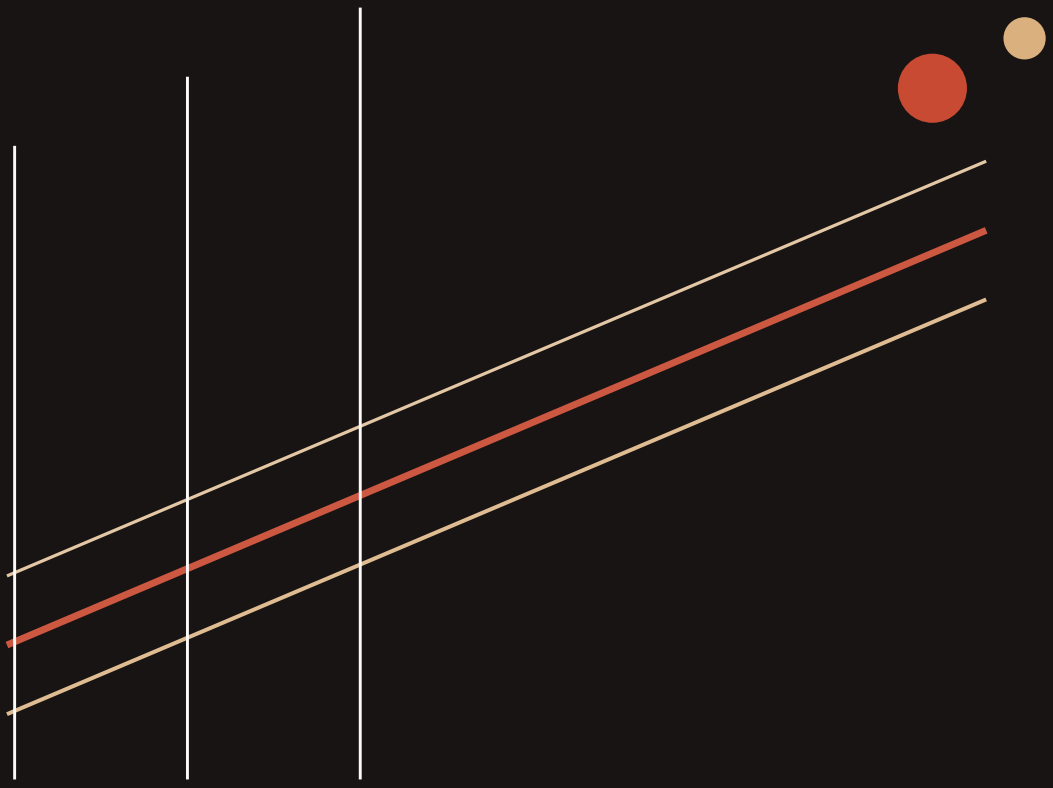
VIBE CODING IN THE AI ERA

Vibe Coding Tauri 2

Request ·Build ·Verify —Shipping six projects with AI

Request | Build | Verify | Repeat



SeungHwan Park

2026

Vibe Coding Tauri 2

Request ·Build ·Verify —Shipping six projects with AI

First edition, 2026.

Set in Apple SD Gothic Neo | Menlo.

Contents

CHAPTER 1

Preface

1.1 Why This Book

In 2026, I built four Tauri 2 apps by myself. One productivity app, one web app I sell as a service, one dashboard wired into a live trading system, and one RL & backtest lab. Alone, every one of them. Start to finish.

The thing to notice is that I am not a developer with time to spare. I have a day job. I have a child at home. One or two hours on a weekday evening and a few hours on weekends — that is all I get. In an earlier era this budget would barely cover a single app. Yet in 2026 I shipped four. One thing changed —**AI agents**. I give Claude Code the context, and during a thirty-minute window after dinner a DB schema appears. The next morning, fifteen minutes before I leave for work, a bug is fixed. The agent types. I only request and verify.

If you showed that sentence to me in 2020 I would have called it a lie. The 2020 version of me spent six months on a single app. Two weeks of design, three months of implementation, one month of bug fixing, one month of deployment. Shipping two apps a year was the ceiling. Building apps on the side while holding a job and raising kids was unthinkable.

What changed is not me. The tech stack got harder. I write Rust, attach WebSockets, embed DuckDB, and launch long-running Python processes inside Tauri. In the old days each of those pieces would have eaten weeks.

The only thing that changed is that **I no longer type the code myself**.

In the AI era, "writing code by hand"is not the goal —it is one means among many. Means can be delegated. I make a request, read the result, point at the wrong parts, and ask again. I run this loop whenever I get a gap in my day. A few months later, four apps exist. This worked **not because I had time, but because I did not have time** and the agent covered for me.

This book is the log of that loop. It is not a code walkthrough. It is **how to work with AI**.

1.2 What This Book Covers / Does Not Cover

Here is what it covers.

- How to frame a request for an AI. How to split context, constraints, goals, and format.
- How to verify what comes back. The difference between "it runs"and "it is correct."
- How to re-run the same request. How to compress the loop when one pass is not enough.
- Where AI got it right and where it got it wrong in each of the six projects. The places I had to overwrite by hand.
- My actual prompt templates, slash commands, and `CLAUDE.md` files.

Here is what it does not cover.

- A Rust syntax tutorial. I do not explain `Result<T, E>`.
- A Svelte 5 syntax tutorial. I use `$state` and `$effect` but do not teach them from scratch.
- A copy of the official Tauri 2 documentation. I touch only the minimum that the narrative needs.
- General prompt engineering. Writing, image generation, and RAG are not in this book.
- The internals of AI models. I will not compare the tokenizers of Claude and GPT.

I draw the boundary this hard for one reason. If a reader is going to finish this book and build their own app **with their own hands**, there is no room for detours.

1.3 The Book's Loop —Request, Build, Verify

The loop this book keeps running has three stages. **Request** is translating my intent into a language the AI understands. **Build** is the AI emitting code and me pasting it in and running it. **Verify** is not checking that it runs —it is checking that it is correct. Whatever Verify catches gets fed back into Request, and one lap is done. I run three to five laps per feature on average. The precise definitions and per-stage checklists live in Chapter 2.

1.4 The Four Apps and Two TUI Projects at a Glance

The six projects are deliberately picked. Each surfaces a different face of AI coding.

Mandai —a Mandal-Art (9×9 goal grid) + GTD + Pomodoro + AI coach. Data lives in a local DuckDB, and OpenAI, Claude, Gemini, and Groq sit behind a single command. The theme is **multi-provider streaming** and **stream cancellation**. The spot the AI missed most often was the one that cancels the previous stream.

readbooks.ai —a live service that translates PDFs into 36 languages. I ported it from Next.js to SvelteKit 5. This is the chapter where I tested how much of a **framework port** I could hand to the AI. Short answer: not 100%.

Trading Dashboard —a 19-screen real-time dashboard. Python FastAPI + WebSocket, an AtomicU16 proxy port, Cloudflare Tunnel, and local charts. The theme is **how to split a sizable app between yourself and the AI**.

RL & Backtest Lab —I launch Python RL training and backtests from Tauri and stream stdout. **Long-running external process control**. This is the area where the AI is weakest. I had to spell out `kill -9` in the prompt before the code handled it.

TUI Trading Monitor —a terminal-based dashboard built with `ratatui`. Real-time positions, P&L, and alerts rendered in the terminal. The theme is **how fast a TUI ships** and why it is a legitimate alternative to a GUI for tools you use yourself.

TUI Log Viewer —a terminal-based log viewer for RL training and backtest output. Filters, search, and live tailing. The theme is **TUI as the fastest path from idea to working tool**. No browser, no WebView, no bundling —`cargo run` and it works.

You do not have to read the six projects in order. Pick the one you care about —the loop looks the same everywhere.

1.5 How to Read This Book

I recommend reading Part I (Chapters 1-5) in order. It contains the premise, the loop definition, the VSCode Copilot workflow, the tool setup, and Tauri 2 fundamentals for the whole book. Skipping these five leaves the later prompt-design discussions floating in air.

Parts II through VI (Chapters 6-17) can be read by app. Each part is self-contained. If Mandai is what you care about, read only Part II. If the trading dashboard is what you care about, read only Part IV. If you want to see that terminal apps are surprisingly viable, jump to Part VI. Each part opens with "The loop pattern you will learn from this app."

The final Part VII (Chapter 18) is the chapter you come back to when applying the method to your own app. Read it once on the first pass, then again when you actually start your own app. It contains templates and failure patterns.

1.6 Full Journey Map

Part	Chapters	Theme
I	1 -5	Why AI, The Loop, Copilot Workflow, Tools, Tauri 2 Fundamentals

Part	Chapters	Theme
II	6 -9	**Mandai** —Design, DB, AI API, Pitfalls
III	10 -11	**readbooks.ai** —Prompting, Verification
IV	12 -14	**Trading Dashboard** —Scale, Real-Time, Deploy
V	15	**RL & Backtest Lab** —Injecting Domain Knowledge
VI	16 -17	**TUI Apps** —Terminal UIs That Ship Fast
VII	18	Applying This to Your Own App

1.7 Prerequisites

To get through this book, you need roughly this background.

- **Ability to read Rust and TypeScript.** You do not have to write them. If you can look at AI-generated code and explain what the ? operator does, that is enough.
- **Comfort with Git and the terminal.** If you already branch and split commits as a habit, you can drop this book's loop straight onto your workflow.
- **Hands-on experience with an AI coding tool.** You should have used at least one of Claude Code, Cursor, GitHub Copilot, or ChatGPT on something real. If you never have, spend a week on a toy project with an AI first, then come back.

On the other side, here is what you **do not** need. Tauri 1 experience, SvelteKit experience, or production Rust experience are all unnecessary. I explain what I need inside the chapters.

1.8 Development Environment

Every piece of code and every screenshot in this book was produced in the following environment.

Item	Version
OS	macOS 15.1 (Sequoia)
Rust	1.87.0
Node.js	22.11 LTS
pnpm	9.15
Tauri	2.1
SvelteKit	2.x + Svelte 5.1
DuckDB	1.1
Python	3.12 (RL Lab only)
Claude Code	latest as of Q1 2026
Cursor	0.45

Most of it runs identically on Windows and Linux. Where it does not, I flag the difference inside the chapter.

1.9 Author's Note

Everything in this book is something I actually went through. I left the loops that failed right next to the loops that worked. There are scenes where I fought with the AI three times before fixing the code myself, scenes where the AI produced plausibly wrong code and I waved it through without noticing, and scenes where I made the same mistake twice. Nothing has been polished smooth. That roughness is the raw material of this book.

Read it as the work log of someone holding a day job, raising a child, and still shipping four Tauri 2 apps as a solo developer. It is not a book written because I had time. It is a record that —even without time —AI agents made it possible. And the fifth app is yours to build.

Ted Park Spring 2026, Seoul

CHAPTER 2

Chapter 1. Why Build with AI —The App You Cannot Build Alone

2.1 Learning Objectives

- Understand why, in 2026, a developer with a day job and a child at home can still ship four desktop apps.
- Feel the productivity gap between the “type everything”era and the “delegate to AI”era, both in numbers and in lived experience.
- Know where the claim “you only need to be able to read AI-generated code”is true and where it becomes a lie.

- Accept the shift in a developer's role from "code author"to "requirements refiner plus verifier."
- Preview the six projects you will build alongside me and the capabilities you will end up with.

2.2 1.1 I Was Busy and Still Shipped Four

In 2026 I shipped four Tauri 2 desktop apps. None of them is a toy. One is the productivity app I use every day. One is sold as a service. The other two are wired into a live trading system. Alone, all four.

The key premise: I am not a developer with time to spare. I have a day job. I have a child at home. One or two hours on a weekday evening and a few hours on weekends —that is the whole budget. I am not a freelancer. I am not a full-time indie developer. I am a regular salaried employee, and parenting waits for me after work.

Under these conditions, the 2020 version of me would not have shipped four apps. Not possible.

In 2020 Tauri 1 was still alpha, SvelteKit did not yet have a name, and Rust was not a language I used. Above all, in 2020 I clung to a single project for six months at a time. One week on visuals, two weeks on feature design, three months to implement, a month to fix bugs, a month to deploy. At that pace, two apps a year was the ceiling —and that was only when I could pour in non-work hours.

In 2026, the same person **under more demanding conditions** shipped four. The stack got harder. I write Rust for the backend, handle WebSockets, embed DuckDB, and launch Python processes. Harder work, less time, more output.

Only one thing changed. **I now barely type.** The AI agent types. I toss context into Claude Code during a thirty-minute gap after dinner, and the next morning I spend fifteen minutes before work verifying the result.

2.3 1.2 Why Now —2026 as the Inflection Point

Why is it **this moment** when a busy salaried developer can ship four apps? A short tour of the technical backdrop.

- **Maturity of conversational coding tools.** Starting late 2024, Claude and GPT-4-class models began handling multi-file context reliably. By 2026 agents and tool use have landed on top, so "read the file, write the file, run the test"all happen in one session.
- **Tauri 2 GA.** Tauri 2 went stable in 2026. The Rust-backend-plus-web-frontend combination is finally production-ready.
- **SvelteKit 5 runes.** Reactive state management got a clean, language-level answer. The quality variance in AI-written frontend code dropped sharply.

All three converged in 2026. If any one of them had slipped a year or two, I would not have shipped four apps.

2.4 1.3 Development, Before: I Was the One Writing Code

I can still reconstruct my day from ten years ago. In the morning I read JIRA tickets. After lunch I opened the editor. I kept seven tabs of Stack Overflow open. I typed `useState` by hand. I fixed typos. I recited React's lifecycle from memory. By 7 p.m. I had finished a single component.

My best tool at the time was **autocomplete**. IntelliSense filled in the back half of `Array.prototype.filter` once I had typed the first half. That was the extent of it.

Ninety percent of the code passed through my fingers. Function names, variable names, type declarations, repeated if/else branches, import statements. I typed them all. What could go wrong? Everything could go wrong. The compiler caught the typos, but nobody caught a bad design.

So I **skimped on design and made up for it with typing**. The longer I paused to think, the later I shipped. "Get it running first, refactor later"was the default. Later never came.

2.5 1.4 Development, Now: The AI Writes the Code, I Decide

2026 is different.

In the morning I jot an idea in a note. "Mandal-Art goal tree + Pomodoro + AI coaching -> store in local DuckDB -> wrap it all in Tauri 2."After lunch I open the Claude Code terminal. I type the prompt.

```
Build me a Mandal-Art 9x9 grid component using Tauri 2 + SvelteKit 5.
Clicking a cell opens an edit modal. When the center cell changes,
have AI propose candidates for the empty cells around it.
Svelte 5 runes only. Break the code by file.
```

Claude returns, in three minutes, a draft of the component, the store, and a Tauri command for the AI suggestions. I read it. Two things bother me. The Svelte 5 syntax is actually Svelte 4, and the AI call runs from the frontend. I append a prompt.

```
Two fixes.
1. Use only $state, $derived, $effect. No reactive variables declared with `let`.
2. Move the AI call into a Tauri command. API keys must not be exposed to the fronte
```

Another three minutes. Passes. Commit.

A six-hour task from the old days finished in twenty minutes. What my hands actually did amounts to three things. **I expressed what I wanted in words. I read the result. I pointed at what was wrong.**

For the same Mandai project, here is a table of **what the AI nailed on the first try, what it got wrong, and what I made it fix**. A reference for readers without the feel for it.

Category	Example
Got it right	9×9 grid layout, Tailwind class composition, skeleton of keyboard navigation

Category	Example
Got it wrong	Mixed in Svelte 4 store syntax, called the AI directly from the frontend, overused `any`
Made it fix	Unify on `$state`/`$derived`, move the AI call into a Tauri command, collapse types into `MandaiError`

This three-row table repeats for every feature. You could fairly describe my day as filling that table out.

2.6 1.5 The Reality of “You Only Need to Read AI Code”

That claim is half right.

The true half: typing drills are obsolete. Memorizing syntax matters less. You do not need to have every branch of Rust’s `Result<T, E>` in your head. The AI writes, you read. If you can pause at a spot and ask “is `?` the right move here?”, that is enough.

The false half: **the bar for “being able to read”is now much higher.**

Suppose the AI hands you this.

```
#[tauri::command]
async fn get_candles(symbol: String) -> Result<Vec<Candle>,
String> {
    let conn = DB.lock().unwrap();
    let rows = conn.prepare("SELECT * FROM candles WHERE symbol
    = ?")?;
    Ok(rows.query_map([symbol], |r|
    Candle::from_row(r))?.collect())
}
```

You need to see **why this is wrong** within three seconds. `unwrap()` can panic. `?` will not auto-convert `rusqlite::Error` into `Result<_, String>`. Holding a synchronous mutex across an `async` function will stall the runtime.

A reader who **only reads** AI-generated code will miss this bug. They run it, nothing crashes, they move on. Three weeks later the production database connections are blocked and the whole app freezes.

The higher reading bar means one thing. **The traps you used to learn by hand while writing code must now be learned on purpose.** This is where I see my juniors slip most often.

2.7 1.6 Role Shift: From Code Reviewer to Requirements Refiner

In 2020 the thing I did most during the day was **write code**. In 2026 the thing I do most during the day is **write prompts**.

Writing a prompt is writing a requirement. It is not the kind of requirement a PM jots down on Confluence. A prompt to an AI must be far more **precise**. Humans accept "just figure it out." AI does not. More precisely: it looks like it does, and then three hours later it is producing something unrelated.

So every day I write these.

- **What**: what to build. A function? A component? A whole app?
- **Why**: why it matters. Where this piece fits inside the larger feature.
- **Constraints**: language, libraries, performance targets, file layout.
- **Format**: the shape of the output. Code blocks, a file list, a diff, depth of explanation.

I split every request into these four chunks. Once the rhythm is in my hands I can pack all four into a single sentence. Chapter 2 covers this as part of the loop.

The **code review** I used to do was reading someone else's PR. The **requirements refinement** I do now is organizing my own head before the AI makes anything. Reviewing is after the fact. Refining is before. That reversal is the single biggest change.

2.8 1.7 A Scene —The PDF Parser Episode from readbooks.ai

Talking in the abstract is hard to feel, so here is a short scene.

I was adding a PDF text extractor to readbooks.ai. My first prompt was this short.

```
Rust-based Tauri command that extracts PDF text.
```

Claude handed back a 20-line command using the `pdf-extract` crate. I ran it. English PDFs worked. A Korean PDF came back as **boxes of broken glyphs**.

In the old days I would have searched "rust pdf korean encoding"on Stack Overflow. Instead I brought evidence and asked again.

```
Korean PDFs come out garbled. Here is the first page of the sample input:
<30 lines of broken text>

List 3 candidate causes and a short fix for each,
then pick the most realistic one and give me the revised command.
```

Claude listed three candidates —encoding, font embedding, missing CMap —and answered: "In this case it is almost certainly a CMap issue. Pure `pdf-extract` has a ceiling here. Consider `pdfium` or shelling out to an external binary."A correct diagnosis.

What matters here is not that Claude failed on the first try. It is that I **fed the evidence (the broken output) back into the prompt** and sharpened the second request. The old me would have asked Google "how do I fix this."The current me **shows the AI exactly what I observed and builds the next hypothesis with it.**

Most of what this book teaches is in this scene. Short requests, evidence-based feedback, enumerated hypotheses, a realistic pick —and the posture of **never treating the AI as an oracle.**

2.9 1.8 The Six Projects We Will Build Together

Across this book I use the six projects I built as raw material. Each one shows a different face of working with AI.

2.9.1 1. Mandai —Productivity Tree + AI Coach

A Mandal-Art (9×9 goal grid) fused with GTD and Pomodoro. All data stacks into a local DuckDB, and four providers —OpenAI, Claude, Gemini, Groq —sit behind a single Tauri command. This app is the textbook for **the multi-provider streaming pattern**.

The spot the AI got wrong most often was stream cancellation. When the user changes the question, the previous stream must stop. At first the UI froze every time. It took three loops to fix.

2.9.2 2. readbooks.ai —Multilingual PDF Translation

Translates PDFs into 36 languages. A live service ported from Next.js to SvelteKit 5. Claude, OpenAI, Gemini, and DeepL hide behind a single `translate_with_provider` command. This app shows how to split **framework porting** and **third-party API integration** between yourself and the AI.

I assumed the Next.js -> SvelteKit port could be handed entirely to the AI. It could not. Routing structure differs, data-fetching models differ, and three big loops were required.

2.9.3 3. Trading Dashboard —A 19-Screen Real-Time Dashboard

Hooks into a Python FastAPI backend over WebSocket. AtomicU16-based proxy port management, Cloudflare Tunnel integration, and local chart rendering all live inside Tauri. This is the textbook for **splitting an app with many screens** across AI prompts.

At 19 screens, you cannot hand it all to the AI at once. I cut it one screen per section and laid the shared layout and WebSocket hook down first. That partitioning strategy is the centerpiece of Chapter 13.

2.9.4 4. RL & Backtest Lab —Long-Running Python Process Controller

Reinforcement-learning training runs and backtests —Python scripts that take hours —are launched from Tauri, and their stdout streams back to the frontend. This is the textbook for **long-lived external processes**.

The AI was weakest here. It never put in the code path for `kill -9`ing a stuck process. I had to hit the problem in person and then spell out "account for SIGKILL"in the prompt.

2.9.5 5. TUI Trading Monitor —Terminal Dashboard

The same trading data from the GUI dashboard, rendered in a terminal with ratatui. Positions, P&L, alerts —all in a single-binary TUI. This project is the textbook for **how fast a TUI ships**. The GUI dashboard took two weeks. The TUI monitor took one day.

2.9.6 6. TUI Log Viewer —From Idea to Tool in Two Hours

A terminal-based log viewer for RL training output. Live tailing, regex search, and stats extraction. Built in two hours with AI. This project proves that **TUI is the fastest path from idea to working tool**.

2.10 1.9 What the Reader Walks Away With

Finishing this book will not make you memorize every Tauri 2 API. You will not become a Rust expert. That is not the goal.

What you walk away with is **the ability to build your own app using the same method**. Specifically, five things.

1. **Training to translate an idea into a request the AI can parse.** The muscle of moving "an app that feels like this"in your head to a prompt that yields actual code.

2. **An eye for judging AI output.** You can distinguish correct code from wrong code, and spot code that is correct but dangerous. In particular, you pre-internalize the common traps in Tauri 2 (permissions, State, IPC, async mutexes).
3. **The pace to keep looping without stalling.** You accept that the first pass is never perfect and you settle into the rhythm of three to five short cycles toward completion.
4. **Six concrete reference projects.** You read the structure of the real apps I built, and the exact spots where the AI contributed and where I overwrote it. Patterns you can lift almost verbatim when you build something similar.
5. **The judgment to pick the right interface.** GUI when users matter, TUI when speed matters, both when monitoring and polish are needed. That choice shapes every prompt that follows.

2.11 1.10 Two Years, in Numbers

Let me close the hand-wavy part of the argument with numbers. The same person (me) in 2020 and in 2026.

Metric	2020	2026
Desktop apps shipped	1	4 + 2 TUI
Lines of code written (approx.)	30k LOC	100k LOC
Share typed by hand	95%	under 10%
Prompts per day	0	40 -80
Commits per day	2 -3	8 -15
Primary languages	TypeScript	TypeScript + Rust + Python

The numbers are rough estimates pulled from my GitHub and shell history. Not a rigorous benchmark —a **sense of scale**. Two things stand out.

First, productivity rose **even as the share of new languages (Rust, Python) increased**. In the old days learning Rust would have eaten half a year. The AI compressed that half-year into "a few minutes of asking whenever I need it, like a search query."

Second, **rising commit count** is evidence that I keep the loops small. Ten commits a day means under an hour per commit. That is the only cadence in which the AI stays useful.

2.12 1.11 Self-Check Before You Start

Getting the most from this book requires something from you. Run through these five.

- ☐ `git`, `npm`/`pnpm`, and `cargo` do not trip you up in the terminal.
- ☐ You have installed at least one AI coding tool (Claude Code, Cursor, Copilot, etc.).
- ☐ You read TypeScript without trouble. Rust knowledge is optional.
- ☐ In the past three months you have "wired up an unfamiliar library in under an hour"at least once.
- ☐ You can carve out at least thirty minutes of focused time daily.

If all five are checked, go straight to Chapter 2. If three or fewer, read Chapter 4's environment setup first and get the machine ready.

2.13 1.12 What This Book Does Not Cover

Being honest about the boundary.

- **General prompt engineering** is out. This book covers prompts only in the context of building desktop apps. For writing, image generation, or RAG, read another book.
- **The internals of AI models** are out. Questions like why Claude writes Rust better than OpenAI are out of scope.
- **A copy of the official Tauri 2 docs** is not what this is. I explain only the minimum concept and then spend my time on "how I used this concept with an AI."

- **No first-try success stories.** Every chapter contains a scene where I was wrong, a scene where the AI was wrong, and a scene where I re-ran the loop. That is the core material.

2.14 Summary

- In 2026 I shipped four Tauri 2 apps while holding a day job and raising a child. Not because I had time, but because **I did not have time and the AI agent filled the gap**.
- The core change is not the tech stack. It is that **I delegated typing to an AI agent**.
- Development used to mean "I am the one writing the code."Now it means "I am the one making decisions."
- "You only need to be able to read AI code"is half-right. **The reading bar** is much higher than it used to be.
- The role shifted from code review (after the fact) to requirements refinement (before the fact). Writing prompts is the longest activity in my day.
- This book uses six projects —Mandai, readbooks.ai, Trading Dashboard, RL & Backtest Lab, TUI Trading Monitor, and TUI Log Viewer —to help you apply the same method to your own app.