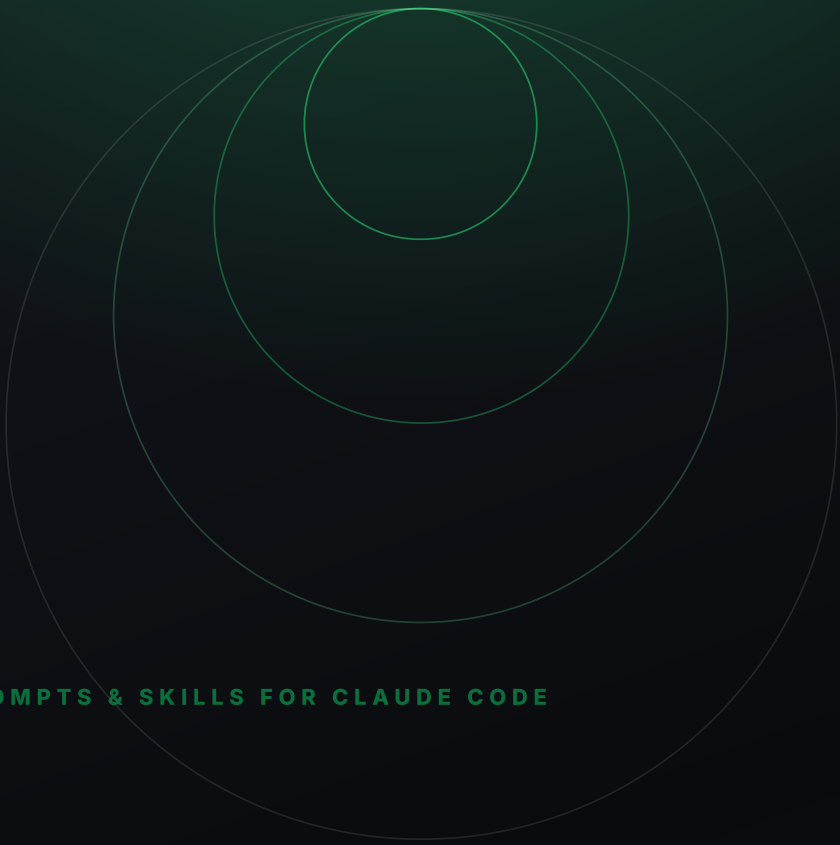


PROMPTS & SKILLS FOR CLAUDE CODE

Forty tested prompts for working with a coding agent, with the exact wording and when not to use them.

Terminal Craft

INDEPENDENT FIELD GUIDE

The Prompt Library

Forty tested prompts for working with a coding agent, with the exact wording and when not to use them.

TERMINAL CRAFT

WELCOME

Before You Begin

You already know the agent can write good code. What you may not have noticed is how much of your day goes into the round trips before it does. The clarifications. The re-runs with slightly different adjectives. The diff that turned out to touch four files you never mentioned. Almost none of that is the model being unhelpful. It is a prompt that carried the goal and left the constraints, the context and the definition of done implicit, so the agent filled in the gaps reasonably and filled them in wrong. This library is forty prompts that stop that happening. They are grouped by the job you are doing: scoping a task, reading an unfamiliar repository, planning before writing, reviewing a diff, rescuing a session that drifted. Each one is written out in full so you can paste it as-is. Each also carries the two lines most prompt collections leave out: what it pairs with, and when it is the wrong tool. Two honest notes before the prompts start. This is an independent publication with no affiliation to Anthropic or any other AI vendor, and no software or subscription is included; you need your own tools. And this is the 2026 Edition, dated on the cover on purpose. Model behaviour shifts between versions, so the final chapter is about what tends to change and how to

adapt a prompt when it does. The structures in this book outlast version numbers. Any specific output will not, and pretending otherwise would waste your money.

— 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

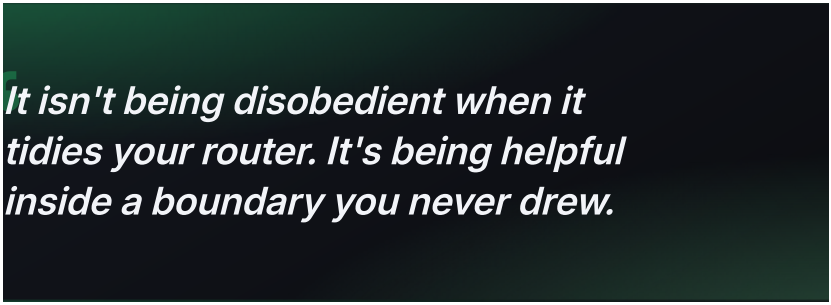
The Anatomy Of A Prompt That Lands

> Before the library, the shape. Almost every prompt in this book is the same four things in the same order, and once you can see that shape you can write your own for situations no book anticipated.

Goal, constraints, context, done

A prompt that lands carries four things. The goal: what outcome you want, in one sentence. The constraints: what must not change, what to leave alone, what conventions to follow. The context: which files, which entry points, what the agent should read before it decides anything. And the definition of done: how you will both know the task is finished.

Most improvised prompts carry the first one and assume the rest. The agent then infers constraints from the code around it, infers context from whatever it happens to open, and infers done from its own sense of completeness. Those inferences are usually reasonable and frequently not what you wanted.



The constraint line does the most work

If you only add one thing to your prompts, add what must not change. 'Do not refactor anything I did not ask about' and 'do not add new dependencies' prevent more surprising diffs than any other sentence available to you, and both take four seconds to type.

This is because an agent optimises for a good result on the task as it understands it, and improving nearby code looks like a good result. It is not being disobedient when it tidies your router; it is being helpful within a boundary you never drew.

Ask for a plan before a diff

The single highest-leverage habit in this entire book is making the agent state its plan before it edits anything. A plan is cheap to read, cheap to correct, and cheap to reject. A diff is none of those, because by the time you are reading

one the work has already been done and rejecting it feels wasteful.

One round of plan review costs perhaps thirty seconds and routinely saves the twenty-minute cycle of reviewing, rejecting and re-running. Every scoping prompt in the next chapter is a variation on this idea.

THE SHAPE EVERY OTHER PROMPT IS BUILT ON

The Four-Part Frame

```
Goal: <one sentence>.
Constraints: do not change <X>; follow the existing patterns in
<path>; no new dependencies.
Context: read <files> before deciding anything.
Done when: <observable condition>.

Restate this back to me in four lines before you start.
```

WHY IT WORKS

It supplies the three things improvised prompts leave implicit, and the restatement forces a mismatch to surface while it is still free to fix.

USE WHEN Any task you would be annoyed to see done wrong

PAIRS WITH The Scope Lock

One-line questions where a restatement costs more than the task.

THE SINGLE SENTENCE WORTH ADDING TO EVERYTHING

The Constraint Line

Do not refactor, rename, reformat or 'improve' anything I did not explicitly ask about. Do not add dependencies. If you think something nearby should change, tell me instead of doing it.

WHY IT WORKS

An agent treats adjacent improvements as part of a good result. This draws the boundary that stops a two-line fix arriving as a nine-file diff.

USE WHEN Append to almost any editing prompt

PAIRS WITH The Four-Part Frame

Sessions where you want a broad cleanup.

LONG SESSIONS THAT LOSE THE THREAD

The Context Budget

Before continuing: summarise in 5 bullets what we have established so far, what is still open, and the current constraint list. Work from that summary from now on.

WHY IT WORKS

Long sessions accumulate contradictory instructions. A restated summary makes the working set explicit and drops what no longer applies.

USE WHEN Any session past about an hour

PAIRS WITH The Four-Part Frame

Short sessions, where it only adds noise.

TASKS THAT FEEL FINISHED BUT AREN'T

The Done Definition

State the observable condition that means this is done: a command that passes, an output that changes, a behaviour I can check. Do not use 'works correctly'.

WHY IT WORKS

Ambiguous completion is why a task can feel finished while the original problem persists. An observable condition cannot be quietly moved.

USE WHEN Anything without an obvious test

PAIRS WITH The Test-First Turn

Exploratory work with no defined endpoint.

Try This: Add One Line

Test the claim on your own work rather than taking it on faith.

1. Take the next three tasks you were going to run anyway.
2. Append the Constraint Line to each, and change nothing else.
3. Count the files touched in each diff, and compare with your usual.
4. If the count drops, that line is now part of how you work.

KEY TAKEAWAYS

- Goal, constraints, context, done. In that order, every time.
- The constraint line prevents more surprising diffs than any other sentence.
- A plan is cheap to reject. A finished diff is not.

CHAPTER 2

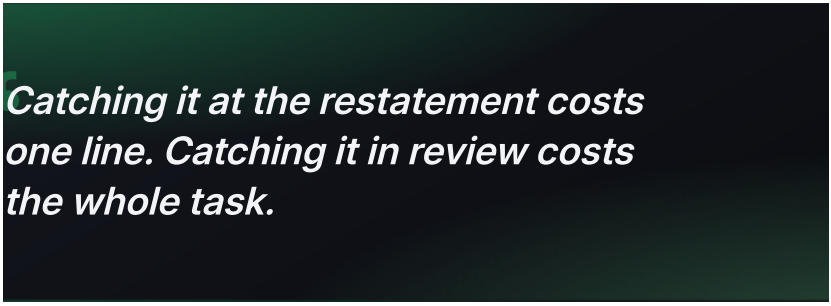
Scoping Prompts

> Scoping is where the money is. Every minute spent pinning down what the task actually is returns several minutes you would otherwise spend reviewing work that answered a different question.

Restatement catches the mismatch early

Asking the agent to restate the task in one sentence before starting sounds like ceremony, and it is the cheapest bug-catching mechanism available. Roughly one time in five the restatement is subtly wrong. A different interpretation of a pronoun, a broader reading of 'fix', an assumption about which layer you meant.

Catching that at the restatement costs one line. Catching it in review costs the whole task, plus the awkward work of separating the correct parts of a diff from the parts built on a misunderstanding.



Name what stays untouched

A scope has two halves and most people only write one. 'Fix the timezone bug in the export' says where to work; it does not say that the export format, the API contract and the test fixtures are off limits. The agent will make sensible decisions about all three, and one of them will surprise you.

Writing the untouched half takes a few seconds and converts a vague instruction into something you can actually verify afterwards, because now there is a stated boundary to check the diff against.

Size the task before starting it

Some tasks are one file and ten lines. Some are secretly a refactor with a bug fix hiding inside. Asking for an estimate of files and risk before any work begins tells you which one you are in, and lets you split the second kind before it becomes a diff nobody wants to review.

The estimate does not need to be accurate. It needs to be a number you can react to, because 'about eleven files' is the moment to stop and split, and you would rather hear it now than after.

BEFORE ANY CHANGE

The Scope Lock

Before writing any code: restate the task in one sentence, list the files you expect to touch, and name what you will NOT change. Wait for my go-ahead before editing anything.

WHY IT WORKS

It surfaces a misunderstanding while it still costs one line, and the list of expected files becomes the boundary you check the diff against later.

USE WHEN Any task larger than about ten lines

PAIRS WITH The Plan Gate

Trivial single-line edits where the round trip is the whole cost.

FINDING THE HIDDEN REFACTOR

The Size Check

Before starting: estimate how many files this touches and rate the risk low/medium/high with one reason. If it is more than four files or medium risk, propose how to split it instead of doing it.

WHY IT WORKS

Large tasks fail as one unreviewable diff. Getting the size stated up front makes splitting the default rather than a rescue operation.

USE WHEN Anything you suspect is bigger than it sounds

PAIRS WITH The Scope Lock

Work you have already deliberately scoped and split.

VAGUE REQUESTS YOU INHERITED

The Ambiguity Sweep

List every ambiguity in this request that could lead you to build the wrong thing. Ask me the three that matter most. Do not start until I answer them.

WHY IT WORKS

It converts your own vagueness into three specific questions, which is far easier to answer than being asked to specify everything in advance.

USE WHEN Tickets written by someone else

PAIRS WITH The Scope Lock

Requests you have already specified precisely.

A TASK THAT IS SECRETLY THREE TASKS

The Split

Split this into the smallest independently shippable steps. Each step must leave the codebase working. Give me the list and wait. Do not start step one.

WHY IT WORKS

Independently shippable steps stay reviewable and let you stop halfway without leaving a broken tree.

USE WHEN Anything estimated over four files

PAIRS WITH The Size Check

Changes that cannot be split without breaking a build.

BEFORE A LONG TASK

The Assumption Dump

List every assumption you are making about my intent, the environment, the data and the conventions. Mark the three most likely to be wrong.

WHY IT WORKS

Assumptions cause the failures that look inexplicable afterwards. Listing them converts silent guesses into things you can correct in ten seconds.

USE WHEN Tasks with unfamiliar context

PAIRS WITH The Ambiguity Sweep

Well-specified mechanical changes.

BEFORE WRITING SOMETHING NEW

The Prior Art Check

Does this codebase already do something similar? Search for it and cite paths before proposing anything new. If it exists, extend it instead.

WHY IT WORKS

Duplicate implementations are the most common avoidable diff, and they are invisible unless someone deliberately looks first.

USE WHEN Any new helper, utility or pattern

PAIRS WITH The Cold-Repo Read

Genuinely novel functionality with no precedent.

RISKY CHANGES

The Rollback Question

If this change turns out to be wrong in production, what is the fastest way to undo it? If the answer is not 'revert the commit', tell me now.

WHY IT WORKS

Migrations and data changes are often much harder to reverse than to apply, and that is worth knowing before rather than after.

USE WHEN Migrations, config and data changes

PAIRS WITH The Plan Gate

Pure code changes behind no state.

Try This: Count the Catches

Find out how often the restatement is wrong on your work.

1. Run the Scope Lock on every task for one week.
2. Each time, note whether the restatement matched what you meant.
3. At the end of the week, count the mismatches.
4. That count is the number of wasted review cycles you avoided.

KEY TAKEAWAYS

- Restatement catches roughly one misunderstanding in five, at a cost of one line.
- A scope has two halves: where to work, and what stays untouched.
- Ask for a size estimate so you can split before it becomes an unreviewable diff.

CHAPTER 3

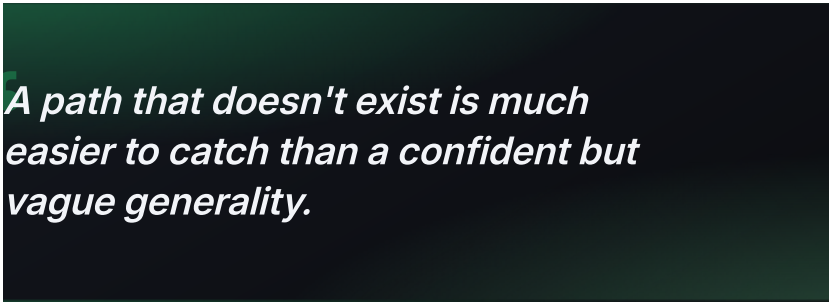
Exploration Prompts

> Reading an unfamiliar codebase is the job agents are unreasonably good at, and the one people most often skip. Twenty minutes of structured exploration prevents the confident change that breaks something three modules away.

Ask for a map, not a summary

A summary of a repository is a paragraph you will forget. A map is a list of entry points, where state lives, how data enters and leaves, and which three files you would have to understand first, with paths you can open. The second is useful; the first is a pleasant description of work you have not done.

Insist on paths and line references throughout. It makes the answer verifiable and it makes invented structure obvious, because a path that does not exist is much easier to notice than a confident but vague generality.



Trace one path end to end

The fastest way to understand a system is to follow a single real request all the way through: where it arrives, what transforms it, what it touches, what it returns. One traced path teaches more than any architectural overview, because it is concrete and because you can check it.

Pick the path closest to whatever you are about to change. You are not trying to understand the whole system; you are trying to know enough about the part you are touching to be surprised less.

Find the landmines before you step on them

Every codebase has places where a reasonable change breaks something distant: global state, implicit ordering, a config value read in six places, a test that passes for the wrong reason. Asking directly for those is remarkably productive, and it is a question people rarely think to ask.

You will not get all of them. You will usually get two or three, and one of those will be the thing that would have cost you an afternoon.

YOUR FIRST SESSION IN AN UNFAMILIAR REPOSITORY

The Cold-Repo Read

Map this repository before changing anything. Give me: entry points, where state lives, how data flows in and out, the build and test commands, and the three files I would need to understand first. Cite exact paths. Say 'unclear' rather than guessing.

WHY IT WORKS

It produces a verifiable map instead of an agreeable summary, and the explicit permission to say 'unclear' reduces confident invention.

USE WHEN First session in any repo you did not write

PAIRS WITH The Trace

Repositories you already know well. It wastes a large read.

UNDERSTANDING THE PART YOU ARE ABOUT TO CHANGE

The Trace

Trace one real <request/command/event> end to end through this codebase: where it enters, every function that transforms it, what it touches, what it returns. Cite file:line at each hop.

WHY IT WORKS

A single concrete path is checkable and memorable, where an architectural overview is neither.

USE WHEN Before changing anything in unfamiliar territory

PAIRS WITH The Cold-Repo Read

Systems where the interesting behaviour is concurrent rather than sequential.

FINDING WHAT BREAKS AT A DISTANCE

The Landmine Sweep

What in this area would make a reasonable change break something far away? Look for global state, implicit ordering, config read in multiple places, and tests that pass for the wrong reason. Cite paths.

WHY IT WORKS

It asks directly for the failure modes that code review misses, and which are invisible to anyone reading only the file being changed.

USE WHEN Before touching shared or legacy code

PAIRS WITH The Trace

Small, well-isolated modules with no shared state.

MATCHING AN UNFAMILIAR CODEBASE

The Convention Read

Before writing: what are this repo's conventions for naming, error handling, testing and file layout? Cite two examples of each from real files.

WHY IT WORKS

Cited examples produce genuine matching; asking for conventions without evidence produces a plausible description of conventions that may not exist.

USE WHEN First change in a new repo

PAIRS WITH The Cold-Repo Read

Repos with no consistent conventions to match.

BEFORE CHANGING SHARED CODE

The Dependency Map

List everything that depends on <file/function>, directly and indirectly. Cite paths. Flag anything outside this module.

WHY IT WORKS

The blast radius of a change to shared code is invisible from the file itself, and cross-module callers are where surprises live.

USE WHEN Editing anything used in several places

PAIRS WITH The Landmine Sweep

Leaf files nothing imports.

CODE THAT LOOKS WRONG

The History Question

This looks wrong to me. Before changing it: find any comment, test or commit that explains why it is like this. If there is a reason, tell me.

WHY IT WORKS

Odd-looking code is often a fix for something you have not encountered yet, and removing it recreates a bug someone already solved.

USE WHEN Anything that looks unnecessarily complicated

PAIRS WITH The Landmine Sweep

Obvious dead code with no references.

TRUSTING A GREEN SUITE

The Test Reality Check

Which tests actually cover the behaviour I am about to change? Cite them. Then tell me which of them would still pass if I broke it.

WHY IT WORKS

A green suite proves very little if nothing exercises the path you are touching, and that gap is invisible until production finds it.

USE WHEN Before relying on tests as a safety net

PAIRS WITH The Trace

Codebases with no test suite at all.

Try This: Verify Three Paths

Exploration is only worth anything if the answer is true.

1. Run the Cold-Repo Read on a repository you know well.
2. Pick three cited paths at random and open them.
3. Note whether each claim was accurate.
4. That accuracy rate is how much you should trust it on a repo you do not know.

KEY TAKEAWAYS

- Demand a map with paths, not a summary you will forget.
- One traced path teaches more than any architectural overview.
- Ask directly for landmines. It is the question nobody thinks to ask.

CHAPTER 4

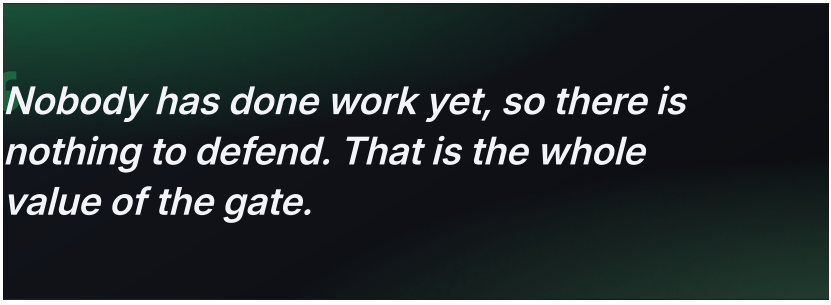
Planning Prompts

> The gap between a good session and a bad one is usually whether a plan existed before the editing started. These prompts are about making the plan explicit, reviewable and cheap to reject.

The plan gate

The pattern is simple: no edits until a plan has been stated and approved. The plan should name the files, the order of changes, what gets tested, and the one thing most likely to go wrong. Four short sections, thirty seconds to read.

The value is not that the plan is correct. It is that a wrong plan is obvious in a way a wrong diff is not, and that rejecting a plan costs nothing emotionally. Nobody has done work yet, so there is nothing to defend.



Ask for the alternative that was rejected

A plan with one approach hides the decision. Asking for the second-best approach and why it lost tells you whether the choice was considered or merely the first idea that appeared, and it occasionally reveals that the rejected option was the one you actually wanted.

It also surfaces assumptions. The reason an approach was rejected is usually a constraint the agent inferred, and seeing that constraint written down is often the moment you realise it was never true.

Decide the test before the code

Asking what test would prove this works, and what test would have caught the original bug, before any implementation produces better implementations. It forces the behaviour to be defined in observable terms, which is the ambiguity that makes tasks go wrong.

It also gives you a definition of done that neither of you can quietly move afterwards, which is a surprisingly common source of a task feeling finished while the original problem persists.

ANYTHING TOUCHING MORE THAN ONE FILE

The Plan Gate

Produce a plan before editing: (1) files in the order you will change them, (2) what each change does in one line, (3) how it will be tested, (4) the single most likely thing to go wrong. Wait for approval.

WHY IT WORKS

A wrong plan is obvious and free to reject. A wrong diff is neither, because the work already exists and rejecting it feels wasteful.

USE WHEN Multi-file work, always **PAIRS WITH** The Road Not Taken

Single-file changes you can review in ten seconds.

CHECKING THE DECISION WAS ACTUALLY MADE

The Road Not Taken

What is the second-best approach here, and why did you reject it? Name the constraint that decided it.

WHY IT WORKS

It distinguishes a considered choice from the first idea, and the stated constraint is often an inference you can correct.

USE WHEN Any plan with architectural consequences **PAIRS WITH** The Plan Gate

Tasks with one obvious mechanical implementation.

DEFINING DONE BEFORE WRITING ANYTHING

The Test-First Turn

Before implementing: what test would prove this works, and what test would have caught the original bug? Write those tests first and show me them failing.

WHY IT WORKS

It converts a fuzzy goal into an observable condition, which removes the ambiguity that most failed tasks are actually made of.

USE WHEN Bug fixes and behaviour changes

PAIRS WITH The Plan Gate

Exploratory spikes you intend to throw away.

PLANS WITH SEVERAL PARTS

The Risk Ranking

Rank the steps in this plan by risk. For the riskiest one, propose how we would find out quickly if it went wrong.

WHY IT WORKS

Risk is unevenly distributed and usually concentrated in one step. Naming it lets you review that step carefully and skim the rest.

USE WHEN Any plan with four or more steps

PAIRS WITH The Plan Gate

Uniformly trivial mechanical steps.

AMBITIOUS PLANS

The Smallest Version

What is the smallest version of this that delivers real value? Describe it, then tell me what the full version adds that the small one does not.

WHY IT WORKS

It surfaces whether the extra scope is wanted or simply the first idea, and the small version often turns out to be enough.

USE WHEN Whenever a plan feels large **PAIRS WITH** The Split

Requirements that are already minimal.

NEW COMPONENTS

The Interface First

Before implementing: show me the public interface only. Signatures, inputs, outputs, errors. No bodies. Wait for approval.

WHY IT WORKS

Interfaces are cheap to argue about and expensive to change later, so reviewing them separately is where the leverage is.

USE WHEN New modules, services or APIs **PAIRS WITH** The Plan Gate

Internal changes with no new surface.

CHANGES TOUCHING STORED DATA

The Migration Order

Give the exact order of operations including deploy steps, and state what happens if we stop halfway. Never leave the system unable to read its own data.

WHY IT WORKS

Multi-step changes to persisted data fail in the middle, and only an explicit order makes that survivable.

USE WHEN Schema and data migrations

PAIRS WITH The Rollback Question

Stateless changes.

Try This: Reject One Plan

The gate only works if you are willing to use it.

1. Run the Plan Gate on your next three multi-file tasks.
2. Read each plan properly rather than skimming to 'approved'.
3. Reject at least one and say specifically why.
4. Notice how much cheaper that felt than rejecting a finished diff.

KEY TAKEAWAYS

- No edits until a plan is stated and approved. Four short sections is enough.
- Ask for the rejected alternative; the reason reveals an inferred constraint.
- Define the test before the code, so 'done' cannot move afterwards.

CHAPTER 5

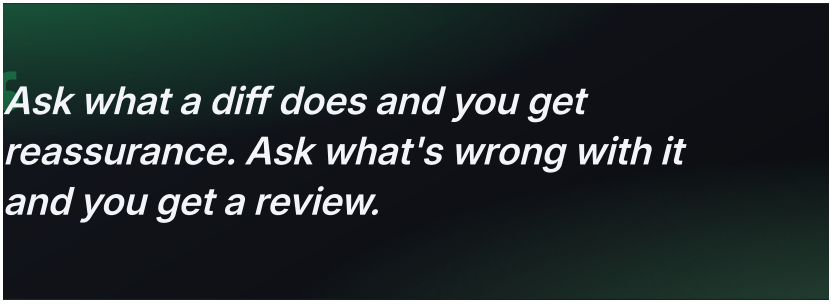
Review Prompts

> Reviewing generated code is a different skill from reviewing a colleague's. The failure modes are different, the confidence is uniform, and the temptation to skim is much stronger because it usually looks right.

Ask for defects, not a description

Ask a model what a diff does and it will describe it accurately and reassuringly. Ask it to name up to three things that are wrong, ranked by blast radius, and you get something useful. The framing decides the answer much more than the model does.

Give it permission to find nothing, too. 'If nothing is wrong, say so in one line' prevents the invented nitpick that wastes your time and trains you to ignore the whole exercise.



Push on the edges

Generated code is usually correct on the path it was written for and thin at the edges: empty input, null, enormous input, concurrent calls, the error branch nobody exercised. Asking specifically about those finds real bugs at a rate that surprises people the first few times.

The reason is straightforward. The happy path was the task; the edges were an inference. And inferences are exactly where this book keeps telling you to look.

Check the diff against the scope you set

You wrote down what would not change. Now check it. Asking the agent to list anything in the diff that falls outside the stated scope catches the quiet improvements: a renamed variable, a reordered import block, a helper extracted for tidiness. Those are what make a review longer than it needed to be.

This closes the loop the scoping chapter opened. A boundary you never check is a boundary you did not really set.

ANY DIFF YOU DID NOT WRITE

The Diff Interrogation

Review this diff as a hostile reviewer. For each change: what breaks if the input is empty, null, or very large? Name the single riskiest line and explain why. Do not summarise what the code does.

WHY IT WORKS

Explicitly forbidding a summary is what turns a reassuring description into a list of defects worth reading.

USE WHEN Before every merge **PAIRS WITH** The Scope Audit

Diffs you have already reviewed line by line yourself.

CATCHING THE QUIET EXTRAS

The Scope Audit

List everything in this diff that falls outside the scope we agreed. Include renames, reformatting, import reordering and extracted helpers. For each, say whether it is required for the task.

WHY IT WORKS

It closes the loop on the boundary you set, and quiet tidying is the most common reason a small diff becomes hard to review.

USE WHEN After any Scope Lock **PAIRS WITH** The Diff Interrogation

Sessions where a general cleanup was the actual request.

ERRORS THAT WILL NEVER BE SEEN

The Silent Failure Hunt

Find every place in this diff where an error could be swallowed, logged and ignored, or turned into a default value. For each, say what a user would experience when it happens.

WHY IT WORKS

Swallowed errors pass review easily because the code looks defensive, and they are the ones that surface as unexplained behaviour weeks later.

USE WHEN Anything with I/O, parsing or external calls

PAIRS WITH The Diff Interrogation

Pure functions with no failure modes.

REVIEWING GENERATED CODE

The Assumption Audit

What does this code assume about its inputs, its environment and its callers that is not enforced anywhere? List each and where it would break.

WHY IT WORKS

Unenforced assumptions read as clean code and are the most common source of failures that look impossible.

USE WHEN Every non-trivial diff

PAIRS WITH The Diff Interrogation

Fully validated boundaries with exhaustive checks.

CODE YOU WILL MAINTAIN

The Naming Pass

Name up to five identifiers here that will be confusing in six months, and propose better ones. Do not change anything yet.

WHY IT WORKS

Naming is the cheapest thing to fix now and the most expensive to fix later, and a fresh reader is well placed to spot it.

USE WHEN Before merging code you will own

PAIRS WITH The Scope Audit

Throwaway scripts.

GROWING CODEBASES

The Duplication Check

Does this diff duplicate logic that already exists here? Cite the existing implementation. If it does, say whether extending it is better.

WHY IT WORKS

Duplication introduced by an agent is easy to miss in review because both copies look reasonable in isolation.

USE WHEN Any new function of substance

PAIRS WITH The Prior Art Check

Deliberate duplication for decoupling.

EXPLANATORY COMMENTS

The Comment Test

For each comment added: does it explain WHY, or does it restate WHAT the code does? Delete the ones that restate.

WHY IT WORKS

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

USE WHEN Diffs with several new comments

PAIRS WITH The Naming Pass

Public API documentation, which is a different job.

Try This: Two Framings, One Diff

Prove to yourself that the framing does the work.

1. Take a diff you already merged and ask 'what does this do?'
2. In a fresh session, run The Diff Interrogation on the same diff.
3. Compare the two answers side by side.
4. Decide which one you would want before your next merge.

KEY TAKEAWAYS

- Forbid the summary. That single instruction turns description into defects.
- The happy path was the task; the edges were an inference. Push on the edges.
- A boundary you never check is a boundary you did not really set.

CHAPTER 6

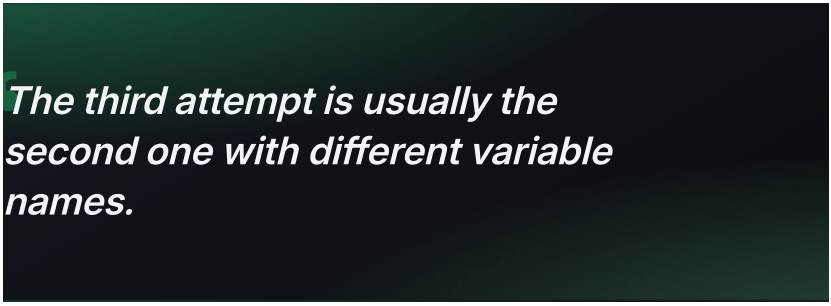
Recovery Prompts

> Some sessions go wrong. The instinct is to keep asking for another attempt, and that instinct is what turns a small problem into an afternoon. This chapter is about stopping, finding out what is true, and keeping the parts worth keeping.

Stop before you diagnose

The first move in every bad session is the same: stop editing and inventory what has already changed. Not because the inventory is interesting, but because every additional speculative edit makes the next diagnosis harder and the eventual cleanup larger.

This is difficult in the moment, because stopping feels like losing the thread. It is the opposite: you cannot diagnose a moving target, and twenty minutes of momentum is exactly what buried the original problem.



Make claims checkable

When something has gone wrong, the most dangerous input is a confident explanation. Requiring every claim about the codebase to be backed by a quoted line converts the conversation from plausible narrative into checkable evidence, and it is remarkable how many confident claims quietly disappear.

Give explicit permission to fail the check. 'If you cannot find the line, say so' produces honest answers much more reliably than any instruction not to hallucinate.

Keep the good work

The temptation after a bad session is to revert everything and start clean. Sometimes that is right. Often a good portion of the work was fine and only the last few speculative changes are the problem, and throwing it all away wastes an hour to avoid ten minutes of sorting.

Ask for the changes to be split into keep, revert and unsure. The unsure pile is usually small and is exactly where your judgement is worth spending.

THE MOMENT YOU NOTICE DRIFT

The Full Stop

Stop editing immediately. List every file you have changed this session and the one-line reason for each. Change nothing else until I reply.

WHY IT WORKS

You cannot diagnose a moving target, and each further speculative edit enlarges the eventual cleanup.

USE WHEN First move in any bad session

PAIRS WITH The Evidence Rule

Sessions that are going fine. It interrupts good flow.

A CONFIDENT EXPLANATION YOU DO NOT TRUST

The Evidence Rule

For every claim you make about this codebase, quote the exact file and line that supports it. If you cannot find one, say 'no evidence' instead of explaining.

WHY IT WORKS

It converts plausible narrative into checkable evidence, and the explicit escape hatch makes honesty easier than invention.

USE WHEN Any confident claim about unfamiliar code

PAIRS WITH The Full Stop

Discussions of approach, where there is nothing to cite.

DECIDING WHAT SURVIVES

The Salvage Sort

Split every change from this session into three lists: keep, revert, unsure. For each in 'unsure', give the one question that would settle it.

WHY IT WORKS

Most bad sessions contain more good work than the frustration suggests, and sorting takes ten minutes where a full reset costs an hour.

USE WHEN Before reverting anything **PAIRS WITH** The Full Stop

Sessions where almost nothing was completed anyway.

THE SAME FAILING FIX, TWICE

The Loop Breaker

You have now tried this approach twice and it failed both times. Do not attempt it again. List three different hypotheses for the root cause, and the cheapest test for each.

WHY IT WORKS

Explicitly forbidding the failed approach is what breaks the loop; without it the third attempt is usually the second one with different names.

USE WHEN The second identical failure **PAIRS WITH** The Evidence Rule

The first failure, where one retry is often legitimate.

RECONSTRUCTING WHAT HAPPENED

The Timeline

Reconstruct this session as a timeline: what I asked, what you did, what failed, what you tried next. Mark where it first diverged from what I asked.

WHY IT WORKS

The divergence point is almost never where the symptom appeared, and finding it is what stops you fixing the wrong thing.

USE WHEN After any confusing session **PAIRS WITH** The Full Stop

Short sessions you can remember accurately.

A BUG YOU CANNOT PIN DOWN

The Minimal Reproduction

Reduce this to the smallest input and shortest path that still reproduces it. Remove everything that is not required. Show me the reduced case.

WHY IT WORKS

Reduction usually reveals the cause on its own, and a small reproduction is the only reliable way to know a fix actually worked.

USE WHEN Any bug with an unclear cause **PAIRS WITH** The Loop Breaker

Bugs that only appear under real load.

DECIDING WHAT TO UNDO

The Revert Boundary

Give me the exact revert command or file list to undo only the speculative changes from the last twenty minutes, keeping the earlier work intact.

WHY IT WORKS

It makes partial recovery concrete, which is what stops people resetting an hour of good work out of frustration.

USE WHEN After the Salvage Sort **PAIRS WITH** The Salvage Sort

Sessions with nothing worth keeping.

WHEN STARTING OVER IS RIGHT

The Fresh Start Brief

We are starting this task again in a clean session. Write the brief I should paste there: goal, constraints, what we learned, what to avoid.

WHY IT WORKS

A clean session with a good brief beats a long polluted one, and the brief captures the learning that would otherwise be thrown away.

USE WHEN When the context is beyond repair **PAIRS WITH** The Timeline

Sessions that only need a course correction.

Try This: Write the Stop Prompt Down

The recovery prompt you cannot find in ten seconds is a prompt you will not use.

1. Copy The Full Stop into whatever you can paste from fastest.
2. Put The Evidence Rule directly underneath it.
3. Use them the next time a session starts drifting, before trying anything clever.
4. Note afterwards how much had already changed without your noticing.

KEY TAKEAWAYS

- Stop first, diagnose second. You cannot diagnose a moving target.
- Require a quoted line for every claim, and allow 'no evidence' as an answer.
- Sort into keep, revert and unsure before throwing an hour of work away.

PUT IT INTO PRACTICE

Your Action Plan

Forty prompts is a reference, not a reading list. Here is the order to actually adopt them in, one at a time.

- [ ] Append the Constraint Line to your next three editing prompts and count the files touched.
- [ ] Run the Scope Lock on every task for one week and note the mismatches it catches.
- [ ] Use the Plan Gate on anything touching more than one file, and read the plans properly.
- [ ] Reject one plan on purpose, and notice how much cheaper it is than rejecting a diff.
- [ ] Run the Cold-Repo Read the next time you open an unfamiliar repository.
- [ ] Verify three cited paths from that read, to calibrate how much to trust it.
- [ ] Trace one real request end to end before your next change in unfamiliar code.
- [ ] Run the Diff Interrogation before your next merge instead of asking what the diff does.
- [ ] Follow it with the Scope Audit and see what quiet tidying turns up.
- [ ] Keep The Full Stop and The Evidence Rule somewhere you can paste in two seconds.

- [ ] Use the Salvage Sort before you revert anything after a bad session.
- [ ] Re-read chapter one whenever a prompt stops working after a model update.

FINAL WORDS

Where You Go From Here

Forty prompts is more than anyone needs at once, and using them that way is the fastest route to using none of them. Take the four from chapter one and two, use them until they are automatic, and only then reach for the rest. The Scope Lock, the Constraint Line, the Plan Gate and the Diff Interrogation will cover most of an ordinary week between them, and everything else in this book is for the situations those four do not reach. Pay attention to the 'not for' line under each prompt. A structured prompt on a trivial task is pure overhead. Reaching for ceremony on a one-line change is how people conclude that this approach is slower. It is, for the tasks it was never meant for. And remember what the cover says. This is the 2026 Edition. Model behaviour will move, some of these wordings will need adjusting, and the way to adapt them is the four-part frame from chapter one rather than a new download. What lasts is the shape: goal, constraints, context, done. Everything else is a variation you can now write yourself.

Breathe. Begin. Return.

---

between versions and runs, so nothing here guarantees a specific result. Payments are processed securely by Digistore24.