Spec-driven development with AI: Get started with a new open source toolkit

github.blog/ai-and-ml/generative-ai/spec-driven-development-with-ai-get-started-with-a-new-open-source-toolkit

Den Delimarsky September 2, 2025

As coding agents have grown more powerful, a pattern has emerged: you describe your goal, get a block of code back, and often… it looks right, but doesn't quite work. This "vibe-coding" approach can be great for quick prototypes, but less reliable when building serious, mission-critical applications or working with existing codebases.

Sometimes the code doesn't compile. Sometimes it solves part of the problem but misses the actual intent. The stack or architecture may not be what you'd choose.

The issue isn't the coding agent's coding ability, but our approach. We treat coding agents like search engines when we should be treating them more like literal-minded pair programmers. They excel at pattern recognition but still need unambiguous instructions.

That's why we're rethinking specifications — not as static documents, but as living, executable artifacts that evolve with the project. Specs become the shared source of truth. When something doesn't make sense, you go back to the spec; when a project grows complex, you refine it; when tasks feel too large, you break them down.

Spec Kit, our new open sourced toolkit for spec-driven development, provides a structured process to bring spec-driven development to your coding agent workflows with tools including GitHub Copilot, Claude Code, and Gemini CLI.

What is the spec-driven process with Spec Kit?

Spec Kit makes your specification the center of your engineering process. Instead of writing a spec and setting it aside, the spec drives the implementation, checklists, and task breakdowns. Your primary role is to steer; the coding agent does the bulk of the writing.

It works in four phases with clear checkpoints. But here's the key insight: each phase has a specific job, and you don't move to the next one until the current task is fully validated.

Here's how the process breaks down:

1. **Specify**: You provide a high-level description of what you're building and why, and the coding agent generates a detailed specification. This isn't about technical stacks or app design. It's about user journeys, experiences, and what success looks like. Who will use this? What problem does it solve for them? How will they interact with it? What outcomes matter? Think of it as mapping the user experience you want to create, and letting the coding agent flesh out the details. Crucially, this becomes a living artifact that evolves as you learn more about your users and their needs.

2. **Plan**: Now you get technical. In this phase, you provide the coding agent with your desired stack, architecture, and constraints, and the coding agent generates a comprehensive technical plan. If your company standardizes on certain technologies, this is where you say so. If you're integrating with legacy systems, have compliance requirements, or have performance targets you need to hit … all of that goes here. You can also ask for multiple plan variations to compare and contrast different approaches. If you make your internal docs available to the coding agent, it can integrate your architectural patterns and standards directly into the plan. After all, a coding agent needs to understand the rules of the game before it starts playing.

3. **Tasks**: The coding agent takes the spec and the plan and breaks them down into actual work. It generates small, reviewable chunks that each solve a specific piece of the puzzle. Each task should be something you can implement and test in isolation; this is crucial because it gives the coding agent a way to validate its work and stay on track, almost like a test-driven development process for your AI agent. Instead of "build authentication," you get concrete tasks like "create a user registration endpoint that validates email format."

4. **Implement**: Your coding agent tackles the tasks one by one (or in parallel, where applicable). But here's what's different: instead of reviewing thousand-line code dumps, you, the developer, review focused changes that solve specific problems. The coding agent knows what it's supposed to build because the specification told it. It knows how to build it because the plan told it. And it knows exactly what to work on because the task told it.

Crucially, your role isn't just to steer. It's to verify. At each phase, you reflect and refine. Does the spec capture what you actually want to build? Does the plan account for real-world constraints? Are there omissions or edge cases the AI missed? The process builds in explicit checkpoints for you to critique what's been generated, spot gaps, and course correct before moving forward. The AI generates the artifacts; you ensure they're right.

How to use Spec Kit in your agentic workflows

Spec Kit works with coding agents like GitHub Copilot, Claude Code, and Gemini CLI. The key is to use a series of simple commands to steer the coding agent, which then does the hard work of generating the artifacts for you.

Setting it up is straightforward. First, install the `specify` command-line tool. This tool initializes your project and sets up the necessary structure.

```
uvx --from git+https://github.com/github/spec-kit.git specify init <PROJECT_NAME>
```

Once your project is initialized, use the `/specify` command to provide a high-level prompt, and the coding agent generates the full spec. Focus on the “what” and “why” of your project, not the technical details.

Next, use the `/plan` command to steer the coding agent to create a technical implementation plan. Here, you provide the high-level technical direction, and the coding agent will generate a detailed plan that respects your architecture and constraints.

Finally, use the `/tasks` command to make the coding agent break down the specification and plan into a list of actionable tasks. Your coding agent will then use this list to implement the project requirements.

This structured workflow turns vague prompts into clear intent that coding agents can reliably execute.

But why does this approach succeed where vague prompting fails?

Why this works

This approach succeeds where “just prompting the AI” fails due to a basic truth about how language models work: they’re exceptional at pattern completion, but not at mind reading. A vague prompt like “add photo sharing to my app” forces the model to guess at potentially thousands of unstated requirements. The AI will make reasonable assumptions, and some will be wrong (and you often won’t discover which aren’t quite right until deep into your implementation).

By contrast, providing a clear specification up front, along with a technical plan and focused tasks, gives the coding agent more clarity, improving its overall efficacy. Instead of guessing at your needs, it knows what to build, how to build it, and in what sequence.

This is why the approach works across different technology stacks. Whether you’re building in Python, JavaScript, or Go, the fundamental challenge is the same: translating your intent into working code. The specification captures the intent clearly, the plan translates it into

technical decisions, the tasks break it into implementable pieces, and your AI coding agent handles the actual coding.

For larger organizations, this solves another critical problem: Where do you put all your requirements around security policies, compliance rules, design system constraints, and integration needs? Often, these things either live in someone's head, are buried in a wiki that nobody reads, or are scattered across Slack conversations that are impossible to find later.

With Spec Kit, all of that stuff goes in the specification and the plan, where the AI can actually use it. Your security requirements aren't afterthoughts; they're baked into the spec from day one. And your design system isn't something you bolt on later. It's part of the technical plan that guides implementation.

The iterative nature of this approach is what gives it power. Where traditional development locks you into early decisions, spec-driven makes changing course simple: just update the spec, regenerate the plan, and let the coding agent handle the rest.

3 places this approach works really well

Spec-driven development is especially useful in three scenarios:

1. **Greenfield (zero-to-one)**: When you're starting a new project, it's tempting to just start coding. But a small amount of upfront work to create a spec and a plan ensures the AI builds what you actually intend, not just a generic solution based on common patterns.
2. **Feature work in existing systems (N-to-N+1)**: This is where spec-driven development is most powerful. Adding features to a complex, existing codebase is hard. By creating a spec for the new feature, you force clarity on how it should interact with the existing system. The plan then encodes the architectural constraints, ensuring the new code feels native to the project instead of a bolted-on addition. This makes ongoing development faster and safer. To make this work, advanced context engineering practices might be needed — we'll cover those separately.
3. **Legacy modernization**: When you need to rebuild a legacy system, the original intent is often lost to time. With the spec-driven development process offered in Spec Kit, you can capture the essential business logic in a modern spec, design a fresh architecture in the plan, and then let the AI rebuild the system from the ground up, without carrying forward inherited technical debt.

The core benefit is separating the stable "what" from the flexible "how," enabling iterative development without expensive rewrites. This allows you to build multiple versions and experiment quickly.

Where we're headed

We're moving from "code is the source of truth" to "intent is the source of truth." With AI the specification becomes the source of truth and determines what gets built.

This isn't because documentation became more important. It's because AI makes specifications executable. When your spec turns into working code automatically, it determines what gets built.

Spec Kit is our experiment in making that transition real. We open sourced it because this approach is bigger than any one tool or company. The real innovation is the process. There is more here that we'll cover soon, specifically around how you can combine spec-driven development practices with context engineering to build more advanced capabilities in your AI toolkit.

And we'd love to hear how it works for you and what we can improve! If you're building with spec-driven patterns, share your experience with us. We're particularly curious about:

- **Making the workflow more engaging and usable**: Reading walls of text can be tedious. How do we make this process genuinely enjoyable?
- **Possible VS Code integrations**: We're exploring ways to bring this workflow directly into VS Code. What would feel most natural to you?
- **Comparing and diffing multiple implementations**: Iterating and diffing between implementations opens up creative possibilities. What would be most valuable here?
- **Managing specs and tasks at scale in your organization**: Managing lots of Markdown files can get overwhelming. What would help you stay organized and focused?

We're excited to see you leverage AI to figure out better ways to translate human creativity into working software.

Get started with Spec Kit >