

FREE SAMPLE

Shipping with Claude Code

The Practical Playbook

*The full contents, the Four Habits, the
prompts that work and the mistakes to avoid*

Pranav S.

FREE SAMPLE · OCTOBER 2026

Contents of the full book

Everything in Shipping with Claude Code, section by section. The parts marked INCLUDED are complete in this sample, and those marked EXCERPT are partly included. Every chapter also ends with prompts to copy, common mistakes, a 10-minute exercise and key takeaways.

How to Use This Book

Who it's for · How to read it · Conventions · The example projects · A note on versions

PART I: FOUNDATIONS

CHAPTER 1

Why Claude Code Is Different EXCERPT

Three ways to use AI for code · What happens when you give Claude Code a task · How to think about it: a capable new colleague · What it's good at, and where it struggles · The idea this book keeps coming back to · The Four Habits · How this book is organized

CHAPTER 2

Setup in 15 Minutes

Before you start · Install it · Sign in · Choose where you'll use it · A tour of your first session · Pick a model and effort level · Permission modes · Set up your terminal · Check everything works

CHAPTER 3

Your First Real Task

Pick a good first task · Step 1: Ask Claude to explain the codebase · Step 2: Create CLAUDE.md with /init · Step 3: Make a small change · Step 4: Review the diff · Step 5: Verify · Step 6: Commit · If it goes wrong · What just happened

PART II: THE CORE WORKFLOW

CHAPTER 4

Prompts That Work EXCERPT

Why prompts matter more with an agent · The four parts of a strong prompt · Before and after · Point at things · Have Claude interview you · Size the task · Steering mid-task · Thinking and effort

CHAPTER 5

CLAUDE.md: Teaching Claude Your Project

What CLAUDE.md is and when it loads · Where it lives · What to put in it · What to leave out · Keep it short · Habit 2, Teach it once · A complete example: Folio's CLAUDE.md · Keep it healthy

CHAPTER 6

Explore → Plan → Code → Verify

Why jumping straight to code fails · Explore: read before you write · Plan: agree on the approach · Code: let it work · Verify: close the loop · When to skip steps · A worked example: discount codes for Folio

CHAPTER 7

Staying in Control

Reviewing diffs · Interrupt and redirect · Checkpoints and /rewind · Git as your safety net · Managing context and avoiding the "dumb zone" · When to start a fresh session · A daily routine · Keep your own skills sharp

PART III: POWER FEATURES

CHAPTER 8

Slash Commands and Skills

Built-in commands you'll use every day · Custom commands and skills · Write your first skill · Arguments · Skills Claude uses on its own · Share skills with your team · Ten skill ideas

CHAPTER 9

Subagents and Parallel Work

Why subagents · The built-in subagents · Create your own · Delegate research · Run several sessions at once · Parallel work: git worktrees · Coordinate parallel work

CHAPTER 10

Hooks

Instructions versus guarantees · How hooks work · Your first hook: format after every edit · Block risky actions · Feed information back to Claude · Get notified when Claude needs you · Test and debug hooks

CHAPTER 11

MCP and Plugins

What MCP is · Add a server · Servers worth knowing · Share servers with your team · Costs and risks · Plugins

PART IV: REAL-WORLD PLAYBOOKS

CHAPTER 12

Build an App from Zero

The goal · Step 1: Write a one-page spec with Claude · Step 2: Scaffold the project and write CLAUDE.md · Step 3: Plan the build as milestones · Step 4: The landing page · Step 5: Checkout with Stripe in test mode · Step 6: Deliver the guide securely · Step 7: Tests · Step 8: Deploy and launch · What made this work

CHAPTER 13

Taming a Legacy Codebase

What makes legacy code hard · Get oriented before you touch anything · A CLAUDE.md for legacy code · Get the tests running first · Add tests before you change anything · Refactor in small, safe steps · Upgrades and migrations · Dead code and dependency upgrades · Know when to stop

CHAPTER 14

Debugging **EXCERPT**

Give it the evidence · Reproduce first · Isolate the cause · Fix it and keep it fixed · Production bugs · When Claude goes in circles · Special cases

CHAPTER 15

Frontend Work

Give Claude eyes · From design to code · The visual loop · Match your design system · Responsive design and accessibility checks · Visual regression tests with Playwright

CHAPTER 16

Proving It Works: Tests, Review and PRs

Five steps of verification · Step 1: Claude checks its own work · Step 2: Hooks enforce the checks · The hand-off: commits and pull requests · Step 3: CI the agent can't edit · Step 4: A review in a fresh context · Step 5: A human

CHAPTER 17

Playbooks by Industry

The same loop, different checks · Payments and fintech · Data pipelines and analytics · Mobile apps · Infrastructure and DevOps · Healthcare, legal and other regulated work · Education, documentation and content · Summary: checks and guardrails by domain

PART V: SCALING UP

CHAPTER 18

Automation and Teams

Headless mode: Claude Code in scripts · Claude in GitHub Actions · Scheduled and cloud runs · Long unattended runs · Build your own agents with the Agent SDK · Roll it out to a team

CHAPTER 19

Cost, Security and Anti-Patterns **EXCERPT**

Where your tokens go · Seeing what you spend · How to spend less · Measure turns per feature · Security · Common mistakes: the top ten

AFTERWORD

Keep Going

The short version · Your first month · Staying current · About this edition · The four books · One last thing

APPENDICES

APPENDIX A

Cheat Sheet

Slash commands · Keyboard shortcuts · Permission modes · Models and effort · CLI commands and flags · File locations · Permission rules · Hooks · Version notes · Official documentation

APPENDIX B

50 Prompts to Copy and Paste **EXCERPT**

Understanding code · Planning · Building features · Fixing bugs · Testing · Refactoring · Review and security · Git, docs and releases · Frontend

APPENDIX C

CLAUDE.md Templates

How to use these templates · Template 1: Next.js and TypeScript web app · Template 2: Python and FastAPI service · Template 3: Django monolith · Template 4: Go service · Template 5: React Native app (Expo) · Template 6: Data and analytics project (dbt)

APPENDIX D

Ready-Made Hooks, Skills and Subagents

Before you paste · Hooks · Skills · Subagents

APPENDIX E

Troubleshooting and Glossary

Start with these checks · Troubleshooting · Glossary

APPENDIX F

Sources

Chapter 3: Your First Real Task · Chapter 5: CLAUDE.md: Teaching Claude Your Project · Chapter 7: Staying in Control · Chapter 9: Subagents and Parallel Work · Chapter 10: Hooks · Chapter 11: MCP and Plugins · Chapter 12: Build an App from Zero · Chapter 13: Taming a Legacy Codebase · Chapter 14: Debugging · Chapter 15: Frontend Work · Chapter 16: Proving It Works: Tests, Review and PRs · Chapter 17: Playbooks by Industry · Chapter 18: Automation and Teams · Chapter 19: Cost, Security and Anti-Patterns · Ideas and terms

Why Claude Code Is Different

By the end of this chapter you'll know what Claude Code actually does when you give it a task, how to think about working with it, and the four habits the rest of this book builds on.

Three ways to use AI for code

Developers use AI to write software in roughly three ways.

Autocomplete. The AI suggests the next few lines as you type. You stay in control of every keystroke, and the AI sees little beyond the file you're in. It's fast and low-risk, and it's limited to what fits on your screen.

Chat. You describe a problem in a chat window, get code back, and copy it into your editor. Then you run it, paste the error back, and repeat. The AI can reason about bigger problems, but you're the one carrying information back and forth. It usually doesn't see your real project or run your tests, so it rarely finds out whether its code worked.

An agent. You describe the outcome you want, and the AI works in your project directly. It reads the files it needs and searches the codebase. It edits code and runs commands like your test suite. It reads the results and keeps going until the job is done or it needs you. That's what Claude Code is.

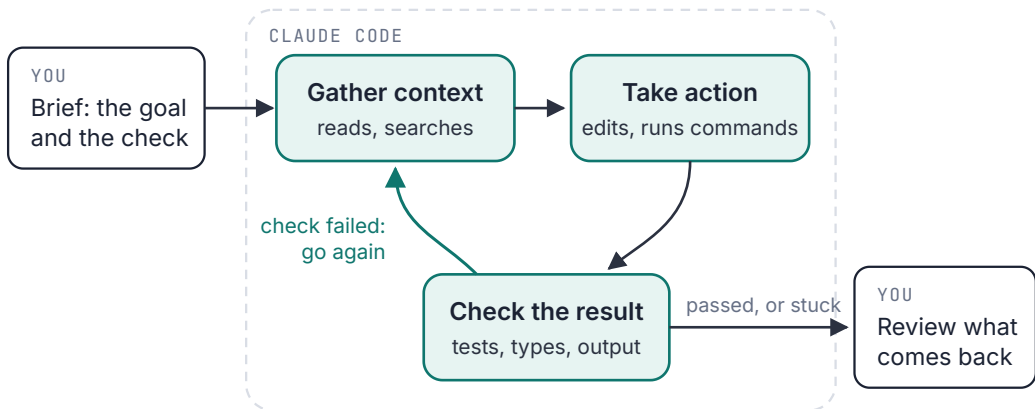
The difference that matters isn't that an agent writes more code. It's that the agent closes the loop. In chat, *you* run the code and report back. With an agent, Claude can run the code itself, see that a test failed, read the failure and fix it, of-

ten several times before you look at anything.

What happens when you give Claude Code a task

When you type a request into Claude Code, it works in a loop:

1. **Gather context.** It reads the files you mentioned and searches for related code. It also checks project files like `package.json` and `CLAUDE.md` to learn how things work.
2. **Take action.** It edits files, creates new ones or runs shell commands. Depending on the permission mode you choose, it either asks you first or has a safety check review each action before it runs. Chapter 2 explains the modes.
3. **Check the result.** It reads what came back: test results, compiler errors, linter warnings, command output.
4. **Repeat or stop.** If the check failed, it goes around again. If the check passed, or it's stuck, it stops and reports back.



The agent loop. You set the goal and the check, Claude goes around until the check passes or it needs you, and you review what comes back.

Here's a representative example. Say you ask:

PROMPT

The `/api/orders` route returns a 500 when the cart is empty. Find out why, fix it, and add a test for the empty-cart case. Run the test suite when you're done.

Claude might do all of this before reporting back:

- Search for the route handler, then read it and the cart helper it calls.
- Notice that the code assumes the cart has at least one item.
- Change the handler to return a 400 with a clear message, and write the new test.
- Run `npm test` and see that a different test now fails because it expected the old behavior.
- Read that test, decide whether to update it, and tell you which way it went.
- Run the suite again to confirm everything passes.

You didn't copy or paste anything. You described the outcome and how to check it, and then you reviewed the result.

NOTE: Claude Code runs in your terminal, in VS Code and JetBrains IDEs, in a desktop app and on the web. The core workflow is the same in all of them. Chapter 2 helps you pick one and points out where they differ.

How to think about it: a capable new colleague

The most useful way to think about Claude Code is as a very capable engineer who joined your team this morning.

This colleague is fast, knows most languages and frameworks well, and doesn't get tired or bored. But they've never seen your codebase, they don't know your team's conventions, and they don't know which parts of the system are fragile.

There's also a twist: every new session is their first morning again. They don't remember yesterday unless someone wrote it down.

That picture explains most of what makes Claude Code go well or badly:

- **They need context.** Give a new colleague a one-line ticket and they'll guess. Give them the relevant files, the pattern to follow and what "done" means, and they'll get it right. The same goes for Claude.
- **They do what you ask, and sometimes a bit more.** Ask them to fix a bug and they'll fix it. They might also "fix" something you didn't want touched, so say what's off-limits.
- **They work better with a way to check.** A colleague who can run the tests catches their own mistakes. One who can't hands you untested code.
- **They don't remember unless you write it down.** That's what `CLAUDE.md` is for. It's a file in your project that Claude reads at the start of every session: the onboarding document you'd write for a new hire. Chapter 5 covers it in detail.
- **You're still the lead.** However good the tool gets, three jobs stay yours: you decide what to build, you check that it's right, and you answer for it. This book gives you tools for all three.

The idea this book keeps coming back to

Give Claude a way to check its work.

Claude can write code it has no way to check, but then you're the only check. Every bug it can't see becomes a bug you have to find, describe and send back. That's slow and expensive, and it's the main reason people give up on AI coding tools.

When Claude *can* check its work, the loop closes. It writes the code, runs the check, sees the problem and fixes it before you ever look. A check can be:

- a test suite (`npm test`, `pytest`)

- a type checker or compiler (`tsc` , `mypy` , `go build`)
- a linter
- a script that calls an API and prints the response
- a screenshot of the page compared with the design
- running the app and following a list of steps

The more of “done” Claude can check on its own, the more you can hand over. The biggest gains rarely come from clever prompts. They come from making the work easy to check.

Automated checks don’t replace you, though. Tests and screenshots catch things that are broken. They don’t catch these:

- a feature that works perfectly but solves the wrong problem
- a security hole nobody wrote a test for
- a change that’s technically fine but bad for the business

That judgment stays with you. That’s why this book keeps telling you to review Claude’s work.

The Four Habits

This book covers a lot of features, but most of the benefit comes from four habits you can start today. Like the idea above and Chapter 6’s explore, plan, code and verify loop, they adapt the practices in Anthropic’s own best-practices guide for Claude Code. Chapters 4, 5 and 7 cover them in depth. Here’s what they are and why they matter.

Habit 1, Brief it

Give Claude a complete brief in a single prompt: where to work, the pattern or limit to respect, and the check that proves the job is done.

A vague prompt makes Claude explore. It reads files it didn't need, guesses at what you meant and often guesses wrong. A good brief puts it to work on the right thing straight away. Compare these two prompts:

PROMPT

Add rate limiting to the API.

PROMPT

Add rate limiting to the routes in `app/api/checkout/`. Follow the pattern in `lib/rate-limit.ts`, which the login route already uses: 10 requests per minute per IP. Don't change any other routes. Add a test that the 11th request in a minute gets a 429, then run `npm test` and `npm run typecheck`.

The second one takes 20 more seconds to write and saves several rounds of corrections. Chapter 4 is all about writing prompts like it.

Habit 2, Teach it once

Correct Claude once, then make the correction a rule in `CLAUDE.md` so later sessions start with it.

Say you tell Claude “we use `date-fns`, not `moment`” and leave it there. It may well make the same mistake in tomorrow's session, because tomorrow it's a new colleague again. Add one line to `CLAUDE.md` and you never have to say it again. Over the weeks these rules add up. Claude stops repeating the mistakes you've already seen, and you stop repeating yourself. Chapter 5 shows you how.

Habit 3, Start fresh

Run `/clear` before you start something unrelated, and `/compact` at a milestone within a long task.

Claude re-reads the whole session on every turn: each file it opened, each command's output, each approach that didn't work. Leftovers from an earlier task make it slower, more expensive and easier to confuse, so `/clear` gives the next task a clean start. `/compact` is for the middle of one long task: once a step is done, it swaps the conversation for a summary and keeps the thread. Chapter 7 covers managing context in detail.

Habit 4, Save points

Commit on a branch whenever the checks pass, so each risky step starts from a known-good state.

Claude can change several files in a single turn, and not every turn goes the way you meant. Claude Code takes a checkpoint before each prompt you send, so `/rewind`, or `Esc Esc` on an empty prompt, can undo a recent step. Checkpoints don't cover every change and they're tied to the session, so the save points that last are git commits. With one behind you, a bad turn costs minutes, not an afternoon. Chapter 7 shows you how.

TIP: If you only remember one thing from this chapter, make it Habit 1. Saying how to check the result does more for quality than any other change you can make to a prompt.

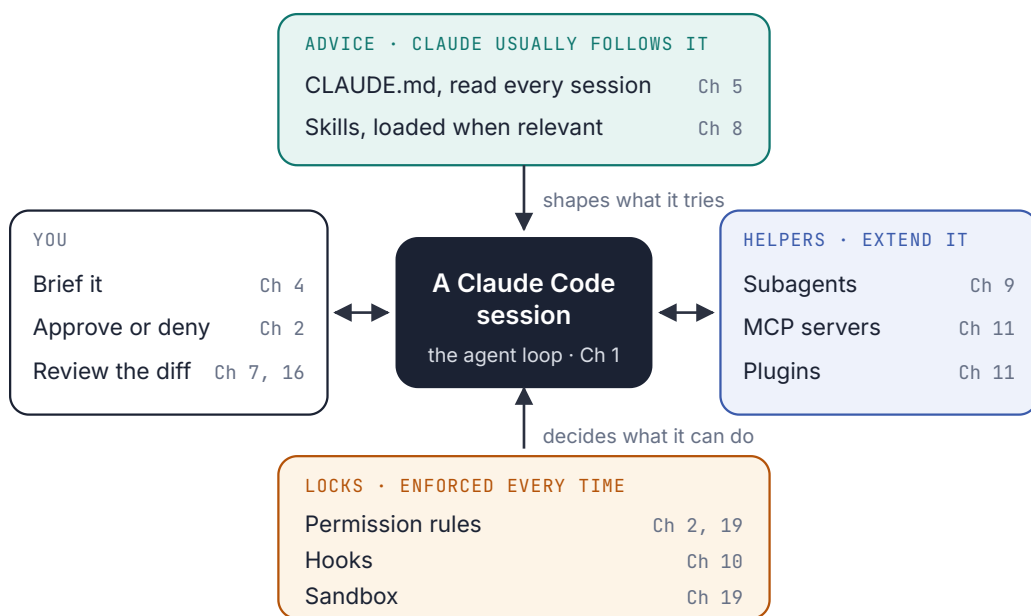
How this book is organized

The book has five parts. You can read it straight through, but each chapter also stands on its own.

- **Part I: Foundations** (Chapters 1–3). Getting set up and doing your first real task.
- **Part II: The Core Workflow** (Chapters 4–7). Prompts, CLAUDE.md, the explore → plan → code → verify loop, and staying in control. If you only read one part, read this one.

- **Part III: Power Features** (Chapters 8–11). Skills, subagents, hooks, MCP and plugins, which let you fit Claude Code to your project and team.
- **Part IV: Real-World Playbooks** (Chapters 12–17). Building an app from zero, legacy code, debugging, frontend work, testing and review, and how the approach changes by industry.
- **Part V: Scaling Up** (Chapters 18–19). Automation, teams, cost and security.

Most of the features in this book sit around one session, and each chapter fills in one part of this picture:



How the pieces fit. Advice shapes what Claude tries, locks decide what it can do, and helpers extend what it can do. You brief it, approve what it asks to do, and review what it hands back.

The appendices are meant to stay open while you work. They hold a cheat sheet, 50 prompts to copy, CLAUDE.md templates, ready-made hooks and skills, a troubleshooting guide, and the sources behind the real incidents and studies in the book.

Two example projects run through the book:

- **Folio** is a small storefront for selling digital downloads, built in Chapter 12 with Next.js, TypeScript and Stripe.
- **Relay** is an eight-year-old Django app that shows how to work with legacy code.

You don't need either one to follow along. Every exercise also works on your own code.

NOTE: The full chapter also covers where Claude Code shines and where it struggles.

Prompts That Work

By the end of this chapter you'll be able to write a prompt that gets a job done in one pass, because it names the files, sets the boundaries and tells Claude how to prove the work is finished. You'll also know how to point Claude at evidence, size a task, steer it mid-task and ask for deeper thinking when a problem is hard.

The four parts of a strong prompt

A strong prompt answers the four questions a new colleague would ask before starting.

Part	The question it answers	Example
Context	Where do I look, and what's the background?	"The receipt email is built in lib/email/receipt.ts."
Goal	What outcome do you want?	"Add the order number to the subject line."
Constraints	What pattern do I follow, and what's off-limits?	"Use formatOrderNumber. Don't change the email body."
Check	How do I prove it's done?	"Update tests/receipt.test.ts and run npm test."

This is Habit 1, Brief it, from Chapter 1. Put the four parts together and you get a prompt like this:

PROMPT

The receipt email is built in `lib/email/receipt.ts`, and order numbers are formatted by `formatOrderNumber` in `lib/orders.ts`. Add the order number to the email's subject line, as "Your Folio receipt (#F-1042)". Use `formatOrderNumber` rather than building the string yourself, and don't change the email body or layout. Update `tests/receipt.test.ts` to cover the new subject, then run `npm test` and `npm run typecheck`.

The check is the part people skip, and it pays off most. Without it, Claude stops when the code *looks* right and leaves the checking to you. With it, Claude runs the command, sees a failure, fixes it and runs it again before you look. The work gets finished faster because Claude does the rounds of fixing instead of waiting for you to report each problem.

A check can be a test suite, a type check, a linter, a script that calls an endpoint or a list of steps to follow in the running app. If there's no way to check, say so, and ask Claude to tell you what it couldn't verify.

Not every prompt needs all four parts. "Fix the typo in the footer in `app/layout.tsx`" is a complete brief. The more files a change touches and the harder it is to undo, the more of the four you should write down.

TIP: Long briefs are easier to write in a real editor. `Ctrl+G` opens the current prompt in your default text editor. For a line break in the prompt box, `Ctrl+J` or `\` followed by `Enter` works in every terminal.

Before and after

Here are six common jobs on Folio. The first prompt in each pair is the one people usually type. The second is a brief.

A bug fix

PROMPT

The download links are broken.

PROMPT

Customers who click the download link in their receipt email see "This link has expired," even minutes after buying. Links are created by `createDownloadLink` in `lib/downloads.ts` and checked in `app/api/downloads/[token]/route.ts`. They should last 24 hours. Write a test in `tests/downloads.test.ts` for a link created 10 minutes ago, confirm it fails, then find and fix the cause. Don't change the token format, because links already sent to customers must keep working. Run `npm test` and `npm run typecheck`.

The second gives the exact symptom, both ends of the code path, a constraint Claude couldn't know (links already sitting in inboxes) and a failing test that proves the fix.

A feature

PROMPT

Add filtering to the admin orders page.

PROMPT

On the admin orders page (`app/admin/orders/page.tsx`), add filters for order status (paid, refunded, failed) and date range. Keep the filters in the URL query string so a filtered view can be bookmarked, the same way `app/admin/products/page.tsx` handles its category filter. Do the filtering in the database query in `lib/orders.ts`, not in the browser. Don't change the Prisma schema. Add tests for the query function in `tests/orders.test.ts`, then run `npm test`, `npm run typecheck` and `npm run lint`.

The second names an existing page to copy, decides where the logic lives, rules out a schema change and says which checks define “done.”

A refactor

PROMPT

Clean up the Stripe code.

PROMPT

`app/api/checkout/route.ts` and `app/api/checkout/bundle/route.ts` both build Stripe Checkout line items, with slightly different code. Move that logic into one function, `buildLineItems`, in `lib/stripe.ts`, and use it in both routes. This is a refactor, so behavior must not change. Don't touch the webhook handler. Run `npm test` before you start and again when you finish, and tell me if any result changed.

“Clean up” has no finish line, while the second describes the end state, says behavior must not change and uses the test suite before and after as the proof.

Tests

PROMPT

Write tests for the pricing code.

PROMPT

Write Vitest tests for `calculateTotal` in `lib/pricing.ts`, in `tests/pricing.test.ts`. Cover an empty cart, one item, three items, a 15% discount on 999 cents (check the rounding), and a discount larger than the subtotal (the total should be 0, never negative). Follow the style of `tests/format.test.ts`, and don't mock anything inside `lib/pricing.ts`. If a test fails, don't change the test or the code. Tell me, because it may be a real bug. Run `npm test`.

The second lists the cases that matter, sets the style and treats a failing test as information rather than something to make go away.

Documentation

PROMPT

Write docs for the project.

PROMPT

Rewrite the "Local setup" section of `README.md` so a new developer can get from git clone to a test purchase. Take the steps from `package.json`, `.env.example` and the `prisma/` folder, not from memory. Cover installing dependencies, the environment variables (names only, plus where to get test values), creating the local database, running the dev server and paying with a Stripe test card. Keep it under 40 lines. Then check every command in the section against `package.json` and list any step you couldn't verify.

The second sets the reader, the sources of truth, the scope and the length, and it makes Claude check its steps instead of writing plausible ones.

A performance problem

PROMPT

The admin dashboard is slow. Speed it up.

PROMPT

app/admin/page.tsx takes several seconds to load locally once there are a few thousand test orders. I suspect lib/orders.ts loads each order's product in a separate query. Before changing anything, log the queries Prisma runs for one page load and time the page. Report the query count and the timing. Then fix the biggest cause without changing what the page shows, and measure again so I can compare.

The second shares your suspicion, asks for a measurement before any change and makes the before-and-after numbers the check. Chapter 14 covers performance debugging in more depth.

None of the “after” prompts is clever. Each one just says what a colleague would need to know, which takes a minute to write and usually gets the job done in one pass.

NOTE: The full chapter goes on to pointing Claude at evidence, letting it interview you, sizing tasks, steering mid-task and choosing an effort level.

Debugging

When Claude goes in circles

Learn to spot a debugging session that has stopped making progress:

- A fix fails, and Claude tries a small variation of it.
- The diff keeps growing, but the reproduction still fails.
- It says “let me try a different approach” several times in a row.
- It changes the test, or wraps code in try/catch blocks that swallow the error.
- It reverts to a fix it already tried.

When you see two of these, stop the session with `Esc`. More turns rarely help, because the context is full of failed attempts and Claude keeps anchoring on them. Instead:

1. **Ask for a summary.** “Summarize what we know for certain, what you’ve tried, and what each attempt ruled out.” Writing it down often exposes the wrong assumption.
2. **Add what you know.** Often you have a hint: “the bug only happens for orders with a discount code,” or “this worked before we upgraded Prisma.”
3. **Start fresh with a better brief.** Run `/clear` and write a new prompt with the evidence, the reproduction and what’s been ruled out. A clean context with a good brief often solves in minutes what the tangled session couldn’t.
4. **Change the approach.** Ask for a plan first (Chapter 6), use a more capable model or a higher effort level for the hard part, or have a subagent review the problem with fresh context (Chapter 9).
5. **Take the wheel.** Ten minutes in a debugger yourself can find the clue that gets Claude back on track.

PROMPT

Stop changing code. Summarize: what we know for certain about this bug, every fix you've tried and why each failed, and what assumptions you're making that we haven't verified. Then suggest one experiment that would test the weakest assumption.

TIP: Never accept a fix that edits the reproduction test to make it pass, unless you agree the test itself was wrong. Better, make it impossible: start the session with `claude --disallowedTools "Edit(./tests/orders.test.ts)"`, as Chapter 13 does for code under characterization tests.

CHAPTER 19 · EXCERPT

Cost, Security and Anti-Patterns

Common mistakes: the top ten

These ten mistakes cause most of the frustration, wasted tokens and risk with Claude Code. Each has a fix you've seen in this book.

1. **Vague prompts.** “Fix checkout” makes Claude explore and guess. → Name the files, the constraint and the check (Habit 1, Chapter 4).
2. **No way for Claude to check its work.** You become the test suite. → Add tests, type checks, scripts or screenshots, and put the command in the prompt (Chapters 1 and 6).
3. **Correcting the same mistake every day.** → Turn the correction into a `CLAUDE.md` rule the first time (Habit 2, Chapter 5).
4. **One endless session.** Stale context makes Claude slower, costlier and more confused. → `/clear` before unrelated work, and `/compact` at a milestone

within a long task (Habit 3, Chapter 7).

5. **Skipping the plan on big changes.** Claude builds the wrong thing, efficiently. → Use plan mode for anything that touches several files (Chapter 6).
6. **Not reading the diff.** Passing tests don't prove the change is right, especially if the tests changed. → Review every diff, and watch test files closely (Chapters 7 and 16).
7. **A bloated CLAUDE.md.** Important rules get lost in 600 lines. → Keep it under about 200 lines, and move workflows into skills and path-scoped rules (Chapters 5 and 8).
8. **Parallel tasks in one folder.** Two sessions edit the same files and break each other's work. → One worktree per independent task (Chapter 9).
9. **Over-broad permissions.** Bash(*) allow rules, or bypass mode on your laptop. → Specific allow rules, deny rules for secrets, sandboxing and containers (this chapter).
10. **Trusting what Claude reads, and what it can reach.** Untrusted content plus production credentials is how agents cause real damage. → Test-only credentials, tight permissions on untrusted input, and human review before anything irreversible (this chapter and Chapter 17).

APPENDIX B · EXCERPT

50 Prompts to Copy and Paste

These prompts are starting points for the jobs you'll do most often. Most follow Habit 1, Brief it: they name what to look at, say what to leave alone and ask for a check. Replace anything in `<angle brackets>` and adjust the file paths to your project.

Most examples use paths from Folio, the storefront built in Chapter 12. Swap in your own.

Understanding code

1. Take a project tour

Use when you're new to a codebase (Chapter 3).

PROMPT

Give me a tour of this project: what it does, how it's structured, the main entry points, and the commands to install, run, test and lint it. Point out anything unusual about the setup. Don't change any files.

2. Trace a feature end to end

Use when you need to know how a flow works (Chapter 6).

PROMPT

Trace what happens when a customer clicks "Buy now," from the button in `components/ui/BuyButton.tsx` to the order being saved. List each file and function in order, with one line on what each does. Don't change any files.

3. Explain a confusing file

Use when a file makes no sense yet.

PROMPT

Explain `lib/pricing.ts` as if I'm new to the codebase. What problem does it solve, what are the inputs and outputs of each exported function, and which parts are surprising or fragile? Don't change any files.

4. Find everything that depends on something

Use when you're about to change a shared type.

PROMPT

I want to change the shape of the Order type in lib/types.ts. Find every file that reads or writes it, group them by how they use it, and tell me which ones would break if I renamed totalCents to amountCents. Don't change any files.

5. Find out why code is the way it is

Use when code looks odd but may be deliberate (Chapter 13).

PROMPT

Use git log and git blame to explain why app/api/webhooks/stripe/route.ts checks event.created against a 5-minute window. Find the commit that added it and summarize the reason from the commit message and the surrounding changes.

6. Map the data model

Use when your task touches the database.

PROMPT

Read prisma/schema.prisma and summarize the data model: each model, its key fields, and how the models relate. Then list any relations or indexes that look missing for the queries in lib/orders.ts. Don't change any files.

7. *Learn the house conventions*

Use when you're writing CLAUDE.md (Chapter 5).

PROMPT

Look at how this codebase handles errors, logging, validation and configuration. For each, describe the pattern with one example file, and note any places that break the pattern. I'll use this to update CLAUDE.md.

8. *Write an architecture overview*

Use when new teammates, or new sessions, need a map (Chapter 13).

PROMPT

Write ARCHITECTURE.md: a one-page overview of how this project is organized, how a request flows through it, where data is stored, and the three things a new developer is most likely to get wrong. Keep it under 80 lines and link to files rather than copying code.

NOTE: That's 8 of the 50 prompts. The full appendix adds prompts for planning, building features, fixing bugs, testing, refactoring, review and security, git and releases, and frontend work.

THE FULL BOOK

Get the Full Book

This sample is a small slice of the book. The full edition has 19 chapters and six appendices, from your first session to team-wide automation.

What you get

- The complete book as a PDF, with clickable bookmarks, that reads well on a laptop, tablet or phone
- All 19 chapters: CLAUDE.md, the explore → plan → code → verify loop, staying in control of long sessions, skills, subagents, hooks, MCP, six real-world playbooks and team automation
- Six appendices: the cheat sheet, 50 prompts to copy, six CLAUDE.md templates, ready-made hooks, skills and subagents, a troubleshooting guide, and the sources behind every real incident in the book
- Companion files: every hook, skill, subagent, rules file, template and script from the book, plus the example plugin and Folio's settings, ready to copy into your project
- A 30-day refund if the book isn't for you

How to buy

1. Go to aiagentsforhumans.com/shipping-with-claude-code and choose Get the book.

2. Pay through the secure checkout, run by Dodo Payments, which handles payment and any sales tax, VAT or GST.
3. Your download link arrives by email straight away, and you can download the book again any time from the Dodo Payments customer portal.

About the author

Pranav S. is an independent author. *Shipping with Claude Code* is a practical guide to getting reliable, repeatable results from Claude Code, written for developers and technical founders who want prompts, settings and workflows they can use the same day.

Its commands, flags and settings were checked against Anthropic's official documentation as of October 2026.

Shipping with Claude Code: Free Sample. A free sample of *Shipping with Claude Code: The Practical Playbook*, first edition, written for Claude Code as of October 2026. © 2026 Pranav S. You're welcome to share this sample with anyone, as long as it stays unchanged and free.

This is an independent, unofficial guide. It is not affiliated with, sponsored by or endorsed by Anthropic. Claude and Claude Code are trademarks of Anthropic, PBC. Claude Code changes often; where anything here differs from the official documentation, the documentation is the source of truth.