely require in-the-moment judgment.**

Layer	How You Invest	Return Profile
Prompt	Re-invest every conversation	One-time return
Context	Write CLAUDE.md once, it keeps working	Compounding return
Harness	Build an automation once, it runs forever	Exponential return

If you take one thing away from this book, make it this: **invest your time in building Context and Harness, not in optimizing Prompts.**

Harness Engineering in Practice: Building Your Own Toolchain

The three-layer model is a theoretical framework — but "what exactly should I build?" is probably what you actually care about. Here's a practical path from nothing to a fully working Harness.

Step 1: Start with CLAUDE.md (Context Layer)

On day one of a new project, create `CLAUDE.md`. You don't need much. Start with three things:

```
# Project Name

## Tech Stack
- Next.js 15 + TypeScript + Tailwind CSS
- PostgreSQL + Drizzle ORM

## Conventions
- Components go in src/components/, organized into feature subdirectories
- API routes go in src/app/api/
- Commit messages in English, format: type: description

## Known Gotchas
- Drizzle's migrate command requires DATABASE_URL to be exported first
- When deploying to Vercel, env variable names cannot start with an underscore
```

That third section — "Known Gotchas" — is especially important. Every time you hit a snag, tell Claude "remember this," and it'll write it into CLAUDE.md. The next time the same situation comes up, Claude will proactively avoid the trap. This is exactly the "mistake → document → iterate" flywheel Boris described.

Step 2: Turn Repetitive Actions into Skills (Harness Layer, Beginner)

After a week or two, you'll notice yourself saying similar things to Claude over and over. Like how after finishing code you always say: "Run the tests, lint it, then commit."

That's your first Skill. Write this in `.claude/skills/ship/SKILL.md`:

```
---
description: Standard workflow for shipping code
---
1. Run all tests and confirm they pass
2. Run eslint --fix to format the code
3. git add changed files (do not add .env or other sensitive files)
4. Generate a concise commit message and commit
5. If a remote branch exists, push to remote
```

From then on, type `/ship` and all five steps run automatically.

Step 3: Add Hooks for Guaranteed Consistency

Skills are suggestions — Claude sometimes forgets them. For things that absolutely cannot be forgotten, use Hooks.

I have a PostToolUse hook that automatically runs type-checking every time Claude edits a TypeScript file. I also have a PostCompact hook that injects three core rules after the context gets compressed, preventing Claude from "forgetting" things during long sessions.

Step 4: Connect to the Outside World with MCP

When your project needs to interact with external services, connect MCP. The three I use most often:

- **Browser MCP** — lets Claude directly control Chrome: take screenshots, fill forms, read web content
- **Feishu MCP** — create documents, send messages, manage knowledge bases
- **Filesystem MCP** — operate on local files, useful for cross-project workflows

What a Complete Harness Looks Like

Using my own writing project as an example: after six months of iteration, it settled into this structure:

```
CLAUDE.md ← Router: dispatches to subdirectories based on task keywords 01-WeChat-
Writing/ CLAUDE.md ← Writing style, editing rules, publishing workflow Projects/2026.03-
SomeProject/ README.md ← Project status and file descriptions .claude/ skills/ huashu-
proofreading/ ← Three-pass editing huashu-research/ ← Structured research huashu-image-
upload/ ← Image generation + upload to image host huashu-feishu/ ← Feishu document
operations ... (60+ skills) settings.json ← Hooks configuration .mcp.json ← MCP server
configuration
```

The root CLAUDE.md stays under 8KB, and its only job is routing: if it sees the keyword "write article" it reads the WeChat writing CLAUDE.md; "make video" triggers the video production one. Each workspace has its own rules, and they don't interfere with each other.

This Harness wasn't built in a day — it was accumulated over six months, one rule at a time. The point isn't scale; it's **making sure that every time you encounter something repetitive, forgotten, or broken,**

you lock the solution into the Context or Harness layer. Over time, your Claude Code becomes a collaborator that understands you better and makes fewer mistakes.

> **Further reading:** If you're interested in the theory and more examples of Harness Engineering, I wrote a dedicated Orange Book — "Harness Engineering" — that breaks down the concept's origins, how teams at OpenAI, Anthropic, and Stripe put it into practice, and a complete methodology for building a Harness from scratch.

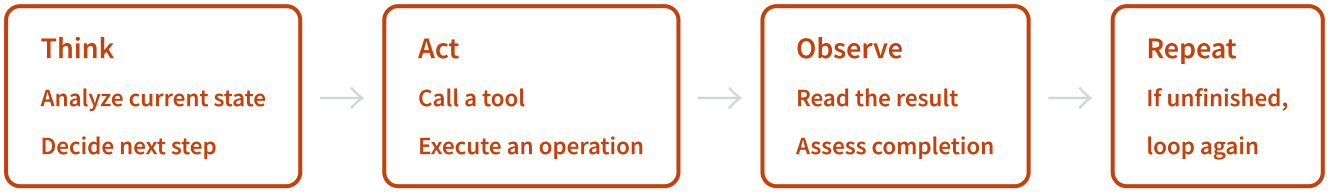
Under the Hood: Claude Code Under the Hood

After spending so much time with Claude Code, have you ever wondered: what actually happens after you press Enter?

I spent some time studying Claude Code's internal architecture. Not to show off, but because once you understand the mechanics, a lot of previously puzzling behavior suddenly makes sense. Why does it sometimes seem to "take the long way around"? Why does it occasionally forget details after `/compact`? Why does Auto mode let some operations through immediately while others require confirmation? None of this is random — there's clear design logic behind it.

The Core Loop: Think → Act → Observe → Repeat

At the heart of Claude Code is an agent loop called TAOR. When you enter a task, it doesn't generate a complete block of code and hand it over. Here's what it actually does:



It thinks through the current state and decides what to do next; calls a tool to execute an action (like reading a file or running a command); observes the result and judges whether the task is complete; if not, it loops back to Think. The whole process might cycle dozens of times before stopping.

This explains why Claude sometimes appears to "take the long way." It's not a linear input-to-output program — it's a cycle of continuous probing and adjustment. Every step is a decision made on fresh observations. Sometimes it tries an approach, finds it doesn't work, backtracks, and tries another path. That's by design, not a bug.

This is also why **giving Claude clear success criteria matters so much**. The loop needs a stopping condition. If your requirements are vague, it doesn't know when "done" is, and it'll keep cycling — changing things back and forth indefinitely. Tell it "stop when tests pass" or "stop once the file is generated," and it converges much faster.

The Tech Stack: React in Your Terminal

Here's a fun fact: the Claude Code interface you see in your terminal is actually rendered by React components.

Claude Code runs on Bun (not Node.js) and uses React's Ink framework to render the terminal UI. It's written entirely in strict-mode TypeScript, with Zod handling schema validation. The entry file is 785KB compressed — substantial for a terminal tool, but it reflects the density of features packed in.

Why does this matter? Because it explains how Claude Code can deliver such a rich interactive experience. Permission confirmation dialogs, multi-line syntax highlighting, progress indicators — things that are difficult to achieve in traditional terminal tools come naturally with React's component model. The "smoothness" you feel isn't an illusion; it's the result of deliberate engineering choices.

40+ Tools, 4 Capability Primitives

Claude Code has over 40 internal tools, each with independent permission controls. But step back far enough, and all capabilities reduce to 4 primitives:

Primitive	What It Does	Typical Tools
Read	Read files, read code, search content	Read, Grep, Glob
Write	Write files, edit code	Write, Edit
Execute	Run commands, execute scripts	Bash
Connect	Connect to external services	MCP tools, WebFetch

The elegant part of this design is the Bash tool. It's a universal adapter that lets Claude use every command-line tool human developers rely on. No need for dedicated integrations for each programming language, no plugins for each framework. `npm install`, `python test.py`, `git push` — Execute + Bash handles all of it. **This is why Claude Code works across virtually any tech stack**, unlike IDE plugins that only support specific languages.

Context Compression: Why Long Conversations "Forget"

You've probably experienced this: after a long conversation with Claude, it suddenly forgets something you mentioned earlier. Or after running `/compact` manually, it becomes hazy on certain details. It's not being lazy.

When the context window approaches capacity, the system compresses the entire conversation history into a summary. That summary becomes the starting point for the next round; the original conversation is discarded. **Compression is lossy.** Core information is preserved, but specific wording, edge-case details, and the tone of what you said — these tend to get lost in the process.

What's worse: in long sessions that go through multiple rounds of compression, **information loss accumulates**. Each compression drops a little more, and after several passes, your earliest context may survive only as a vague shadow.

> **核心建议**
> Practical tip: Put important constraints and requirements in CLAUDE.md rather than stating them once in the conversation. Conversations get compressed; CLAUDE.md gets re-read every time. This echoes the conclusion from the Three-Layer Model: sink information into the Context Layer.

The Permission System: More Than Yes/No

Auto mode isn't a simple blanket approval. Internally, a classifier rates the risk of each operation as LOW, MEDIUM, or HIGH. Reading a file is typically LOW — approved automatically. Writing a config file is MEDIUM or HIGH — you're asked to confirm.

Some files are hardcoded as protected: `.gitconfig`, `.bashrc`, `.zshrc` and other system-level configs are handled with extra caution regardless of permission mode. There's even a defense against path traversal attacks, preventing malicious code from bypassing permission checks via unicode characters or mixed case.

Each time a permission confirmation dialog appears, the explanation text you see isn't a preset template — it's generated in real time. The system makes a separate LLM call to produce it. So the wording varies slightly each time, and that's intentional, not instability.

Automatic Memory Maintenance

Claude Code runs a background sub-agent that periodically organizes your memory files (your CLAUDE.md and related configuration). It works in four steps: reviewing existing content, extracting new useful information, consolidating duplicate entries, and trimming sections that have grown too long. The goal is to keep memory at a reasonable size — roughly 200 lines.

This is why, after using Claude Code for a long time, you feel like it understands you better and better. It's not entirely that the model is getting smarter — it's that **your preferences, habits, and project context are being slowly accumulated and maintained by this memory system**.

Understanding Claude Code's internal mechanics isn't about cracking open a black box for its own sake. It's that once you know how the loop turns, how context gets compressed, and how permissions are evaluated, you can collaborate with it more effectively. You don't need to understand engine mechanics to drive a car — but once you do, you know when to shift gears.

A Shifting Identity: From Writing Code to Building Products

This shift is happening faster than most people expect.

Boris Cherny, the creator of Claude Code, has publicly said that over 90% of his code is generated by Claude Code. His day-to-day work now looks more like: describing requirements, reviewing output, making architectural decisions. He has a line I find striking: "My job now looks more like a product manager with strong technical judgment."

My own experience is more extreme. I have never written code by hand — every product I've built was created with AI, including Kitty Light (which reached #1 on the App Store paid chart). Many people find this hard to believe at first, but once you've actually used Claude Code, it makes complete sense. What determines whether a product is good has never been how clever the code is — it's how precisely the requirements were defined, and how smooth the user experience is.

What this means is that the critical skills are shifting:

Old Skills (Declining Importance)

- Syntax fluency
- Framework API memorization
- Manual debugging techniques
- Accumulating code templates

New Skills (Rising Importance)

- Requirement decomposition
- Architectural judgment
- Output quality review
- Product taste

Note that I said "declining importance," not "useless." Understanding code still has value — it helps you describe requirements more precisely and evaluate output more accurately. But you no longer need to be able to write a complete application from scratch. What you need is the ability to judge whether an application is well built.

The core of this shift is: **from "how to write it" to "what to write."**

You used to spend 80% of your time on "how to implement this feature" and 20% on "what features should exist." Now the ratio is reversed. Claude Code solves the "how to write it" problem, but the "what to write" question is largely yours to answer. You need to think it through yourself: what problem does this product solve? Who are the target users? What's the core experience? Which features are essential, and which can be cut?

> **核心建议**
> If you're anxious about "will AI replace me," try shifting the frame: focus on learning to define requirements, design interactions, and review quality. These abilities won't lose value as AI gets stronger.

How to Keep Up With a Fast-Moving Tool

Looking back at Claude Code's feature timeline, the pace of iteration has been genuinely fast:

- 024. MCP protocol launches, giving Claude Code the ability to connect to external services
- 025.(Public beta released, transitioning from an internal tool to a public product
- 025.(GA release, significant stability improvements
- 025.(SubAgents launched, enabling Claude to spin up sub-processes for parallel work
- 025.(Hooks introduced, making event-driven automation possible
- 025.1 Skills system released, letting the community share and reuse capability packages
- 026.(Agent Teams officially launched, bringing multi-agent collaboration into practical use
- 026.(Computer Use arrives, giving Claude the ability to control the screen; Voice Mode lets you speak directly to the terminal
- 026.(Source code accidentally made public, giving the community its first complete look at the agent system's architecture

One major feature roughly every two months. That means some of the specific steps in this book may need updating within three months of publication.

How do you keep up? Here are a few reliable information sources:

Primary official sources:

- Claude Code official changelog: detailed notes for every update
- Anthropic official blog: major features come with in-depth articles
- Anthropic Academy: over a dozen free courses covering everything from basics to advanced topics

From the creator and team:

- Boris Cherny's X account (@bcherny): the creator of Claude Code, regularly shares tips and design thinking
- howborisusesclaudecode.com: Anthropic's official practical guide page
- "How Anthropic Teams Use Claude Code" whitepaper: real workflows from the official team

High-quality podcast interviews (for understanding design philosophy):

- Lenny's Podcast: Boris on product design and the future of AI-assisted programming
- Pragmatic Engineer: in-depth conversation from a technical perspective

- YC Lightcone: founder perspective on how AI tools are changing the way products get built

> **注意**
> Don't try to track every minor update. Your time should go toward building things with the tool, not studying the tool itself. Spending 30 minutes browsing the changelog once a month is enough.

What really deserves your attention isn't specific feature changes — it's the direction. Over the past year and a half, three threads have remained constant:

1. **Autonomy keeps increasing.** From needing step-by-step instructions to independently planning and executing.
2. **Context windows keep growing.** From 8K to 200K to 1M — the scale of project Claude can "see" keeps expanding.
3. **Collaboration patterns keep evolving.** From single agent to SubAgents to Agent Teams — multi-agent collaboration keeps getting more natural.

This means that "how to collaborate with AI" — what you're learning today — won't go stale. Specific commands may change, but the core loop of "describe requirements → review output → iterate" isn't going anywhere in the foreseeable future.

Recommended Resources

If you finish this book and want to go deeper, here's my curated list. It's short — but every item is worth your time.

Essential

Resource	Type	Why I Recommend It
Claude Code Best Practices	Official docs	The authoritative source for all techniques, regularly updated
DeepLearning.AI x Anthropic Course Series	Video course	Co-produced by Andrew Ng's team and Anthropic — systematic and comprehensive
Anthropic Academy	Free courses	Over a dozen free courses covering Prompt Engineering through Agent development
How Anthropic Teams Use Claude Code	Whitepaper	Real workflows from the official team — not theory, practice

Going Deeper

Resource	Type	Why I Recommend It
awesome-claude-code	GitHub repo	Community-curated collection of plugins, Skills, and best practices
Claude Code Ultimate Guide	Community docs	Practical tips compilation covering edge cases the official docs don't reach
howborisusesclaudecode.com	Official page	Boris's complete workflow, continuously updated
Boris on Lenny's Podcast	Podcast	A deep conversation on "what happens after coding is solved"
Boris on Pragmatic Engineer	Interview	How Claude Code evolved from a side project into a core tool

A Closing Note

While writing this book, I kept coming back to a question: what's the endgame for AI-assisted programming?

I thought about it for a long time without a clean answer. But one thing I'm fairly confident about: over the past year and more, I've used Claude Code to build over a dozen products — from iOS apps to Chrome extensions to PDF generation tools. In every case, the final quality of the product wasn't determined by the AI's capability ceiling. It was determined by my judgment about what "good" means.

AI can write a login page in 30 seconds. But whether you need a login page, what the user should see after logging in, what a smooth experience actually feels like — those are still human calls to make.

So I have just one piece of advice: don't spend too much time studying the tool. Go build something. Find a problem you genuinely want to solve, open a terminal, and start talking to Claude. When you get stuck, flip through this book. Then keep going.

The distance from idea to product is now short enough that you might not have fully adjusted to it yet.

§11 Hands-on Project: Chrome Extension

Hands-on Project: Build a Chrome Extension

Build a Chrome browser extension from scratch using Claude Code. Not a Hello World toy — a real product that people actually use. You'll experience the complete journey from requirements analysis to packaging and installation, including the most authentic debugging and refactoring moments.

I built my Bilibili creator assistant plugin out of pure laziness. Dozens of comments needed replies every day — open the backend, click each comment one by one, type a response, hit send. After two weeks of that, I'd had enough. I opened a terminal and told Claude: "Help me build a Chrome extension that automatically replies to Bilibili comments."

The first version was up and running in about two hours. It worked, but all the code was crammed into a single content.js — 1,211 lines — where any change felt like defusing a bomb. I spent a few days later doing a v3.0 refactor: content.js shrank to 175 lines, and all the business logic was split into 7 independent modules. That process taught me far more than building the first version ever did.

In this chapter, I'll walk you through the complete workflow. The goal is "maintainable" — just "runnable" isn't nearly enough.

Why Chrome Extensions

After everything covered in previous chapters, a Chrome extension is a natural next step:

The technical barrier is just right. No need to learn Swift or set up a backend. HTML + CSS + JavaScript — exactly what Claude Code does best. Manifest V3's rules are more complex than V2, but Claude has a solid grasp of MV3. You describe what you want; it handles Service Workers, permission declarations, and all those details automatically.

There's a real use case. You use a browser every day. A good extension can genuinely change your daily workflow. Compared to building a todo list you'll never open twice, building a tool you use every single day is a completely different motivational experience.

The feedback loop is instant. Change the code, go to the Chrome extensions page, click "Reload," refresh the page, and you see the result. No compilation, no deployment, no waiting for review. That instant feedback lets you iterate fast.

Phase 0: Requirements Analysis — What Exactly Do You Want It to Do

The mistake I made early on was jumping straight to "build an auto-reply plugin." Claude gave me a very basic version: detect a new comment, reply "Thanks for your support." Functional, but not what I wanted at all.

Later I learned to switch into Plan mode first and talk through the requirements. Press `Shift+Tab` to enter Plan mode:

```
I want to build a Bilibili creator assistant Chrome extension. Core requirements:
1. Automatically scan video comments
2. Match comments against keyword rules and auto-reply
3. Support both global rules and per-video rules
4. Don't re-reply to comments that have already been replied to
5. A simple management panel to view running status

Help me analyze these requirements first and propose a technical approach.
```

In Plan mode, Claude outputs a complete technical proposal — file structure, technology choices, analysis of tricky parts. Don't rush to say "great, let's start." Read the proposal carefully and raise any questions or changes you want to make.

> 核心建议
>
> **Lesson learned:** Time spent discussing requirements in Plan mode is always less than time spent tearing things apart and rebuilding later. My first version skipped the discussion and went straight to coding — two hours. The v3.0 refactor took several days. If I'd spent 30 minutes upfront in Plan mode thinking through the architecture, most of that refactor work could have been avoided.

Phase 1: Project Initialization

Once the plan is confirmed, switch back to normal mode and have Claude create the project:

```
Based on the approach we discussed, create the Chrome extension project. Start with the skele
- manifest.json (MV3)
- background.js (Service Worker)
- content.js (Content Script)
- popup.html + popup.js (management panel)
- lib/ directory (business modules)

Only request permissions that are strictly necessary.
```

Claude will generate a complete project structure. manifest.json is the soul of a Chrome extension. Ours looks roughly like this:

```
{
  "manifest_version": 3,
  "name": "Bilibili Creator Assistant",
  "version": "3.0.0",
  "permissions": ["storage", "alarms", "activeTab"],
  "host_permissions": ["*://*.bilibili.com/*"],
  "background": {
    "service_worker": "background.js"
  },
  "content_scripts": [{
    "matches": ["*://*.bilibili.com/video/*"],
    "js": ["content.js"]
  }],
  "action": {
    "default_popup": "popup.html"
  }
}
```

A few key decisions worth explaining:

manifest_version: 3. MV2 has been deprecated by Chrome. Claude is reliable on this point — it automatically uses MV3 syntax. But be careful: many Chrome extension tutorials online still use MV2, and if you ask Claude to reference those tutorials, it may end up mixing the two APIs. I recommend adding a line in CLAUDE.md: "Must use Manifest V3."

Service Worker instead of Background Page. The biggest change in MV3 is that the background script becomes a Service Worker, meaning the browser can terminate it at any time. You can't use `setInterval` for scheduled tasks — you need `chrome.alarms`. Claude knows this distinction, but if your CLAUDE.md doesn't mention it, it will occasionally fall back to the old pattern.

Minimal permissions. Only request what you actually need. The Chrome Web Store review process is more likely to reject extensions with excessive permissions. And users will hesitate when they see a long list of permission requests.

Phase 2: Core Architecture — Who Owns What

The most common mistake in Chrome extension development is "putting code in the wrong place." MV3 has three runtime environments with completely different responsibilities:

Environment	File	Can Do	Cannot Do
Service Worker	background.js	Scheduled tasks, global state, message routing	Access page DOM, read page cookies
Content Script	content.js	Manipulate page DOM, read page context	Call chrome.alarms and similar APIs directly
Popup/Options	popup.js	User interface, configuration management	Run after the user closes the popup

My v1 lesson was stuffing everything into content.js. Scanning logic, API calls, rule matching, reply sending, state management — all in one file. 1,211 lines. Touching any single feature meant hunting through the whole thing.

The v3 architecture looks like this:

```
background.js (Command Center)
  ├── Owns the timed scan loop (chrome.alarms, once per minute)
  ├── Routes all messages (popup/content/widget → centralized handling)
  └── Manages global state (on/off switch, rate limit level, etc.)
       ↕ chrome.runtime.sendMessage
content.js (Front-line Scout)
  ├── Extracts current video info (BV ID, title, etc.)
  ├── Proxies API calls (requires page cookies)
  └── Forwards logs and status to the floating widget
       ↕ Message passing
lib/ (Business Brain)
  ├── db.js        → Unified data layer, all storage reads/writes go here
  ├── scanner.js   → Comment scanning engine
  ├── rules.js     → Rule matching logic
  ├── api.js       → Bilibili API wrapper
  ├── ai.js        → AI reply generation
  └── rate-limiter.js → 4-level rate limiting with graceful degradation
```

The key design principle: **content.js only does what it absolutely must.** What must it do? Read page cookies (background.js can't access these in MV3) and manipulate page DOM. Everything else — all business logic — lives in background.js and lib/.

You can describe this architecture to Claude Code like this:

```
Architecture principles:
1. content.js is a thin shell — only extracts video info and proxies API calls
2. background.js is the command center — owns the scan loop and state management
3. All business logic goes in independent modules under lib/
4. All data storage goes through lib/db.js — no direct chrome.storage calls elsewhere
5. Modules communicate via message passing, not shared state

Write these into the project's CLAUDE.md.
```

> 核心建议
>
> **Lesson learned:** Have Claude help you write the project's CLAUDE.md and encode the architecture principles there. Every future conversation will automatically respect them. This is the "give the AI a map" concept from §05 applied in practice.

Phase 3: Building Modules One by One

With the architecture defined, development becomes a matter of implementing one module at a time. I recommend this order:

Step 1: Data layer (db.js). This is the foundation. Every other module depends on it.

```
Build lib/db.js first — a unified data access layer. It needs these interfaces:
- getSystemState() / setSystemState(partial)
- getConfig() / setConfig(partial)
- getRules() / setRules(rules)
- getVideo(bvid) / setVideo(bvid, data)
- markReplied(bvid, rpid)

Storage structure:
- 'sys:state' → system state (on/off switch, rate limit level, etc.)
- 'sys:config' → configuration (scan interval, reply delay, etc.)
- 'sys:rules' → rules (global rules + per-video rules)
- 'v:{bvid}' → data for each individual video

All writes must be atomic: read → merge → write, to prevent concurrent overwrites.
```

Consolidating from 20+ scattered storage keys into 5 structured keys made every subsequent module simpler. No more "where is this data stored anyway?" — everything goes through db.js.

Step 2: Rule matching (rules.js). The smallest module at 62 lines. Input: a comment and a set of rules. Output: the match result. Pure function, no side effects, very easy to test.

Step 3: API wrapper (api.js). Wrap Bilibili's comment fetching and reply sending into clean functions. Note that Bilibili's API requires page cookie authentication, so these functions actually execute in the content.js context and return results via message passing.

Step 4: Scanning engine (scanner.js). Wire the previous three modules together. The core logic: fetch comments → filter already-replied ones → match rules → generate reply → send.

There's one key design choice here: dependency injection. scanner.js doesn't call the API directly — it receives functions passed in from outside:

```
async function scanOneVideo(bvid, rules, config, {
  sendReplyFn,        // function to send a reply
  fetchCommentsFn,    // function to fetch comments
  logFn               // logging function
}) {
  // business logic
}
```

Why this design? Because scanner.js runs inside background.js, but API calls need to execute in the content.js context. With dependency injection, background.js can pass in a function that "relays the call to content.js via messaging," while scanner.js doesn't need to know any of that detail.

Step 5: Rate limiter (rate-limiter.js). Bilibili restricts comment frequency. Reply five times in a second and your account gets temporarily throttled. Our rate-limiting strategy uses 4-level graceful degradation:

Level	Reply Interval	Scan Interval	Trigger
normal	5 seconds	2 seconds	Default
slow	15 seconds	5 seconds	Rate limit warning received
slower	30 seconds	10 seconds	3 consecutive errors
paused	Paused	Paused	Account throttled — resumes after 1 hour

A successful reply decrements the error counter, automatically recovering back to normal. Like a spring — compress it, and it bounces back.

Step 6: Background script (background.js). Assemble all the modules and implement the scan loop. Woken by chrome.alarms once per minute, scanning videos in priority order (currently open tabs first).

Step 7: Content Script and UI. Build the interface last. content.js exposes just a few simple methods to the floating widget; popup.js handles the management panel.

Phase 4: Debugging Tips

Debugging a Chrome extension is different from debugging a normal webpage. A few practical tips:

Service Worker debugging. On the `chrome://extensions` page, you'll see a "Service Worker" link under your extension. Clicking it opens a dedicated DevTools instance. console.log output from background.js

appears here — not in the webpage's console.

Content Script debugging. You can see content.js logs directly in the webpage's DevTools console. Keep in mind that content.js runs in an isolated execution environment and doesn't share global variables with the page's own JavaScript.

Reloading. After changing code, go to `chrome://extensions` and click the refresh button on your extension. Popup and options pages need to be closed and reopened. content.js requires a refresh of the target page. Service Worker changes require clicking "Update" next to the "Service Worker" link.

Claude Code can help you write debugging utilities. For example, have it add a logging system to content.js that captures all key operations — invaluable for tracking down problems. In v3, we built a simple log system that retains the last 200 entries, auto-scrolls to clear old ones, and displays them in the popup panel.

Phase 5: Installation and Testing

During development, you don't need to publish to the Chrome Web Store. On `chrome://extensions`, enable "Developer mode," click "Load unpacked," and select your project directory.

A testing checklist (have Claude generate one for you):

- Basic functionality: add a rule → scan comments → auto-reply
- Deduplication: the same comment is never replied to twice
- Rate limiting: rapid consecutive replies automatically slow down
- Persistence: close the browser and reopen it — state and data are preserved
- Multiple tabs: open several video pages simultaneously, each running independently

From 1,211 Lines to 175: The Refactoring Story

This refactor deserves its own section, because it's a textbook example of Claude Code-assisted refactoring.

v1 worked, but the pain points were real. One file, content.js, 1,211 lines — API calls, scanning logic, rule matching, and UI updates all tangled together. Every feature change meant hunting for code before you could even start. On top of that, MV3 Service Workers can go to sleep at any time, making setInterval-based scheduling unreliable.

The refactor wasn't something I planned from the start. One day I wanted to add a new feature — AI-assisted replies — and found it completely impossible to work with in a 1,211-line content.js. That's when I decided a refactor was unavoidable.

I told Claude:

```
The current content.js is 1,211 lines with all logic mixed together. I want to refactor it ir
- background.js as the operational hub
- content.js as a thin shell (only video info extraction and API proxying)
- lib/ directory for all business modules

Don't do this all at once — work in steps. First help me analyze the current code and map out
```

Claude produced a detailed migration plan, breaking the 1,211 lines into 8 functional chunks and annotating where each belonged. Then we executed it step by step — test after each migration, confirm nothing regressed, then move on.

The final result:

File	Lines	Responsibility
background.js	468	Message routing + scan orchestration
content.js	175	Video info extraction + message bridging
lib/db.js	410	Unified data layer
lib/scanner.js	322	Comment scanning engine
lib/api.js	137	Bilibili API wrapper
lib/ai.js	127	AI reply generation
lib/rate-limiter.js	100	Rate limiting with graceful degradation
lib/rules.js	62	Rule matching
lib/migrate.js	253	v2→v3 data migration

Total line count actually increased (from 1,211 to 2,000+), but each file has a clear, singular responsibility that can be understood and modified in isolation. content.js went from "does everything" to "does only what it must." The point of refactoring was never to reduce lines of code — it's to reduce cognitive overhead.

> 注意
>
> **Watch out:** The most common mistake during refactoring is trying to do it all at once. Don't ask Claude to rewrite everything in one shot — migrate in steps, confirming functionality works after each one. During the v3 refactor, I ran a full test cycle in the browser after every module migration. Slower is fine; stability beats speed.

Ideas for Your Chrome Extension Project

If you want to build a Chrome extension along these lines, it doesn't have to be a Bilibili plugin. A few directions to consider:

- **Web annotation tool**: highlight selected text and save it locally. Involves content script DOM manipulation + storage persistence
- **AI translation assistant**: select a paragraph and call an AI to translate it. Involves content script + external API calls
- **Social media timer**: track time spent on each website, alert when you go over. Involves background alarms + multi-site content scripts
- **GitHub enhancer**: display CI status on PR pages, auto-apply labels. Involves GitHub API + content script injection

Whichever you choose, the core workflow is the same: Plan mode to analyze requirements → define architecture (who owns what) → write CLAUDE.md to lock in the architectural principles → implement module by module → debug and test.

Chrome extensions are a great starting point because they prove one thing: Claude Code's capabilities extend far beyond building webpages. Browser plugins, automation tools, even OS-level scripts — if it can be solved with code, Claude Code can help you build it. In the next two chapters, we'll push that boundary even further.

§12 Hands-on Project: Content Creation Automation

Hands-on Project: Content Creation Automation

Claude Code's most underrated capability isn't writing code. In this chapter, I'll show you my actual daily workflow: using Claude Code for content creation. From topic selection, research, writing, and proofreading to publishing — fully automated. When you treat Claude Code as a "general-purpose Agent," its value multiplies several times over.

I run a media operation. Roughly 30% of my time goes to AI-assisted development, and 70% goes to content: writing WeChat articles, creating Xiaohongshu posts, writing video scripts, and producing research reports. For a long time, I used Claude Code for development and ChatGPT or the Claude web interface for writing. Then one day I noticed a problem: every preference, rule, and style guideline I'd built up in Claude Code vanished the moment I switched to the web version. Every new conversation meant re-explaining everything from scratch — "don't use em-dashes," "use Chinese quotation marks," "search sources from the past three months."

That day, I moved my writing rules into CLAUDE.md. Then I added a few Skills. Then I built a complete content production pipeline. Looking back, this may be the most valuable thing I've ever done with Claude Code. Not any specific coding project — this content automation system. It saves me time every single day.

Nothing in this chapter is theoretical. Every step is something I actually use daily. You can copy it wholesale, or adapt it to your own needs.

Step 1: The CLAUDE.md Routing System

If you work on a single project, one CLAUDE.md is enough. But if you have multiple work contexts — say you're doing writing, development, and video production simultaneously — you need a routing system.

A routing system is simple: one root-level CLAUDE.md acts as a "traffic controller." Based on what you say, it figures out what you're trying to do, then loads the corresponding subdirectory CLAUDE.md.

My project structure:

```
Writing/
├── CLAUDE.md              ← Router (~8KB)
├── 01-WeChat/
│   ├── CLAUDE.md          ← Full rules for WeChat articles
│   └── projects/
├── 02-Xiaohongshu/
│   ├── CLAUDE.md          ← Xiaohongshu writing rules
│   └── projects/
├── 03-Video/
│   ├── CLAUDE.md          ← Video script rules
│   └── projects/
├── 04-References/
│   └── SHARED-RULES.md    ← Cross-workspace shared rules
├── .claude/skills/         ← 27 Skills
└── 10-OrangeBook/          ← The book you're reading now
```

The most important element in the root CLAUDE.md is the routing table:

```
## Workspace Routing

After receiving a task, determine the workspace and load the corresponding CLAUDE.md:

| Keywords                          | Workspace       | Load File                          |
|-----------------------------------|-----------------|------------------------------------|
| article, sponsorship, WeChat      | WeChat Writing  | /01-WeChat/CLAUDE.md               |
| Xiaohongshu, notes, image-text    | Xiaohongshu     | /02-Xiaohongshu/CLAUDE.md          |
| video script, video production    | Video           | /03-Video/CLAUDE.md                |
| Orange Book, epub                 | Orange Book     | /10-OrangeBook/CLAUDE.md           |

Routing principle: ambiguous task → ask for clarification. Multiple workspaces involved → loa
```

When I say "write a WeChat article about Claude Code," Claude Code will:

1. Read the root CLAUDE.md, spot the keyword "WeChat" → match to WeChat Writing workspace
2. Automatically load `/01-WeChat/CLAUDE.md` and apply the writing rules
3. Start working according to the WeChat style guidelines

The benefits: the root CLAUDE.md stays lean (~8KB) and doesn't waste context window; each workspace's rules can be maintained independently without interference; adding a new workspace only requires one new routing line.

> 核心建议
>
> **You can do the same:** If you manage multiple projects — work projects, side projects, study notes — create a unified working directory, use the root CLAUDE.md as a router, and give each project its own rules. You don't need to copy my structure exactly. The core idea is: **one entry point, multiple destinations**.

Step 2: Shared Rules for Cross-Project Consistency

Some rules don't belong to any single workspace — they apply to all content. For example:

- Research should draw from sources within the past 3 months
- Three-pass proofreading (content review → style review → detail review)
- Image workflow (screenshot → AI-generated image → upload to image host)
- File naming conventions

These live in `SHARED-RULES.md`, referenced by every workspace's CLAUDE.md.

The three-pass proofreading process is worth unpacking, because it directly impacts output quality.

Pass 1: Content review. Check factual accuracy, logical flow, and structural completeness. Don't touch writing style in this pass.

Pass 2: Style review. This pass specifically targets "AI-ness." AI-generated text has a few telltale patterns:

AI Pattern Type	Example	How to Fix
Filler phrases	"In today's era," "To summarize," "It's worth noting that"	Delete entirely — meaning is preserved
AI sentence structures	"Not A, but B," "Not only A, but also B"	Replace with natural, conversational transitions
Overly formal language	"possess," "exhibit," "embody," "empower"	Swap for everyday vocabulary
Mechanical structure	Every paragraph starts with "First / Second / Finally"	Vary the order; use narrative flow to connect ideas
Neutral stance	Wishy-washy both-sides-ism that offends no one	Take a clear, explicit position
Missing specifics	"Many users report..." (How many? Who?)	Add concrete numbers and names

Pass 3: Detail review. Sentence length 15–25 words, paragraph spacing, bold highlights (roughly 10 per article), quotation mark style, em-dash usage (at most 1–2 in the entire piece).

Once these three-pass rules are in SHARED-RULES.md, Claude applies this standard no matter which workspace I'm writing in. And I can trigger a proofreading session at any time using the `/proofreading` Skill — no need to manually describe the process each time.

Step 3: Create Your First Skill

§ 07 covered the basics of Skills. Here's a real example of how to build one from scratch.

The first Skill I ever made was the three-pass proofreading workflow described above. Here's how to create it:

```
# Create the skill directory inside your project
mkdir -p .claude/skills/huashu-proofreading
# Write SKILL.md
```

The SKILL.md format:

```
---
name: proofreading
description: |
  Three-pass proofreading workflow to reduce AI detection rate below 30%.
  Auto-trigger: user says "proofread," "less AI-sounding," "too robotic," "revise this"
  Manual trigger: /proofreading
---

# Three-Pass Proofreading

## Pass 1: Content Review
- Fact-check: verify all data, dates, version numbers
- Logic check: confirm cause-and-effect relationships hold
- Structure check: identify any missing key points

## Pass 2: Style Review
Check for the following 6 AI patterns (see table) and fix each:
1. Filler phrases → delete
2. AI sentence structures → rewrite as conversational
3. Overly formal language → replace with everyday words
4. Mechanical structure → vary; use narrative flow
5. Neutral stance → add a clear personal position
6. Missing specifics → add concrete data

## Pass 3: Detail Review
- Sentence length: aim for 15–25 words
- Paragraph length: 3–5 sentences
- Bold highlights: roughly 10 per article
- Quotation marks: use „" style, not ""
- Em-dashes (—): at most 1–2 in the entire piece
```

Once written, type `/proofreading` in Claude Code and it will run this workflow on the current document. You can also just say "proofread this for me" — Claude recognizes the keyword "proofread" and auto-triggers the Skill.

There's an important mental shift here: **Skills are written for the AI, not for you.** You don't need to memorize all the checks in the three-pass process. You just need to remember one command: `/proofreading`. Every detail is encapsulated in the Skill — Claude handles execution.

Once you develop the habit of packaging repetitive work into Skills, your library grows fast. I currently have 27 Skills covering every stage of content production:

Category	Skill Name	What It Does
Writing	/proofreading	Three-pass proofreading
	/article-edit	Article editing with progress tracking
	/topic-gen	Generates 3–4 topic directions
	/article-to-x	Adapts WeChat articles into X (Twitter) threads
Video	/video-outline	Video script outline with title strategy
	/script-polish	Script refinement
	/danmaku-gen	Generates live comment/bullet-chat copy
Research	/research	Structured research with auto-archiving
	/info-search	Information search with source verification
	/material-search	Searches personal asset library
Publishing	/book-pdf	Full Orange Book build pipeline
	/md-to-pdf	Markdown to PDF conversion
Images	/image-upload	AI image generation + upload to image host
	/design	Infographic design
Integration	/feishu	Create and send Feishu documents

Every Skill has been refined through real-world use. I didn't write 27 Skills all at once — each one emerged from daily work. Whenever I ran into a repetitive operation, I packaged it into a Skill. It took about two months to accumulate this many.

Step 4: Full Workflow Demo — From Topic to Publication

I've walked through the individual components. Now let me string them together with a complete example. Say it's today and I want to write a WeChat article.

10:00 — Topic selection. I open Claude Code in the terminal and navigate to my writing project directory:

```
Claude Code recently shipped some new features. Help me brainstorm a few WeChat article topic
```

Claude spots the keyword "WeChat," routes automatically to the WeChat workspace, and loads the writing rules. Then it triggers the `/topic-gen` Skill, producing 3–4 topic directions, each with a suggested title, outline, and effort estimate (⭐ to ⭐⭐⭐).

I pick the second direction — effort rating ⭐⭐.

10:10 — Research.

```
Go with option two. Start with research — gather Claude Code's major updates and user feedbac
```

This triggers the `/research` Skill. Claude will:

1. Immediately create a research file: `_knowledge_base/research-claude-code-updates-20260403.md`
2. Append findings to the file after each search round (prevents data loss if the conversation is interrupted)
3. After 3 search rounds, output an interim summary
4. Deliver a structured final report: key facts, sources, gaps, and writing recommendations

This "search-and-save-as-you-go" design came out of necessity. Once I was mid-research when the session got compressed because the context was too long — everything I'd found up to that point was gone. After that, every research Skill enforces real-time file writes.

10:30 — Draft.

```
Research looks good. Based on the findings, write a WeChat article.
Requirements:
- Save to projects/2026.04-Claude-Code-Updates/draft.md
- Around 3,000 words
- Clear personal opinion — don't write it like a press release
```

Claude creates the project directory and starts writing. The draft gets saved to an md file. Note: the root CLAUDE.md has a standing preference: "articles must be written to an md file, never directly in the reply." This is because chat replies disappear when the session ends — md files persist.

11:00 — Proofreading.

```
/proofreading draft.md
```

The three-pass review runs automatically. Claude reads the file, goes through each check in sequence, and outputs revision suggestions. Once I confirm, it edits the file directly. A typical pass catches 10–20 AI-pattern issues.

11:20 — Images.

```
Create a cover image for this article.
```

This triggers the image workflow. Claude generates an image prompt from the article content, calls an AI image generator, uploads the result to an image host, and inserts the URL at the right place in the article. The whole process is automated — I just check whether I like the image.

11:30 — Publishing.

```
Proofreading is done. Send the article to a Feishu document.
```

This triggers the `/feishu` Skill. Claude converts the md file to Feishu document format, creates the document via API, sets permissions, and delivers it to me. I do one final manual review in Feishu, then copy it into the WeChat editor and publish.

From topic selection to a publication-ready document: roughly ninety minutes. Without this system, the same work would take half a day.

Step 5: Advanced — Multi-Agent Parallelism

The workflow above is sequential: finish one step before moving to the next. But in practice, many steps can run in parallel. When I was writing this book, I had four Agents running simultaneously:

```
# Terminal 1: Research Agent — finding the latest Claude Code updates
claude -p "Research Claude Code's major updates from the past three months..." --dangerously-

# Terminal 2: Writing Agent — drafting §07 on extensibility
claude -p "Based on the following outline, write §07..." --dangerously-skip-permissions

# Terminal 3: Proofreading Agent — reviewing the completed §05 chapter
claude -p "Proofread fragments/part5-claude-md.html..." --dangerously-skip-permissions

# Terminal 4: Image Agent — creating visuals for the reviewed chapter
claude -p "Create an illustration for §03..." --dangerously-skip-permissions
```

Each Agent runs independently without interference. With four Agents running at once, the overall build efficiency of the book improved several times over. API costs came to about $50 — far cheaper than hiring an assistant.

> 注意
>
> **Parallelism requires task independence.** If Agent B depends on Agent A's output, they can't run in parallel. "Research first, then write" must be sequential. But "write §03" and "write §05" can run in parallel — they operate on different files and won't conflict.

Step 6: Skills + Hooks + MCP Working Together

By this point, we've used Skills (to encapsulate recurring workflows) and CLAUDE.md (to persist rules). Add Hooks and MCP, and you have a complete Harness.

What do Hooks do? Hooks are event-driven. For example:

- Every time a new file is created, automatically check whether the filename follows naming conventions
- Every time an md file is edited, automatically run formatting (normalize quotation marks, remove extra blank lines)
- Before every commit, check for unresolved TODO markers

The value of Hooks is that they prevent things from slipping through. People forget to check naming conventions — Hooks don't.

What does MCP do? MCP lets Claude Code connect to external services. In my content creation workflow, MCP is primarily used for:

- Connecting to Feishu: creating and editing documents directly
- Connecting to the browser: operating pages and reading information in Chrome
- Connecting to the file system: accessing reference materials outside the writing directory

Here's how the four components relate:

```
CLAUDE.md  → Defines rules and preferences (passive — auto-loaded every conversation)
Skills     → Encapsulates workflows (triggered manually or by keywords)
Hooks      → Automated checks (event-driven — no human involvement needed)
MCP        → Connects to the outside world (extends Claude's capability boundaries)
```

If Claude Code were an employee: CLAUDE.md is the job handbook, Skills are the techniques they've learned, Hooks are the good habits they've internalized, and MCP is the set of tools they can use. Combine all four, and you have a complete "AI employee" system.

Building Your Own Content Automation System

You don't need to build a 27-Skill system overnight. I recommend three phases:

Week 1: Establish your CLAUDE.md. Start with a root CLAUDE.md and write down 3–5 rules you care most about — your writing style preferences, file naming conventions, the project structure you use. It doesn't need to be extensive. Anything is better than nothing.

Week 2: Create your first Skill. Spend the week noticing which instructions you repeat to Claude most often. Package that into a Skill. Maybe it's "proofread this," maybe it's "translate this," maybe it's "summarize my meeting notes." Whatever it is, just build one and start using it.

Week 3 onward: Expand as needed. Each time you hit a repetitive operation, ask yourself: "Is this worth turning into a Skill?" If you do it more than three times a week, the answer is yes. Three months in, you'll have 10–15 Skills covering most of the repetitive work in your day.

The key isn't quantity — it's habit. When you habitually package repetitive tasks into Skills, lock rules into CLAUDE.md, and use MCP to connect external services, your overall work efficiency reaches a new level. This is what § 10's Harness Engineering looks like in practice.

A note for non-programmer readers

Nothing in this chapter requires you to write code. Creating a CLAUDE.md is writing a rules document. Creating a Skill is writing an operations manual. Configuring MCP is filling in a few parameters. Writers, marketers, product managers — anyone whose work involves repetitive output can use this approach directly. Claude Code isn't just a programming tool; it's a general-purpose AI workstation. This may be the single most important point in this book.

§13 Hands-on Project: From Zero to #1 Paid App on the App Store

Hands-on Project: From Zero to #1 Paid App

This chapter isn't about technology — it's about a complete product story. From a single offhand comment from my girlfriend to reaching #1 on the App Store paid chart, only a weekend stood between them. But the one hour of development isn't the point. The five minutes of judgment that came before, and the months of iteration that followed — that's the part of this story actually worth telling.

One afternoon in November 2024, I was recording a Cursor tutorial video. My girlfriend was sitting beside me checking the frame, when she suddenly said: "Instead of making a screen flashlight, what about a fill-light color card?"

I was a little confused. What was a fill-light color card supposed to be? I'd never heard the term before. But I didn't ignore the comment — I opened Xiaohongshu (China's Instagram) and searched "fill-light color card."

The results stopped me cold for a few seconds. Several posts had hundreds of thousands of likes. People were using solid-color images as makeshift fill panels for selfies, sharing comparisons of different color lighting effects, and asking for "a fill-light app where you can adjust color and brightness" — but no one could find a good one. I realized immediately: this was a real, high-frequency need. The demand had already been validated; it just hadn't been turned into a product yet.

Five minutes later I opened Cursor (I hadn't started using Claude Code at that point) and described the feature I wanted in plain language. An hour after that, the first version of Kitty Light was live on the App Store.

What Matters More Than Writing Code: Five Minutes of Judgment

Looking back on this later, I realized the most critical thing wasn't the development process — it was getting a few key judgments right in those five minutes:

First, I didn't dismiss advice from a non-programmer. My girlfriend doesn't know code, but she understands user needs. The phrase "fill-light color card" came from something she'd seen on Xiaohongshu — it represented real users speaking their real language. If I had only bounced ideas off programmer friends, I almost certainly wouldn't have thought of this direction.

Second, I spent three minutes validating the demand. Not a gut feeling that "someone probably wants this," but actually opening Xiaohongshu and searching. Posts with hundreds of thousands of likes are the best market research report you can get. Cost: three minutes. Return: confirmation that this was a genuine, high-frequency need.

Third, I committed to the minimum viable product. The first version did exactly one thing: display a solid color fullscreen, with adjustable color and brightness. No filters, no timer, no social sharing, no membership system. Just a full-screen light you could change the color of.

This decision-making process is identical to the logic you should use when deciding what to build with Claude Code. Technical ability isn't the bottleneck — Claude Code can help you write code of any complexity. The real bottleneck is: does anyone actually need what you're building?

> **A Suggestion for Readers**
> Before reading this chapter, think of an app you'd like to build. Don't think about "whether it's technically feasible" — think first about "who would use it, in what situations, and whether anything on the market already solves this." If you can find people on Xiaohongshu, Bilibili, or X discussing a real need but no great product to meet it, that's worth building.

Using AI to Build an iOS App

Kitty Light was originally built with Cursor — that was late 2024, when Claude Code wasn't as mature as it is today. If I were starting this project fresh, I'd use Claude Code throughout. Here's how that would look:

Create the SwiftUI project. Start by creating a blank iOS project in Xcode (this step must be done manually, since Xcode project initialization requires a GUI to select configuration options). Once it's created, launch Claude Code from inside the project directory:

```
This is a SwiftUI project. I want to build a fill-light app. Core features:
1. Full-screen solid color display, user can choose the color
2. Adjustable brightness (from 0 to maximum)
3. A few preset color cards (warm white, cool white, warm yellow, pink, etc.)
4. Clean interface — open and use immediately

Take a look at the project structure, then help me implement the core functionality.
```

Claude Code will read the project structure and start writing SwiftUI code. The core technology behind a fill-light app is actually quite simple — just a full-screen `Color` view with a color picker and a brightness slider. Claude handles tasks like this well — clear goals, clear technical path — and the first version usually runs right away.

Iteration is the real work. The first version works, but it's far from polished. User feedback will tell you what comes next:

- "Can you add a camera preview? I don't want to keep switching apps" → core feature of the Pro version
- "Color selection is too fiddly — can you make it preset cards instead?" → the color card system
- "I want to pick a custom color" → HSB color picker
- "The brightness doesn't go high enough" → system brightness API + screen brightness overlay

Every iteration follows the same loop: user feedback → describe the need in Claude Code → implement → test → ship.

```
Users are saying they want to see the camera preview while using the fill light.
Please add a feature: the top half of the screen shows a live camera feed,
the bottom half shows the fill-light color, with a draggable divider between them.
```

Claude will modify the existing code to integrate `AVCaptureSession` for camera support. This is technically complex — it involves camera permissions, live preview, and layout management — but Claude Code handles it well, because these are all standard patterns in SwiftUI and AVFoundation.

App Store Submission: Where Things Go Wrong

Writing code probably accounts for only 30% of the total process. The remaining 70% is everything related to getting onto the App Store: developer account, certificate configuration, privacy policy, app review.

Developer account. Apple Developer Program, ¥688/year. Must be registered with your real identity. Review takes about 1–2 days. If you don't already have an Apple developer account, this is the only step that requires waiting.

Certificates and signing. This is the most counterintuitive part of iOS development. You need to create certificates, configure provisioning profiles, and set up signing in Xcode. Claude Code can help you understand what each step means, but the actual work happens in Xcode and on the Apple Developer website.

```
I already have an Apple Developer account. Help me walk through the complete steps
from development to App Store submission — including certificate setup,
provisioning profile creation, and Xcode build configuration.
List out each step with specific instructions and common pitfalls to watch for.
```

Claude will give you a detailed checklist. It can't operate Xcode's GUI for you, but it can tell you exactly what each option should be set to and why.

App review. Apple's review process is significantly stricter than the Chrome Web Store. Common reasons for rejection:

- Missing privacy policy page (required even if your app collects no data at all)
- Screenshots that don't match the actual functionality
- Use of private APIs
- Functionality too simple (Apple may determine that "anything achievable via a website shouldn't be an app")

Kitty Light passed on the first submission — the functionality was clear, the interface was clean, and there were no gray areas. Review took about 24 hours.

Going Viral: Luck or Inevitability

In the first week after launch, I posted a note on Xiaohongshu — just a simple before-and-after comparison showing the fill-light effect.

Then it exploded.

Within three days, the post had 1.18 million views and 73,000 likes. App downloads crossed 30,000. The Pro version (with the camera feature, priced at ¥6) shot to #1 on the App Store paid chart and stayed there for over a month.

Journalists later asked me "how did you do it?" Honestly, I was a little surprised myself. But looking back, a few things stand out:

The need was strong enough. Selfie fill lighting is a high-frequency, must-have use case. Not "nice to have" — "can't do without." The enormous volume of fill-light color card posts on Xiaohongshu proved the need was real.

The product was simple enough. Open the app and you get a full-screen color card. No registration, no tutorial, no learning curve. Products that work the moment you open them spread the most easily, because users can show off the entire feature set in a 3-second clip.

Controversy drove traffic. Plenty of programmers questioned in the comments whether "an app like this can really make money" and whether "something built in an hour deserves to be paid for." The controversy worked in our favor — it got more eyes on the post. Users debating each other became a free traffic engine.

Fast iteration built a reputation. During the two days the app went viral, I shipped three updates overnight — fixing issues from user feedback and adding the most-requested features. At the same time I resisted the urge to edit the original Xiaohongshu post or publish new ones, because the platform algorithm was actively recommending it and any interference might have disrupted distribution.

> 注意
>
> **A Realistic Expectation Check**
>
> The vast majority of indie apps quietly disappear. What you see are the few that made the charts — behind them are thousands of apps no one has ever heard of. Kitty Light's success involved a significant element of luck: picking the right niche, catching Xiaohongshu's recommendation algorithm at the right moment, and having a product format that was perfectly suited to short-video sharing. These factors can't be reliably replicated. The right mindset for building products is: execute every step as well as you can, but don't assume a particular outcome.

What Came Next: From Viral Hit to Product Line

Kitty Light's trajectory exceeded my expectations.

The Pro version continued receiving updates with more professional features. Building on the same technical foundation, I launched a mini-program version (for users without iOS devices) and a HarmonyOS version (for Huawei users). Then came Kitty Album — a new product aimed at the same user base (selfie enthusiasts).

An app built in one hour eventually grew into a small product line. In traditional development this would have been nearly impossible — writing the code alone would have taken months. AI-assisted development compresses the cost of building to near zero, enabling rapid experimentation and rapid iteration.

People's Daily called this model the "hand-crafted economy." When CCTV came to do an interview, the reporter asked me to demonstrate writing code with AI on camera. I opened Claude Code and built a working mini-product in 10 minutes. Apparently, this may have been the first time Claude Code appeared on Chinese national media.

Replicating This Process with Claude Code

Kitty Light was built with Cursor, but the core methodology applies directly to Claude Code. If you want to build an iOS app, here's the complete process:

Step 1: Validate the need (30 minutes). Search your idea on Xiaohongshu, Bilibili, and Douyin. See if people are already talking about the problem. Don't worry about how many competitors exist — competition means there's a market. What you're looking for is: are the existing competitors all mediocre? What are users complaining about most?

```
I want to build [your app idea]. Help me analyze:
1. How real is this need (how many people actually want this)
2. What competitors exist and how they're rated
3. What differentiation angles I could pursue
4. What the minimum viable product should include
```

Step 2: Define the MVP (30 minutes). Use Plan mode to discuss the minimum viable product with Claude. Cut everything non-essential. Ask yourself one question: if this app could only do one thing, what would that thing be?

Step 3: Build (1–4 hours). Create the project in Xcode, then develop inside the project directory with Claude Code. For a simple utility app, a few hours is usually enough to reach something that actually runs.

Step 4: Test and polish (1–2 days). Use it yourself, have friends try it, and note every friction point. Have Claude Code fix them one by one. This step matters more than development itself.

Step 5: Submit (1–2 days). Prepare App Store assets (screenshots, description, keywords, privacy policy), build and upload from Xcode, wait for review.

Step 6: Promote and iterate (ongoing). After launch, share on social platforms, collect user feedback, and iterate quickly.

Of these six steps, Claude Code is directly involved in Step 1 (analysis), Step 2 (planning), Step 3 (development), and Step 4 (fixes). Steps 5 and 6 require you to operate Xcode and social platforms yourself — but Claude Code can help you write the app description, prepare keywords, and analyze user feedback.

CLAUDE.md Recommendations for iOS Development

If you're serious about iOS development, consider putting the following in your project's CLAUDE.md:

```
# iOS Project Rules

## Tech Stack
- SwiftUI (avoid UIKit unless SwiftUI can't handle it)
- Minimum deployment target: iOS 16
- Swift Package Manager for dependencies

## Architecture
- MVVM pattern
- Each View has a corresponding ViewModel
- Network requests go in a Service layer
- Persistence via SwiftData or UserDefaults

## Code Style
- File naming: UpperCamelCase (ContentView.swift)
- Variable naming: lowerCamelCase (isLoading)
- Max 200 lines per file — split if exceeded
- Comments in English

## Common Pitfalls
- Don't perform async operations inside a View's body
- Use @StateObject instead of @ObservedObject to own a ViewModel
- Device testing requires signing configured in Xcode first
```

These rules help Claude Code generate code that aligns with iOS best practices, reducing the amount of cleanup needed later.

What This Chapter Is Really About

Technology is the least important part of this story. Claude can write the SwiftUI code, the submission process is documented online, and every platform has tutorials on promotion.

What actually determines success or failure comes down to three non-technical factors:

Whether you can spot the need. The idea for Kitty Light came from a single offhand comment from my girlfriend. If I'd had headphones on that afternoon and missed it, none of what followed would have happened. Staying observant about everyday life matters more than any programming skill.

Whether you can exercise restraint. The first version had one feature. No social layer, no memberships, no ads. Radical simplicity is a radical competitive advantage. I've seen too many indie developers turn their apps into feature-bloated monstrosities — everything's there, but nothing's done well.

Whether you can keep iterating. Going viral is an accident; continuing to deliver value is a choice. After Kitty Light took off, I shipped several more versions, each addressing real problems users had reported. It's not glamorous work — but that's the actual reason a product succeeds.

AI coding tools have made "writing code" a non-bottleneck. That means the dimension of competition has shifted from "who can get code written" to "who can find problems worth solving." Claude Code gives you the ability to build anything you can imagine — but deciding what to build, and what not to build, will always be your call.

§14 Slash Commands: A Deep Dive

Slash Commands: The Hidden Powertools

Claude Code's slash commands (prefixed with /) go far beyond /help and /clear. This chapter digs deep into the underrated commands that could each transform how you work. From context management to parallel exploration, from cost control to session branching — these are Claude Code's hidden levers.

Most people use Claude Code with just two things: typing requests directly, and occasionally running `/help` to see what's available. It's like buying a car and only ever driving straight — never discovering reverse, cruise control, or lane assist.

In this chapter, I'll organize the most valuable slash commands by use case, explaining each one clearly: what it does, when to use it, and how to get maximum value out of it.

Context Management — The Most Underrated Capability

Claude Code's context window is finite (currently supporting 1M+ tokens). That sounds enormous, but a session running for several hours — combined with project code, conversation history, and tool output — can push right up against that limit. How well you manage context directly determines whether you can complete complex tasks.

/compact — Smart Compression, Not Simple Deletion

`/compact` is one of the commands I use most. It compresses conversation history into a distilled summary, freeing up context space.

The key word is "smart." It doesn't just delete what came before — it uses AI to summarize: preserving key decisions, code change records, and your stated preferences, while discarding only redundant intermediate steps.

When to use it:

- **After 20+ rounds of conversation.** Every exchange gets resent with the full history — the longer it gets, the more tokens you burn.
- **When shifting task direction.** If the first half was spent on Feature A and you're now moving to Feature B, compact it to remove the A-specific noise.
- **When Claude feels slower.** Longer context means slower responses. You'll notice a speed improvement after compacting.

Advanced usage: you can pass instructions to `/compact`, telling it what to preserve:

```
/compact Keep all architecture decisions and bug fix records; compress away the detailed code
```

This way, the important information survives compression while the irrelevant intermediate steps get cleaned out.

> **核心建议**
>
> **From experience:** Don't wait until the context is full to compact. My habit is to compact after each milestone. For example: finish requirements analysis → compact → start coding → finish coding → compact → start testing. This keeps each phase well-supplied with context space.

/context — See Where Your Tokens Are Going

`/context` shows a detailed breakdown of your current context: how much is taken up by system prompts, CLAUDE.md, Skills, and conversation history.

The value of this command is visibility. When you suspect the context is nearly full, run `/context` first to see exactly what's consuming space. Maybe it's a particularly large CLAUDE.md, too many MCP tools loaded, or a conversation history that's grown too long. You need the data to make the right call.

/clear — A Clean Slate

`/clear` wipes the entire conversation history and returns to the initial state. It's equivalent to closing the terminal and reopening `claude`, just faster.

My recommendation: **before starting an unrelated new task, always /clear first.** Many people chain Feature A and Feature B in the same session, leaving all of A's context sitting there taking up space. /clear is free, and the tokens it saves are real.

Safety Nets — Mistakes Aren't Fatal

/rewind — Surgical-Precision Rollback

Press `Esc` twice or type `/rewind` to enter rollback mode. You'll see three options:

- **Rewind conversation:** Undo the last few exchanges and restart from a specific point
- **Rewind code:** Keep the conversation, but restore files to their previous state
- **Rewind everything:** Roll back both conversation and code together

"Rewind code but keep conversation" is the most powerful option. Say Claude modified 10 files and you realize the direction was wrong. Rewind the code — files are restored — but your discussion remains intact. You can say, "That approach didn't work. Let's try a different angle; this time only touch the 3 core

files." Claude still knows what you discussed and why the first attempt failed, so the second attempt is usually much better.

/fork — Parallel Universe Exploration

`/fork` branches the current conversation into a new, independent thread. Both branches share the prior history but develop independently from that point forward.

A typical use case:

```
You: Help me design a user authentication system
Claude: Option A is JWT stateless authentication...

You: /fork
(New branch begins)
You: Let's set aside the JWT approach. What about Session + Redis?
Claude: Option B is session-based authentication...

(Both branches run in parallel; compare and pick the better fit)
```

There's no cost to "going down the wrong path." /fork lets you explore multiple directions simultaneously and choose the best one at the end.

Efficiency Tools — Do More, Spend Less

/cost — How Much Have You Spent

`/cost` shows the token consumption and approximate cost for the current session.

This command looks simple, but it changed how I use Claude Code. Before I started checking /cost, I had no sense of what each session was costing me. After looking, I realized: a long session might run $2–5, a large-scale refactor might hit $10+. The point isn't to be frugal — it's to make a conscious decision about whether a task is worth that many tokens.

Suggested moments to check /cost:

- Before starting a large task (set expectations)
- When a conversation feels too long (decide whether to /compact or /clear)
- At the end of the day (understand your total usage)

/model — Switch Models on the Fly

`/model` lets you switch the active model mid-session.

A cost-effective pattern: use Sonnet for everyday tasks (fast and cheap), then switch to Opus for problems requiring deep reasoning (slower, but more capable). Sonnet costs roughly 1/5 of Opus, and for most coding tasks — CRUD, simple bug fixes, boilerplate generation — it's more than sufficient.

```
/model sonnet   ← everyday tasks
/model opus     ← complex architecture decisions, large-scale refactoring
```

/fast — Toggle Fast Mode

`/fast` switches to fast mode. Fast mode uses the same model but produces output more quickly. If you don't need Claude to think deeply and just want it to complete a well-defined task quickly, enabling fast mode gives a noticeable speed boost.

/btw — A Side Question Without Breaking the Flow

`/btw` is a severely underrated command. It lets you ask an unrelated question while Claude is in the middle of working — Claude answers it, then picks up right where it left off.

```
(Claude is refactoring your code...)
/btw What's the difference between readonly and const in TypeScript?
(Claude answers quickly, then continues refactoring)
```

Key detail: /btw responses don't enter the conversation history. They're "one-time" — they don't pollute the context or consume extra tokens. Perfect for those sudden small questions that pop up while you're in the middle of something.

Project Management — Making Claude Remember Everything

/init — Create Your First CLAUDE.md

`/init` scans the current project structure and auto-generates an initial CLAUDE.md file. It reads your package.json, README, and code structure to infer the project's tech stack and conventions.

The generated CLAUDE.md is a starting point, not a finished product. You'll need to layer in your own rules. But it's far easier than starting from scratch — at minimum, the tech stack and project structure are things the AI can figure out on its own.

/memory — Audit Claude's Memory

`/memory` lists all memory files currently loaded by Claude: project-level CLAUDE.md, user-level CLAUDE.md (`~/.claude/CLAUDE.md`), and the auto-generated CLAUDE.local.md.

CLAUDE.local.md is Claude's own notebook. When your project has auto-memory enabled, Claude automatically records useful information it discovers during work — build commands, debugging patterns, architecture decisions — into this file. The next time you open a new session, that knowledge loads automatically.

Check `/memory` periodically to see what Claude has recorded. Sometimes what it wrote is wrong or outdated, and you'll need to correct it manually.

/permissions — Security Management

`/permissions` manages Claude Code's permissions. You can pre-authorize certain operations (such as allowing `npm test` to run automatically) or restrict others (such as prohibiting file deletion).

For beginners, I recommend starting with default permissions (every action prompts for approval) and gradually opening things up as you get comfortable. For high-frequency operations (running tests, git status), pre-authorization eliminates a lot of confirmation steps.

Code Quality — Let AI Review Your Work

/review — Automated Code Review

After finishing a set of code changes, type `/review` and Claude will audit all uncommitted modifications and provide improvement suggestions. It's like having a 24/7 code reviewer on call.

It checks for:

- Potential bugs (null checks, boundary conditions)
- Performance issues (unnecessary loops, large data set operations)
- Security problems (SQL injection, XSS, hardcoded secrets)
- Code style (naming consistency, file organization)

/simplify — Three-Angle Code Refinement

`/simplify` is more interesting. It launches three parallel Agents that each review your code changes from a different angle — reusability, quality, and efficiency — then consolidates the findings and auto-applies the fixes.

This isn't a theoretical review; it directly modifies your code. When it's done, you review the diff and confirm.

Advanced Operations — Unlocking More Possibilities

/doctor — Health Check

`/doctor` runs a series of diagnostic checks: whether the CLI is up to date, whether authentication is valid, whether required tools are installed, and whether environment variables are configured correctly.

When something inexplicable goes wrong, run `/doctor` first. The problem is often in the environment configuration, not in Claude itself.

/vim — Vim Mode

`/vim` enables Vim key bindings. For Vim users, this is a gift — you can use Vim editing commands (d, c, y, w, and other motion commands) directly in the input box, without switching to an editor to modify your

prompt.

/terminal-setup — Terminal Integration

`/terminal-setup` automatically configures your terminal so that Shift+Enter inserts a newline in the input box (instead of sending the message). Supports VS Code terminal, iTerm2, Alacritty, Warp, and other popular terminals.

/export — Export Session Transcript

`/export` exports the current session as plain text. Useful for writing project documentation, retrospective reports, or sharing a conversation with a colleague to show what was discussed.

Command Combinations: Workflow Patterns

Individual commands are useful, but combining them is where the real power lies. Here are a few workflow patterns I've validated in practice:

Pattern 1: Long Session Management

```
1. /clear          ← clean starting point
2. Complete Phase 1
3. /compact        ← compress Phase 1 details
4. Complete Phase 2
5. /compact        ← compress again
6. Complete Phase 3
7. /cost           ← check total consumption
```

This pattern extends sessions that would otherwise last 30–60 minutes into several hours, while keeping Claude's understanding of the project intact.

Pattern 2: Exploratory Development

```
1. Discuss requirements, align on direction
2. /fork           ← Branch A: Approach One
3. (Implement Approach One in Branch A)
4. Return to original branch
5. /fork           ← Branch B: Approach Two
6. (Implement Approach Two in Branch B)
7. Compare both results, choose the better one
```

Pattern 3: Safe Refactoring

```
1. /review         ← audit current code state first
2. Have Claude begin refactoring
3. If direction is wrong → press Esc twice → /rewind to roll back code
4. Refactoring complete → /simplify  ← three-angle refinement
5. /review         ← final audit
6. git commit
```

Pattern 4: Economy Mode

```
1. /model sonnet   ← use the cheaper model for everyday tasks
2. Hit a complex problem → /model opus  ← switch to the stronger model
3. Problem solved → /model sonnet  ← switch back
4. /cost           ← see how much you saved
```

The essence of slash commands

Slash commands aren't a "feature list" — they're workflow infrastructure. Each command looks simple in isolation, but when you combine them into workflow patterns, your efficiency improves qualitatively. It's like Git commands: add, commit, and push are each simple on their own, but together they form an entire version control system. Claude Code's slash commands work the same way: /compact + /context + /clear = a context management system; /fork + /rewind = a safe exploration system; /cost + /model = a cost control system.

练习 Try It Yourself

Exercises — Try It Yourself

Each chapter comes with 2–3 hands-on exercises. You don't have to do them all — just pick what interests you. Finishing even one puts you ahead of everyone who only reads.

§ 02 After Installation

Exercise 1: Say hello to Claude. After installing, open your terminal, type `claude`, and ask it in your own language: "Who are you? What can you help me with?" See how it responds. Then type `/status` to check your account status and current model.

Exercise 2: Try piped input. Find any text file on your computer (e.g., README.md) and run `cat README.md | claude -p "Summarize this file in one sentence"`. Get a feel for non-interactive mode.

§ 03 After Your First Project

Exercise 3: Build something you actually want. Don't follow a tutorial. Think of something you do repeatedly in daily life — organizing photos, batch-renaming files, generating a daily task list — then tell Claude Code: "Build me a command-line tool that does X." When it's done, screenshot it and show a friend.

Exercise 4: Give it a deliberately vague request. Type "make me something interesting" and see how Claude responds. Then give it a specific, concrete request and compare. The goal is to feel how much clarity of requirements affects output quality.

§ 04 After the Core Workflow

Exercise 5: Walk through Plan mode. Press `Shift+Tab` to switch to Plan mode, then describe a feature you want to build. Review Claude's proposed plan, give feedback, and go back and forth 2–3 rounds. Notice the difference between "discuss first, then build" versus "just go do it."

Exercise 6: Try Auto mode. Switch to Auto mode and give Claude a clear task (e.g., "Create an index.html in the current directory with a personal introduction page"). Watch it complete the task automatically. Pay attention to which actions it executes directly and which ones still prompt for confirmation.

§ 05 After CLAUDE.md

Exercise 7: Create your first CLAUDE.md. In any project directory, run `/init`, then open the generated CLAUDE.md and add 3 rules of your own. For example: write code comments in English, use English for commit messages, name files in kebab-case. Then start a new conversation and verify that Claude follows your rules.

Exercise 8: Test forgetting. Tell Claude a rule mid-conversation (e.g., "start every reply with an emoji"), then chat for 10 rounds and see if it still remembers. Then put that rule in CLAUDE.md, start a new conversation, and compare. Feel the difference between in-context memory and CLAUDE.md memory.

§ 06 After Advanced Conversation Techniques

Exercise 9: Same request, two phrasings. First describe a request vaguely ("help me fix this page") and record Claude's behavior. Then describe the exact same request precisely ("change the heading on line 15 of index.html from h2 to h1, and set the font size to 24px"). Compare the output quality and your satisfaction with each approach.

Exercise 10: Try @file references. In a conversation, type `@` and select a file, then ask Claude: "What does this file do, and what could be improved?" Experience the difference between proactively giving Claude context versus letting it find things on its own.

§ 07 After Extending Claude's Capabilities

Exercise 11: Write your first Skill. Think of something you repeat to Claude every day (e.g., "format this markdown file — headings with ##, lists with -"). Write it as a rule in `.claude/skills/format/SKILL.md`, then invoke it with `/format`.

Exercise 12: Add an MCP. Run `claude mcp add` to add any MCP server (the filesystem MCP is a good starting point), then have Claude read a file at a specific path through MCP. Feel the shift from "Claude can only see the current directory" to "Claude can see anywhere."

§ 08 After Multi-Agent Collaboration

Exercise 13: Run two Claudes simultaneously. Open two terminal windows, each with a running Claude Code instance. Give them different tasks (one writes HTML, the other writes CSS), then merge the results. Experience what parallel work actually feels like.

§ 09 After Building a Product

Exercise 14: A weekend project. Pick something you genuinely want to build — a Chrome extension, a personal website, a small utility — and spend a full weekend on it. Start in Plan mode, break it into tasks, and build incrementally. When you're done, share it with at least one friend.

Exercise 15: Review your CLAUDE.md. After all these exercises, your CLAUDE.md should have accumulated some rules. Open it up: are there any duplicates? Anything you no longer need? Clean it up and keep it under 200 lines. This is your first CLAUDE.md maintenance pass.

Self-Check: Are You Ready to Build on Your Own?

After completing the exercises above, check yourself against this list:

Capability	How to verify	Chapter
Installation & setup	Can install Claude Code from scratch and complete a first conversation	§ 02
Basic interaction	Can have Claude complete a small end-to-end project	§ 03
Working modes	Know when to use Plan mode vs. Auto mode	§ 04
Memory system	Have a non-empty CLAUDE.md file	§ 05
Effective communication	Can write specific, verifiable requirement descriptions	§ 06
Extended capabilities	Have written at least one Skill or configured at least one MCP	§ 07
Parallel thinking	Have tried running two or more Claude Code instances simultaneously	§ 08
Boundary awareness	Know what Claude can't handle and when you need to step in	§ 09b

Hit 6 out of 8, and you're ready to start building independently. Pick up the rest as you go.

Appendix A What 510,000 Lines of Code Told Us

What 510,000 Lines of Code Told Us

In late March 2026, the complete source code of Claude Code leaked unexpectedly. As someone who uses this tool every day, I spent a good amount of time going through it. Here's what I think is most worth knowing.

A Textbook Packaging Mistake

On March 31, 2026, security researcher Chaofan Shou spotted an anomaly on npm: the Claude Code v2.1.88 package weighed in at 59.8MB — several times larger than normal. The culprit was a `.npmignore` file that failed to exclude `.map` files, causing complete source maps to be bundled into the release.

What are source maps? They're mapping files between compiled code and the original source. With them, anyone can reconstruct the original TypeScript code. This leak involved roughly 1,900 TypeScript files totaling 510,000 lines of code.

Anthropic pulled the package within hours, but multiple mirror repositories had already appeared on GitHub. The code was out in the wild.

What I find most ironic is this: buried in that source code was a complete "Undercover Mode" anti-leak system, specifically designed to conceal AI involvement. And the real leak happened because someone forgot one line in a build script. This is probably the most classic story in software engineering — the most sophisticated defenses tend to fall to the most basic oversights.

To be clear: **this leak did not involve model weights, training data, or any user information.** What leaked was purely the client-side tool code — the CLI program that runs on your machine.

The Tech Stack: Some Surprising Choices

The first thing that caught my eye was the technology choices. Some were expected; others genuinely surprised me.

Component	Technology	Notable
Runtime	Bun	Not Node.js
UI Framework	React + Ink	React running in the terminal
Language	Strict-mode TypeScript	Comprehensive type safety
Schema Validation	Zod v4	Runtime type checking
Entry Point	main.tsx	785KB single file after compilation

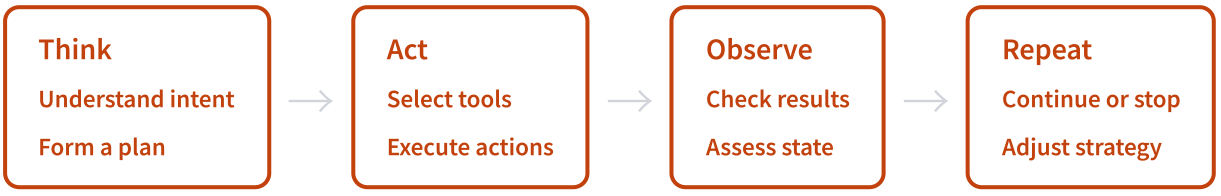
Why Bun over Node.js? One word: speed. Claude Code constantly spawns subprocesses, reads and writes files, and handles heavy concurrent requests. Bun outperforms Node.js in all of these areas — cold start time especially. For a command-line tool, those few hundred milliseconds between pressing Enter and seeing a response directly affect the experience.

Even more interesting is using React to render the terminal UI. Why not just splice strings together with console.log? Because Claude Code's terminal interface is actually quite complex: real-time progress bars, collapsible code diffs, permission confirmation prompts, multi-level nested tool call displays. The state management requirements are fundamentally the same problem as web frontend. React + Ink lets them build the terminal UI with a component-based approach, rather than writing spaghetti code to manually refresh the screen.

The scale of the codebase is also worth noting: the tool system alone is 29,000 lines, and the query engine is 46,000 lines. This is a serious engineering project, not a hastily written script.

The Agent Loop: TAOR Is the Core

Claude Code's core working loop is called TAOR: Think-Act-Observe-Repeat. Everything that happens after you type something in the terminal is driven by this loop.



Why does Claude sometimes take a few detours to finish a task? Because the TAOR loop isn't a one-shot execution. After each step, Claude re-examines the result and re-evaluates what to do next. It's not running a pre-written script — it's making decisions in real time. This is actually a lot like how humans work: you don't plan every line before you write code either. You write a bit, run it, check the output, and adjust.

Claude Code has over 40 built-in tools, but if you look carefully, every tool can be reduced to four capability primitives: **Read (retrieve information), Write (write to files), Execute (run commands), Connect (interface with external services).** The Bash tool is the universal adapter — any system command can go through it. That's why Claude Code often reaches for Bash even when dedicated file read/write tools are available. Flexibility is its greatest weapon.

Context management is another design highlight. Claude Code structures the system prompt in a modular way: some parts are static and can be cached (tool definitions, base instructions), while others are refreshed dynamically (current git status, CLAUDE.md content). The static parts leverage Anthropic's Prompt Caching to save tokens; the dynamic parts keep information accurate in real time.

Then there's the Context Compaction implementation. When a conversation grows too long and the context is close to overflowing, Claude Code automatically triggers a "compaction" pass — asking the model to summarize the prior conversation and replace the full history with a shorter representation. **This is a lossy operation.** So if you've noticed long conversations getting worse, it's not your imagination — some details genuinely get lost in compression. Knowing this, you understand why opening a fresh session periodically is a good habit.

The Permission System: Far More Complex Than You'd Think

Claude Code's permission system is considerably more complex than I expected. It's not a simple allow/deny binary — it's a three-tier model:

Mode	Behavior	Best For
Interactive	Prompt confirmation for every action	Beginners / sensitive projects
Auto	Safe actions run automatically; risky ones require confirmation	Day-to-day development
YOLO	Almost all actions run automatically	Trusted environments / CI

YOLO mode — yes, Anthropic actually calls it that — is backed by an ML-driven classifier. It categorizes the risk of each action into three levels: LOW, MEDIUM, and HIGH. Reading a file is LOW; running `rm -rf` is HIGH. The classifier isn't a simple rule matcher — it's a real machine learning model that factors in command content, target path, current project context, and more.

The defensive layer is equally thorough. The source code contains a list of protected files, including sensitive paths like `/etc/passwd`, `~/.ssh`, and `~/.aws/credentials`. More interesting is the path traversal defense: the code checks for unicode encoding bypasses (e.g., using zero-width characters to obfuscate paths), case-variant attacks (on Windows, `C:\Windows` and `c:\windows` are the same path), backslash injection, and similar techniques.

One detail that really stuck with me: **every permission decision triggers a separate LLM call to generate an explanation.** That means when Claude declines an operation, the explanation text isn't a pre-written template — it's generated by the model in real time. This ensures the explanation is relevant to the specific situation, but it also means every refusal costs extra tokens. Security isn't free.

The Future Hidden in Feature Flags

Perhaps the most fun part of the source code is the features that haven't shipped yet. The feature flags give a window into what Anthropic is building and thinking about. Here are a few worth discussing.

KAIROS is an always-on background assistant. Unlike the current model — where you have to open a terminal and type a command — KAIROS runs continuously in the background, listening to GitHub webhooks and proactively stepping in when a new PR, Issue, or CI failure appears.

It has a 15-second "blocking budget": each proactive action can use at most 15 seconds of compute before it's automatically queued. It also maintains an append-only log of everything it did while you weren't watching. **If it ships, the AI assistant shifts from "you go to it" to "it comes to you."**

ULTRAPLAN offloads complex planning tasks to the cloud. Local Claude Code has response time constraints, but some tasks genuinely require deep thinking. ULTRAPLAN lets you send a planning task to Opus 4.6 in the cloud with up to 30 minutes of thinking time. Go grab a coffee and come back to the result. Anthropic clearly understands something: not every problem can be solved by responding faster. Some problems just need time.

Coordinator Mode is a four-phase multi-agent orchestrator. The code shows four clearly defined phases: Research → Synthesis → Implementation → Verification. Each phase can spin up a different number of sub-agents working in parallel. This is more structured than the current Agent Teams — it's more like a real project manager coordinating a team.

Then there's **BUDDY**, a Tamagotchi-style AI pet system. 18 species, 5 rarity tiers, deterministic gacha mechanics. The code is surprisingly polished — not the kind of thing you'd throw in casually. This might be Anthropic's April Fool's Easter egg (the leak happened on March 31), but it might also be the team genuinely wrestling with a question: how do you make a command-line tool feel less cold? Can you build some kind of emotional connection between users and AI?

There are other flags worth noting: interleaved thinking mode (letting the model weave in reasoning steps during generation), 1M context window support (the current default is 200K), and resumable agents (pause and resume long-running tasks). Each one hints at the direction Claude Code is heading.

The Designs That Sparked Controversy

After the leak, the most heated community discussions weren't about technical architecture — they were about the values behind certain design decisions.

Undercover Mode allows Anthropic employees to automatically hide AI involvement when working on public open-source repositories. Specifically: when the tool detects that the current repo is public and the user is an Anthropic employee, it automatically strips the AI co-authorship markers from commit messages. Community reactions were sharply divided. One camp argued that AI participation in open-source contributions should be transparently disclosed, as a matter of respect for other contributors. The other camp held that code quality is the only metric that matters, and authorship is irrelevant. The debate itself is fascinating, because it touches on a bigger question: in the age of AI, what does "author" even mean?

Anti-Distillation is another intriguing mechanism. Claude Code injects fake tool definitions at runtime — definitions that look real but are deliberately wrong. The goal is to prevent competitors from scraping Claude's outputs to distill training data. If someone tried to train their own model directly on Claude Code's requests and responses, these incorrect definitions would "poison" the training data. Effective, but contested: is this legitimate self-defense or data poisoning?

Emotion detection might be the most unexpected finding. Claude Code uses regular expressions to detect negative emotions in user messages — frustration, anger. Why regex instead of an LLM for sentiment analysis? The answer is pragmatic: regex is 10,000 times faster than an LLM call, and the results are deterministic. When negative emotion is detected, Claude adjusts its tone to be more patient and more specific. It's a sound engineering choice — but it also raises a question: does it bother you that AI is "reading your mood" without you knowing?

> **My take:** The existence of these controversies tells us something important. AI tools are no longer purely technical products — they've entered territory where ethical boundaries need to be negotiated. It's the same arc social media went through: first "this is so convenient," then "wait, what data is it collecting about me?" Claude Code has arrived at the same crossroads.

What This Means for You

If you're a regular user, understanding these internal mechanics will help you use the tool more effectively. The TAOR loop explains why Claude sometimes takes multiple attempts to complete a task — that's not "making mistakes," it's "iterating." The lossy nature of Context Compaction explains why long conversations degrade over time, and why opening a fresh session periodically is a smart strategy. The three-tier permission model lets you choose the right level of trust for each situation, rather than being forced to choose between fully manual and fully automatic.

If you're a developer, Claude Code's architecture is worth studying seriously. This is probably the most mature AI Agent system implementation you can currently access. The TAOR loop design, the tool system abstraction, the layered context management strategy, the risk-tiered permission model — each one is an

engineering pattern worth borrowing. You don't have to design your own agent system from scratch; these 510,000 lines of code have already worked through an enormous number of hard problems.

I'll close with an observation I think is worth holding onto. **This leak was an accident, but it objectively accomplished something good: it helped far more people understand how AI Agent systems actually work.** Before this, most users' mental model of Claude Code was "a very smart command-line tool." Now we know how much engineering detail, how many trade-offs, and how many future possibilities live underneath the surface. That understanding builds both confidence in the tool and genuine respect for it.

After all, you can only truly master a tool when you understand both its limits and the logic behind its design.

Appendix B Connecting Chinese AI Models

Connecting Chinese AI Models

If you have difficulty accessing the Anthropic API from mainland China, or want to try running Claude Code on domestic models, this guide covers the setup process for all major platforms.

How It Works

Claude Code supports routing requests to third-party APIs via environment variables. Most major Chinese AI platforms offer Anthropic-compatible endpoints (the `/anthropic` path) that connect directly — no proxy layer required.

Key environment variables:

```
export ANTHROPIC_BASE_URL="https://xxx/anthropic"        # Vendor's Anthropic-compatible endp
export ANTHROPIC_AUTH_TOKEN="your-api-key"                 # Vendor API Key
export ANTHROPIC_MODEL="model-name"                        # Default model
export ANTHROPIC_DEFAULT_SONNET_MODEL="model-name"         # Sonnet role mapping
export ANTHROPIC_DEFAULT_HAIKU_MODEL="model-name"          # Haiku role mapping
export ANTHROPIC_DEFAULT_OPUS_MODEL="model-name"           # Opus role mapping
```

You can also set these in the `"env"` field of `.claude/settings.json` — same effect, but no manual export needed each time.

Direct-Connect Platforms at a Glance

The following five platforms provide Anthropic-compatible endpoints and connect directly to Claude Code with zero additional setup:

Platform	Recommended Model	Anthropic Endpoint	Highlights
DeepSeek	deepseek-chat	api.deepseek.com/anthropic	Very low pricing; new users get 5M free tokens
Zhipu GLM	GLM-4.7	open.bigmodel.cn/api/anthropic	Offers a ¥20/month Coding Plan
Kimi	kimi-k2.5	api.moonshot.cn/anthropic	256K extra-long context window
MiniMax	MiniMax-M2.7	api.minimaxi.com/anthropic	Exceptional value at $0.30/M input tokens
Alibaba Cloud Bailian	qwen3.5-plus	dashscope.aliyuncs.com/apps/anthropic	One API key to access models from multiple providers

Per-Platform Setup

DeepSeek

Register at platform.deepseek.com and obtain an API key, then:

```
export ANTHROPIC_BASE_URL="https://api.deepseek.com/anthropic"
export ANTHROPIC_AUTH_TOKEN="your-deepseek-api-key"
export ANTHROPIC_MODEL="deepseek-chat"
export ANTHROPIC_DEFAULT_SONNET_MODEL="deepseek-chat"
export ANTHROPIC_DEFAULT_HAIKU_MODEL="deepseek-chat"
export ANTHROPIC_DEFAULT_OPUS_MODEL="deepseek-chat"
```

DeepSeek V3.2 supports 128K context at extremely low cost (cache hits as cheap as $0.028/M tokens). Its Anthropic-compatible endpoint ignores some advanced fields (such as `mcp_servers` and `metadata`), but it's fully capable for everyday coding tasks.

Zhipu GLM

Get your API key from the bigmodel.cn console:

```
export ANTHROPIC_BASE_URL="https://open.bigmodel.cn/api/anthropic"
export ANTHROPIC_AUTH_TOKEN="your-zhipu-api-key"
export ANTHROPIC_MODEL="GLM-4.7"
export ANTHROPIC_DEFAULT_SONNET_MODEL="GLM-4.7"
export ANTHROPIC_DEFAULT_HAIKU_MODEL="GLM-4.5-Air"
export ANTHROPIC_DEFAULT_OPUS_MODEL="GLM-4.7"
```

Zhipu also offers a dedicated Coding Plan (¥20/month) and an international endpoint at `api.z.ai/api/anthropic`. The latest GLM-5 is a 745B-parameter flagship model with 200K context support.

Kimi (Moonshot AI)

Get your API key from platform.moonshot.cn:

```
export ANTHROPIC_BASE_URL="https://api.moonshot.cn/anthropic"
export ANTHROPIC_AUTH_TOKEN="your-moonshot-api-key"
export ANTHROPIC_MODEL="kimi-k2.5"
export ANTHROPIC_DEFAULT_SONNET_MODEL="kimi-k2.5"
export ANTHROPIC_DEFAULT_HAIKU_MODEL="kimi-k2.5"
export ANTHROPIC_DEFAULT_OPUS_MODEL="kimi-k2.5"
```

Kimi K2.5's standout features are its 256K context window and multimodal capabilities; automatic caching can cut costs by up to 75%.

MiniMax

Get your API key from platform.minimaxi.com:

```
export ANTHROPIC_BASE_URL="https://api.minimaxi.com/anthropic"
export ANTHROPIC_AUTH_TOKEN="your-minimax-api-key"
export ANTHROPIC_MODEL="MiniMax-M2.7"
export ANTHROPIC_DEFAULT_SONNET_MODEL="MiniMax-M2.7"
export ANTHROPIC_DEFAULT_HAIKU_MODEL="MiniMax-M2.7"
export ANTHROPIC_DEFAULT_OPUS_MODEL="MiniMax-M2.7"
```

MiniMax M2.7 cache-hit pricing goes as low as $0.06/M tokens, making it one of the best value options available.

Alibaba Cloud Bailian (Qwen)

Get your API key from the Bailian console:

```
export ANTHROPIC_BASE_URL="https://dashscope.aliyuncs.com/apps/anthropic"
export ANTHROPIC_API_KEY="your-dashscope-api-key"
export ANTHROPIC_MODEL="qwen3.5-plus"
export ANTHROPIC_DEFAULT_SONNET_MODEL="qwen3.5-plus"
export ANTHROPIC_DEFAULT_HAIKU_MODEL="qwen3.5-flash"
export ANTHROPIC_DEFAULT_OPUS_MODEL="qwen3-max"
```

Bailian's unique advantage is that a single API key can access models from multiple providers — the full Qwen lineup, DeepSeek, Kimi, GLM — making it a one-stop model aggregation platform.

Platforms That Require a Proxy

Volcengine (Doubao) and Tencent Cloud (Hunyuan) currently only offer OpenAI-compatible interfaces. You'll need LiteLLM for protocol translation:

```
# Install LiteLLM
pip install 'litellm[proxy]'

# Create config.yaml
# model_list:
#   - model_name: "doubao"
#     litellm_params:
#       model: "openai/your-endpoint-id"
#       api_base: "https://ark.cn-beijing.volces.com/api/v3"
#       api_key: "your-ark-api-key"

# Start the proxy
litellm --config config.yaml --port 8000

# Then point Claude Code at LiteLLM
export ANTHROPIC_BASE_URL="http://localhost:8000/anthropic"
```

The community-maintained claude-code-router tool is also available, supporting multi-provider configuration and dynamic model switching.

Things to Keep in Mind

> 注意
>
> **Capability differences:** When substituting a Chinese model for Claude, some of Claude Code's advanced features — such as complex multi-step reasoning and precise tool-call sequencing — may not perform as well. Start with simpler tasks to evaluate quality before using it on production work.
>
> **Network setup:** When using Chinese models, make sure your terminal's network environment can reach the domestic API endpoints without issue.
>
> **API changes:** The information above is current as of April 2026. Endpoint URLs, model names, and pricing for each platform can change at any time — consult the official documentation for each platform before proceeding.

Appendix C Complete Command Reference

Command Reference

A quick reference for all Claude Code CLI commands, in-session commands, keyboard shortcuts, and configuration options. As of April 2026.

CLI Commands

Commands starting with `claude` used in the terminal:

Command	Description
`claude`	Start an interactive session
`claude "query"`	Start a session with an initial prompt
`claude -p "query"`	Non-interactive (print) mode — outputs result and exits
`claude -c`	Continue the most recent session in the current directory
`claude -r "name"`	Resume a specific session by ID or name
`claude -w`	Start in an isolated git worktree
`claude update`	Update to the latest version
`claude auth login`	Log in (supports --email, --sso, --console)
`claude auth logout`	Log out
`claude auth status`	Check authentication status
`claude agents`	List all configured sub-agents
`claude mcp`	Manage MCP servers
`claude plugin`	Manage plugins
`claude remote-control`	Start the Remote Control server
`cat file | claude -p`	Pipe content as input

Common CLI Flags

Flag	Description
`--model`	Specify model (e.g. sonnet, opus, or full model name)
`--permission-mode`	Permission mode: default, acceptEdits, plan, auto, dontAsk, bypassPermissions
`--add-dir`	Add an additional working directory
`--effort`	Effort level: low, medium, high, max (Opus 4.6 only)
`--max-turns`	Limit agent turns (print mode only)
`--max-budget-usd`	Maximum API call budget in USD (print mode only)
`--output-format`	Output format: text, json, stream-json
`--mcp-config`	Load MCP servers from a JSON file
`--bare`	Minimal mode: skips hooks, skills, plugins, MCP, and CLAUDE.md
`--append-system-prompt`	Append text to the end of the system prompt
`--verbose`	Enable verbose log output
`--debug`	Debug mode, with optional category filter (e.g. "api,hooks")
`--chrome`	Enable Chrome browser integration
`--name, -n`	Set the session name
`--allowedTools`	List of tools that don't require permission prompts
`--disallowedTools`	List of tools to disable entirely
`--dangerously-skip-permissions`	Skip all permission confirmations (isolated environments only)

In-Session Slash Commands

Navigation & Session Management

Command	Description
`/clear`	Clear session history (aliases: /reset, /new)
`/compact [instructions]`	Compact the session, optionally specifying what to preserve
`/context`	Visualize current context usage
`/cost`	Show token usage statistics
`/resume [session]`	Resume a previous session (alias: /continue)
`/rename [name]`	Rename the current session
`/branch [name]`	Branch the current session (alias: /fork)
`/rewind`	Roll back to a previous checkpoint (alias: /checkpoint)
`/diff`	Interactive diff viewer
`/copy [N]`	Copy the most recent assistant reply to clipboard
`/export [filename]`	Export session as plain text
`/exit`	Exit (alias: /quit)

Configuration & Modes

Command	Description
`/model [model]`	Select or switch model
`/fast [on|off]`	Toggle Fast Mode (faster output)
`/effort [level]`	Set model effort level: low/medium/high/max/auto
`/plan [description]`	Enter Plan mode (discuss only, no execution)
`/vim`	Toggle Vim/Normal editing mode
`/voice`	Toggle voice input
`/config`	Open settings panel (alias: /settings)
`/permissions`	Manage tool permission rules
`/sandbox`	Toggle sandbox mode
`/theme`	Change color theme
`/color [color]`	Set the prompt bar color

Extensions & Integrations

Command	Description
`/memory`	Edit CLAUDE.md files and manage auto memory
`/init`	Initialize a project CLAUDE.md
`/skills`	List all available Skills
`/hooks`	View Hook configuration
`/mcp`	Manage MCP server connections
`/plugin`	Manage plugins
`/agents`	Manage agent configuration
`/ide`	Manage IDE integrations
`/chrome`	Configure Chrome integration
`/add-dir <path>`	Add a working directory

Information & Diagnostics

Command	Description
`/help`	Show help
`/status`	View version, model, account, and connection status
`/doctor`	Diagnose installation and configuration issues
`/usage`	Show plan usage and rate limit status
`/stats`	Visualize usage statistics
`/insights`	Generate a usage analysis report
`/release-notes`	View changelog
`/tasks`	List background tasks
`/pr-comments [PR]`	Fetch GitHub PR comments
`/security-review`	Analyze security vulnerabilities in the current branch

Special Prefixes

Prefix	Description	Example
`/`	Command/Skill completion	`/proofreading`
`!`	Execute a Bash command directly	`! npm run test`
`@`	File path mention/completion	`@src/index.ts`

Keyboard Shortcuts

Core Actions

Shortcut	Description
`Ctrl+C`	Cancel current input or stop generation
`Ctrl+D`	Exit session
`Shift+Tab`	Cycle through permission modes (default → plan → auto → ...)
`Alt+P`	Switch model
`Alt+T`	Toggle extended thinking
`Alt+O`	Toggle Fast Mode
`Ctrl+R`	Reverse search command history
`Ctrl+B`	Send running task to background
`Ctrl+T`	Toggle task list display
`Ctrl+O`	Toggle verbose output (expand MCP call details)
`Ctrl+V / Cmd+V`	Paste image from clipboard
`Ctrl+G`	Open current input in an external editor
`Ctrl+L`	Redraw screen
`Esc + Esc`	Rewind or summarize
`Hold Space`	Voice input (push-to-talk)

Multi-line Input

Method	Action
Backslash newline	`\` + Enter
macOS default	Option + Enter
iTerm2/WezTerm/Kitty	Shift + Enter
Control sequence	Ctrl + J
Paste	Paste multi-line text directly

Text Editing

Shortcut	Description
`Ctrl+K`	Delete to end of line
`Ctrl+U`	Delete to beginning of line
`Ctrl+Y`	Paste deleted text
`Alt+B`	Move cursor back one word
`Alt+F`	Move cursor forward one word
`Up/Down`	Navigate command history

Environment Variables

Variable	Description
`ANTHROPIC_API_KEY`	Anthropic API key
`ANTHROPIC_BASE_URL`	API endpoint URL (for third-party models)
`ANTHROPIC_AUTH_TOKEN`	Authentication token (used by some third-party platforms)
`ANTHROPIC_MODEL`	Default model name
`ANTHROPIC_DEFAULT_SONNET_MODEL`	Model mapped to the Sonnet role
`ANTHROPIC_DEFAULT_HAIKU_MODEL`	Model mapped to the Haiku role
`ANTHROPIC_DEFAULT_OPUS_MODEL`	Model mapped to the Opus role
`CLAUDE_CODE_DISABLE_NONESSENTIAL_TRAFFIC`	Disable non-essential network requests (set to 1)
`API_TIMEOUT_MS`	API call timeout in milliseconds
`NODE_EXTRA_CA_CERTS`	Path to additional CA certificates (corporate networks)
`SSL_CERT_FILE`	Path to SSL certificate file

settings.json Core Configuration

Config file location: `~/.claude/settings.json` (global) or `.claude/settings.json` (project-level).

Option	Description
`hooks`	Hook configuration (PreToolUse, PostToolUse, Stop, etc.)
`env`	Environment variable overrides (e.g. API keys, model config)
`permissions`	Tool permission rules (allow/deny lists)
`autoMode`	Custom classifier rules for Auto mode
`availableModels`	Restrict the list of selectable models
`defaultShell`	Default shell (bash or powershell)
`cleanupPeriodDays`	Session cleanup interval (default: 30 days)
`claudeMdExcludes`	Exclude specific CLAUDE.md files (glob pattern)
`attribution`	Customize attribution for git commits and PRs

Appendix D Frequently Asked Questions

Frequently Asked Questions

A curated collection of high-frequency questions from official documentation, GitHub Issues, and community discussions. Check here first when something goes wrong.

Installation & Configuration

Q: After installation, typing `claude` gives "command not found"

The most common issue. Your shell's PATH hasn't been updated. Fully close and reopen your terminal, or check whether `~/.zshrc` (macOS) / `~/.bashrc` (Linux) contains the Claude PATH entry. Run `/doctor` for automatic diagnostics.

Q: macOS installation throws a dyld error or Abort trap

Claude Code requires macOS 13.0 or later. You'll need to upgrade your OS first.

Q: Can't connect on a corporate network

Corporate VPNs may use SSL man-in-the-middle certificates. You'll need to configure `NODE_EXTRA_CA_CERTS` to trust the custom CA certificate. Contact your IT department for the certificate path.

Q: "Git not found" error on Windows

Install Git for Windows and make sure to check "Add Git to PATH" during setup. Spaces in Windows paths can also cause MCP path resolution failures.

Tips & Best Practices

Q: What should I put in CLAUDE.md?

Start with three things: tech stack, coding conventions, and known pitfalls. Keep it concise — Claude can effectively follow roughly 100–150 user instructions. Run `/init` to auto-generate an initial file. See § 05 for details.

Q: What do I do when the context window fills up?

Claude currently supports up to 1M token context. Auto-compression kicks in at around 83.5% usage. Use `/compact [focus]` to compress manually, and `/context` to see how each component is consuming context. For long tasks, consider breaking them into multiple shorter sessions.

Q: How do I get better results from Claude Code?

Five core tips: (1) Use Plan mode to discuss the approach before executing; (2) Provide exact file paths and line numbers; (3) Don't attach too many MCP servers — each one consumes context; (4) Break large tasks into smaller sessions; (5) Give clear completion criteria ("stop when the tests pass" rather than vague requests).

Q: How do I create custom commands?

Create `.md` files in the `.claude/commands/` directory, write the steps in natural language, and use the `$ARGUMENTS` placeholder where needed. Invoke them with `/command-name`. See § 07 for details.

Q: How do I use Claude Code in CI/CD?

Use `claude -p "query"` for non-interactive mode, which supports text/json/stream-json output formats. In GitHub Actions, store your API Key in Repository Secrets. Over 60% of teams integrate via GitHub Actions.

Billing & Usage

Q: What's the difference between Pro ($20), Max ($100/$200), and API Key?

Plan	Price	Usage	Best For
Pro	$20/mo	~5x free tier	Light personal use
Max 5x	$100/mo	~25x free tier	Heavy personal use
Max 20x	$200/mo	~100x free tier	Power users / small teams
API Key	Pay per token	Unlimited	CI/CD, enterprise integration

Q: What do I do when I hit the Rate Limit? (429 error)

Four ways to mitigate: (1) Use `/compact` to compress context and reduce input tokens; (2) Split large sessions into smaller ones; (3) Avoid peak hours (5am–11am Pacific Time); (4) Upgrade to the Max plan.

Q: How do I save tokens?

(1) Break large files into smaller, single-responsibility files; (2) Disable MCP servers you're not using; (3) Compress regularly with `/compact`; (4) Batch related operations into a single prompt.

Q: How much more does Agent Teams cost compared to regular usage?

Roughly 7x. Each Teammate maintains an independent context window — effectively a separate Claude instance running in parallel.

Q: Why am I being asked to set an API Key after subscribing?

The system may have detected an existing `ANTHROPIC_API_KEY` environment variable from a previous project. Run `unset ANTHROPIC_API_KEY` to clear it, then `claude login` to re-authenticate.

Permissions & Security

Q: Is Auto mode safe?

Auto mode is the officially recommended balance between efficiency and safety. Low-risk operations (reading files) are approved automatically; high-risk ones (deleting files, pushing code) still require confirmation. Toggle it with `Shift+Tab`.

Q: What is `--dangerously-skip-permissions`?

It bypasses all permission confirmations — and the word "dangerously" in the name is not a joke. Only use this inside an isolated container or VM, and pair it with `--network none` to prevent data leakage. Never use it for day-to-day development.

Q: Will Claude send my code somewhere external?

Claude Code's sandbox restricts the Bash tool's filesystem and network access. However, MCP tools may connect to external services. Review the access scope of each MCP server, and use environment variables instead of hardcoding sensitive tokens.

Errors & Troubleshooting

Q: What's the difference between 529 Overloaded and 429 Rate Limit?

529 means Anthropic's servers are overloaded — it's not your fault, and it usually resolves within 30 seconds. 429 means you've hit your personal rate limit, and you'll need to compact your context or wait for it to reset.

Q: Claude forgot earlier instructions after a long conversation

Context compression can lose early details. The fix: put important constraints in CLAUDE.md rather than just stating them in conversation; use a PostCompact Hook to automatically re-inject critical rules after compression; and save intermediate results to files after completing each batch of tasks.

Q: Claude started drifting off course mid-task

Clear verification criteria are especially important here. Telling it "stop when the tests pass" or "just generate the file" converges much faster than vague requests like "help me optimize this." Try Plan mode to align on the approach before executing.

Q: MCP server connection failure

Common causes: spaces in paths, wrong scope configured (global vs. project), or too many servers connected (5+ servers can add 50+ tool descriptions that consume context). Start with the minimal set and add servers one at a time.

Q: The ~/.claude directory keeps growing

This is a known issue. Claude Code has no automatic disk management — you'll need to clean it up manually from time to time. Old session files can be deleted, but keep `settings.json` and `CLAUDE.md`.

> **核心建议**
>
> For any strange issue, run `/doctor` first. This built-in diagnostic tool automatically checks installation, network, configuration, and other common problems — and often provides a direct solution.

Appendix E Glossary

Glossary

A quick reference for core concepts used throughout this book. Listed alphabetically with English-Chinese pairings.

A

Term	Definition
Agent 智能体	An AI system capable of autonomous planning, execution, observation, and adjustment. Claude Code is itself an Agent — it reads code, writes code, runs commands, handles errors, and completes the entire loop automatically. This is fundamentally different from a traditional chatbot that simply responds to each individual message.
Agent Teams 多Agent团队	Claude Code's multi-instance collaboration feature. You can launch multiple Claude Code processes, each responsible for a different module, functioning like a small development team. Officially released in February 2026.
Agentic Coding Agent式编程	A programming paradigm where AI Agents — rather than humans — write the code. The core shift is from "writing code" to "giving instructions," with the AI handling planning and execution. Claude Code is the most representative Agentic Coding tool available today.
API Key API密钥	A credential required to call an AI model service. Using Claude Code requires either an Anthropic API Key or a Pro/Max subscription. When configuring third-party models, you'll also need the corresponding platform's API Key.
Auto Mode 自动模式	A permission mode in Claude Code. When enabled, low-risk operations (such as reading files) are approved automatically, while medium- to high-risk operations (such as deleting files) still require confirmation. Best suited for scenarios where you trust Claude to work independently.

C

Term	Definition
CLAUDE.md	Claude Code's memory file. Placed in the project root directory, it contains project conventions, architectural decisions, coding style guidelines, and more — all of which Claude reads automatically on startup. Think of it as a persistent project map for your AI. Supports three levels: global (~/.claude/), project, and subdirectory.
Claude Code	A terminal-based AI coding tool developed by Anthropic. It runs in the command line and handles the full development lifecycle — writing, debugging, testing, and deploying code — through natural language interaction. Created by Boris Cherny and publicly released in February 2025.
Claude Opus / Sonnet / Haiku	The three tiers of the Claude model family. Opus is the most powerful (complex reasoning), Sonnet strikes a balance (the go-to for everyday coding), and Haiku is the fastest (simple tasks). Current versions are Opus 4.6, Sonnet 4.6, and Haiku 4.5.
Compact 压缩	Claude Code's context compression mechanism. When a conversation grows too long, the system compresses the history into a summary to free up space. Can be triggered manually with `/compact`. Compression is lossy — important constraints should be written into CLAUDE.md rather than mentioned only in conversation.
Computer Use 屏幕操作	Claude Code's ability to interact with the computer screen — viewing screen content, moving the mouse, clicking buttons, and typing text. Launched in March 2026, this extends Claude's reach from "operating code only" to "operating any software."
Context Window 上下文窗口	The total amount of information a model can "see" in a single session. Claude currently supports a maximum context of 1M tokens — roughly equivalent to the entire codebase of a large project. The larger the window, the more complete Claude's understanding of the overall project.

H

Term	Definition
Harness 工具链/脚手架	The automated working environment built around an AI model. Includes Skills, Hooks, MCP, and more. Investment in the Harness layer yields exponential returns: build it once, and it runs indefinitely. See the three-layer model in §10 for details.
Harness Engineering Harness工程	An AI collaboration methodology. The core idea: rather than endlessly refining your prompts, invest your energy in building Context (memory files) and a Harness (automated toolchain) — and earn compounding, exponential returns over time.
Hooks 钩子	Claude Code's event-triggered mechanism. Automatically executes shell commands when specific events occur — such as before or after a tool call, or when a notification is sent. Similar to Git Hooks, but designed for AI workflows. Configured in settings.json.

M

Term	Definition
MCP Model Context Protocol 模型上下文协议	An open protocol proposed by Anthropic that allows AI models to connect to external tools and data sources. Through MCP, Claude Code can access databases, call APIs, interact with web pages, and more. Released in November 2024, it has since become an industry standard.
Multi-Agent 多Agent	A working mode where multiple AI instances collaborate in parallel. In Claude Code, you can open multiple terminal windows simultaneously, each running its own Claude instance to handle different tasks. Agent Teams is the official framework for multi-agent collaboration.

P

Term	Definition
Plan Mode 规划模式	A working mode in Claude Code. When enabled, Claude discusses plans without taking any action, letting you confirm the approach before execution begins. Ideal for upfront planning on complex tasks. Toggle with `Shift+Tab`.
Prompt Engineering 提示词工程	The practice of optimizing the text you give to an AI in order to get better output. In the context of Claude Code, a good prompt should clearly state the goal, provide constraints, and define success criteria — rather than obsessing over exact phrasing.

S

Term	Definition
Skills 技能包	Reusable capability modules for Claude Code. A Skill is a SKILL.md file that defines the workflow and standards for a specific task. You can write your own or install them from the community. See §07 for a detailed introduction.
SubAgent 子Agent	A child-process Agent spawned by Claude Code. The primary Agent can delegate tasks to SubAgents for parallel execution while continuing other work itself. Launched in July 2025, this is the foundational capability behind multi-agent collaboration.

T

Term	Definition
TAOR Loop Think-Act-Observe-Repeat	The core working loop of Claude Code. Think about the current state → Take an action → Observe the result → Repeat if not done. A single task may cycle through this loop dozens of times before completion. Understanding this mechanism helps you give better instructions.
Token	The basic unit by which AI models process text. Roughly 4 English characters or 1–2 Chinese characters equals 1 token. Claude Code usage and cost are both calculated in tokens. 1M tokens is approximately 750,000 words of English text.
Tool Use 工具调用	An AI model's ability to call external tools — such as reading files or executing commands. Claude Code has 40+ internal tools, which reduce to four core primitives: Read, Write, Execute, and Connect. Each tool call has its own independent permission controls.

V

Term	Definition
Voice Mode 语音模式	Claude Code's voice interaction feature. Simply speak to the terminal to issue commands, and Claude responds with voice. Launched in March 2026, it's ideal for continuing coding work while on the move, doing chores, or in other hands-free situations.

Other Common Abbreviations

Abbr.	Full Form	Description
CLI	Command Line Interface	The terminal-based interface through which users interact with a program. Claude Code is a CLI tool that runs in the terminal.
GA	General Availability	The official public release of a product. Claude Code reached GA in May 2025.
IDE	Integrated Development Environment	A software environment for writing and managing code, such as VS Code, Cursor, or JetBrains. Claude Code can be integrated as an IDE extension.
LLM	Large Language Model	A large-scale AI model trained on text. Claude is Anthropic's family of LLMs.
MVP	Minimum Viable Product	The simplest version of a product that can be used to validate an idea. With Claude Code, you can rapidly build an MVP from concept to testable prototype.
PR	Pull Request	A request to merge code changes into a codebase. Claude Code can create a PR with a single instruction, automatically generating the title and description.

Appendix F The State of Agentic Coding

The State of Agentic Coding in 2026

This appendix traces the evolution of AI coding from "autocomplete" to "autonomous agents," and what that landscape actually looks like in 2026. The data comes from Anthropic's official reports, industry research, and my own hands-on experience.

From Copilot to Agent: Three Generations of AI Coding Tools

The history of AI-assisted coding is short, but the pace of evolution has been remarkable. It breaks down into roughly three generations:

Generation	Representative Products	What It Can Do	User's Role
First Gen: Autocomplete (2021–2023)	GitHub Copilot, TabNine	Complete the current line or code block based on context	You write code; AI helps you type faster
Second Gen: Conversational Coding (2023–2024)	ChatGPT, Claude web interface	Explain code, generate functions, answer questions	You ask questions; AI gives answers; you paste into your editor
Third Gen: Agentic Coding (2024–present)	Claude Code, Cursor, Windsurf	Autonomously read code, write code, run tests, fix bugs, manage files	You define the goal; AI executes independently

The critical leap happened between the second and third generations. This wasn't just "smarter autocomplete" — it was a qualitative shift: AI transformed from "a tool that answers questions" into "an agent that can act autonomously."

Two capabilities drove this change: **tool use** and **multi-step reasoning**. Second-generation AI could only output text — you copied and pasted. Third-generation AI can directly manipulate the filesystem, run terminal commands, call APIs, and navigate across multiple files. It doesn't just "tell you what to do" — it "goes ahead and does it."

What the Data Shows

In early 2026, Anthropic published a report based on real Claude Code usage data. Several key findings stood out:

78% of sessions involve multi-file operations

This tells us that most real-world coding tasks aren't "write one function" — they require coordinated changes across multiple files. Adding an API endpoint, for instance, might mean touching the routing file, controller, data model, test files, and documentation simultaneously. Second-generation AI tools could only see one file at a time, making such tasks painful. Agentic tools are a natural fit for this kind of multi-file coordination.

Average session length grew from 4 minutes to 23 minutes

Users are no longer "ask one question, get an answer, move on." An average session length of 23 minutes signals that people are using Claude Code to complete entire tasks — not asking for directions, but getting in the car and riding along. For complex tasks like building a project from scratch, sessions can run for hours.

44% market share in complex tasks

This is Claude's market share figure in the agentic coding space. Complex tasks — multi-step, multi-file — are where agentic tools compete most fiercely. Simple tasks like completing a single line of code have become highly commoditized, but quality differences in complex tasks remain substantial.

Top three use cases

Use Case	Share	Typical Tasks
Feature Development	43%	From requirements to implementation, including code, tests, and documentation
Code Understanding	28%	Reading unfamiliar codebases, understanding architecture, tracking down bugs
Refactoring & Optimization	18%	Performance tuning, code reorganization, clearing technical debt

"Code understanding" at 28% is higher than most people would expect. AI coding tools aren't just for writing new code — comprehending existing code is an equally core need. Many developers spend more time reading code than writing it.

Enterprise Adoption Today

AI coding is no longer just a toy for individual developers. Enterprise adoption is accelerating rapidly:

Rakuten: After deploying Claude Code, sprint cycles shrank from 24 days to 5 days. Not because developers were typing faster, but because AI automated a large volume of repetitive work: code reviews, test writing, documentation updates.

Zapier: 97% of engineers use AI coding tools in their daily work — a shift from "a few people experimenting" to "standard practice for everyone."

Market size projections: The AI coding tools market is expected to grow from $7.84 billion in 2025 to $52.62 billion by 2030, at a compound annual growth rate of 46.3% — faster than most technology subsectors.

Eight Key Trends

Based on Anthropic's report and industry observation, here are eight defining trends in Agentic Coding for 2026:

Trend 1: From Assistance to Leadership

Early AI coding was "humans write code, AI assists." Increasingly, the dynamic has flipped: "AI writes code, humans review." Especially for well-defined tasks — CRUD operations, form validation, API wrappers — having AI write and humans review is far more efficient than the reverse.

Trend 2: Context Windows Define the Ceiling

The core constraint on agentic coding isn't model intelligence — it's context window size. The larger the window, the more code the AI can simultaneously see and process, and the better it handles complex tasks. From the original 4K tokens to today's 1M+, each expansion has brought a meaningful leap in capability.

Trend 3: Engineering Craft Matters More Than Model Capability

Claude Code's competitive edge doesn't come solely from the Claude model itself. The toolchain design (how files are read and written, how commands are executed), the permission system (balancing safety and convenience), and persistence mechanisms (CLAUDE.md, Memory) — these engineering-layer decisions shape the user experience just as much as the model's raw capability. This is what § 10 calls Harness Engineering: the engineering wrapper around the model determines how much of that model's capability is actually unlocked in practice.

Trend 4: Explosion in Non-Coding Use Cases

Perhaps the most underappreciated shift in Claude Code is that it's breaking out of the "coding tool" category entirely. Writing articles, building presentations, managing files, automating workflows — any

task that amounts to "processing information according to rules" is, at its core, something an agentic tool can handle. The content creation automation described in § 12 is just one example. This trend will keep accelerating.

Trend 5: From Tool to Platform

Skills, Hooks, and MCP aren't standalone features — they're the three pillars of a platform. Together, they transform Claude Code from "an AI coding tool" into "an extensible AI work platform." Anyone can write a Skill, build an MCP Server, or configure Hooks to turn Claude Code into their own custom instrument. The logic mirrors that of a smartphone's App Store: a platform's value lies not in how many built-in features it ships with, but in how rich an ecosystem can grow around it.

Trend 6: Enterprise CLI Tools Go Mainstream

Feishu launched a CLI tool. DingTalk launched a CLI tool. WeCom now has a CLI interface. Even AI video generation tools like LibTV shipped a Claude Code Skill on day one of their product launch.

What does this mean? More and more enterprise software will have two modes of interaction: a graphical interface for humans, and a command-line interface for AI. You'll use Claude Code to send messages via Feishu, create approval workflows in DingTalk, or push notifications through WeCom. AI becomes your "universal remote control," while enterprise tools become the "devices" it can operate.

Trend 7: Products Designed for AI, Not Just for Humans

Traditional software design centers on the question "how do humans interact with this most conveniently?" The emerging trend is designing from the ground up for "how does AI call this most conveniently?" — shipping Skill files, API interfaces, and MCP Servers so that AI agents become first-class users.

This isn't science fiction. When a video generation tool ships a Claude Code Skill on day one, it signals that the product team believes AI invocation may be a more primary use pattern than manual operation. That bet will be embraced by more and more product teams over the next two years.

Trend 8: "Can't Code" Is No Longer a Barrier

The deepest impact of Agentic Coding isn't making programmers more productive — it's enabling non-programmers to build software. Kitty Light (§ 13) is a case in point: someone who had never written a line of code built the #1 paid app in the App Store using AI.

This is not an outlier. As agentic tools mature, the barrier to "programming" will keep falling. Future competition won't be about who can write code — it'll be about who can spot problems worth solving, define clear product requirements, and make sound judgments. Code becomes the execution layer, no longer the bottleneck.

What These Trends Mean for You

If you're a programmer: your value won't disappear, but it will shift — from "writing code" to "designing systems, defining interfaces, making technical decisions." AI that can write code is increasingly abundant; people who can judge what code should be written have always been scarce.

If you're not a programmer: you already have the capacity to build software. What you lack isn't technical skill — it's what this book has covered: how to collaborate with AI, how to define requirements, how to validate results, how to iterate and improve. None of that depends on whether you can write code.

Whichever you are, Claude Code is a starting point for engaging with this future. The tools will keep changing, the trends will keep accelerating — but the core capabilities of understanding problems, defining solutions, and validating outcomes will never go out of style.

Appendix G CLAUDE.md Template Collection

CLAUDE.md Templates for Every Project Type

Different project types call for different CLAUDE.md files. This appendix provides four battle-tested templates you can copy and adapt immediately. Each template includes comments explaining the reasoning behind every section.

From .cursorrules to CLAUDE.md

Before Claude Code, I was already writing .cursorrules files in Cursor — fundamentally the same idea as CLAUDE.md: tell the AI what your project is, how to write code, and what rules to follow. I later built a VSCode extension with a dozen cursorrules templates for different tech stacks (iOS, React, Vue, mini-programs, Chrome extensions, and more) to help users get set up quickly.

When I switched to Claude Code, those rules transferred almost directly. The core structure is identical:

.cursorrules	CLAUDE.md	Purpose
Role & Goal	Project Overview	Tell the AI what it's working on
Tech Stack Rules	Technical Standards	Constrain code style and technology choices
Three-Step Workflow	Workflow	Define how things get done
Problem-Solving Patterns	Troubleshooting Guide	What to do when things go wrong

If you've been using Cursor and already have a .cursorrules file, you can copy its contents directly into CLAUDE.md. The format is fully compatible.

Template 1: Frontend Project (React/Next.js)

```
# Project Name

## Project Overview

A Next.js 15-based XXX platform using React 19 + Tailwind CSS + shadcn/ui.

## Tech Stack
- **Framework**: Next.js 15 (App Router)
- **UI**: Tailwind CSS + shadcn/ui
- **State Management**: Zustand
- **Data Fetching**: React Query
- **Types**: TypeScript strict mode
- **Package Manager**: pnpm

## Project Structure

```
src/
├── app/            # Next.js routes and pages
├── components/     # Reusable components
│   ├── ui/         # shadcn/ui base components
│   └── features/   # Business feature components
├── hooks/          # Custom Hooks
├── lib/            # Utility functions and config
├── types/          # TypeScript type definitions
└── styles/         # Global styles
```

Code Standards

- Components: function components + Hooks, no class components
- File naming: kebab-case (my-component.tsx)
- Exports: named exports, no default export (except page files)
- Styling: Tailwind first, custom CSS via CSS Modules
- State: useState for local, Zustand store for cross-component
- No component file should exceed 150 lines — split it if it does

Common Commands

- Start dev server: `pnpm dev`
- Run tests: `pnpm test`
- Build: `pnpm build`
- Add shadcn component: `pnpm dlx shadcn@latest add [component-name]`

Gotchas

- Don't use useState/useEffect inside Server Components
```

```
- Use next/image for images, not native img tags
- All API calls go through src/lib/api.ts — don't fetch directly inside components
```

Template 2: iOS App Project (SwiftUI)

```
# Project Name

## Project Overview
A SwiftUI iOS app that does XXX. Minimum deployment target: iOS 17.

## Project Status

| Module | Status | Notes |
|--------|--------|-------|
| Core Features | ✅ Done | Basic functionality live |
| User System | 🔨 In Progress | Login / signup / profile |
| Push Notifications | ⏳ Not Started | Needs APNs configuration |

## Tech Stack
- **UI Framework**: SwiftUI
- **Architecture**: MVVM
- **Persistence**: SwiftData
- **Networking**: URLSession + async/await
- **Dependency Management**: Swift Package Manager

## Project Structure
```
ProjectName/
├── App/   # App entry point and configuration
├── Views/   # SwiftUI views
│   ├── Home/
│   ├── Settings/
│   └── Shared/   # Shared UI components
├── ViewModels/   # View models
├── Models/   # Data models
├── Services/   # Network / storage service layer
└── Utilities/   # Helpers and extensions
```

## Code Standards
- Each View has a corresponding ViewModel
- Use @StateObject to own a ViewModel, not @ObservedObject
- Async operations use async/await, not completion handlers
- Colors defined in Asset Catalog, never hardcoded
- File naming: UpperCamelCase (HomeView.swift, HomeViewModel.swift)
- No file should exceed 200 lines

## Common Commands
- Run in Xcode: Cmd+R
- Run tests: Cmd+U
- Clean build cache: Cmd+Shift+K

## Gotchas
```

```
- Don't perform async work inside a View's body (use the .task modifier)
- Device testing requires a signing Team set in Xcode
- @Published properties must be updated on the MainActor
- Camera / photo library permissions require Usage Descriptions in Info.plist
```

Template 3: Content Creation Project

```
# Writing Project

## Project Position

This is a content creation workspace. I'm [your identity], primarily producing [your content

## Workspace Routing

| Keyword | Workspace | Rules File |
|---------|-----------|-----------|
| WeChat, article | Article Writing | /articles/RULES.md |
| video, script | Video Production | /videos/RULES.md |
| social, tweet | Social Media | /social/RULES.md |

## Writing Style

- Voice: [your style — e.g., conversational, warm, unpretentious]
- Quotes: use 「」 not ""
- Em dashes (——): no more than 1-2 per piece (a telltale AI marker)
- Bold: ~10 instances per piece, marking key sentences
- Paragraphs: 3-5 sentences each — longer blocks are tiring to read
- Banned words: [list words you never want to see — e.g., synergy, in conclusion, it's worth

## Editorial Standards

Three-pass review:
1. Content pass: Are the facts accurate? Is the logic sound? Anything missing?
2. Style pass: Does it feel AI-generated? (filler phrases, mechanical structure, neutral tone
3. Detail pass: Sentence length, paragraph spacing, consistent formatting

## Output Standards

- Article drafts: written to .md files, never pasted directly into the chat
- File location: inside the relevant project folder
- Naming convention: YYYY.MM-title.md

## Common Commands

- /proofreading → three-pass editorial review
- /research → structured research
- /topic-gen → topic ideation
```

Template 4: Backend API Project (Python/Node.js)

````
# Project Name

## Project Overview
A REST API service built on [framework], handling XXX business logic.

## Tech Stack
- **Runtime**: Python 3.12 / Node.js 22
- **Framework**: FastAPI / Express
- **Database**: PostgreSQL
- **ORM**: SQLAlchemy / Prisma
- **Cache**: Redis
- **Deployment**: Docker + AWS ECS

## Project Structure
```
src/
├── api/           # Route and endpoint definitions
│   └── v1/        # API versioning
├── models/        # Data models / schemas
├── services/      # Business logic layer
├── repositories/  # Data access layer
├── middleware/    # Middleware (auth, logging, error handling)
├── config/        # Configuration files
└── tests/         # Tests
    ├── unit/
    └── integration/
```


## Code Standards
- API paths: RESTful style, lowercase + hyphens (/user-profiles)
- Error handling: unified error format { code, message, details }
- Auth: JWT Bearer Token, secret read from environment variables
- Validation: validate at the entry layer (Pydantic/Zod), don't repeat in the business layer
- Logging: structured logs (JSON format), include request_id

## Environment Variables

```
DATABASE_URL=       # PostgreSQL connection string
REDIS_URL=          # Redis connection string
JWT_SECRET=         # JWT signing secret (.env file, never committed)
API_KEY=            # External API key (.env file, never committed)
```


## Common Commands
- Start dev: `docker compose up` or `uvicorn main:app --reload`
- Run tests: `pytest` / `pnpm test`
- Database migration: `alembic upgrade head` / `npx prisma migrate dev`
````

```
- View API docs: `http://localhost:8000/docs`

## Security Rules

- ❌ Never hardcode secrets, passwords, or tokens in source code
- ❌ Never log sensitive data (user passwords, tokens, etc.)
- ✅ All user input must be validated and sanitized
- ✅ SQL queries use parameterized queries — never string concatenation
- ✅ All API endpoints require authentication (except /health and /docs)
```

5 Principles for Writing a Good CLAUDE.md

Principle 1: Specific beats vague. "Code style should be good" is useless. "Functions no longer than 30 lines, files no longer than 200 lines, named exports only" is actionable. The AI needs rules it can actually execute.

Principle 2: Concise beats comprehensive. CLAUDE.md is loaded into the context window at the start of every conversation. Under 200 lines is ideal; beyond 500 lines you're just burning tokens. If you have too many rules, break them into sub-files and reference them from the main CLAUDE.md.

Principle 3: Documented gotchas are more valuable than best practices. The AI already knows best practices. But "using useState inside a Server Component throws an error" — that's a project-specific pitfall the AI has no way of knowing. Without writing it down, you'll keep hitting it. The most valuable section of any CLAUDE.md is usually "Gotchas."

Principle 4: Keep it alive. CLAUDE.md isn't a one-and-done document. Every time you notice Claude making the same mistake, add a rule. Do a cleanup pass once a month to remove stale rules. Treat it as a living document.

Principle 5: Start with 3 rules, then grow. Don't try to write the perfect CLAUDE.md on day one. Start with the 3 things you care about most — say, tech stack, file naming conventions, and security requirements. Use it for a few days, add more as needs arise. Three months from now you'll have a CLAUDE.md that fits your workflow perfectly.

Claude Code: The Complete Guide

Orange Book Series by HuaShu

HuaShu

The Definitive AI Programming Guide for Engineers and Product Managers

Based on Anthropic's official documentation and Boris Cherny's public talks

Download Latest Version →

This guide is continuously updated

Get the latest version at: huasheng.ai/orange-books

Bilibili · WeChat: HuaShu · X/Twitter · YouTube · Xiaohongshu · Website

Created by HuaShu • v2.0 • April 2026