Code We Understand

A Field Manual for Building Software With AI — Sample Chapters

William Kilpatrick

This is a free sample of *Code We Understand*. It contains the Preface, Chapter 1, and Chapter 3 of the full book.

The full book — including the detailed practice chapters, the worked Tidepool walk-through, and the appendices — is available at `codewe-understand.com`.

hello@codeweunderstand.com

Preface

This book is about how to build software, real software, the kind that does useful work in the world with an AI coding assistant as your collaborator.

It is written for two kinds of readers, and I have tried to serve both. The first is someone who already writes code and wants to understand how to work with AI tools without the output becoming mediocre. The second is someone who has never written code professionally but has an idea they want to build, and is willing to learn what it takes to build it well. Both of you are welcome here.

If you are the second kind of reader, I want to tell you directly: you can do this. Not everything, and not every kind of project, but a surprising amount. The method in this book has worked for a person with a non-technical background to build real, working tools. But I want to be honest about what it asks of you. You will be learning two things in parallel: software development, and the specific practice of working with AI on it. This book teaches the second. It does not teach the first from scratch. It will show you a method that makes your learning of the first efficient, provided you are willing to learn alongside it.

If you are starting from no coding background at all, spend a week or two on the foundations before starting Chapter 2 seriously. The command line and git sections of Appendix D are the right starting points.

If you are the first kind of reader, I want to tell you something different: most of what I am going to say will seem obvious in retrospect, and yet I watch experienced engineers skip these steps constantly. The method is not complicated. The discipline of actually following it is

where the value lives.

There is one kind of reader I cannot help: someone looking for shortcuts. This book is not about how to generate code faster. It is about how to work with an AI collaborator in a way that produces code you actually understand and can maintain. If what you want is to generate a pile of code, accept it, ship it, and move on, there are other books and other methods. The results are faster and worse, and I have nothing against people who choose that path. But it is not the path this book teaches.

A note on examples. The primary example running through the book is a small project called Tidepool, a tool that monitors a handful of websites every morning and emails you the new things it finds. Tidepool is small enough to build in a single focused session, real enough to teach the method, and general enough that you can mentally substitute your own project for it. The finished version is available at the Tidepool repository located on GitHub at `https://github.com/CodeWeUnderstand/tidepool`. I recommend not looking at it until you have worked through Chapter 14 yourself, because the value of the walk-through is in the decisions, not the artifact. In later chapters I will also reference fragments of a larger project I have been building, without naming it, to show how the same method scales.

A note on the AI. Throughout the book I will refer to "the agent" rather than to a specific AI product. The method works with any capable AI coding assistant. At the time of writing, that includes Claude Code, Cursor, Aider, Codex CLI, and others. The specific commands I show are from Claude Code, because that is what I use and what I can speak to with real experience. Readers using other tools will need to translate the specifics, but the practices and the reasoning are the same.

Chapter 1

The Three Modes of Use, and Why Two of Them Fail

There are three ways people use AI coding assistants. The first two are common and produce bad results. The third is uncommon and produces good ones. Understanding which mode you are in, or which mode you are drifting toward, is the first act of discipline.

The Paranoid Mode

In the paranoid mode, the builder treats the AI with suspicion. They ask it questions, copy its answers into another window, read the answers carefully, and then write the code themselves. The AI functions as a reference document: a faster version of Stack Overflow, a more conversational version of documentation.

The paranoid mode produces good code, because the human is writing every line. It also produces code very slowly, because the human is writing every line. The paranoid builder uses AI as a search engine, and accepts the cost of being their own translator between the AI's suggestions and their own implementation.

There is nothing wrong with this mode. Many excellent engineers use AI this way, and the code they produce is their own in every meaningful sense. But the mode leaves value on the table. The AI is capable of much more than being a reference document, and if you are willing to let it do more, you can build significantly faster without meaningfully lower quality, provided you have discipline.

If paranoid mode is where you are comfortable and you are producing

the work you want to produce, this book may not change your mind. But read on, because you may find a middle ground.

The Trusting Mode

In the trusting mode, the builder treats the AI as a capable collaborator and largely accepts its output. They describe what they want, the AI produces code, they skim the code, they run it, and if it works, they move on.

This is the mode most people fall into when they first discover how capable AI coding assistants have become. It is intoxicating. Work that used to take hours takes minutes. A developer can produce ten times their previous output. The temptation to simply trust, to review lightly and ship, is very strong.

The trusting mode produces bad results over time, and the reason is subtle. Individual outputs from a capable AI are often correct. The problem is cumulative. Every time you accept output without full understanding, you add a small piece of code to your project that you do not fully understand. Initially this does not matter. The code works. But as the project grows, two things happen.

First, the pieces you do not understand start to interact. A bug appears. You do not know where it is, because you do not know the code well enough to reason about it. You ask the AI to help you debug, and the AI suggests fixes, which you also do not fully understand. You are now two layers deep in code you did not write and do not understand, and the fix-and-patch cycle becomes the only tool you have.

Second, your judgment erodes. Each small acceptance teaches you that review is optional. Over a few weeks of sustained trusting use, the threshold at which you would push back drops. You begin to accept things you would have questioned earlier. The AI becomes not a collaborator but an authority. And because it is occasionally, subtly wrong, and because you have lost the habit of checking, the wrongness compounds.

The trusting mode is where most of the horror stories come from. It is how builders end up with codebases they do not understand, powered by decisions they did not make. It is not a moral failing. It is an easy mistake with strong tailwinds. Avoiding it requires structure.

The Structured Mode

In the structured mode, the AI does real work, as much as it is capable of doing well, but every piece of that work passes through explicit review gates. The gates are cheap individually. In aggregate they compound into something valuable: a codebase the human genuinely understands, produced faster than the human could have produced it alone.

The rest of this book is about how to operate in the structured mode. The method is not complicated. It is a set of seven practices that, together, create the conditions for good work. None of the practices are original. None of them is clever. The only insight is that you have to actually do them, every time, even when it feels like you could skip a step.

Before we get to the practices, let me describe what structured mode feels like in practice.

When I build something in structured mode, I am not faster than the AI. I am about the same speed as the AI. What I am is faster than I would be alone, and I produce output I understand as well as if I had written it myself. The sessions are cognitively demanding, more demanding than solo work, because I am constantly evaluating someone else's proposals, but they are also short. Two hours is a long session. Thirty minutes is common.

I commit small increments of work with descriptive messages, so that in six months I can read my own git history and know what I was doing and why. I write down the decisions I made, and the reasoning, so that future-me, or a collaborator, or the agent itself in a later session can pick up where I left off.

I push back on the agent regularly. Not because it is usually wrong, but because pushing back is how I stay engaged. An agent that never hears "no" will not reliably produce work I can trust. The no's are part of the calibration.

And I stop when the work is done, even when I could keep going. This is the hardest practice and the one most people skip. Momentum is seductive. Fatigue is invisible until you notice, after the fact, that you approved something you would have caught earlier in the session. Stopping is how you protect the quality of your own attention.

That is structured mode. The rest of this book teaches how to do it.

Chapter 3

The Seven Practices: An Overview

This chapter introduces all seven practices briefly. The next seven chapters each take one practice and go deep. They cover what it is, when to do it, what it looks like in practice, and what goes wrong when you skip it.

If you read nothing else from this book, read this chapter. The practices, stated briefly, are the entire method. Everything else is elaboration.

Practice One: Load context deliberately.

At the beginning of every session with the agent, make sure it has the context it needs to do good work. This means a project-level document that tells it what the project is, what the conventions are, what is forbidden, and what is expected. Without this context, the agent produces generic work. With it, the agent produces work that belongs to your project.

The context document has a specific name in Claude Code. It is called CLAUDE.md and lives in the project root. Other tools use other names. The name does not matter. The existence of the document does.

Before doing any real work in a session, verify the context loaded correctly. A simple prompt like "Describe this project in three sentences, without running any commands" is enough. If the agent describes the project correctly, you are ready. If not, fix the context before proceeding.

Practice Two: Plan before code.

When you want the agent to build something new, the first thing you ask for is not code. It is a plan. Describe what you want. Ask the agent to describe what it would need to build, what external tools it would use, and what ambiguities exist in the specification that you should resolve before implementation begins.

The agent, asked to plan, will surface real ambiguities. The agent, asked to code, will plow through those ambiguities with confident guesses. The planning step is where most mistakes get caught, before they have been written into the codebase.

A good plan is short. A few paragraphs, maybe a list of questions. If the plan is longer than what you could read in five minutes, the task is too big and should be broken down.

Practice Three: Probe real data.

Before writing production code that depends on something external, an API, a data source, or a third-party service, run a small exploratory query and look at the real output. Not synthetic test data. The real thing.

This catches the class of bugs where the code is correct but the assumptions about the external thing are wrong. An API parameter that does not mean what it looks like it means. A default ordering that returns old content first. A response format that has changed since the documentation was written. Every one of these is cheap to catch at the probe stage and expensive to catch after implementation.

The probe is usually five to twenty lines of throwaway code. Its only purpose is to show you what the real data looks like. It never becomes part of the project.

Practice Four: Adjudicate with reasoning.

When the agent asks you a question during planning, answer with reasoning, not just a choice. “Use strict regex” is a choice. “Use strict regex, because I want the code to fail loud if the format ever changes, rather than silently accept something unexpected” is reasoning.

The reasoning is what teaches the pattern for next time. It is also what you will refer back to when you come to the code six months from now and wonder why it is the way it is. A codebase full of arbitrary-seeming decisions is a codebase nobody wants to maintain. A codebase where every nontrivial decision has a recorded reason is a codebase you can actually evolve.

Reasoning should also be recorded somewhere durable. In code comments where the decision lives. In commit messages. In a notes file. Do not rely on your memory or the terminal scrollback.

Practice Five: Verify end-to-end.

After the agent implements something, run it against real inputs and look at the output. Do not just read the code and conclude it looks right. Do not just run automated tests and conclude they passed. Look at the actual output of the actual function with actual data.

This catches the gap between "the code compiles" and "the code does what was specified." That gap is bigger than most people expect. The agent is very good at producing code that looks like it does what you asked. The only reliable way to know if it actually does is to run it.

End-to-end verification does not have to be elaborate. A single invocation of the function with a realistic input and a print statement showing the output is enough. If the output matches what you expected, you are done. If not, investigate.

Practice Six: Commit in small increments.

Every meaningful piece of work gets its own commit with a descriptive message. Not every edit, not every file change, but every meaningful unit of work. Adding a new feature is a commit. Fixing a bug is a commit. Refactoring a function is a commit.

The commit message is part of the commit. Not an afterthought. A good message tells future-you what changed and why. "Updated fetcher" is a bad message. "fetcher: add retry logic for transient API failures, with exponential backoff" is a good message.

Small, well-labeled commits create a breadcrumb trail through your project. Six months from now, when you are debugging something

strange, the trail will tell you exactly when each piece was built, what the thinking was, and what changed in each step. Without the trail, debugging becomes archaeology.

Practice Seven: Stop when the work is done.

This is the hardest practice. When you have completed a meaningful unit of work, a feature is done, a bug is fixed, or a session has a clean resting point, stop. Even if you have energy. Even if you could keep going.

The quality of your judgment degrades faster than your output does. Three hours into a session you are producing code at the same rate but evaluating it less carefully. You miss things. You approve edits you would have questioned earlier. The last piece of work in a long session is almost always of lower quality than the first piece, and the difference is invisible until you notice the bug later.

The hardest version of this practice is: stop when you are on a roll. Momentum feels productive. It sometimes is. But extending a session because it feels good to be productive is exactly how trusting mode creeps in. A short session ending at a clean resting point is better than a long session ending at fatigue.

Stop. Commit. Close the terminal. Come back later.

How the practices fit together

Individually, each practice is small. None of them takes much time. The value is in how they combine.

Together, they form a loop: **load context, plan, probe, adjudicate, implement, verify, commit, stop.** Each loop produces one committable unit of work. A session might be one loop. A complex session might be two. Rarely more.

The next seven chapters take each practice and explore it in detail. They cover what it looks like in practice, the specific prompts I use, the mistakes people make, and the reasoning behind the design. If a practice seems obvious, you can skim. If it seems strange, slow down and read carefully, because the strangeness is usually where the value is hidden.

The practices as a checklist

If you want a single artifact to keep next to your terminal while you build, here it is.

Before the session:

- ☐ Context file exists and is current
- ☐ Notes from the last session are on hand
- ☐ I am in a mental state appropriate for careful work

At the start:

- ☐ Start the agent in the project directory
- ☐ Verify context loaded, ask the agent to describe the project
- ☐ Review the notes file: what was done, what is open, what is next

For each piece of work:

- ☐ Plan before code, ask the agent to describe what it would do, with ambiguities
- ☐ Adjudicate ambiguities with reasoning, not just choices
- ☐ Probe real data if anything external is involved
- ☐ Implement in reviewable chunks: imports, constants, body
- ☐ Verify end-to-end with real inputs and look at real outputs
- ☐ Commit with a descriptive message explaining what and why

At the end of the session:

- ☐ Update the notes file with what was done
- ☐ Confirm the repo is in a clean committed state
- ☐ Stop. Actually stop. Close the terminal.

Two of the appendices are essential reference rather than supplementary material: Appendix B collects the prompt templates you will see surfaced inline in the chapters that follow, and Appendix C catalogues the anti-patterns to watch for in yourself and in the agent. Read both before your first real session, and return to them as you build.

Where to go from here

You have just read the argument (Chapter 1) and the full method (Chapter 3). The rest of the book is the elaboration — seven chapters, one per practice, each covering what the practice looks like in real sessions, the prompt templates I use, the specific mistakes I watch for, and the reasoning behind the design. The book closes with a complete walk-through of building Tidepool from scratch so you can see every practice applied to a real project.

If the method here seems worth living with, the full book is available at `codeweunderstand.com`.

— *William Kilpatrick*