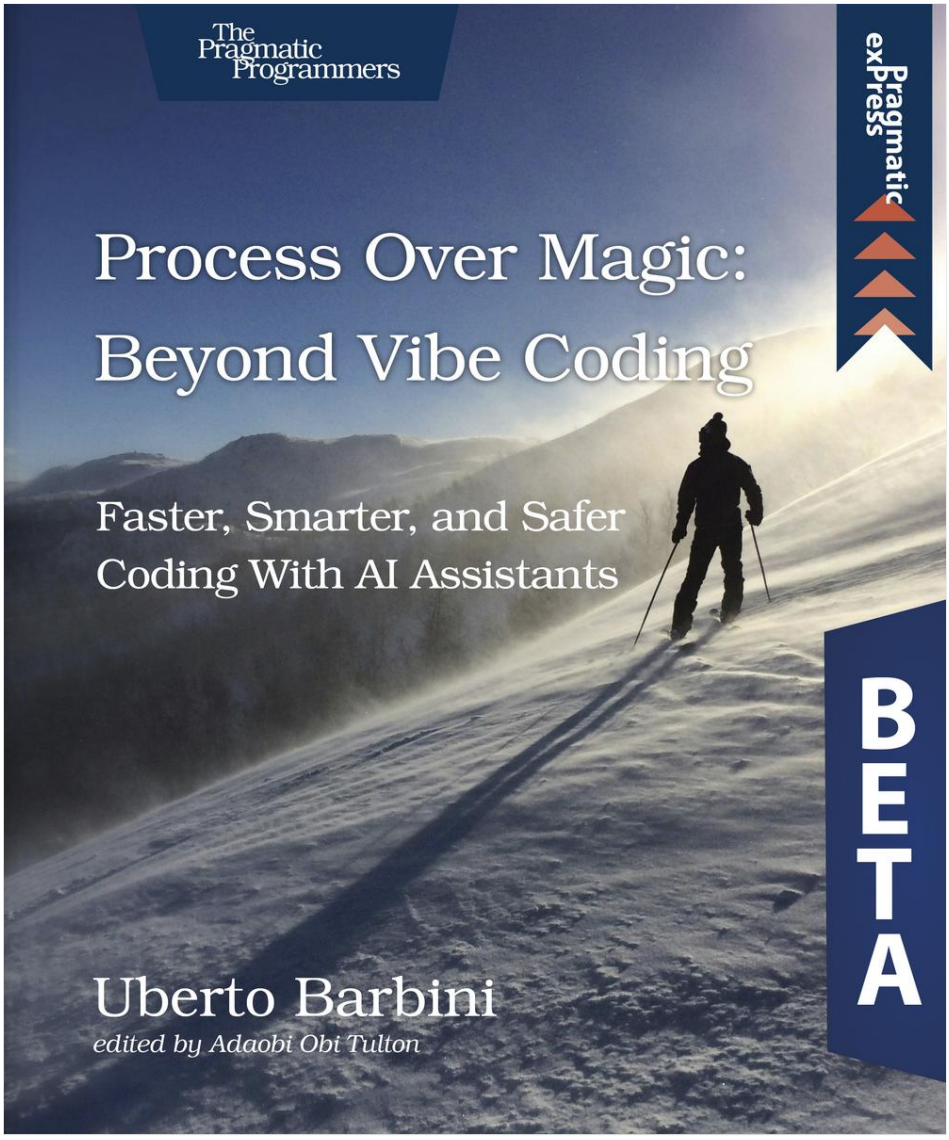

This extract shows the online version of this title, and may contain features (such as hyperlinks and colors) that are not available in the print version.

For more information, or to purchase a paperback or ebook copy, please visit https://www.pragprog.com.

An AI with Good Habits

And just like with human developers, good habits make all the difference for the LLM as well. So how do we tell the LLM which habit to follow? Well, we ask the LLM to write it down in a special document called the ruleset.

What's the Ruleset?

All LLM-based assistants allow you to define a persistent prompt that gets automatically appended to every request. This is often referred to as a ruleset because it sets clear expectations the LLM will follow throughout the session.

The conversation history itself works as a sort of temporary ruleset, since previous messages are kept in context. But defining a strong, persistent ruleset is arguably the single most important factor in getting consistent, high-quality results from your assistant, especially when generating code.

When working in a team, it becomes even more critical. Since conversation history isn't shared across developers, a clearly written ruleset ensures that everyone gets the same baseline behavior, regardless of when or how they interact with the assistant.

Many people use persona-style prompts, like:

```
"You are a senior developer and an expert in Scala and Play Framework."
```

While that can help a little, it's far more effective to spell out exactly what you expect, not just the role but how that role should behave.

For example:

```
"Provide a detailed explanation of how Play Framework handles dependency
injection, including an example. Structure your response as if you were
explaining to a senior backend engineer who is new to Play Framework."
```

This is much more useful. You're not asking the LLM to impersonate someone; it's not an actor. You're giving it clear instructions for tone, depth, and structure. And that works better.

Also, a quick reality check: Telling the LLM to "be careful," "be precise," or "don't make mistakes" doesn't help the way people often assume it will. These phrases only affect the style of the response, not the actual quality of the output. The model just sounds more confident, but it doesn't improve the quality of outcome.[1]

1. https://arxiv.org/pdf/2504.13656

Concrete, actionable rules works much better. For example:

```
Never commit code before running all tests and verifying they pass.

When refactoring, never modify test assertions, only the implementation.

Always update the README and specification file after finishing a task.
```

These kinds of instructions are easy for the model to follow, and more importantly, easy for you to verify.

One common cause of hallucination is when the assistant tries too hard to "make something work" even if the input is vague or flawed. To avoid this, your ruleset should include guidance like:

```
If instructions are unclear, ask follow-up questions before continuing.

If a request is impossible, explain why is so and suggest alternatives.
```

It's also helpful to include project-specific guidelines, like preferred technologies, coding styles, or architecture patterns. The more relevant the ruleset is to your project, the better the assistant will be at staying on track.

Finally, the ruleset should be relatively short—around 1,000 to 2,000 words—so it can easily fit within the LLM's context window without taking up too much space meant for actual content.

Your Code, Your Rules

Since I started working with AI assistants, I've gradually shaped an approach—or a kind of self-discipline, if you prefer—that I've now applied across many projects. It delivers high-quality code in very little time, and it's simple to follow. The only real requirement is to stick with it and stay focused.

I'll explain the approach here, and later in this chapter we'll see how it works in practice. But the main point is this: to get consistent results, you need to define your rules and follow them. No shortcuts, no guessing, just a clear, repeatable process that keeps the assistant aligned with what you're trying to build.

That said, the goal isn't for you to follow my rules blindly. These are just the ones that work well for me. What really matters is that you find a set of rules that fit your own workflow, team, and way of thinking. Still, I recommend starting by understanding mine, and I'll explain the reasoning behind each one as we go.

First of all, maybe you're a Test-Driven Development (TDD) fan like I am, or maybe you've never been too impressed by it. Either way, when working with an LLM, it's critical to write tests alongside the code.

Why? Because tests become the main feedback loop for the AI. Unlike human developers, the assistant can't reason abstractly or run the full application on its own. Tests are how we help it check its work and keep it honest. They let the assistant refine the output, catch most issues early, and often get things working before we even hit run.

If we let the AI write large chunks of code without tests, we're much more likely to end up with a mess. And if we wait until the end to add tests, we often discover the code is hard—or even impossible—to test without refactoring. That's wasted time. Writing the tests first, or at least together with the code, helps us avoid all that. At the end of the day it actually saves time, but we have to be specific:

- We don't want tests without real assertions
- We don't want the same assertion repeated in multiple tests
- We do want meaningful coverage that evolves with the code

Once testing is in place, we can talk about code design. For that, we can rely on the classic Four Rules of Simple Design (from Kent Beck):

1. The code works—it contains and passes the tests
2. It reveals intention—anyone reading it should understand what it's doing
3. No duplications—follow the Don't Repeat Yourself principle
4. Fewest elements—no extra abstractions, just what's needed

Next—and this is just my personal preference (feel free to use your own style)—I want the LLM to adopt a functional style whenever possible:

- Explicit over implicit—use parameters instead of private fields; prefer pure functions to class methods
- Immutable over mutable—immutable data is easier to reason about and debug
- Declarative over imperative—composing small functions is clearer than nesting conditionals and loops
- Stateless over stateful—stateful code is easier to write at first, but much harder to debug later

At the architecture level we aim for:

- Clear modularization—large applications should be broken into manageable, focused modules

- Separation of concerns—each module should have one well-defined responsibility
- Low coupling—changing one module shouldn't break everything else

And above all, we want to keep it simple: short functions, minimal dependencies, No unnecessary complexity.

These aren't strict rules; sometimes you'll need to make the less elegant choice to get things done. But as general guidelines, they're really effective. If you instruct your LLM to follow them, you'll end up with much cleaner, more maintainable code.

These rules are battle-tested and make a solid starting point. But your own codebase might have different priorities. Maybe you prefer a more object-oriented style, for example, and that's totally fine. Feel free to experiment and adjust the rules to fit your way of working.

Just make sure to clearly explain your expectations to the AI assistant. The more context it has, the better it can follow your lead.

Now we also need to add a few rules about the process itself. The LLM should:

1. Read the specification document before starting to code, and follow it carefully
2. Update the specification after coding, adding a brief log of what was done and why
3. Write and maintain a README with clear instructions on how to run the application
4. Run all tests before declaring anything as "done"

Keeping everything in Markdown documents gives us several key advantages:

- Consistency across LLM sessions—context doesn't get lost between prompts or sessions
- No need for long prompts—all the guidelines live in the docs
- Easy collaboration—multiple developers and different AI assistants can work together more seamlessly on the same codebase

Finally at the end you can add a section for your preference about specific technologies of the project—for example, using Poetry to manage dependencies in Python or using expression style for functions in Kotlin.

Here's a sample ruleset file you can use as a starting point. Keep it short and focused—you can always add more rules later. Just remember that context size is limited when working with LLMs, so every word counts.

```
### Interaction Rules

* Ask clarifying questions if input is unclear.
* Explain why and suggest alternatives if task is not feasible.
* Use structured, readable formatting (headings, lists, code blocks).
* Follow tone/structure instructions; do not simulate personas.

### Coding Standards

* Write meaningful tests with assertions for all code.
* Avoid duplicated test assertions.
* Maintain evolving test coverage.
* Apply Four Rules of Simple Design:

  1. Code works (passes tests)
  2. Reveals intent
  3. No duplication
  4. Minimal elements

* Prefer functional style:

  * Use explicit parameters
  * Prefer immutability
  * Prefer declarative over imperative
  * Minimize state

### Architecture

* Modularize by concern, not by technical layer
* One responsibility per module
* Low inter-module coupling
* Short functions, no overengineering

### Workflow

* Read `spec.md` before coding
* Update `spec.md` after task (log changes)
* Write and pass tests before finalizing
* Keep a `README.md` with setup/run info
* Store all docs/specs in Markdown

### Commit Strategy

* One prompt = one commit
* Each commit:

  * Self-contained
  * Includes tests
  * Uses 50/70 commit message format

### Safe Practices

* Do not change test assertions during refactoring
* Do not skip failing tests
```

```
* Do not invent unknown APIs; ask if you unsure

### Project Preferences (Example)

* Python: use Poetry
* Kotlin: expression-bodied functions
* JS: use ES Modules
* Follow `.editorconfig` + linter rules

### Goal

Produce consistent, safe, testable, and maintainable code.
Stick to the rules—no shortcuts.
```

To put all of this into practice, let's build an application with a clear goal defined upfront. Unlike the more exploratory projects of the previous chapter, this time we're solving a specific problem and we want to build it in a way that's easy to maintain and adapt later. Less vibing, more discipline.