Spec Driven Development

AI Native Software Engineering

Kevin Ryan

7c5d0c7 · 2026.03.09

Move beyond ad-hoc prompting to structured workflows where the spec is the source of truth and code follows.

Spec Driven Development

Spec Driven Development

AI Native Software Engineering

Kevin Ryan

First Edition, 2026

Licensing — What You Can and Can't Do

This project uses three licenses working together. Here's what that means in practice.

The Book Content (text, diagrams, examples)

License: CC BY-NC-ND 4.0 + DEL v1.0

You are free to read, share, and reference the book, quote excerpts with attribution, use it for personal learning and education, and link to it from any context.

AI systems may retrieve and cite the content in responses (with attribution), quote limited excerpts ($\leq$250 words or 10%, whichever is smaller), and reference it in RAG and search systems.

You may not modify the text or create derivative works, use it for commercial purposes, train AI/ML models on the content without express written permission from the author, or republish or redistribute modified versions.

The Code and Tooling (build pipeline, scripts, templates)

License: MIT + DEL v1.0

You are free to use, copy, modify, and distribute the code, use it commercially, and build your own projects with it.

AI systems may train on the code, retrieve and reference it, and fine-tune models using it.

The only requirement is attribution — keep the copyright notice and license text intact.

Getting Permission

If you want to do something these licenses don't cover — including AI training on the book content — contact the author directly.

Spec Driven Development: AI Native Software Engineering

First edition, 2026

Build 7c5d0c7 · 2026.03.09

Contents

It's turtles all the way down

— Attributed to everyone, owned by no one.

Author's Note

Fair warning: you're reading an early beta. Chapters are incomplete, some exist only as outlines, and the content changes with every commit. If something reads like it was written at midnight with too much coffee — it probably was. Raise a ticket and let me know. Treat this as a living document, not a finished book.

This book is written using the methodology it describes. The specifications, the toolchain, the CI/CD pipeline, the provenance tracking — all of it lives in the same repository as the prose. Every chapter is generated from a spec. When something doesn't work, I fix the spec, not the output. The book is its own case study, iteration is part of the process.

I'm building this in the open because specification-driven development gets better through practice and shared experience. The methodology is still young and I don't have all the answers. Nobody does yet. That's exactly why I want to hear from you — especially if you've got:

- Case studies or war stories from working with AI coding agents in the real world
- Experience practising specification-driven development on your own projects, even rough early attempts
- Technical feedback on the book's content, structure, or accuracy — including the parts you think are wrong

Contributing works the same way as any open-source project. The repository is on GitHub. Open an issue, submit a pull request, or just reach out directly:

- Email: kevin@kevinryan.io
- Website: kevinryan.io

Anyone who contributes direct effort to this book gets credited as a contributor. Given enough eyeballs, all bugs are shallow — and that includes the specs.

Kevin Ryan
February 2026

Preface

Something fundamental is changing in how we build software, and it's moving fast.

In February 2026, Spotify's co-CEO told investors that the company's best developers hadn't written a single line of code since December. Engineers specify what they want in natural language from Slack on their phones, AI generates the implementation, and they review and merge before they reach the office. Spotify shipped over fifty new features in 2025 using this approach. At Anthropic, the head of Claude Code hasn't written code by hand in over two months — shipping dozens of pull requests a day, every one generated by AI. The tool is now substantially writing itself.

This isn't the future. It's the present. And most of us don't have a methodology for it.

AI coding tools have gone from curiosity to daily driver in the space of a couple of years. Most of us are still figuring out what works. We prompt, we iterate, we get something that mostly functions, and we move on. The industry has started calling it "vibe coding" — half joke, half admission that we're making it up as we go.

Specification Driven Development is an attempt to change that. The core idea is simple: instead of prompting and fixing, you write specifications. Instead of maintaining code, you maintain the specs that generate it. The specification becomes the artifact. Code becomes a side effect.

This isn't a contrarian position any more. It's what the industry's most advanced practitioners are converging on independently. Tessl raised $125 million on the thesis that software development should be "spec-centric, not code-centric." NVIDIA's Jensen Huang wants his 30,000 engineers to stop writing code entirely and focus on problem discovery. Anthropic has shifted hiring from language specialists to generalists, because — in their words — "the model can fill in the details." The pattern is the same everywhere: the specification is becoming the primary engineering contribution.

I'm writing this book now — early, incomplete, in the open — because this methodology is emerging in real time and I want to capture it while the lessons are fresh. SDD isn't mine to gatekeep. The ideas in this book draw on work happening across the industry, from Thoughtworks to Microsoft to Tessl to individual practitioners who are discovering the same patterns independently. The sooner we have a shared language and a shared set of practices, the sooner we all benefit.

My hope is that this book becomes a catalyst. A starting point for collaboration, not the final word. The repository is public. The specs are visible. The methodology is documented in the same repo that uses it. If you disagree with something, open an issue. If you've found a better pattern, contribute it. If you've hit a wall that SDD doesn't address, that's exactly the kind of problem I want to hear about.

These technologies are transformative. How we adapt to them matters — not just for individual productivity, but for the engineering community as a whole. I want us to handle this transition well. I want the practices we develop now to be ones we're proud of in five years.

I also just find the subject fascinating. I want to be in the thick of it with my sleeves rolled up, building things, breaking things, and writing down what I learn. If that sounds like your kind of project, you're welcome here.

Kevin Ryan
February 2026

Part I

Foundation

Chapter 1

The Fifth Generation

Programming has always been about abstraction. Each generation moves further from the machine and closer to human intent.

First came machine code—raw binary instructions for the processor. Then assembly language gave those instructions human-readable mnemonics. High-level languages like FORTRAN and COBOL abstracted away the registers and memory addresses. Object-oriented programming abstracted away the procedures. Each step let programmers express more with less, trading direct control for leverage.

We're now entering the fifth generation. The abstraction is no longer syntax or structure—it's intent itself.

1.1 The Abstraction Ladder

Consider what happened at each level:

Machine code required you to think like a processor. Every operation explicit, every memory address managed by hand. A simple loop might take dozens of instructions.

Assembly gave you names. `MOV`, `ADD`, `JMP`. Still one-to-one with machine operations, but readable. You could finally share code with other humans.

High-level languages gave you constructs. Loops, functions, variables with types. The compiler handled the translation. You stopped thinking about registers.

Object-oriented and functional paradigms gave you composition. Classes, modules, interfaces. You stopped thinking about memory layout and started thinking about relationships.

Each transition felt like a loss to some programmers. Assembly programmers distrusted compilers. C programmers distrusted garbage collection. The pattern repeats: what feels like giving up control is actually gaining leverage.

1.2 The Fifth Generation Shift

The fifth generation abstracts away the code itself.

This sounds radical, but it follows the same pattern. You describe what you want—the specification—and the system generates the implementation. The specification becomes the artifact you maintain. The code becomes ephemeral, regenerated as needed.

This isn't new in concept. Code generation has existed for decades. What's new is the capability. Large language models can generate working code from natural language descriptions, handle ambiguity, and adapt to context in ways that templated code generators never could.

But capability without methodology is chaos. That's where most developers are stuck right now.

1.3 The Current Mess

Watch a developer using an AI coding assistant today. They type a prompt, get some code, paste it in, run it, see an error, type another prompt, get more code, paste that in. It works, sort of. They move on.

This is vibe coding. It produces working software through iteration and luck. It doesn't scale. It doesn't transfer. It leaves no trail of intent.

The problem isn't the AI. The problem is that we're using a fifth-generation tool with second-generation thinking. We're still writing code—we're just dictating it to a machine that types faster than we do.

1.4 Specification as Artifact

The shift requires inverting how we think about what we produce.

In traditional development, code is the artifact. Requirements, designs, and documentation exist to support the code. When they drift apart, we update the documents (or more often, we don't).

In specification-driven development, the specification is the artifact. Code exists to implement the specification. When they drift apart, we fix the code or refine the specification—but the specification remains the source of truth.

This changes everything: what you version control, what you review, what you test against, how you communicate with your team.

The code still matters. It still runs. It still has bugs. But it's no longer what you maintain. You maintain the specification, and the code follows.

1.5 What This Means for You

If you're a developer who's been using AI tools and feeling like something isn't quite working, you're right. You've been given a fifth-generation tool and told to use it like an autocomplete.

The rest of this book will show you how to actually use it.

You'll learn to write specifications that generate correct code on the first try—or at least, code that fails in predictable, fixable ways. You'll learn to structure projects so that specifications and code stay synchronized. You'll learn to validate generated code against the specification that produced it.

Most importantly, you'll learn to think about code as a side effect of specification, not the other way around.

This is the fifth generation. The abstraction is intent. Let's learn to use it.