Why AI Prompting Fails

AI is often good at making the change you named. Prompting fails when that same rule also has to hold somewhere else.

By Gregory Tomlinson

AI coding tools are often at their best when the work is local.

Rename a field. Tighten a validation rule. Update a serializer. Adjust an error path. Generate a test. Ask for one bounded change in one visible place and the result can feel almost unnervingly effective.

That early success hides a different kind of failure.

As systems get larger, more changes stop being truly local. A rule that sounds simple in English often has to hold in more than one place in the code. “Preserve this file.” “Tighten this behavior.” “Make this atomic.” The prompt may point at one place while the real rule has to survive in several.

That is where AI prompting fails.

This is the failure boundary the article is about: the prompt can carry the local edit, but it fails when it has to carry the full rule across multiple places in the system.

The failure is easy to miss because the output often looks correct in the place the prompt targeted while leaving the rule broken somewhere else.

A concrete failure: deletion changed, overwrite did not

The clearest example came from a system that backed up my writing docs into a Git repo.

At one stage of the work, the rule sounded simple: preserve manual files inside managed directories. But that rule had to hold in two places: deletion and overwrite or re-export.

That distinction mattered.

If manual files survived deletion but got wiped during overwrite, the rule was still broken.

In the code-only workflow, the prompt was usually scoped to deletion. The request looked like a local fix: preserve manual files during deletion. The model updated the deletion path accordingly.

But overwrite was not named in the prompt.

So it stayed as it was.

During re-export, the old directory was still removed, and the manual files were still destroyed.

The issue was not bad output in the usual sense. The issue was that one path changed and the other did not.

That is a much more useful way to describe what goes wrong in iterative AI coding. The result can look reasonable in the place you asked about and still leave the rule broken in the rest of the system.

What I was actually testing

I was not trying to answer whether AI can write code. It can. That is not the debate.

I was testing a narrower question: when prompting is the only thing carrying scope forward, where does it fail?

To test that, I used two workflow types across staged requirement deltas.

In the first workflow, changes were made directly through conversational prompts. The code changed, but there was no persistent artifact being updated to carry the rule forward across later iterations.

In the second workflow, each tightening step was first written into a versioned contract, and implementation followed from that updated artifact.

So the real comparison was not human coding versus AI coding.

It was one way of carrying scope versus another.

In one workflow, the prompt had to carry the rule forward.

In the other, the rule was written down first in something that survived the next change.

That difference mattered most once the rule stopped being local.

What held up in the experiment

Across structured runs using Claude Opus 4.6 and GPT-5.2, the broad pattern held.

Changes that stayed in one place held up better.

Rules that had to survive in more than one place did not.

When a requirement stayed local, conversational iteration could remain stable. Once the rule had to land in multiple places, the code-only path repeatedly changed the place named in the prompt and left the unnamed place behind.

That was the repeated result.

And it was not just one bad run. Across the structured executions, the same pattern kept coming back. Changes that stayed in one place generally held. But once the rule had to survive in more than one place, the code-only track kept changing deletion and leaving overwrite or re-export behind. In the reported runs, that happened with Claude Opus 4.6 and with GPT-5.2 in ChatGPT Code Interpreter. Even prior awareness of the failure did not fix it. The pattern held because the prompt carried the part it named and dropped the part it did not.

Not random nonsense. Not generic model failure. A specific failure of scope.

The prompt named one place, so that place changed.

The prompt did not name another place, so that place lagged.

That is the pattern the article is naming.

Why “just prompt better” is not a complete answer

A natural response to all of this is simple: write better prompts.

That is partly true, but it is not the whole story.

In the experimental work, sufficiently explicit prompts could eliminate the failure. Once the prompt encoded the full rule, the gap disappeared.

But that is exactly the point.

The prompt had to become structurally equivalent to a contract before the failure stopped.

That is the boundary this article is naming: prompting fails when it is the only thing preserving the rule across iterations.

For a one-off change, that may be enough.

For ongoing iterative work, it is not.

Explicit prompting can work, but only when it fully names the rule and every place where that rule has to hold. In practice, that means the prompt is doing the same job as a durable specification artifact.

What actually helped

What helped was a written artifact outside the prompt and outside the code that stated what the system was supposed to do and what had to stay true.

In practice, that often meant a contract.

Not a loose note. Not a remembered intention. Not a reviewer trying to reconstruct what everyone probably meant. A contract in the engineering sense: a durable artifact that names required behavior, preserved rules, boundaries, and non-negotiable invariants.

Its value was mechanical, not abstract: it moved scope out of the prompt and into something that persisted across iterations.

More importantly for this argument, it offloaded scope from the prompt into an artifact that survived the next change.

That was the key difference in the manual-file case.

In the code-only track, the prompt was usually some version of “Preserve manual files during deletion.” That is why deletion changed and overwrite did not. In the contract-first track, the rule had to be written globally: preserve manual files in deletion, overwrite or re-export, and any reset behavior that could wipe the directory. Writing the rule that way forced both paths into scope before implementation changed.

A useful contract patch could not just say “preserve manual files” and stop there. To be useful, it had to name the actual places where the rule could be violated: deletion, overwrite or re-export, and any reset behavior that could wipe the directory.

That is why the artifact helped. It did not help by being fancy. It helped because it forced the rule to be written at the level where the whole change had to hold.

The bounded result is the useful one

This is where it matters not to overstate the result.

The useful claim is not that contracts magically solve AI coding.

The useful claim is narrower: explicit source-of-truth artifacts materially improve AI-assisted iteration, especially when a rule has to survive across more than one place in the system.

That is also why the result is credible.

It is bounded.

The artifact improved iteration. It did not eliminate every downstream enforcement or translation problem.

For day-to-day software delivery, the main point is simpler:

If a rule has to hold in multiple places, leaving that scope implicit is where prompting fails.

What this means for teams using AI coding tools

The practical takeaway is simple: when a rule spans multiple places, name those places explicitly.

Not:

preserve manual files during deletion

But:

preserve manual files during deletion, overwrite or re-export, and any directory reset path

For teams scaling AI coding workflows, three operating lessons follow.

The workflow change is straightforward. When the rule is local, a prompt may be enough. When the rule is cross-cutting, stop treating it like a local edit. Write the rule down first, name the places where it has to hold, and then ask for the change against that written rule. That gives the next prompt, the next review, and the next edit the same target instead of making each step rediscover the scope from memory.

The workflow rule is simple: if a change affects more than one place, and the prompt names only one of them, only that named place should be expected to change. Teams need to stop treating cross-cutting rules like local edits. Either name every affected place in the prompt, or write the rule down first in an artifact that does it for you.

Separate local edits from system rules

These are not the same class of work.

A local rename or isolated validation change can often be handled conversationally with little trouble. A rule that has to hold across handlers, persistence, overwrite behavior, resets, tests, and user-visible outputs is a different problem. Treating both as “just another prompt” is what creates false confidence.

Do not assume review repairs the problem

Review still matters.

But if the workflow depends on reviewers rediscovering missing scope after the fact, then the important rule still lives in memory instead of in the artifact driving the change. That is a weak operating model, especially as iteration gets cheaper and faster.

Give important rules a durable home outside the prompt

If a rule matters, it should live somewhere that persists across iterations.

That does not require a massive specification program. It does require something more durable than whatever was said in the last chat window.

Once the rule has a durable home, later changes have something explicit to answer to besides the current code and the current prompt.

That is the real benefit.

AI is often very good at making the change you asked for.

The harder question is whether it also preserved the rule everywhere else that rule still had to hold.

That is where prompting fails.