PRACTICAL TECH GUIDE

Spec-Driven Development

Building Software From Clear, Executable Specs

INTENT • SCOPE • CONSTRAINTS • TESTS • REVIEW

Spec-Driven Development

Building Software From Clear, Executable Specs

Blue J. Lion
Quiet Line Press (quietlinepress.com)

First Edition: July 2026

Published by Quiet Line Press (quietlinepress.com)

Table of Contents

Introduction

Spec-Driven Development in Five Minutes

If you only have a few minutes, here is the practical short version.

Many software problems start before code.

They start when the request is loose, the scope is blurry, the edge cases are unstated, and everyone fills in the gaps differently.

That is where spec-driven development helps.

Here, a spec is not a giant document. It is a small, clear description of what should change, what should stay unchanged, how success will be checked, and what is out of scope.

A useful spec usually answers a few practical questions:

1. what problem are we solving

2. what behavior should change

3. what must stay the same

4. what is explicitly out of scope

5. how will we know the work is done

The common mistakes are also predictable:

1. treating vague tickets as if they were already buildable

2. mixing requirements, design ideas, and future wishes together

3. leaving non-goals unstated and then calling the drift a surprise

4. writing acceptance criteria that sound nice but cannot be checked

5. creating long documents that nobody uses once coding starts

What this book will help you do:

1. turn rough requests into executable specs

2. write clearer scope, constraints, and non-goals

3. use examples and edge cases to reduce ambiguity

4. connect specs to tests, review, and definition of done

5. use specs as a practical foundation for AI-assisted development

Book Positioning

This book is for software engineers, technical leads, product-minded builders, and teams who want software changes to be easier to build, review, and verify.

It stands on its own, but it also fits naturally beside `Harness Engineering` and the earlier book on `AI Skills`.

This book focuses on making the work itself clearer before implementation begins. The aim is practical: show how a small amount of clarity up front can prevent a lot of drift, rework, and argument later.

For most teams, the best fit is a spec-anchored workflow: the spec lives with the change, guides the implementation, and gets updated when the intended behavior changes.

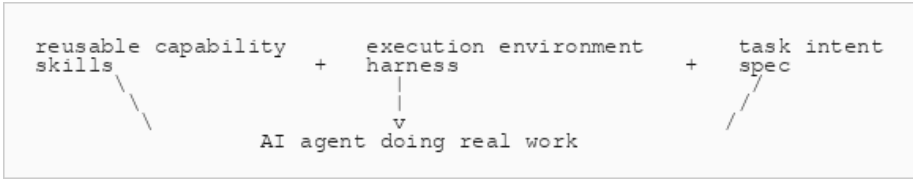
Skills, harness guides, and specs may all use Markdown artifacts, but they do different jobs.

- skills provide reusable know-how for one kind of work

- the harness shapes the environment around the work

- the spec defines what this specific change should achieve

Or as a single workflow:



That distinction matters because the files may look similar, but they guide different parts of the work:

1. what the agent knows how to do repeatedly

2. what the agent is allowed and expected to do here

3. what this change is supposed to accomplish

How The Pieces Fit Together

The simplest end-to-end shape in this book is:

request or task -> spec -> plan -> implement -> checks -> review

In this book, a task or request is the starting ask, a spec clarifies the intended change, a plan reduces implementation uncertainty, checks provide evidence, and review compares the delivered result to the spec and evidence.

Not every change needs a separate file for every step. A small task may keep some of them in one ticket, one pull request, or one conversation.

In practice, checks and review may loop a bit. Review often sends a change back for more checks, but the usual pattern is still to run checks before final review.

Companion Project

The companion project for this book is `order_rules_lab`, a small internal order and refund admin application.

It is a good fit for spec-driven development because small wording changes can change real behavior: eligibility rules, refund limits, status transitions, API fields, and reviewer expectations.

The project starts in a familiar state: rough tickets, mixed assumptions, partial tests, and a lot of behavior that people think is obvious until it is time to implement a change.

Small companion examples for it appear throughout the book:

- `docs/starting-state/order-rules-lab-starting-state.md`

- `docs/tasks/order-rules-lab-weak-ticket.md`
- `docs/specs/order-rules-lab-spec-example.md`
- `docs/specs/order-rules-lab-contract-example.md`
- `docs/plans/order-rules-lab-implementation-plan.md`
- `docs/verification/order-rules-lab-test-mapping.md`
- `docs/reviews/order-rules-lab-change-report.md`

The runnable app and these companion files belong together, so the code and the spec artifacts can be inspected side by side.

To run it locally:

git clone https://github.com/nextframedev/order_rules_lab.git
cd order_rules_lab
npm install
npm start

Then open `http://localhost:4400`.

Chapter 1. Why Teams Drift Without Specs

Many teams believe they have a spec when they really have a request.

The request may be useful. It may even be enough for a teammate who already knows the system well. But that does not make it a strong foundation for implementation.

A short line like `support partial refunds for approved orders` sounds clear until the work begins.

Does partial mean one refund ever, or multiple refunds up to a limit? Are shipping fees included? What status changes are allowed? Should the API response shape stay the same? What should happen when the order is already fully refunded?

Those are not edge concerns added later. They are part of the work.

Why Requests Drift

Drift usually does not begin because people are careless. It begins because the original request leaves too much open to interpretation.

One developer fills the gaps one way, a reviewer imagines something else, and an AI coding agent follows whichever interpretation seems most likely from the available context.

Then the team spends time debating the result after code already exists.

That gets expensive fast.

Spec-driven development tries to move more of that thinking earlier, while the change is still cheap.

Prompt-Driven And Spec-Driven

One contrast is prompt-driven work versus spec-driven work.

Prompt-driven work starts with a loosely written request and hopes the prompt will carry the missing structure.

Spec-driven work tries to add that structure before implementation starts: goal, boundaries, important behavior, non-goals, and checks.

This difference matters even more with AI coding agents. Better output often starts before the prompt.

What A Spec Changes

A useful spec does not need to describe every detail of the solution. It needs to reduce the important ambiguity.

That usually means intent should be clearer than implementation.

The spec should say what needs to be true. The code can still decide how to get there, within the stated boundaries.

In practice, that often means the spec should make a few things explicit:

1. the intended behavior change

2. the boundaries of the change

3. the important non-goals

4. the checks that will prove the result

That alone changes the quality of the whole workflow.

The implementation gets clearer. The tests get easier to choose. The review gets less subjective. The definition of done gets harder to fake.

This is one reason specs matter even more when AI coding agents are involved. A human teammate can sometimes recover from a vague request through discussion, history, and intuition. An agent usually has less of that shared context.

Weak requests do not stay weak when an agent touches them. They get amplified.

Order Rules Lab

In `order_rules_lab`, a weak ticket might say:

```
support partial refunds for approved orders
```

That is a reasonable starting request, but it is not yet a buildable spec.

It leaves open questions about refund limits, status rules, API behavior, and what should happen to existing checks.

A spec does not need to turn that into a long document. It only needs to turn it into something the team can implement and verify with less guessing.

The companion examples for this chapter are `docs/starting-state/order-rules-lab-starting-state.md` and `docs/tasks/order-rules-lab-weak-ticket.md`.

After this chapter:

1. distinguish a rough request from a usable spec

2. see why drift often starts before implementation

3. treat ambiguity as part of the work, not as something to clean up later

About the Author

Blue J. Lion has over 20+ years of experience in software development, with a focus on programming, data security, and privacy. He has worked across engineering and product environments, building practical solutions and tools.

Beyond software, he enjoys creating simple, thoughtful products-ranging from books and visual tools to creative projects that explore the intersection of technology and everyday life.

In his free time, he enjoys running, swimming, and working on new ideas.

Quiet Line Press

Author Portfolio