PROMPTS & SKILLS FOR CLAUDE CODE

Twelve skills that turn your most-repeated instructions into permanent project setup.

Terminal Craft

INDEPENDENT FIELD GUIDE

The Daily Driver Skills

Twelve skills that turn your most-repeated instructions into permanent project setup.

TERMINAL CRAFT

WELCOME

Before You Begin

Everyone who works with a coding agent builds a private list of things they always say. Match the existing style. Don't refactor what I didn't ask about. Show me the plan first. You type them so often they stop feeling like instructions and start feeling like punctuation, and then, on a tired Thursday evening, you forget one. That is reliably the session that produces the surprising diff. Anything you have typed more than five times is not a prompt any more. It is a project standard that happens to live in your head, and it belongs in a file where it applies whether you remembered it or not. This guide turns twelve of those into permanent setup. Each arrives as a complete definition: what it does, the situation that should trigger it, where the file goes, and how to check it is firing rather than sitting on disk being politely ignored. There is also a chapter on the discipline nobody writes about: keeping the set small. A bloated collection slows every session and makes it impossible to tell which instruction produced which behaviour, which is how people end up with twenty files and no idea why the agent is doing something odd. Two notes before we start. This is an independent publication with no affiliation to Anthropic or any other AI vendor, and no software is included. And it is

the 2026 Edition: file locations and selection behaviour move between versions, so chapter six covers how to tell when something has changed and what to do about it rather than pretending the details are permanent.

— The Terminal Craft Team

HOW TO USE THIS GUIDE

Getting the Most From This Book

This is a reference, not a reading list. The prompts work best adopted a few at a time until they are automatic. Here's how to get the most from the pages ahead.

1. **Use it as a reference.**
   Every chapter ends with a *Try This* exercise and a set of *Key Takeaways*. Read the chapter once, then come back to the prompts when you need them.

2. **Adopt a few at a time.**
   Take four prompts and use them until they are automatic. Reaching for forty at once is the fastest way to use none of them.

3. **Read the 'not for' line.**
   A structured prompt on a trivial task is pure overhead. Knowing when not to use one matters as much as knowing the wording.

4 **Adapt, don't discard.**

When a model version changes and a prompt stops landing, rebuild it from the four-part frame rather than hunting for a new download.

Whenever you're ready, turn the page and let's begin.

CONTENTS

What's Inside

CHAPTER 1

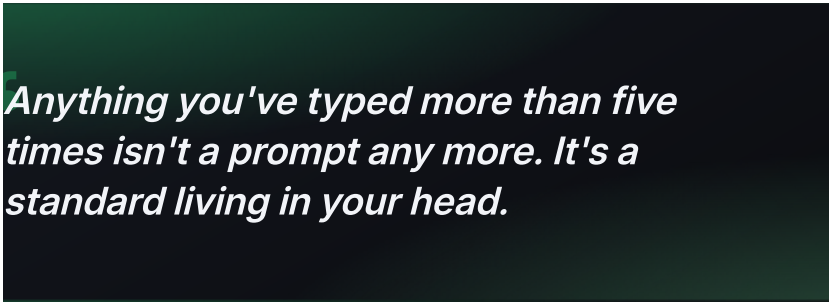
Prompt, Skill, Or Neither

> Before writing anything permanent, a filter. Most instructions people want to make permanent should stay in a prompt, and a few should not exist at all. Getting this wrong is how a helpful setup becomes a slow, contradictory one.

The five-times rule

If you have typed an instruction five times, it is a candidate. If you have typed it five times and been annoyed each time you forgot it, it is not a candidate. It is overdue. That is the entire test, and it is deliberately mechanical because the alternative is guessing at what feels important.

Things that pass this test are almost always standards rather than tasks: how code should look, when to stop and ask, what must never happen without permission. Things that fail it are usually one-off context that belonged in the prompt where you first typed it.



What should stay a prompt

Anything specific to today's task. The files involved, the bug you are chasing, the constraint that applies to this change and no other. Making those permanent is how a setup starts contradicting itself two weeks later, when the constraint no longer applies and nobody remembers writing it.

A good check: if you would be surprised to see this instruction applied to an unrelated task next month, it is a prompt, not a skill.

What should not exist at all

Instructions that restate the obvious take up space, dilute everything around them and change nothing. Write good code. Be careful. Follow best practices. All of it is noise. Politeness instructions are the same. If you cannot describe the specific behaviour that changes when the instruction fires, it does not belong anywhere.

Be ruthless here. Every line you add is a line loaded into every session, competing with the lines that actually matter.

DECIDING WHAT DESERVES TO BE PERMANENT

The Five-Times Audit

Read the last twenty things I asked you in this project. List any instruction I repeated three or more times, exactly as I phrased it. Rank by how often it appeared.

WHY IT WORKS

It finds the repeated instructions from evidence instead of memory, and people are consistently wrong about which ones they repeat most.

USE WHEN Before writing your first skill **PAIRS WITH** The Skill Draft

A brand new project with no history to read.

TURNING A REPEATED INSTRUCTION INTO A DEFINITION

The Skill Draft

Turn this instruction into a skill definition: a description naming the SITUATION it applies to (not the capability), the scope it covers, and the instructions themselves. Keep it under 15 lines.

WHY IT WORKS

The description decides whether the skill is ever selected, and drafting it separately stops it becoming an afterthought.

USE WHEN Each instruction that passed the audit **PAIRS WITH** The Five-Times Audit

Task-specific context that should stay in a prompt.

Try This: Find Your Repeats

Evidence beats intuition about what you actually type.

1. Run the Five-Times Audit on your current project today.
2. Write down the three instructions that came back most often.
3. For each, ask: would I want this applied to an unrelated task next month?
4. Only the ones you answer yes to become skills.

KEY TAKEAWAYS

- Typed it five times and been annoyed when you forgot? It's overdue, not a candidate.
- Task-specific context stays in the prompt. Standards become skills.
- If you can't name the behaviour that changes, the instruction belongs nowhere.

CHAPTER 2

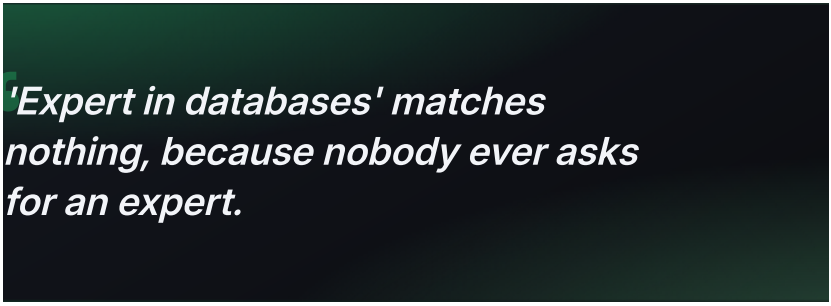
The Anatomy Of A Skill

> A skill is four things, and only one of them decides whether it ever runs. Most people spend their effort on the part that matters least.

The description is the whole game

A skill is selected by matching its description against the situation at hand. That means the description is not a summary for humans. It is the matching surface. 'Use when reviewing a database migration' matches a real moment. 'Expert in database best practices' matches nothing, because nobody ever asks for an expert.

Write it as the situation, in the words you would actually use when you are in it. The instructions inside can be as long as they need to be; the description has one job and it is not describing your expertise.



Scope keeps a growing set predictable

Every skill should name what it does not cover. Without that, the fourth skill you write starts firing on requests the second one was handling, and you get behaviour that changes depending on which matched first, which looks exactly like the tool being unreliable.

A sentence is enough: 'This covers migrations only, not schema design or query performance.' It costs nothing now and saves an afternoon of confused debugging later.

Instructions should be checkable

Write instructions you could verify in a diff. 'Prefer clarity' cannot be checked. 'Never introduce a new dependency without asking first' can. The second changes behaviour; the first changes nothing except the length of the file.

Where a rule has exceptions, name them. An instruction with an unstated exception gets ignored the first time the exception occurs, and once ignored it tends to stay ignored.

FIXING A DESCRIPTION NOBODY MATCHES

The Situation Rewrite

This description names a capability. Rewrite it as the situation it applies to, in the words someone would use while in that situation. Give me three options.

WHY IT WORKS

Selection matches situations, not credentials, and three options make the best phrasing obvious by comparison.

USE WHEN Every description you write

PAIRS WITH The Scope Sentence

Descriptions that already name a concrete moment.

PREVENTING COLLISIONS BEFORE THEY HAPPEN

The Scope Sentence

Write one sentence saying what this skill does NOT cover, specific enough that a neighbouring skill could not claim the same territory.

WHY IT WORKS

Unstated scope is why two skills fight, and the behaviour that results looks like unreliability rather than a configuration problem.

USE WHEN Every skill, from the second one onward

PAIRS WITH The Situation Rewrite

A project with exactly one skill.

REMOVING INSTRUCTIONS THAT DO NOTHING

The Checkability Pass

For each line in this skill: could I verify it was followed by reading a diff? Mark yes or no. Rewrite or delete every no.

WHY IT WORKS

Unverifiable instructions consume context in every session and change no behaviour at all.

USE WHEN Before installing any skill **PAIRS WITH** The Scope Sentence

Skills whose whole purpose is tone rather than output.

Try This: Rewrite One Description

Take the skill you most suspect is being ignored.

1. Read its description and ask whether it names a situation or a capability.
2. Run the Situation Rewrite and pick the option closest to how you actually talk.
3. Add a scope sentence saying what it does not cover.
4. Run the Checkability Pass over the instructions and delete every unverifiable line.

KEY TAKEAWAYS

- The description is a matching surface, not a summary. Situations, not credentials.
- Name what a skill does not cover, from the second skill onward.
- If you can't verify an instruction from a diff, it changes nothing.

CHAPTER 3

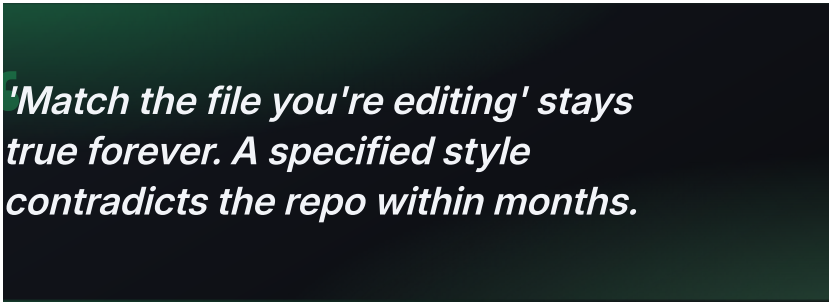
The Convention Skills

> The first four skills to install are the ones that stop you writing the same review comment forever. These are pure standards: how code should look and behave in this repository, applied without anyone having to remember them.

Style is the cheapest win

Matching the surrounding code is the single most common source of otherwise avoidable review friction: naming, comment density, error handling, file layout. It is also trivially expressible as a standard, and it applies to essentially every change.

The key is to say match what is there rather than to specify a style. Specified styles go stale and contradict the repository within months; 'match the file you are editing' stays true forever.



Boundaries stop the helpful extras

The second convention skill is the constraint line made permanent: no refactoring what was not asked about, no new dependencies, no reformatting, and tell me instead of doing it. This is the standard that most reduces surprising diffs, and it works far better as a skill than as something you remember to type.

Include the escape hatch. An agent that notices something wrong nearby should say so. The rule is about acting without permission, not about staying silent.

Comments and naming, decided once

Two small standards worth making permanent: comments explain why rather than restating what, and identifiers are named for a reader who arrives in six months. Both are things people ask for repeatedly in review and almost never write down.

They are also easy to verify in a diff, which makes them good skills by the test from the previous chapter.

MATCHING THE REPOSITORY AUTOMATICALLY

House Style

Match the surrounding code: same naming, same comment density, same error handling, same file layout. Never introduce a new pattern without saying so first and waiting for a reply.

WHY IT WORKS

Matching what exists stays true as the codebase evolves, where a specified style goes stale and starts contradicting the repo.

USE WHEN Every session **PAIRS WITH** Boundary Guard

Repositories with no consistent style to match yet.

STOPPING UNREQUESTED IMPROVEMENTS

Boundary Guard

Do not refactor, rename, reformat or improve anything outside the stated task. Do not add dependencies. If something nearby is wrong, describe it to me instead of changing it.

WHY IT WORKS

It is the constraint line made permanent, and it removes the single largest category of surprising diff.

USE WHEN Every session **PAIRS WITH** House Style

Sessions where a broad cleanup is the actual request.

COMMENTS THAT SURVIVE

Comment Discipline

Comments explain WHY, never restate WHAT the code does. If a comment would just narrate the line below it, delete it and make the code clearer instead.

WHY IT WORKS

Restating comments rot immediately and train readers to skip comments generally, which costs you the useful ones.

USE WHEN Every session **PAIRS WITH** Naming Standard

Public API documentation, which is a different job.

IDENTIFIERS A STRANGER CAN READ

Naming Standard

Name things for someone reading this in six months with no context. No abbreviations that are not already used in this repo. Flag any existing name you think is misleading rather than renaming it.

WHY IT WORKS

Naming is cheapest to fix at write time and most expensive later, and the flag-don't-rename rule keeps it inside the boundary.

USE WHEN Every session **PAIRS WITH** Comment Discipline

Throwaway scripts nobody will read again.

Try This: Install Two, Not Four

Adding four standards at once makes it impossible to see what each did.

1. Install House Style and Boundary Guard today. Leave the other two.
2. Run three ordinary tasks and read the diffs properly.
3. Note whether the files-touched count dropped and whether review comments changed.
4. Only then add Comment Discipline and Naming Standard.

KEY TAKEAWAYS

- Say 'match what is there' rather than specifying a style that will go stale.
- Boundary Guard removes the largest category of surprising diff.
- Install two at a time so you can see what each one actually changed.

CHAPTER 4

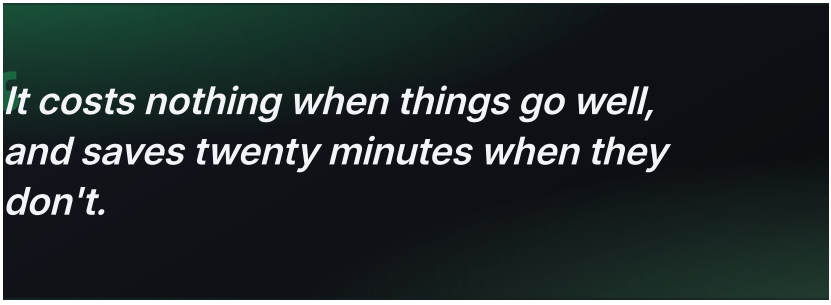
The Workflow Skills

> The second group is about sequence rather than style: when to stop, when to ask, what has to exist before code does. These change how a session runs, which makes them the ones you notice most.

The plan gate, made permanent

Asking for a plan before edits is the highest-leverage prompt in the whole system, which makes it an obvious skill. As a standard it needs a threshold, because not every change needs ceremony. Three files is a reasonable line for most projects.

Below the threshold it stays out of the way. Above it, you get a plan you can reject for nothing, which is the entire point.



Tests as a companion, not an afterthought

A standard worth having: when behaviour changes, name the test that would have caught the old bug, and write it before the fix. This produces better fixes because it forces the behaviour to be stated in observable terms first.

It also catches the specific failure where a fix addresses a symptom nobody can describe. If no test can be written, that is usually a sign the problem is not yet understood.

Commit discipline and the stopping rule

Two smaller workflow standards. First, commits are not made without being asked. A surprising number of frustrations come from work being committed before it was reviewed. Second, an explicit stopping rule: after two failed attempts at the same approach, stop and propose alternatives rather than trying a third variation.

That second one is the loop breaker from the recovery pack, installed permanently. It costs nothing when things go well and saves twenty minutes when they do not.

ANYTHING ABOVE THE THRESHOLD

Plan Gate

For changes touching more than three files: produce a plan with the file list, what each change does, how it is tested, and the likeliest failure. Wait for approval before editing.

WHY IT WORKS

A plan is free to reject; a finished diff is not. The threshold keeps small work fast.

USE WHEN Multi-file work **PAIRS WITH** Test Companion

Projects where nearly every task spans many files by nature.

BEHAVIOUR CHANGES AND BUG FIXES

Test Companion

Whenever you change behaviour, name the test that would have caught the old bug. If it does not exist, write it and show it failing before you write the fix.

WHY IT WORKS

It forces the behaviour into observable terms, which is exactly the ambiguity most bad fixes are made of.

USE WHEN Every bug fix **PAIRS WITH** Plan Gate

Codebases with no test infrastructure at all.

KEEPING THE HISTORY YOURS

Commit Discipline

Never commit, amend, push or create a branch unless I explicitly ask. Stage nothing. When work is ready, tell me and stop.

WHY IT WORKS

Work committed before review is work you have to undo publicly, and the reflex to commit when finished is strong.

USE WHEN Every session **PAIRS WITH** Plan Gate

Automated pipelines where committing is the whole task.

LOOPS, MADE IMPOSSIBLE

The Stopping Rule

After two failed attempts at the same approach, stop. Do not try a third variation. List three different hypotheses for the root cause and wait.

WHY IT WORKS

The third attempt is usually the second one with different names, and the rule is much more reliable installed than remembered.

USE WHEN Every session **PAIRS WITH** Test Companion

Exploratory work where repeated attempts are the method.

Try This: Set Your Threshold

The plan gate only helps if it fires at the right moment.

1. Look at your last ten tasks and count files touched in each.
2. Pick the threshold that would have caught the two you most wish you had reviewed.
3. Install Plan Gate with that number rather than the default three.
4. Adjust once after a fortnight, based on how often it fired unnecessarily.

KEY TAKEAWAYS

- The plan gate needs a threshold, or ceremony swallows the small tasks.
- Write the test that would have caught the bug before writing the fix.
- The stopping rule is the loop breaker, installed instead of remembered.

CHAPTER 5

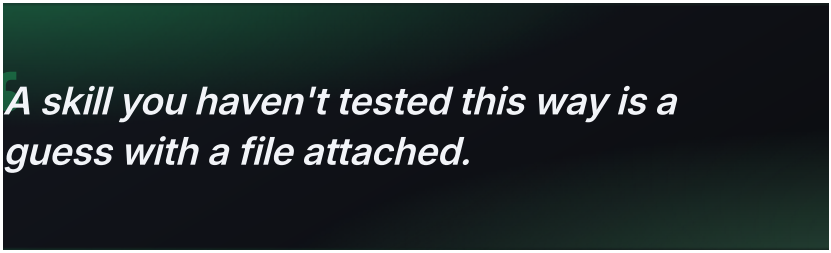
Installing And Verifying

> A skill on disk is not a skill in effect. This chapter is about the gap between the two, which is where almost all the disappointment in this area lives.

Where the file goes

Skills live in project configuration, and the exact path depends on your tooling and its version. The stable idea is that project-level configuration applies to everyone working in that repository, while user-level configuration applies to you across all projects, and you want standards at the project level, so a colleague gets them too.

Check your tool's current documentation for the exact location rather than trusting any book, including this one. That is the kind of detail that moves between versions.



Proving it fires

The verification is simple and almost nobody does it: write three requests that must trigger the skill and three that must not, then run all six. If the three that should trigger it do not, the description is the problem. If the three that should not do trigger it, the description is too broad.

Do this before you rely on a skill in real work. A skill you have not tested this way is a guess with a file attached.

Watching for silent failure

The characteristic failure mode is silence. Nothing errors, nothing warns, the agent simply behaves as though the file does not exist, because as far as selection is concerned, it does not.

So build a habit of noticing. If a standard you installed stops showing up in diffs, treat that as a signal rather than an annoyance: re-run the six-request test and look at the description first, because that is where the fault almost always is.

PROVING A SKILL WORKS BEFORE YOU TRUST IT

The Fires-Or-Not Test

Write three requests that must trigger this skill and three that must not. I will run all six. Predict the result for each before we start.

WHY IT WORKS

Prediction before running is what turns the test into information rather than a demonstration.

USE WHEN Before relying on any skill

PAIRS WITH The Silent-Failure Check

Skills you have already validated and not changed since.

A STANDARD THAT QUIETLY STOPPED APPLYING

The Silent-Failure Check

In the change you just made, which of my project standards applied and which did not? Name each and quote the line of the diff that shows it.

WHY IT WORKS

It converts an invisible failure into a visible one, and the answer points straight at the description that stopped matching.

USE WHEN When a standard seems to have lapsed

PAIRS WITH The Fires-Or-Not Test

Sessions where no standards were relevant anyway.

AFTER ADDING ANYTHING PERMANENT

The Install Review

List every project-level instruction currently in effect, in the order they load. Flag any two that could apply to the same request.

WHY IT WORKS

It is the only reliable way to see the whole loaded set, and overlaps are invisible until they produce contradictory behaviour.

USE WHEN After each new skill **PAIRS WITH** The Silent-Failure Check

The very first skill, where there is nothing to overlap with.

Try This: Six Requests

Fifteen minutes that tell you whether your setup is real.

1. Pick your most important skill.
2. Write three requests that must trigger it and three that must not.
3. Predict each result, then run all six.
4. Every surprise is a description problem. Fix it there, not in the instructions.

KEY TAKEAWAYS

- Standards belong at project level so colleagues get them too.
- Three must-trigger and three must-not requests, predicted then run.
- Silent failure is the norm, and a lapsed standard is a description problem.

CHAPTER 6

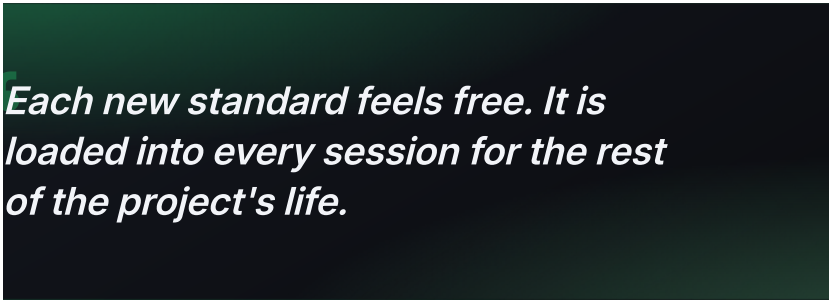
Keeping The Set Small

> The final discipline, and the one that decides whether this still works in six months. Every skill you add is loaded into every session, competing for attention with the ones that matter.

Twelve, not fifty

There is a strong pull toward more. Each new standard feels free, and each one is reasonable in isolation. But a large set slows every session, makes selection less predictable, and makes it impossible to attribute a behaviour to a cause.

Twelve is not a magic number; it is a budget. When you want a thirteenth, the honest question is which of the twelve it replaces, and that question is much more useful than it sounds.



Pruning on evidence

Every quarter, go through the set and ask for each: when did this last visibly change a diff? Anything you cannot answer for is a candidate for removal. Do not remove it on suspicion. Remove it, work for a fortnight, and see whether anything degrades.

The ones that turn out to matter announce themselves quickly. The ones that do not were costing you context in every session for nothing.

What changes between versions

Three things move with tool versions: where files live, how selection is decided, and how much of a long instruction is loaded at once. Prose and standards survive; paths and mechanisms do not, which is why this is the 2026 Edition and says so.

When something stops working after an update, check in that order: location first, then whether the description still matches, then whether the instructions are being truncated. It is almost always the first two, and re-reading a book is rarely the fix.

KEEPING THE BUDGET HONEST

The Quarterly Prune

For each project standard currently installed, tell me the most recent change where it visibly affected the output, and quote that line. Mark any you cannot evidence.

WHY IT WORKS

Evidence beats opinion about which standards earn their place, and unevidenced ones are pure context cost.

USE WHEN Every quarter **PAIRS WITH** The Replacement Question

Sets younger than a month, where there is nothing to judge yet.

WANTING TO ADD A THIRTEENTH

The Replacement Question

I want to add this standard. Which of my existing ones does it overlap with, and which one would you remove to make room? Argue for the removal.

WHY IT WORKS

Forcing a trade keeps the set at a size where behaviour stays attributable.

USE WHEN Every addition after the budget is full **PAIRS WITH** The Quarterly Prune

The first few skills, where there is room.

AFTER A TOOL VERSION CHANGES

The Post-Update Check

Something stopped applying after an update. Check in this order: is the file still in the right location, does the description still match a real request, is the instruction being truncated? Report which.

WHY IT WORKS

It checks the three things that actually move between versions, in the order they most often break.

USE WHEN After any tool update that changes behaviour

PAIRS WITH The Silent-Failure Check

Failures that predate the update.

Try This: Set the Budget

A budget you wrote down is a budget you will keep.

1. Count your current standards. Write the number down.
2. Decide your ceiling. Twelve is a reasonable default.
3. Put a quarterly reminder in your calendar titled 'prune standards'.
4. The next time you want to add one over the ceiling, run the Replacement Question first.

KEY TAKEAWAYS

- Twelve is a budget, not a magic number, so a thirteenth should replace something.
- Prune on evidence: which diff did this visibly change?
- After an update check location, then description, then truncation. In that order.

PUT IT INTO PRACTICE

Your Action Plan

From a list in your head to a working setup, in the order that keeps every change attributable.

- [ ] Run the Five-Times Audit and write down the three instructions you repeat most.
- [ ] For each, ask whether you'd want it applied to an unrelated task next month.
- [ ] Draft the first skill with the Skill Draft prompt, keeping it under fifteen lines.
- [ ] Rewrite its description as a situation, not a capability.
- [ ] Add a scope sentence saying what it does not cover.
- [ ] Run the Checkability Pass and delete every unverifiable line.
- [ ] Install House Style and Boundary Guard first. Just those two.
- [ ] Run three ordinary tasks and read the diffs before adding anything else.
- [ ] Run the six-request Fires-Or-Not Test on each skill before relying on it.
- [ ] Set your plan-gate threshold from your own last ten tasks, not the default.

- [ ] Write your budget down, and put a quarterly prune reminder in the calendar.
- [ ] After any tool update, check location, then description, then truncation.

FINAL WORDS

Where You Go From Here

The point of this book is not twelve files. It is that the standards you care about should apply on the evenings you are too tired to mention them, because those are the evenings they matter most. Start with two. Install House Style and Boundary Guard, run a few ordinary tasks, and read the diffs properly. You should see the files-touched count drop within the week. Add the rest one at a time, test each with six requests before you trust it, and keep the whole set small enough that you can still say which instruction produced which behaviour. When something stops applying, resist the urge to rewrite the instructions. It is nearly always the description, and nearly always because it drifted back toward naming a capability instead of a situation. And when the tooling changes underneath you, and it will, check the three things in chapter six rather than starting over. The standards you wrote are still right. Only the plumbing moved.

Breathe. Begin. Return.
