"The rules my AI coding agent is not allowed to break"

The rules my AI coding agent is not allowed to break

Left to its own defaults, an AI coding agent is an enthusiastic shortcut machine. It will wrap the failing call in `try?` and move on. It will force-unwrap because the optional is "obviously" non-nil. It will handle the three easy cases, skip the malformed one, and report the task done. It will tell you "tests pass" without running them, because the build was green and the build "should" mean the tests are fine. Every one of those is a small lie that compiles.

I did not start with a rulebook. I started with corrections.

The first ones were one-offs, the kind you mutter while you fix them yourself. Don't swallow that error. Don't force-unwrap there. Run the tests before you tell me they passed. But the same corrections came back, session after session, and a correction you give twice is a rule you have not written down yet. So I started writing them down, one repeated annoyance at a time, in a file the agent loads before it touches a line of my code.

What I did not expect was the side effect. To make a rule a machine will actually obey, you have to state it precisely enough that it cannot be wriggled around, and every time I forced that precision I found a place where I had been hand-waving for years. The rulebook was supposed to discipline the agent. It ended up disciplining me.

That file is public now. It is `rules-swift`: the rules I point every coding agent at as always-loaded

context. It is two layers in one repo, the Swift and Apple-platform craft on top and a language-agnostic standard of care vendored underneath in `core/` (which is also its own repo, `rules-engineering-discipline`, if you work in another language). The whole thing rests on one line, "choose the optimal path, not the fastest one," and everything else is a specific way of not betraying it.

But the repo is the destination. The journey was three turns, each one a correction I kept making until it became a rule, and each rule turning around to teach me something about my own work.

The first turn: the fix you ask for is usually a symptom

The correction I made most often was not about a bug. It was about where the bug was being fixed.

The example that finally made me write the rule was a retry. The agent had wrapped a call that should never fail in a retry loop, and the retry made the failure disappear. But a call that should never fail and sometimes does has a bug one layer down, and the retry was hiding it. The fix belonged in the layer that owns the data, not in a loop around the symptom.

The rule I wrote is almost embarrassingly basic: before writing code, state the problem's real invariant in one sentence, and if you cannot, you are not ready to write it. Trace a symptom to the layer that owns it before you patch anything. The moment I made the agent do that, I had to admit how often I did the opposite, reaching for the nearest familiar shape instead of deriving from the actual constraint. Its companion clause turned out to be just as much about me: reason from the source, not from memory. A remembered API signature is a hypothesis, not a citation. I had been quoting my own memory as evidence for years.

The second turn: validations are values, and the best one is the one you delete

The more I corrected the agent's validation code, the more I recognized my own. One big `validate()` function, a tree of `if` statements, a mutable `errors` array I appended to as I went. That is how almost everyone does it, and it is how I did it.

The turn came from Matt Polzin's OpenAPIKit idiom: validations are not control flow, they are values. A `Validation<Subject>` is a description, a check, and a `when` predicate, composed with an algebra instead of nested conditionals. Two things in that rewired how I think about correctness. First, the description states the correct state, never the failure. You write "Operations contain at least one response," and the framework derives the negative by prefixing "Failed to satisfy:". One source of truth, two readings, where I used to write the message and the check as two things that quietly drifted apart. Second, the strongest validation is the one you delete: if a type carries its own invariant, a non-empty collection, a smart constructor that rejects bad input at `init`, the bad state cannot be built and needs no check at all.

And the part with teeth, the one I now miss in every codebase that lacks it: every public type is either validated or excluded with a one-line reason, and a coverage gate in CI fails the build if someone adds a public type that is neither. "We validate our inputs" stopped being something I said in a meeting and became something the build proves on every commit.

The third turn: no singletons, not even the sanctioned one

This is the bluntest rule in the set, and the one I argued with myself about the longest. No singletons, ever. Not `static let shared`. Not even the Singleton that the Gang of Four book explicitly sanctions on page 127. The rule names that exact temptation and refuses it: when you catch yourself about to document one as "legitimate per p. 127," stop, and do the injection.

It reads as dogma until you see the reason, and the reason was the lesson. Every dependency should appear at the type's `init`, where the coupling is visible, testable, and removable. A singleton conjures its collaborator out of process-wide state at runtime, and the instant it does, that dependency is invisible in the call graph and welded to everything that touches it. The test I now hold a module to is mechanical: copy it out of the repo with only its declared dependencies and build it. If it does not lift out clean, it has a hidden dependency, and a hidden dependency is a bug that has not happened yet.

Where it landed: a machine will not give you the benefit of the doubt

Here is the thing the whole journey was actually about, the one I did not see coming. A colleague who reads "validate carefully" fills in the rest from experience. An agent does not. Vague rules produce vague work, instantly and at scale. So every rule had to become checkable: not "be rigorous" but "no claim of done without a fresh command in this response," not "handle errors" but the exact list of forbidden silencers (`try?`, force-unwrap, `xfail`, `sleep()` to make a symptom vanish).

That is what made the corrections worth writing down. Every place a rule resisted being made checkable was a place I had been getting by on the benefit of the doubt, the same benefit a machine refuses to extend. The whole set collapses into one question the agent has to answer before it calls anything finished, and writing it for the agent is how I finally started asking it of myself:

> Did I take the optimal path, or a path I am hoping no one inspects?

Honesty is one of the rules, so the limits: a rules file does not enforce itself. A model that has read "no force-unwrap" is not a force-unwrap that never ships, which is why the checks a machine can decide live in git hooks and CI and the prose only covers the judgment calls. The set is also shaped to my stack, a Swift monorepo with an OpenAPI-first workflow, and the Swift opinions are opinions. The engineering-discipline core travels anywhere; the rest you should fork and disagree with in writing.

The map, if you want it

If any of this sounds like a correction you have made more than once, take the map. Clone `rules-swift`, point your agent at it, and watch how much stops needing to be said twice. The core spine comes vendored with it; the standalone `rules-engineering-discipline` is there if you want it on its own. Star it if it saves you a repeated correction. That is the whole ask.