PROMPTS & SKILLS FOR CLAUDE CODE

Using a second, independent AI to review work before it ships.

Terminal Craft

INDEPENDENT FIELD GUIDE

The Second-Opinion Stack

Using a second, independent AI to review work before it ships.

TERMINAL CRAFT

WELCOME

Before You Begin

Ask a model to review something it just produced and it will explain, clearly and confidently, why the approach was sound. It is not being evasive. It is reasoning from the same assumptions that produced the code, so the mistake is invisible from where it is standing. Proofreading your own writing is famously unreliable for the same reason. A second model, handed the diff cold with no stake in the approach, reads it as evidence rather than as a conclusion it already reached. That change of position is the entire mechanism, and it is available to anyone who already pays for more than one AI tool. This guide is the method: what the reviewing model needs in order to be useful rather than agreeable, the questions that produce defects instead of summaries, how to judge which disagreements between two models are worth your time, the shared blind spot where both agree and both are wrong, and how to keep the whole thing cheap enough that you actually run it on every meaningful diff rather than admiring it as an idea. Two notes. This is an independent publication with no affiliation to any AI vendor, and no software, licence or subscription is included. You need your own tools, and this assumes you have access to at least two. And it is the 2026 Edition:

which models are strongest at which kind of review moves constantly, so this teaches a method for testing that on your own work rather than naming winners that will be wrong by spring.

— 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

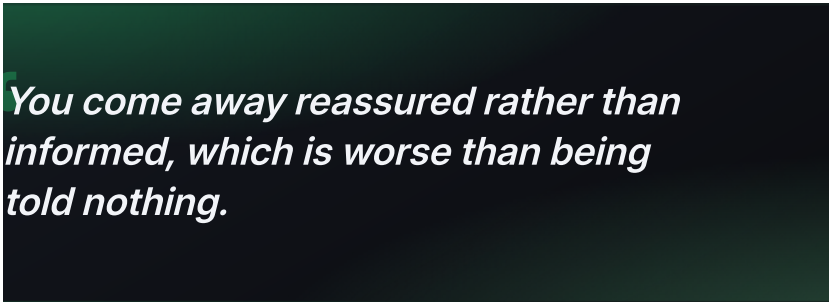
Why Self-Review Fails

> The failure is structural rather than a shortcoming of any particular model, which is why the fix has to be structural too.

Same assumptions, same blind spot

Code is produced by reasoning from a set of assumptions: what the input looks like, what the caller guarantees, what the surrounding system does. If one of those assumptions is wrong, the code is wrong, and a review that reasons from the same assumptions will conclude the code is right.

Nothing about that is a defect of intelligence. It is what happens when the reviewer and the author share a premise. Human teams have the same problem, which is why code review exists at all.



Explanation is not verification

Asked about its own work, a model produces an explanation, and a fluent explanation is extremely persuasive. More persuasive than a correct one is obliged to be. You come away reassured rather than informed, which is worse than being told nothing.

The tell is that self-review almost never finds a defect of consequence. If your review step has never changed a decision, it is a ritual rather than a control.

What a cold reader has that the author does not

No memory of why the approach was chosen. No investment in it. No context that quietly filled a gap the code did not actually fill. It reads what is there instead of what was meant, which is the reading you need before shipping.

That is also why the handoff matters so much: give it too much of the author's framing and you have recreated the original blind spot in a second window.

FINDING OUT IF YOUR CURRENT STEP DOES ANYTHING

The Self-Review Test

Review the code you just wrote and name any defect. Then tell me honestly: is there any change here you would defend that a stranger might reject?

WHY IT WORKS

The second question is the useful one. It surfaces the defended decision, which is where shared-assumption failures hide.

USE WHEN Before deciding you need a second tool

PAIRS WITH The Cold Handoff

Work that already goes through independent review.

MAKING THE PREMISES VISIBLE

The Assumption Surface

List every assumption this code makes about its inputs, its callers and its environment. Mark each as enforced in code, checked by a test, or simply assumed.

WHY IT WORKS

Shared-assumption failures become visible the moment the assumptions are written down as a list.

USE WHEN Before any handoff

PAIRS WITH The Cold Handoff

Trivial changes with no external surface.

Try This: Check Your Ritual

Find out whether your review step has ever changed anything.

1. Think back over your last ten reviews by the authoring model.
2. Count how many produced a change you actually made.
3. If the answer is zero or one, it is a ritual.
4. Read the next chapter and run one real handoff this week.

KEY TAKEAWAYS

- A reviewer sharing the author's assumptions will confirm the author's mistake.
- Fluent explanation is more persuasive than correctness requires it to be.
- If your review step has never changed a decision, it is not a control.

CHAPTER 2

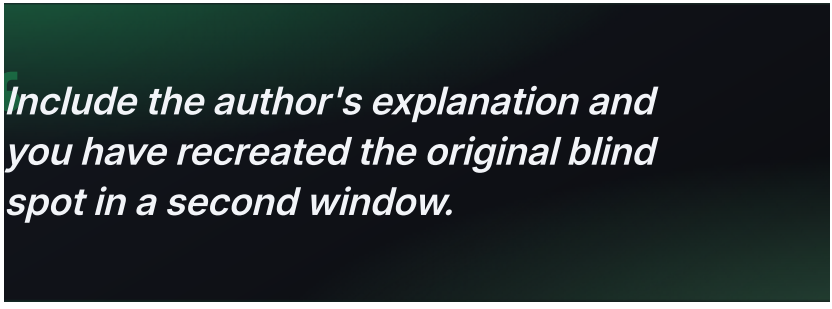
The Handoff Package

> The reviewing model needs enough to judge and not so much that it inherits the author's framing. Getting that balance right is most of the method.

Four things, no more

The diff. The task it was meant to solve, in one sentence. The constraints that applied. And the relevant surrounding code: the callers, the types, the tests. That is enough to judge whether the change does what was asked without importing the reasoning that produced it.

What to leave out: the author model's explanation of its approach. Including it is the most common mistake and it reliably converts an independent review into agreement.



State the stance explicitly

Tell the reviewer it did not write this and that the author was overconfident. Both framings measurably change what comes back. The first removes any assumed ownership. The second licenses criticism that politeness would otherwise soften.

This is not a trick. It is stating the actual situation, which happens also to be the framing that produces useful output.

Give it somewhere to look

A review of a diff in isolation can only comment on the diff. If the reviewing tool can see the repository, say so and ask for citations. If it cannot, paste the callers and the type definitions, because most real defects live at the boundary rather than inside the changed lines.

A reviewer that cannot see the caller cannot tell you the caller passes null.

BRIEFING THE REVIEWING MODEL

The Cold Handoff

You did not write this and you have no stake in it. Here is the diff, the one-sentence task it was meant to solve, the constraints, and the surrounding code. Assume the author was overconfident.

WHY IT WORKS

The stance is the active ingredient: no ownership, and explicit permission to be unimpressed.

USE WHEN Every meaningful diff **PAIRS WITH** The Defect Hunt

Changes too small to justify the round trip.

DECIDING WHAT TO INCLUDE

The Context Trim

For this review, what is the minimum context needed to judge correctness: which callers, which types, which tests? List them and leave out any reasoning about why the approach was chosen.

WHY IT WORKS

It stops you pasting the author's explanation, which is the single most common way an independent review collapses into agreement.

USE WHEN Preparing any handoff **PAIRS WITH** The Cold Handoff

Self-contained changes with no external surface.

REVIEWS WHERE THE TOOL CANNOT READ THE REPO

The Boundary Package

I will paste the diff plus the direct callers and type definitions. Tell me first exactly which of those you need to judge correctness, so I do not paste the wrong things.

WHY IT WORKS

Most real defects live at the boundary, and asking first prevents a review that can only comment on the changed lines.

USE WHEN Tools without repository access

PAIRS WITH The Cold Handoff

Reviewers that can read the codebase directly.

Try This: Withhold the Explanation

Run the same review twice and watch the framing do the work.

1. Take a recent diff and hand it to a second model with the author's explanation included.
2. In a fresh session, hand over the same diff without the explanation.
3. Compare the two responses side by side.
4. Note how much of the first one simply agreed.

KEY TAKEAWAYS

- Diff, task, constraints, surrounding code. Never the author's reasoning.
- State that it did not write this and that the author was overconfident.
- A reviewer that cannot see the caller cannot tell you the caller passes null.

CHAPTER 3

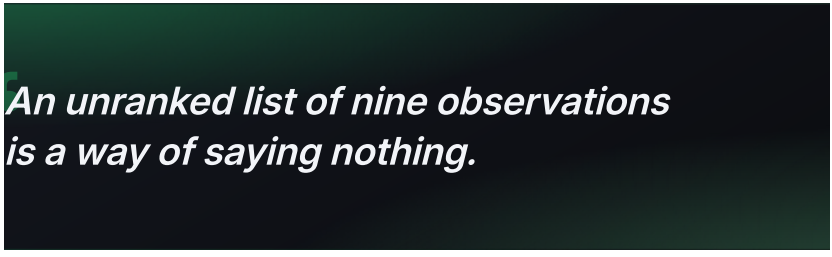
Review Prompts That Bite

> The framing decides the answer much more than the model does. These are the questions that produce defects rather than descriptions.

Forbid the summary

Ask what a diff does and you get a competent, reassuring description. Explicitly forbidding the summary and asking for up to three things that are wrong, ranked by blast radius, changes the output completely. It is the highest-leverage sentence in the whole method.

Ranking matters too. An unranked list of nine observations is a way of saying nothing; three ranked by consequence is a decision aid.



Give permission to find nothing

Always add: if nothing is wrong, say so in one line. Without it you get invented nitpicks: a manufactured concern about naming to satisfy the request. That wastes your time and teaches you to discount the whole exercise.

A reviewer that sometimes says 'this is fine' is a reviewer worth reading when it does not.

Push where inference happened

Generated code is usually right on the path it was written for and thin where the author inferred rather than knew: empty input, null, enormous input, concurrent access, the error branch nobody exercised, and what a user experiences when something fails.

Asking specifically about those finds real defects at a rate that surprises people the first few times they try it.

THE CORE REVIEW QUESTION

The Defect Hunt

Do not summarise what this does. Name up to three things that are wrong, ranked by blast radius. If nothing is wrong, say so in one line.

WHY IT WORKS

Forbidding the summary and ranking by consequence turns a description into a decision aid.

USE WHEN Every handoff **PAIRS WITH** The Edge Interrogation

Reviews where you want an explanation of unfamiliar code.

WHERE INFERENCE REPLACED KNOWLEDGE

The Edge Interrogation

For each change: what happens with empty input, null, a very large value, and two concurrent calls? Name the first one that breaks and what the user would see.

WHY IT WORKS

The happy path was the task; the edges were an inference, and inferences are where the defects are.

USE WHEN Anything with I/O, parsing or shared state **PAIRS WITH** The Defect Hunt

Pure functions with fully constrained inputs.

ERRORS THAT WILL NEVER BE SEEN

The Silent Failure Sweep

Find every place an error could be swallowed, logged and ignored, or replaced by a default. For each, describe what the user experiences when it happens.

WHY IT WORKS

Swallowed errors pass review because the code looks defensive, then surface weeks later as unexplained behaviour.

USE WHEN External calls, parsing, filesystem work

PAIRS WITH The Edge Interrogation

Code with no failure modes.

CHANGES TO SHARED INTERFACES

The Contract Check

Does this change what any caller can rely on: types, nullability, ordering, timing, error behaviour? List each and say whether callers were updated.

WHY IT WORKS

Contract changes are invisible in the diff and expensive everywhere else, which is the worst possible combination.

USE WHEN Anything touching a shared interface

PAIRS WITH The Defect Hunt

Purely internal implementation changes.

CATCHING THE QUIET EXTRAS

The Scope Audit

List everything in this diff outside the stated task: renames, reformatting, reordered imports, extracted helpers. For each, say whether it was required.

WHY IT WORKS

Quiet tidying is the most common reason a small diff becomes hard to review, and a cold reader spots it immediately.

USE WHEN Every review PAIRS WITH The Defect Hunt

Changes where cleanup was the request.

Try This: Two Framings

Prove to yourself that the wording does the work.

1. Take a diff you already merged.
2. Ask a second model 'what does this do?' and read the answer.
3. In a fresh session, run The Defect Hunt on the same diff.
4. Decide which one you would want before your next merge.

KEY TAKEAWAYS

- Forbidding the summary is the single highest-leverage sentence in the method.
- Always permit 'nothing is wrong' or you will get manufactured nitpicks.
- Push on edges and error paths. That is where inference replaced knowledge.

CHAPTER 4

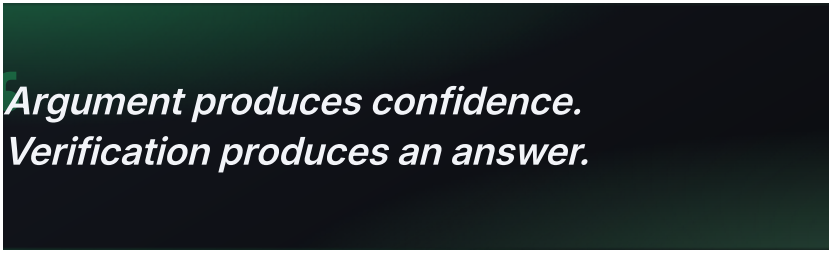
Reading Disagreement

> Two models will disagree. Most disagreements are noise, some are the most valuable signal available to you, and telling them apart takes one question.

Checkable versus stylistic

The question is whether the disagreement is decidable against the repository. 'This can be null here' is checkable. Open the caller. 'This should be a separate function' is taste, and no amount of arguing between two models will settle it.

Spend your attention entirely on the checkable ones. They are rarer and they are the only category where a wrong answer costs you something real.



Make them check rather than argue

When a checkable disagreement appears, do not ask either model who is right. Ask one of them to verify the claim against the repository and show the line. Argument produces confidence; verification produces an answer.

This is also where you find out that both were wrong, which happens more often than the neatness of the process suggests.

Divergence maps where to look

Independently of who wins, the places where two models disagree are a good map of where the code is ambiguous, under-specified or subtle. That is useful even when the disagreement itself is unresolvable.

If they disagree about what a function guarantees, the function's contract is unclear, and that is worth fixing regardless of which reading was correct.

A CONCRETE CONFLICT

The Tiebreak

Model A says X, model B says Y. Which of these is checkable against the repository? Check it, quote the line, and tell me who was right.

WHY IT WORKS

Verification produces an answer where argument only produces more confidence.

USE WHEN Any factual disagreement **PAIRS WITH** The Taste Filter

Disagreements about style, which have no fact to check.

DECIDING WHAT TO IGNORE

The Taste Filter

Sort these review comments into: checkable against the code, versus matters of taste. I only want to spend time on the first list.

WHY IT WORKS

Most review noise is taste presented with the same confidence as fact, and sorting is faster than adjudicating.

USE WHEN Any long list of comments **PAIRS WITH** The Tiebreak

Short reviews with one or two points.

USING DISAGREEMENT AS A MAP

The Ambiguity Marker

You two disagreed about what this function guarantees. Regardless of who is right: what is unclear about its contract, and how would I make it unambiguous?

WHY IT WORKS

Disagreement about behaviour is evidence the contract is under-specified, which is worth fixing whoever was correct.

USE WHEN Repeated disagreement about the same area

PAIRS WITH The Tiebreak

One-off differences of emphasis.

Try This: Sort Before You Read

Sorting first stops taste from consuming the attention facts deserve.

1. Take the next review that produced more than four comments.
2. Run The Taste Filter before reading them properly.
3. Work only through the checkable list.
4. Note how much of the original list you never needed to think about.

KEY TAKEAWAYS

- Only checkable disagreements deserve your attention. Taste never resolves.
- Ask for verification against the repo, not for an argument about who is right.
- Where they disagree about behaviour, the contract is under-specified.

CHAPTER 5

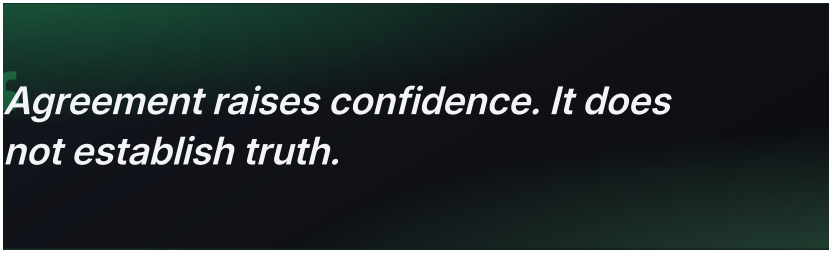
When Both Are Wrong

> The uncomfortable chapter. Two independent reviewers agreeing feels like confirmation, and sometimes it is two instances of the same blind spot arriving at the same wrong place.

Shared training, shared gaps

Different models are not independent in the way two people from different backgrounds are. They have overlapping training data and similar failure tendencies, so they can share a misconception about a library's behaviour, a security pattern, or an idiom that used to be correct and no longer is.

Agreement raises confidence. It does not establish truth, and treating it as though it does is how a confident consensus reaches production.



The domains where agreement is weakest

Recent library changes, anything version-specific, your own internal conventions, security specifics, and performance characteristics of your actual data. In all of those, both models are reasoning from general patterns rather than from your situation.

For those areas, agreement between models is close to worthless. Documentation, a benchmark or a test is the only thing that settles it.

Keep one human check

The method reduces defects; it does not remove your judgement from the loop. The minimum is that you read the diff yourself and satisfy yourself that it does what you asked. Not line by line necessarily, but properly.

Anyone selling you a review process that removes that step is selling you a way to ship faster in exchange for occasionally shipping something nobody understood.

WHEN BOTH AGREE AND IT MATTERS

The Consensus Check

You both approved this. What would have to be true for you both to be wrong? Name the assumption you share and how I could check it.

WHY IT WORKS

It converts agreement into a testable statement instead of leaving it as reassurance.

USE WHEN High-consequence changes

PAIRS WITH The Version Reality Check

Low-stakes changes where the cost of being wrong is small.

LIBRARY BEHAVIOUR EITHER MIGHT HAVE LEARNED WRONG

The Version Reality Check

Which claims here depend on a specific version of a library or framework? For each, quote the documentation for the version this project actually uses.

WHY IT WORKS

Version-specific behaviour is where shared training data most reliably produces shared, confident errors.

USE WHEN Anything touching a third-party API

PAIRS WITH The Consensus Check

Code with no external dependencies.

KEEPING YOURSELF IN THE LOOP

The Human Minimum

Summarise in three lines what I should verify myself before merging this, chosen so that a person can check it in two minutes.

WHY IT WORKS

It makes the irreducible human check small and concrete rather than an aspiration you skip when busy.

USE WHEN Every merge **PAIRS WITH** The Consensus Check

Nothing. This one always applies.

Try This: Ask What Would Have To Be True

The most useful question to ask a consensus.

1. The next time both reviewers approve something consequential, do not merge yet.
2. Run The Consensus Check and read the shared assumption.
3. Verify that one assumption yourself.
4. Note how often it turns out to be the thing worth checking.

KEY TAKEAWAYS

- Different models are not independent, because overlapping training means overlapping blind spots.
- Agreement is worth least on version specifics, your conventions, security and performance.
- Keep a two-minute human check. A process that removes it is selling you speed for surprises.

CHAPTER 6

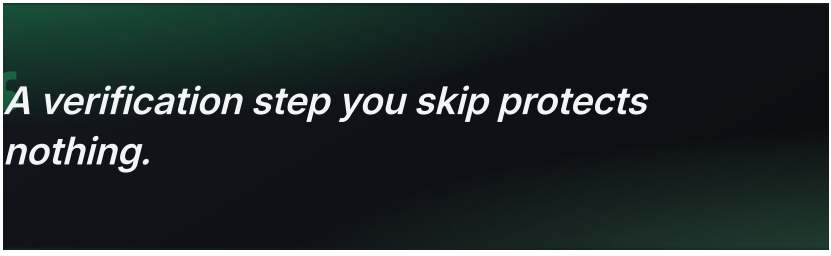
Keeping It Cheap

> A verification step you skip protects nothing. This chapter is about making the method small enough that it survives a busy week, which is the only test that matters.

Two prompts is the working version

The full method has a handoff, five review questions, a tiebreak and a consensus check. The version you will actually run is the Cold Handoff plus the Defect Hunt, and it takes under two minutes.

Run that on every meaningful diff. Reach for the rest only when the change is consequential or the first pass produced something you did not expect.



Decide the threshold in advance

Deciding in the moment always resolves toward skipping. A rule decided in advance removes the negotiation entirely: anything touching shared code, data, auth or a public interface gets a second opinion.

Write it next to your other project standards, so it is a rule rather than a good intention.

Notice when it stops paying

Keep a rough count for a month: how many reviews produced a change you actually made? If it is very low, either your threshold is too broad or your prompts have drifted back toward asking for summaries.

The method is worth what it catches. Measuring that is what stops it becoming the ritual described in chapter one, just with an extra window open.

THE VERSION THAT SURVIVES A BUSY WEEK

The Two-Minute Review

You did not write this. Here is the diff and the task. Do not summarise it. Name up to three things that are wrong, ranked by blast radius. If nothing is wrong, say so in one line.

WHY IT WORKS

It is the handoff and the defect hunt in one paste, which is the difference between a method you use and one you admire.

USE WHEN Every meaningful diff **PAIRS WITH** The Threshold Rule

Consequential changes that deserve the full pass.

REMOVING THE IN-THE-MOMENT DECISION

The Threshold Rule

Draft a one-sentence rule for when a change must get a second opinion, based on what this project actually contains: shared code, data, auth, public interfaces.

WHY IT WORKS

Decisions made in the moment resolve toward skipping; a written rule does not negotiate.

USE WHEN Once, per project **PAIRS WITH** The Two-Minute Review

Solo throwaway work.

CHECKING THE METHOD STILL PAYS

The Value Count

Over my last ten second opinions, how many produced a change I actually made? If it is under three, tell me whether my threshold is too broad or my prompt has drifted toward summaries.

WHY IT WORKS

A control that catches nothing is a ritual, and only counting reveals which of the two causes is at work.

USE WHEN Monthly **PAIRS WITH** The Threshold Rule

The first few weeks, with too little history.

Try This: Write the Threshold Down

One sentence, next to your other project standards.

1. Decide what always deserves a second opinion in this project.
2. Write it as a single sentence and put it with your standards.
3. Use the Two-Minute Review for a fortnight without exception.
4. Count how many produced a change you made, and adjust the threshold from that.

KEY TAKEAWAYS

- The version you will run is two prompts and two minutes. Everything else is for consequential changes.
- Decide the threshold in advance, because in-the-moment decisions resolve toward skipping.
- Count what it catches, or it becomes the same ritual with an extra window open.

PUT IT INTO PRACTICE

Your Action Plan

From a review that always approves to one that occasionally saves you. Small steps, in order.

- [ ] Count how many of your last ten self-reviews changed anything. If it is zero, it is a ritual.
- [ ] Pick a second tool you already have access to.
- [ ] Hand off a recent diff with the Cold Handoff wording and read what comes back.
- [ ] Never include the authoring model's explanation of its approach.
- [ ] Use The Defect Hunt: forbid the summary, rank three by blast radius.
- [ ] Always allow 'nothing is wrong' so nitpicks are not manufactured.
- [ ] Add the edge interrogation for anything with I/O, parsing or shared state.
- [ ] Sort long review lists into checkable versus taste before reading them.
- [ ] Resolve checkable disagreements by verification against the repo, not argument.
- [ ] On consequential changes, ask what would have to be true for both to be wrong.

- [ ] Write your threshold rule down next to your other project standards.
- [ ] Count monthly how many second opinions produced a change you actually made.

FINAL WORDS

Where You Go From Here

The whole method reduces to one move: stop asking the author. Everything else in this book is refinement on that, and if you only ever run the Cold Handoff followed by the Defect Hunt, you will already catch things that would otherwise have shipped. Keep it to two minutes so it survives the weeks when you are busy, because a review step that gets skipped under pressure is worth exactly nothing, and pressure is when the mistakes happen. Be honest about the limits too. Two models are not two independent experts. They share training and therefore share blind spots, and their agreement is weakest where you most want reassurance: version-specific behaviour, your own conventions, security and performance on your real data. Keep your own two-minute read of the diff. And count what the method catches. If it stops catching anything, that is not proof your code got better. It is a signal to check whether your prompts drifted back toward asking for a summary.

Breathe. Begin. Return.
