ITNEXT · Follow publication

Up your AI Development Game with Spec-Driven Development

11 min read · 3 days ago

Mario Bittencourt Follow

In late 2025, the AI development landscape is at an interesting point. At the same time, AI in general continues to dominate the news cycle, with headlines of the massive investments by the usual suspects — Nvidia, OpenAI, AMD — and alerts of the bubble that may or may not burst in the immediate future.

At the same time, the tools and discourse around using AI for software development seem to have distanced themselves from the “developers are no longer needed” / vibe coding is the future, to more balanced approaches.

One of these approaches is specification-driven development, also referred to as requirements-driven development.

In this article, I want to cover a recent contender in the arena, SpecKit, why it can be a natural progression to your AI toolkit, and what is still missing from it.

Before we do a quick status check on current practices, don't forget to subscribe to the **Architecture Corner Newsletter** for more updates on software architecture and development practices.

AI Development

I see the development landscape with many contenders that can be grouped into two categories:

- Vibe coding

They are focused on delivering an experience that can take a prompt, generate a full application, and even deploy it.

- AI assisted

The more “traditional” AI assistant, where you have code completion, code generation, and tool calling.

In the latter category, we find Visual Studio Code+Github Copilot, Cursor, and Windsurf as the most popular. The IDE manipulates the user interactions to add more context in the form of the current and surrounding files, diffs, and custom instructions.

I covered the usual ways to interact with them and how to achieve the best results in a previous article.

Suppose you are at the beginning of your AI-assisted development. In that case, I recommend following the traditional approach so you can make yourself comfortable with using AI and see where it falls short of your expectations.

Life is Good... But it Can Be Better!

Once you have applied all the techniques, your next move, like many, is to invest in having a set of custom instruction files that aim to capture your development process. See this as one example, where it tries to lay down rules the AI assistant must follow when generating code.

This is an attempt to generate a more predictable process and, even if not directly, try to enable a harmonized approach across the members of your development team.

With that in mind, new contenders, like Kiro, propose to make the process a first-class citizen part of the experience. Without depending on custom instructions, it makes the requirements (specification) a key part that is produced and maintained throughout the developer interaction.

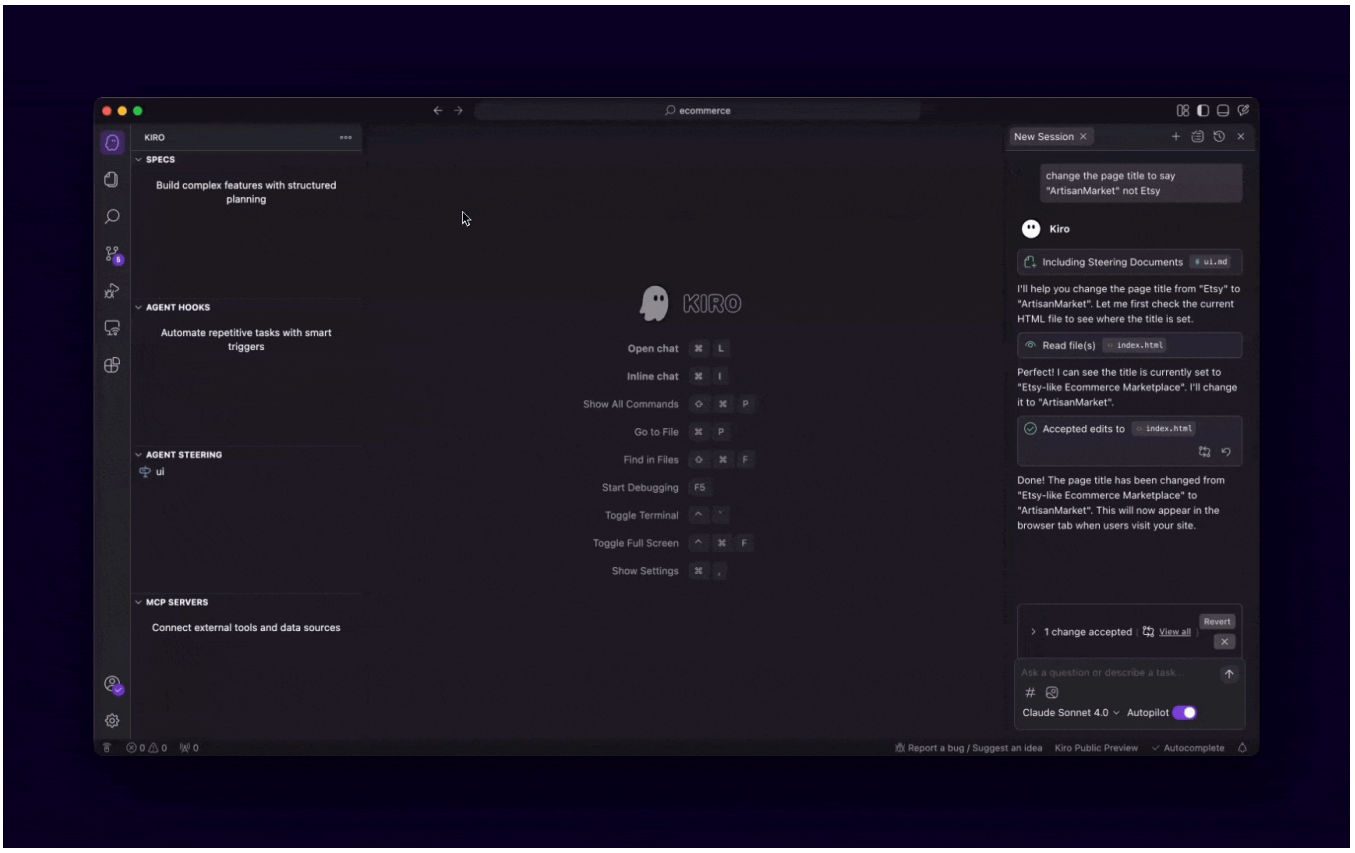

Figure 1. Kiro IDE with the specification being at the center. Source Kiro.dev

It essentially follows a Requirements → Design → Implementation approach

Figure 2. Kiro's workflow and commonly generated artifacts.

Because this process is integrated with the IDE and the generated artifacts and part of your project, they can be source-controlled and shared, which can meet us in the

sweet spot: keeping the human in the loop, having the benefits of code generation, governance of what is generated, and being available as new features are added.

Meet SpecKit

SpecKit is an open source alternative to the Spec Driven approach that Kiro presented. While Kiro offers its capabilities as a fork of Visual Studio Code + proprietary solution, SpecKit approaches it differently.

It is a series of custom prompt files and supporting shell scripts. Together, it proposes a workflow with some mandatory and optional steps you should follow.

Figure 3. SpecKit general flow. Circular nature indicates we can do multiple iterations.

Constitution — enables you to define a series of mandatory conditions your project must follow.

Specification — enables you to define the functional requirements of the application/feature you want to develop.

Clarification — [OPTIONAL] enables you to address unclear items from your specification

Plan — performs an analysis of the specification and constitution, and then produces artifacts like the research, data model, API details, etc

Checklist — [OPTIONAL] creates quality checklists that validate requirements completeness, clarity, and consistency

Task — break down the plan into smaller tasks

Analyze — [OPTIONAL] looks for inconsistencies between the generated artifacts and enables you to adjust them

Implement — does the code generation

So let's see a sample process.

Installation

You can use SpecKit with both greenfield and brownfield projects, which means you can start from scratch or add to an existing project :)

It offers two main options for installing it:

- Using uv

uv tool install specify-cli — from git+https://github.com/github/spec-kit.git

- Using uvx

uvx — from git+https://github.com/github/spec-kit.git specify init PROJECT_NAME (for new projects) or — *here* (for existing ones)

The recommended one is via the UV tool, as it is installed once and available everywhere without the need to reach GitHub servers every time.

Once it is installed, you can use it to "initialize" your project

Figure 4. Installing SpecKit

You can indicate what tool you want to use, and if you want to install PowerShell or bash scripts

Figure 5. Choosing the shell script format to be used.

And then it begins the installation

Figure 6. Adding the prompts, scripts and templates to the project.

If you open your project, you will see it has created two folders: .github and .specify

The *.github* will contain prompt files representing the commands it will offer us.

Figure 7. The new commands are just custom prompts added.

The *.specify* contains the artifacts, scripts and templates that SpecKit will use.

Figure 8. The templates for most artifacts and helper scripts that are triggered by the commands

No other changes to your system took place, and since all the files are within the project, they can be added to the source version control. They are regular shell or .md files, so you can even further customize them if you want to.

Constitution

SpecKit calls these the non-negotiable aspects that your application will have. They are the closest thing to the copilot-instructions.md that you may already have.

I added the coding standards, folder structure, specific libraries I use, and definitions for my tests.

Figure 9. Sample constitution defining our standards and non-negotiable project-wide expectations.

It will then start working and update the constitution.md, plan and task templates.

Figure 10. Updated constitution and templates.

If you open the *constitution.md,* you can see it even tracks the changes.

Figure 11. The constitution has a changelog indicating what changed.

The beauty is that you do not need to think of all your non-negotiables in one shot. You can send additional constitution commands.

Figure 12. You can iterate to improve the constitution.

It would update the constitution further to follow the newly provided information.

Figure 13. Changes are indicated so you know recent changes.

Once you are ready, you can move to the next stage.

Specification

The specification step is where you should describe the feature you are trying to develop. It is supposed to focus mainly on the what and why you are building, not the how.

For example, let's imagine we have a movie tracking application we want to develop, it would contain the following (showing just the beginning)

Figure 14. Adding the specification of my application/features.

It would create a new branch and a spec folder to contain your specification

Figure 15. A new git branch is created to host the specification (and future artifacts).

This is important because it enables you to experiment and develop the feature outside the main branch. After running the specification, it would contain what you requested, including user stories and requirements

Figure 16. Fragment of the generated specification, showing some of the functional requirements.

Like the constitution step, you can run the specify command on multiple items if you notice that a requirement is missing.

Clarification

So far, we have built/updated our constitution and added the specification of the features we want to develop. Before we can proceed, one recommended step is to look for any ambiguities that need clarification.

The clarify command assesses the constitution and specification and provides a list of 5 questions to address any items.

Figure 17. First of the 5 questions to help resolve unclear requirements.

You choose among the options or provide a short answer, and it updates the artifacts based on your answer.

Plan

With the constitution and specifications reviewed, you are ready to start the planning phase, where the model will take both into account and perform some research to propose the implementation plan.

Figure 18. Starting to plan the implementation from research and other decisions (data model, api contracts, etc).

The actual number and type of artifacts it will create may vary, depending on the type of application, but in our example, it generated the following files:

Figure 19. Generated artifacts from the plan phase

For example, the *data-model.md* contains its proposal of how the entities will be represented in the persistence layer.

Figure 20. The data model proposed to match our specification.

Checklist

This is the newest addition to the SpecKit toolchain and offers you the ability to create a quality checklist for topics of interest, such as UX, Security, API, etc.

When invoked, it will ask you 3 questions on the topic you selected first. In the example below, I selected security as the topic.

Figure 21. Questions to help define what aspects of the selected topic the checklist will contain. Varies according to the topic.

After you provide your answers, it will generate, under the checklists folder, one or more files that contain the aspects it should verify. In our case, it looks like this

Figure 21. The items from our security checklist that should be solved before moving on.

Like the preceding steps, it can be called multiple times with different topics to create focused checklists. This step is optional.

Tasks

By now, we have iterated with the constitution, specification, and from those, done the research to come up with your implementation plans/decisions. Before we jump into the implementation, we need to break the work into smaller phases.

This approach helps both the LLM to have specific and focused instructions and for you as a way to follow and review the code as it is generated.

Figure 22. Task generation broke the plan into user stories and tasks to be done.

And a sample of it looks like the list below, with tasks marked in [P] as parallel opportunities, in case your tooling allows.

Figure 23. The individual tasks with their parallel support are tagged.

Analyze

Still with me? :) The final step before the actual implementation takes place is an optional validation check. We have generated several artifacts, and not all of them

may be in sync.

The analyze command aims to address this by cross-referencing and offering you a “final” opportunity to address any divergences before the implementation kicks in.

You can check a snippet of the one below

Figure 24. Cross-referencing the constitution, specification and implementation tasks leads to finding discrepancies.

The idea is that you should address the ones with the highest severity ahead of the implementation to make sure they get done the way you expect — or as close as possible.

I instructed it to solve the critical ones, and the results were the following

Figure 25. After addressing all critical issues, we can move on.

Implementation

That's it, the time has finally come to start the implementation.

It will start by looking at any of the checklists you have created and tell you if there are open items

Figure 26. When the implementation command runs, it first alerts you of any unsolved checklists.

You can choose to continue as is or address them before moving forward. If you choose to continue, the process will begin by following the phases and user stories created in the tasks.md. Now it is a matter of following the process and deciding on which approach to take: accept the files as is, review them one by one as they progress through the tasks, or just at the end.

As it progresses, it will mark the tasks that it has executed

Figure 27. Implementation progress via the tasks

And voilà

Figure 28. The end result.

If you want to see the code it generated, check this repository.

My Review

After this journey, you must wonder what my conclusion is on Spec Driven Development, and more specifically, in SpecKit. The TLDR is definitely **positive**, at least for me, but let me detail why and what aspects I learned as I used it in both green and brownfield projects.

SpecKit proposes a very detailed AI-assisted development process. It leverages helper scripts and LLM with custom prompts to take your input, be it the *constitution* or *specification*, to expand in ways that will probably hit close to what you would want for most domains.

Additional sanity/quality checks in the form of the *clarify* and *analyze* do a great job of finding those ambiguities that you would normally skip, or worse, see them when you start coding and have to ask someone mid-activity, or potentially be blocked by the lack of a response.

Implementation went well for the most part, with the progress through the phases happening one at a time and for bigger phases even into smaller increments. For example, in my application, I saw Phase 3 broken into four implementation cycles, the first with 45% complete, the second with 60%, then 69% and the rest in a final one.

All in all, it was able to take and deliver **a working solution** that followed the specification at a faster pace than I could have done if I had to type all the code. And because it used the constitution, the generated code was close to the style I wanted, with some exceptions. For example, it created the interfaces following an *ISomething* pattern, which is not something I instructed, but at the same time, not something I explicitly told how to do beforehand.

Another issue is that it did not create a login/logout support. I assumed it would be given the mention of authentication in the Constitution. The reminder here is to be explicit with the expectations. The models suggest things you haven't explicitly asked, but will not always fill in the blanks.

Working in a brownfield was better than expected for one reason: **smaller scope**. If you check my example project, you will see that the number of phases and tasks it had was significant, and in some cases, the number of created/updated files exceeded 30. That made reviewing the files before pressing “Keep” more challenging and tiresome.

The recommendation here could be to stick to smaller features, which will increase the chances of the human in the loop actually checking for the code as it gets generated instead of at the end.

Failing to do so may be similar to joining a new team and having to learn the domain by reading the entire code for the first time. Possible? Yes, but not necessarily the most enjoyable experience. And this leads me to one final aspect that may move you away from this approach: the process itself.

To adopt SpecKit means following the process, with 5–8 steps. Some steps are meant to be executed more than once. That can be a tedious process for some and overkill for true “vibe coding” followers.

If you and your projects do not fall under the previously mentioned category, then SpecKit has the chance to bring true benefits to your productivity.

You want to know more or want support adopting GenAI, reach out here.

Until the next time!

. . .

Want to check this article in a podcast format? Check the link below

Generative Ai Tools | Software Development | Software Architecture | Spec Kit

Follow

Written by Mario Bittencourt

1.7K followers · 22 following

Principal Software Architect

Responses (1)

Bgerby

What are your thoughts?

Jochen Froelke
12 hours ago

great article. When to Provide the Big Picture to AI?

I keep wondering about this:
When working with AI, should I provide the big picture — what the overall solution should be able to do — or should I just describe single features to avoid... more

1 Reply

More from Mario Bittencourt and ITNEXT

In ITNEXT by Mario Bittencourt

Software Estimations and Agile—Friends or Foes?

Software estimations and agile have been under heavy criticism for a while, so nothing better than putting them together in the same...

Sep 22 360 4

In ITNEXT by Max Silva

The Real Cost of Server-Side Rendering: Breaking Down the Myths

There's a growing narrative in the web development community that Server-Side Rendering (SSR) is nothing more than an expensive burden on...

Oct 5 33 1

In ITNEXT by Animesh Gaitonde

System Design: Building TikTok-Style Video Feed for 100 Million Users

A deep dive into architecture, scalability, and real-time data delivery at scale

May 20 764 21

In SSENSE-TECH by Mario Bittencourt

A Look at the Functional Core and Imperative Shell Pattern

If you come from an enterprise background, you have most likely been working with Java or C# and automatically associate the analysis and...

May 19, 2023 223 1

See all from Mario Bittencourt

See all from ITNEXT

Recommended from Medium

Julien Reichel

How I Built a Multi-Model AI Testing App in 4 Days (with Copilot)

A behind-the-scenes build story of how I rapidly created a multi-model AI testing app using GitHub Copilot as my co-developer.

Oct 16 86 1

Dave Patten

Spec-Driven Development: Designing Before You Code (Again)

AI has changed how we build software - developers now code alongside assistants that analyze repos, write tests, and design architectures...

Oct 13 51 1

In ITNEXT by Alexandre Daubois

Beyond the Hype: The Actual Tech Stack Building SpaceX's Starship

I looked at SpaceX job ads. I wanted to find the code that builds Starship. What I found was smart and practical.

1d ago 17

In Netflix TechBlog by Netflix Technology Blog

How and Why Netflix Built a Real-Time Distributed Graph: Part 1—Ingesting and Processing Data...

Authors: Adrian Taruc and James Dalton

6d ago 626 14

In JavaScript in Plain English by Marcin Krasowski

Leveraging AI to speed up UI documentation in Storybook

The silent debt of undocumented components sooner or later occurs in many development projects. What begins as a few simple UI elements...

Oct 16 69 1

Bhavyansh

Why I Threw Away My Service Layer and My Code Got Simpler

After years of building “clean architecture” systems with service layers, I tried something radical: I didn’t build one.

4d ago 34 9

See more recommendations