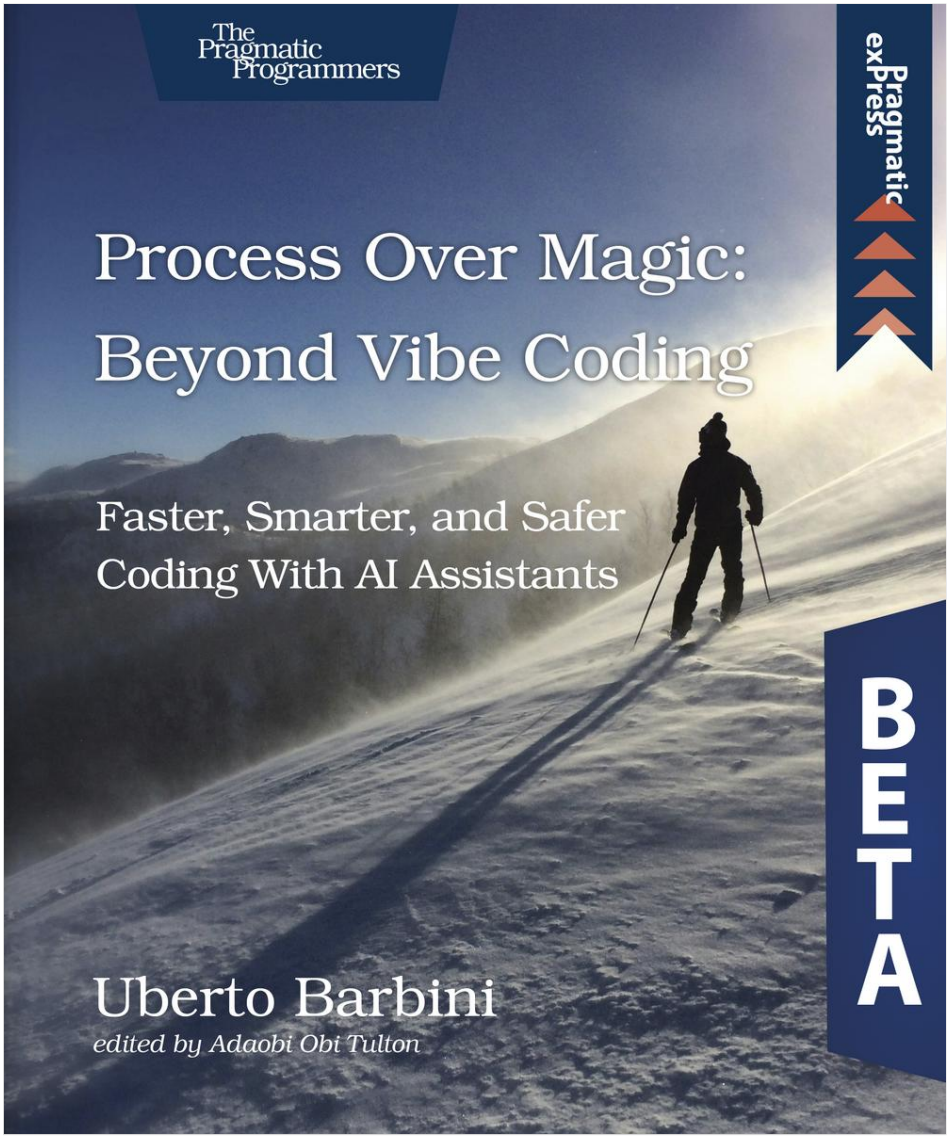


This extract shows the online version of this title, and may contain features (such as hyperlinks and colors) that are not available in the print version.

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

Enhancing Big Code Projects with AI

When working on large codebases—especially those meant to be maintained for years by teams of dozens or even hundreds of developers—we need a different mindset than when building small greenfield projects.

These systems have history, complexity, and often a fair share of technical debt. Understanding them takes time and effort, so we need to be very careful about how we use AI.

There's a quote by Alberto Brandolini that fits perfectly here: "It is the developers' understanding, not the domain experts knowledge that becomes production code."[1]

What we really don't want is production code that reflects only the LLM's understanding. That would make it legacy code from day one — because nobody would truly understand it.

AI can help us deepen our understanding, generate ideas, and speed up some tasks, but we must not delegate responsibility. It's tempting to focus on specs and let the AI handle the rest, but in the end, it's the code that runs — not our documents.

Our real value as developers lies in how well we understand both the domain and the code. No AI — at least until it becomes sentient — can replace that. Until then, we need to review every change carefully and make sure it matches our mental model of the system.

Given the complexity of big projects, we won't walk through a full end-to-end example here. That would be hard to follow unless you've already worked on the same system. Also, different projects require different techniques, and no single example could realistically demonstrate them all.

So instead of one long session, this chapter will focus on a series of practical, focused recipes—techniques that are useful when working on large, mature codebases with the help of AI. So let's dive in.

Starting Clean

Before we get into the more interesting stuff, let's start with recipe zero. It might sound like a repeat of what we've already covered, but better safe than sorry, especially when you're working in a team with other people.

1. https://x.com/ziobrando/status/634668319006683136

Don't start prompting until the code is in a clean, working state. If the project is half-edited, uncommitted, or broken in some corner, the assistant will almost certainly get confused. It might misunderstand the current state, suggest changes in the wrong places, or make everything worse.

So before doing anything else, save your work and commit whatever is working. That gives you a clear baseline and a safety net. If something goes wrong, you can always roll back.

Just as important: never commit code you haven't reviewed or aren't sure about. Even when the assistant produces something that looks fine, it still needs your judgment. One careless commit can create a chain of problems down the line.

And finally, be clear about what the assistant should touch. Tell it where to work and what to leave alone. The more precise your instructions, the better the result—and the less cleanup you'll need after.

Working in Small Steps

The first recipe is simple: when working on a large project, it's better to move in small steps. This is different from greenfield development, where we can often build things in bigger chunks. In legacy or complex systems, small, focused changes are safer and easier to manage.

At the current state of technology, we can't expect the AI assistant to understand how everything works in a large, tangled codebase. It won't magically guess the architecture or internal conventions, and even if we document them explicitly, the system might simply be too big for the model to hold in context.

That's why we need to guide the assistant precisely. Instead of asking for entire features, we ask for a specific refactor, a test for one function, or a small utility method. Keeping the scope tight leads to better suggestions and fewer surprises.

Another challenge is that we often don't know the codebase very well ourselves. That makes it easy to give the assistant impossible instructions—asking it to modify something that doesn't exist, or to write code that goes against the project's conventions. This is where the assistant's behavior becomes important. Rather than framing it as a "senior expert engineer who always knows best," it's often more effective to tell the model something like:

"You are a junior developer doing your best. If an instruction seems unclear, wrong, or unfamiliar, ask for more information before continuing."

This kind of prompt encourages the model to be cautious and collaborative. It also reduces hallucinations, as noted in a research from Anthropic,[2] where uncertainty and clarification are shown to improve output quality.

Even with clear instructions, though, LLMs still make small, beginner-level mistakes—especially in large or messy codebases. They might forget an import, leave in dead code after a refactor, or accidentally duplicate something. That's normal. In these cases, it's almost always faster to spot and fix the obvious issues yourself, rather than trying to describe the problem in detail in natural language.

Another useful trick I often use is writing pseudocode directly into the file. Just a few lines that explain what I want to happen. Then I ask the assistant to turn that into real code. This is often quicker and more precise than trying to explain everything in plain English inside the chat window.

Of course, you can use small-step instructions in smaller projects too. But in those cases, the assistant usually has an easier time figuring things out on its own, so giving it larger tasks or asking for complete features lets you move faster. As models continue to improve, they'll get better at handling bigger and more complex projects, so this balance will likely shift over time. But for now, in the messy real-world codebases most of us deal with, breaking things down is still the key to getting good results.

Documenting Behavior Through Tests

Let's take Kondor-Json as an example.[a] It's a library I wrote to handle JSON precisely in Kotlin, without relying on boilerplate DTOs just to bridge the gap between domain models and JSON schemas. It's built around functional programming principles , which you can read more about in my other book, *From Objects to Functions [Bar23]*.

In this kind of code, where correctness matters and logic can be subtle, it's easy to miss edge cases. One effective use of AI here is to ask it to generate tests for tricky inputs or scenarios we might have overlooked. The assistant doesn't need to understand the whole system, just the function and its expected behavior.

Now let's use the LLM to write a few more tests. This not only helps verify behavior but also documents how the feature is supposed to work. Well-written tests can serve as a form of living documentation, especially in parts of the code that are hard to explain in plain comments.

For this example (and some of the ones that follow), we're using JetBrains IntelliJ IDEA with Junie,[b] JetBrains' LLM assistant. It works especially well when dealing with Java and Kotlin code. Watch the video at https://youtu.be/kUAagcC6HKE.

2. https://arxiv.org/pdf/2507.21509

We start by asking Junie to check whether any test cases are missing:

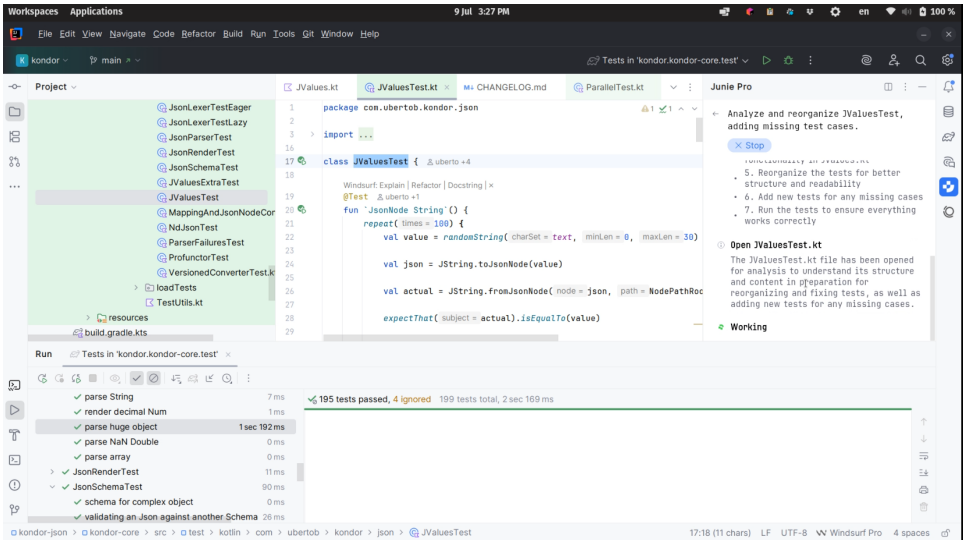
Junie prepares a step-by-step plan and starts analyzing the existing tests. It quickly spots that the converters for ‘BigInteger‘ and ‘BigDecimal‘ aren’t fully covered, and also notes the absence of tests for special ‘Double‘ values like ‘NaN‘ and ‘Infinity‘.

It then writes the missing tests, runs them, detects a small mistake in one of the assertions, fixes it, and reruns the tests automatically. Our role here is simple: review the changes (which only involve new tests) and commit the result.

---

a. https://github.com/uberto/kondor-json
b. https://www.jetbrains.com/junie/

As you can see from the image below this tedious task has been completed in just over five minutes, something that would’ve taken much longer by hand. Good job, Junie!



Understanding with Ask Mode

LLMs are great for small well-defined changes, but what if we don’t know the code well enough to even ask for specific tasks?

That’s where the second recipe comes in. We can use AI to help us learn and get familiar with a new codebase, language, or library. This approach is especially useful when we’ve just joined a project or are working in unfamiliar territory.

The idea is simple: instead of asking the assistant to write code to solve a problem, we ask to tell us how to do something without writing any code, and then we write it ourselves. In a way, it's the opposite of the previous recipe. But there's a big advantage: by typing the code ourselves, we absorb much more of what's going on around it. We understand the structure, spot conventions, and notice patterns we'd easily miss by just reading or copy-pasting.

Since we're the ones writing the code, we don't have to worry about hallucinations (hopefully!). And if something isn't clear or doesn't work, we can just ask the assistant for clarification. It becomes a patient, helpful domain expert, rather than a code generator.

Most assistants even have an Ask Mode, which answers questions without producing code and instead gives us a small document with a plan we can follow step by step.

Fixing a Tricky Bug in Kondor-Json

To put the "ask the AI" recipe into practice, I recorded a session where the AI helped me fix a tricky bug in the new version of Kondor — my JSON library for Kotlin, which we saw a few pages earlier. Watch video at https://youtu.be/HrZhIfeHTeU.

A bit of context: Kondor has a converter that specializes in translating a field with a map of key-value pairs into a raw JSON object, and vice versa, without needing to know in advance what the map contains.

In version 4.0, a major change was introduced: instead of first converting JSON into a tree of JsonNodes and then into Kotlin objects, the new parser goes directly from the raw JSON string to the target objects. This provides a big performance boost (the main reason for the 4.x rewrite), but it also required a lot of careful internal changes.

One of the trickiest parts was updating the JMap converter. I had forgotten some of its internals and was struggling to adapt it to the new architecture.

Enter the AI. Rather than asking it to "fix the code," I asked Junie (JetBrains' LLM assistant) to explain how the code needed to change. It offered a few suggestions, some of which were subtly incorrect. But even those half-right answers were helpful. Writing them down and thinking through the logic helped me see what was missing, and ultimately led me to the correct solution.

As you can see in the video, we tried several dead-end approaches before finally reaching a clean solution. That's part of the process—and the learning. Not only did I fix the bug faster than I would have on my own, I also deepened my understanding of how to convert to the new parser in the process.

This approach it's a bit like rubber duck debugging, except the duck actually talks back (even if it doesn't always get everything right). So you can use it

every time you want to not only fix the bug but also get familiar with the code and understand why it's happened.