PROMPTS & SKILLS FOR CLAUDE CODE

How to write skills that get selected, stay scoped and survive a version change.

Terminal Craft

INDEPENDENT FIELD GUIDE

The Skill Author's Playbook

How to write skills that get selected, stay scoped and survive a version change.

TERMINAL CRAFT

WELCOME

Before You Begin

You wrote a skill. It is well organised, the instructions are sensible, and the agent ignores it completely. This is the most common experience in this whole area, and it almost never means the instructions were wrong. The skill was simply never selected, because its description named a capability rather than a situation, and nothing in a real request ever matched it. Selection is the entire game. A brilliant skill with a vague description is invisible; a modest skill with a precise one runs every day and quietly improves your work. Almost everything else people worry about when writing skills is secondary to that one fact, and it is the chapter most authors skip. This playbook is twenty-four worked patterns for the craft: how to write a description that matches the moment, how to scope a skill so the fifth one you write does not break the second, how to split long instructions so only the relevant part loads, and a six-request test procedure that proves a skill fires when it should and stays silent when it should not. Each pattern comes with the anti-pattern it replaces, because seeing the wrong version next to the right one teaches faster than either alone. Two notes. This is an independent publication with no affiliation to any AI vendor, and no software is

included. And it is the 2026 Edition: file locations and selection mechanics move between versions, so the final chapter is about telling the difference between a skill that broke and a mechanism that changed.

— 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

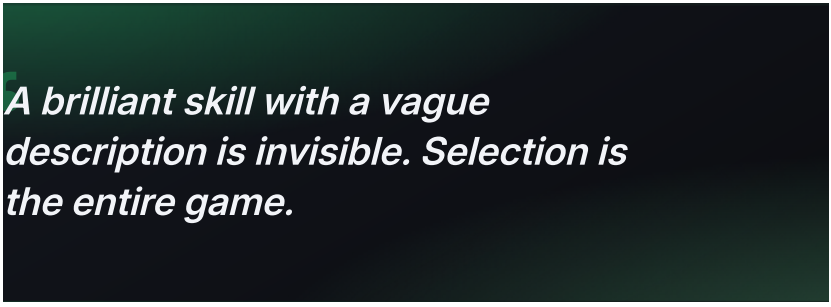
Why Skills Get Ignored

> Before any craft, the diagnosis. There are exactly three reasons a skill does nothing, and they are not equally likely. The first accounts for most of them.

It was never selected

By far the most common cause. Selection happens by matching the description against the situation, and a description that names your expertise matches nothing, because requests describe problems rather than asking for specialists.

The tell is that nothing changes at all. No partial application, no odd behaviour, just the same output you would have got with no skill installed. Total silence means selection, not instructions.



It was selected and lost an argument

Less common but distinctive: the skill fired and something else overrode it. Usually a direct instruction in the conversation, occasionally another skill with overlapping scope. The signature here is partial application: some of the instructions visibly took effect and others did not.

This is a scope problem, and chapter three is about it. It is rarer than people assume, because most authors conclude 'conflict' when the real answer was silence.

It was selected and could not be followed

The rarest case: instructions that are unverifiable, contradictory or impossible in the current context. 'Prefer elegance' cannot be followed because it does not describe a decision. 'Always write tests' collides with a repository that has no test framework.

The fix is the checkability test. Could you verify from a diff that this was followed? If not, rewrite it until you could.

WORKING OUT WHICH OF THE THREE YOU HAVE

The Silence Test

Did any project skill apply to that last response? Name each one that did and quote the part of your output it shaped. If none applied, say none.

WHY IT WORKS

It separates 'never selected' from 'selected and overridden', which need completely different fixes.

USE WHEN Any skill that seems inert

PAIRS WITH The Description Autopsy

Skills you have never installed.

A SKILL THAT IS NEVER SELECTED

The Description Autopsy

Here is my skill description. Write three real requests a developer would type that should match it. Then tell me honestly whether this description would match them.

WHY IT WORKS

Generating the requests first, then checking, exposes the gap between what you meant and what the description says.

USE WHEN Total silence from a skill

PAIRS WITH The Silence Test

Descriptions already validated by the six-request test.

INSTRUCTIONS THAT CANNOT BE FOLLOWED

The Checkability Test

For each line of this skill: could I verify from a diff that it was followed? Mark yes or no, and rewrite every no as something observable.

WHY IT WORKS

Unverifiable instructions consume context in every session and change nothing, which reads as the skill being ignored.

USE WHEN Before installing anything

PAIRS WITH The Description Autopsy

Skills about tone rather than output.

PARTIAL APPLICATION

The Override Trace

Some of this skill applied and some did not. For each instruction that did not, tell me what overrode it: a direct request, another skill, or an impossibility.

WHY IT WORKS

Partial application has a specific cause, and naming it prevents rewriting instructions that were fine.

USE WHEN Some instructions took, others did not

PAIRS WITH The Silence Test

Complete silence, which is a selection problem.

Try This: Diagnose Before Rewriting

Most people rewrite the instructions when the description was the problem.

1. Take the skill you most suspect is inert.
2. Run the Silence Test on a request that should have triggered it.
3. If nothing applied, do not touch the instructions at all.
4. Go straight to the description and run the Autopsy.

KEY TAKEAWAYS

- Total silence means selection failed. Do not rewrite the instructions.
- Partial application means scope or an override, which is a different fix entirely.
- If you cannot verify a line from a diff, it cannot be followed.

CHAPTER 2

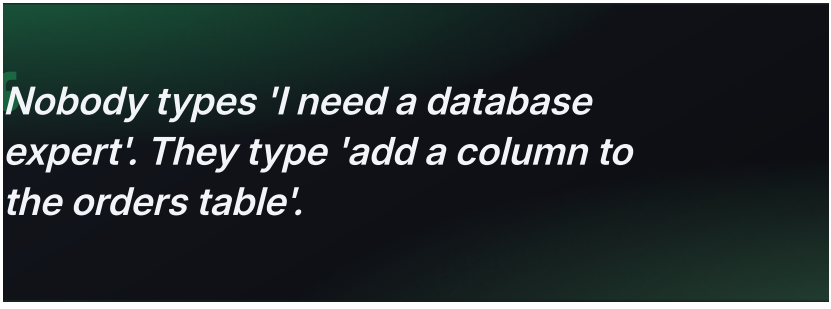
Writing The Description

> The description is a matching surface, not a summary. Everything in this chapter follows from taking that literally.

Situations, not capabilities

'Expert in database performance' describes you. 'Use when reviewing or writing a database migration' describes a moment someone is in. Only the second can match a request. Nobody types 'I need a database expert'. They type 'add a column to the orders table'.

Write the description in the vocabulary of the request, not the vocabulary of the solution. That single reframing fixes the majority of inert skills.



Name the trigger words

Include the words that will actually appear in a request that needs this skill. If the skill is about migrations, the description should contain migration, schema, column, table, because those are what a real request says.

This feels crude and it is effective. You are not writing prose to be admired; you are writing the surface a matcher will compare against.

Length is a tool, not a virtue

A description long enough to include real trigger vocabulary and a scope boundary is right. Longer than that starts matching things it should not, and shorter than that matches nothing at all.

Two sentences is a good target: one naming the situation with its vocabulary, one naming what it excludes. If you cannot say the situation in a sentence, the skill is probably two skills.

ANY DESCRIPTION NAMING A CAPABILITY

The Situation Rewrite

Rewrite this as the situation it applies to, using the words someone would type while in that situation. Give me three options, from narrow to broad.

WHY IT WORKS

Three options make the right breadth obvious by comparison, which is hard to judge from a single attempt.

USE WHEN Every description **PAIRS WITH** The Vocabulary Sweep

Descriptions already naming a concrete moment.

MAKING SURE REAL REQUESTS MATCH

The Vocabulary Sweep

List the ten words most likely to appear in a request that needs this skill. Which of them are missing from my description?

WHY IT WORKS

Matching happens on the words that actually appear, and authors systematically use solution vocabulary instead of request vocabulary.

USE WHEN After the situation rewrite **PAIRS WITH** The Situation Rewrite

Skills triggered by explicit invocation only.

A DESCRIPTION THAT FIRES TOO OFTEN

The Breadth Check

Write three requests that should NOT trigger this skill but probably would, given this description. Then narrow it so they would not.

WHY IT WORKS

Over-broad descriptions are as harmful as narrow ones, and they are much harder to notice because something always happens.

USE WHEN Skills that seem to fire everywhere

PAIRS WITH The Vocabulary Sweep

Skills that are not firing at all.

DESCRIPTIONS THAT GREW

The Two-Sentence Cut

Reduce this to two sentences: one naming the situation with its real vocabulary, one naming what it excludes. Nothing else.

WHY IT WORKS

The constraint forces a decision about what the skill is for, and a description you cannot cut to two sentences is usually two skills.

USE WHEN Any description over four lines

PAIRS WITH The Situation Rewrite

Genuinely complex triggers that need enumeration.

Try This: Request Vocabulary

Write down what a real request looks like before touching the description.

1. Write three requests, in your own typing voice, that should trigger the skill.
2. Underline the words that carry the meaning.
3. Check how many of those words appear in your description.
4. Add the missing ones and re-run the six-request test.

KEY TAKEAWAYS

- Describe the moment someone is in, not the expertise you have.
- Include the vocabulary a real request uses, because matching happens on those words.
- Two sentences: the situation, and what it excludes.

CHAPTER 3

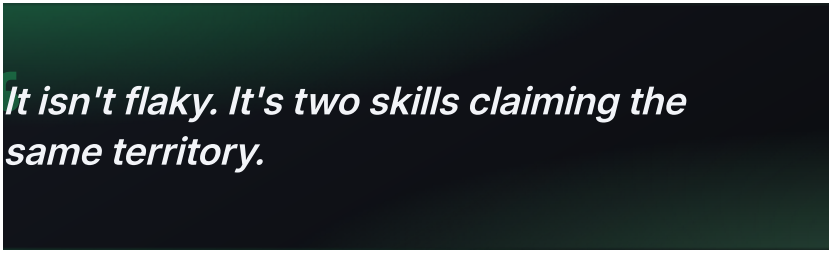
Scope And Collisions

> One skill needs no boundaries. Five do. This chapter is about the failure that arrives around the fourth one and looks exactly like unreliability.

Overlap produces nondeterminism

When two descriptions could match the same request, which one applies starts depending on details you cannot see. The result is behaviour that changes between sessions for no visible reason. It is the most frustrating failure mode in this area, because it looks like the tool is flaky.

It is not flaky. It is two skills claiming the same territory, and the fix is to narrow one until they cannot both match.



Exclusions are cheap insurance

Every skill past the first should say what it does not cover, specifically enough that a neighbour cannot claim the same ground. 'This covers migrations only, not schema design or query performance' takes ten seconds and prevents an afternoon.

Write the exclusion when you write the skill, not when the collision appears. By then you are debugging behaviour rather than reading two descriptions side by side.

Prefer few, broad skills to many, narrow ones

The instinct is to write a skill per situation. That produces a large set with heavy overlap and unpredictable selection. A smaller number of well-scoped skills, each covering a coherent area, behaves far better.

When you find yourself writing the fourth variant of the same idea, the answer is usually one skill with a broader description rather than four competing ones.

EVERY SKILL AFTER THE FIRST

The Collision Check

Here are my skill descriptions. List every pair that could match the same request, and for each, write the request that would be ambiguous.

WHY IT WORKS

The ambiguous request makes an abstract overlap concrete, which is what makes it fixable.

USE WHEN After each new skill **PAIRS WITH** The Exclusion Sentence

A set of exactly one.

CLAIMING TERRITORY PRECISELY

The Exclusion Sentence

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

WHY IT WORKS

Ten seconds at authoring time against an afternoon of debugging nondeterministic behaviour later.

USE WHEN Every skill you write **PAIRS WITH** The Collision Check

Skills covering a isolated area.

FOUR VARIANTS OF ONE IDEA

The Merge Question

These skills overlap heavily. Would one skill with a broader description and internal branching behave better? Draft it and tell me what is lost.

WHY IT WORKS

Fewer, well-scoped skills select more predictably than many narrow competitors, and 'what is lost' keeps the decision honest.

USE WHEN When a set starts feeling repetitive

PAIRS WITH The Collision Check

Skills that serve different situations.

OVERLAP YOU CANNOT REMOVE

The Precedence Question

These two must both exist and could both match. Which should win, and how do I express that in the descriptions rather than in the instructions?

WHY IT WORKS

Expressing precedence in descriptions keeps selection predictable; expressing it in instructions only helps after both have already fired.

USE WHEN Unavoidable overlaps

PAIRS WITH The Exclusion Sentence

Overlaps you can simply narrow away.

Try This: Read Them Side By Side

Collisions are obvious on paper and invisible in behaviour.

1. Paste all your skill descriptions into one document.
2. Run the Collision Check and read the ambiguous requests it produces.
3. For each pair, narrow one description until the request is no longer ambiguous.
4. Add an exclusion sentence to any skill that does not have one.

KEY TAKEAWAYS

- Overlap makes selection nondeterministic, which reads as the tool being flaky.
- Write the exclusion when you write the skill, not when the collision appears.
- Few broad, well-scoped skills beat many narrow competing ones.

CHAPTER 4

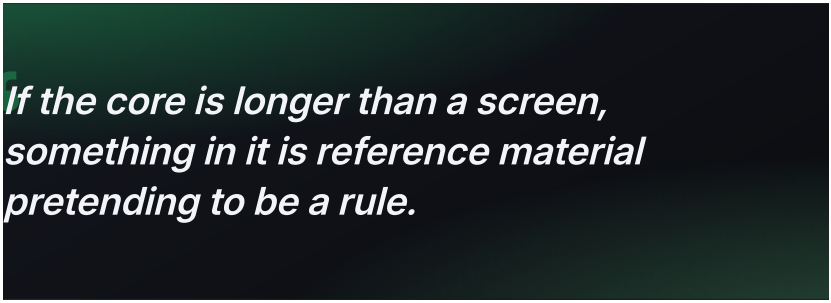
Progressive Disclosure

> A long skill has a problem the author never sees: it competes for space with everything else loaded. This chapter is about splitting instructions so the depth is available without the cost being constant.

The core and the reference

Split any substantial skill in two: a short core with the rules that apply every time it fires, and reference material consulted when needed. The core stays small enough to be loaded without hesitation; the depth is available on demand.

As a rule of thumb, if the core is longer than a screen, something in it is reference material pretending to be a rule.



Point rather than paste

Reference material belongs in files the skill points to, not inlined. A skill saying 'for the full checklist see checklist.md' costs a line; a skill containing the checklist costs the checklist every time it fires, including the many times the checklist is irrelevant.

This is the single biggest lever on how heavy a skill set feels, and it is invisible until the set is large enough to hurt.

Make the pointer specific

A pointer only works if it says when to follow it. 'See reference.md' is weak; 'when the change touches authentication, read auth-rules.md before proposing anything' is strong, because it names the condition rather than leaving it to judgement.

Conditions in the core, detail in the file. That split is the whole pattern.

A SKILL THAT HAS GROWN LONG

The Core Extraction

Split this skill: which lines are rules that apply every time it fires, and which are reference consulted occasionally? Give me two files.

WHY IT WORKS

It surfaces how little of a long skill is needed on every invocation.

USE WHEN Any skill longer than a screen

PAIRS WITH The Pointer Rewrite

Short skills that are already only rules.

MAKING A REFERENCE ACTUALLY GET READ

The Pointer Rewrite

Rewrite this pointer to name the CONDITION under which the file must be read, not just its existence. Start with 'when'.

WHY IT WORKS

A pointer without a condition is a suggestion, and suggestions lose to whatever is already in context.

USE WHEN Every reference file you add

PAIRS WITH The Core Extraction

Files meant purely as background.

UNDERSTANDING WHAT A SET COSTS

The Weight Check

List every project instruction currently loaded and roughly how long each is. Which three take the most space, and what fraction of sessions actually need them?

WHY IT WORKS

It makes the invisible cost visible, and the answer usually points straight at one bloated skill.

USE WHEN Quarterly, or when sessions feel slow

PAIRS WITH The Core Extraction

Sets of two or three short skills.

KEEPING CORES HONEST

The One-Screen Rule

Cut this skill's core to what fits on one screen. Tell me exactly what you moved out and where it went.

WHY IT WORKS

An explicit constraint forces the reference/rule distinction that authors avoid making voluntarily.

USE WHEN Any core over about forty lines

PAIRS WITH The Pointer Rewrite

Skills whose rules are long and all conditional.

Try This: Split Your Longest

The longest skill is almost always mostly reference.

1. Open your longest skill and mark each line: rule or reference.
2. Move everything marked reference into a separate file.
3. Replace it with a pointer that starts with 'when'.
4. Run the six-request test to confirm the core still fires correctly.

KEY TAKEAWAYS

- Core = rules that apply every time. Everything else is reference.
- Point rather than paste. Inlined reference costs you on every invocation.
- A pointer without a condition is a suggestion, and suggestions lose.

CHAPTER 5

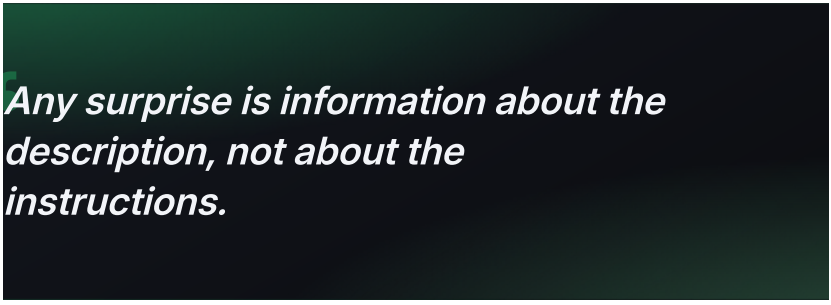
Testing A Skill

> Almost nobody tests a skill, which is why almost everybody has skills that do not fire. The procedure takes fifteen minutes and it is the difference between a setup and a hope.

Six requests, predicted first

Write three requests that must trigger the skill and three that must not. Predict the outcome of each before running any of them. Then run all six. The prediction is what turns a demonstration into a test, because it makes a surprise visible.

Any surprise is information about the description, not about the instructions. That is worth repeating, because the instinct on failure is always to go and edit the instructions.



Test the boundaries, not the centre

The obvious triggering request will usually work. The useful tests are the edges: a request that mentions the topic in passing, one that uses different vocabulary for the same situation, one adjacent to another skill's territory.

Those are where descriptions turn out to be too narrow or too broad, and they are exactly the cases real work produces.

Re-test after every change

Editing a description invalidates previous results. So does adding a neighbouring skill, because the set is what determines selection, not each skill alone.

This sounds heavy and takes about three minutes once you have the six requests written down, which is the actual reason to write them down.

PROVING A SKILL FIRES

The Six-Request Test

Write three requests that must trigger this skill and three that must not, using varied vocabulary. Predict each result. I will run all six and tell you what happened.

WHY IT WORKS

Prediction before execution is what makes a surprise visible rather than explainable after the fact.

USE WHEN Before relying on any skill **PAIRS WITH** The Edge Set

Skills you have not finished writing.

TESTING WHERE IT ACTUALLY BREAKS

The Edge Set

Write three edge-case requests: one mentioning the topic in passing, one using different vocabulary for the same situation, one bordering another skill's territory.

WHY IT WORKS

The centre almost always works. Descriptions fail at the edges, and real work lives there.

USE WHEN After the basic six pass **PAIRS WITH** The Six-Request Test

Skills with a single, unmistakable trigger.

KEEPING A GROWING SET HONEST

The Regression Set

Collect the six requests for each of my skills into one numbered list I can re-run after any change. Note which skill each expects.

WHY IT WORKS

The set determines selection, so adding one skill can silently break another. A list makes re-testing three minutes rather than an afternoon.

USE WHEN Once you have four or more skills

PAIRS WITH The Six-Request Test

Very small sets that change rarely.

CHECKING IT FIRED IN REAL WORK

The Live Confirmation

In the work you just did, which project skills applied? Quote the specific part of your output each one shaped.

WHY IT WORKS

Test-bench behaviour and real-session behaviour differ, and quoted output is the only proof that survives scepticism.

USE WHEN Periodically, in ordinary work

PAIRS WITH The Regression Set

Sessions where no skill was relevant.

Try This: Write the Six Down

The requests are worth more written than remembered.

1. For your most important skill, write three must-trigger and three must-not requests.
2. Save them in the same folder as the skill, in a file called tests.md.
3. Predict, then run all six.
4. Re-run them the next time you add or edit any skill.

KEY TAKEAWAYS

- Predict before running. That is what converts a demonstration into a test.
- Test the edges: passing mentions, different vocabulary, neighbouring territory.
- Adding one skill can break another, so keep the requests written down.

CHAPTER 6

Maintenance Over Versions

> Tools update. When something stops working afterwards, the useful question is whether your skill broke or the mechanism moved, and they need completely different responses.

Three things that move

Where files live, how selection is decided, and how much of a long instruction is loaded at once. Those are the mechanisms, and they change without asking you. Your descriptions, scopes and rules are content, and they do not.

So after an update, check in that order: location, then whether the description still matches, then whether the instructions are being truncated. It is almost always one of the first two.

Symptoms tell you which

Everything stopped at once, across all skills? Location. One skill stopped while others kept working? Description or a new collision. Long skills behave oddly while short ones are fine? Truncation.

That triage takes a minute and saves you from rewriting content that was never the problem, which is the most common wasted afternoon in this area.

What to keep stable on purpose

Write descriptions in plain request vocabulary rather than tool-specific jargon, keep cores short, and keep the regression list current. All three make version changes cheap, because none of them depend on a mechanism staying put.

A setup built that way survives updates with an afternoon of adjustment. One built around a specific version's quirks gets rebuilt every release, which is the treadmill this whole system exists to get you off.

AFTER A TOOL VERSION CHANGE

The Update Triage

Something stopped working after an update. Ask me: did everything stop, or one skill, or only the long ones? Then tell me which of location, description or truncation to check first.

WHY IT WORKS

The symptom pattern identifies the cause in a minute and prevents rewriting content that was fine.

USE WHEN Any post-update misbehaviour

PAIRS WITH The Regression Set

Failures that predate the update.

DESCRIPTIONS THAT WILL AGE BADLY

The Jargon Sweep

Find any tool-specific or version-specific vocabulary in my descriptions and rewrite it in plain request language.

WHY IT WORKS

Descriptions written in the vocabulary of a version stop matching when the vocabulary changes; plain request language does not.

USE WHEN Quarterly

PAIRS WITH The Update Triage

Descriptions that must name a specific tool.

A SET YOU HAVE STOPPED THINKING ABOUT

The Annual Re-Read

Read all my skills as if you had never seen them. Which would you delete, which would you merge, and which describe a problem I no longer have?

WHY IT WORKS

Sets accumulate skills written for problems that were solved another way, and nobody notices from inside.

USE WHEN Once a year, or after a big project shift

PAIRS WITH The Jargon Sweep

Sets under three months old.

KNOWING WHAT YOU WOULD LOSE

The Portability Check

If I moved to a different tool tomorrow, which of my skills would transfer as written, which would need reworking, and which are pure mechanism?

WHY IT WORKS

It separates the standards you have decided from the plumbing you have configured, which is worth knowing before you are forced to.

USE WHEN Before adopting a new tool

PAIRS WITH The Annual Re-Read

Setups you have no intention of moving.

Try This: Triage Before Editing

The next update will break something. Decide now how you will respond.

1. Write the three checks on a note: location, description, truncation.
2. Next time something breaks after an update, run the triage before opening any skill file.
3. Fix only what the triage names.
4. Re-run the regression list to confirm nothing else moved.

KEY TAKEAWAYS

- Location, selection and loading move. Your descriptions and rules do not.
- All skills broken means location. One means description. Long ones means truncation.
- Plain request vocabulary and short cores are what make updates cost an afternoon.

PUT IT INTO PRACTICE

Your Action Plan

The craft, in the order that matters. Selection first, everything else after.

- [ ] Run the Silence Test before touching any instructions. Diagnose, don't assume.
- [ ] Rewrite every description as a situation, in request vocabulary.
- [ ] Run the Vocabulary Sweep and add the missing trigger words.
- [ ] Cut each description to two sentences: the situation and what it excludes.
- [ ] Add an exclusion sentence to every skill past the first.
- [ ] Run the Collision Check after each new skill and narrow any ambiguous pair.
- [ ] Split any skill longer than a screen into a core and a reference file.
- [ ] Rewrite every pointer to start with 'when' and name the condition.
- [ ] Write three must-trigger and three must-not requests for each skill.
- [ ] Predict, then run all six. Every surprise is a description problem.

- [ ] Keep the requests in tests.md and re-run after any change to the set.
- [ ] After an update, triage location / description / truncation before editing anything.

FINAL WORDS

Where You Go From Here

If you take one thing from this playbook, take the diagnosis order. When a skill does nothing, the overwhelming probability is that it was never selected, and the instinct to go and improve the instructions is not just wasted effort, it usually makes the file longer and the problem worse. Check whether it fired. If it did not, the description is the only thing that matters. Write descriptions the way requests are actually typed, not the way solutions are described. Give every skill an exclusion sentence from the second one onward. Keep the cores short and put the depth behind a pointer with a condition on it. And write the six requests down, because a set of skills is a system rather than a collection, and adding one can quietly break another. Do those things and you will have something rare: a setup you can reason about, that survives a version change with an afternoon of adjustment, and that you can hand to a colleague without a paragraph of apology about which parts do not really work.

Breathe. Begin. Return.
