

Contents

Vibe Coding	1
Preface: This Book Is For You	3
Part One: Why This Changes Everything	5
Chapter 1: The Old World Is Dead	5
Chapter 2: What Actually Changed	6
Chapter 3: But I’m Not Technical	8
Chapter 4: Who Is Already Doing This	10
Chapter 5: The Democratization of Creation	11
Part Two: The Foundation	15
Chapter 6: What Is Software, Really?	15
Chapter 7: The Anatomy of Modern Software	16
Chapter 8: The Development Environment	18
Chapter 9: Your First Conversation with Claude	19
Chapter 10: The Collaboration Model	20
Chapter 11: Building on Solid Ground: Tests	21
Chapter 12: Version Control: Your Safety Net	23
Part Three: Building Your First Real Project	25
Chapter 13: Choosing Your First Project	25
Chapter 14: Describing What You Want	26
Chapter 15: The Build Cycle	28
Chapter 16: Debugging When Things Go Wrong	29
Chapter 17: Working with Databases	31
Chapter 18: Making It Look Good	32
Chapter 19: Deploying to the World	33

Part Four: Going Further	35
Chapter 20: When Things Get Complex	35
Chapter 21: User Accounts and Authentication	36
Chapter 22: Payments and Business Logic	38
Chapter 23: Mobile Applications	39
Chapter 24: Working with Existing Systems	40
Part Five: The Bigger Picture	41
Chapter 25: What You're Really Learning	41
Chapter 26: The Teacher Who Builds Tools	42
Chapter 27: The Small Business Revolution	43
Chapter 28: What the Critics Get Wrong	44
Chapter 29: Ethics and Responsibility	45
Chapter 30: Where We Go From Here	46
Part Six: Practical Projects	49
Chapter 31: Project Walkthrough: Personal Journal	49
Chapter 32: Project Walkthrough: Small Business Tool . .	50
Chapter 33: Project Walkthrough: Community Tool	52
Part Seven: Reference	55
Chapter 34: Prompt Patterns That Work	55
Chapter 35: Common Problems and Solutions	56
Chapter 36: Security Checklist	57
Chapter 37: Deployment Checklist	57
Chapter 38: Continuing Your Journey	58
Conclusion: The Power Is Yours	61
Appendix A: Setting Up Your Mac	63
Appendix B: Setting Up Ubuntu Linux	65
Appendix C: Glossary	67
Appendix D: Resources	69
Part Eight: Advanced Tutorials	71
Chapter 39: Tutorial: Building an Event RSVP System . .	71
Chapter 40: Tutorial: Customer Feedback Collection	73
Chapter 41: Tutorial: Personal Finance Tracker	75

Chapter 42: Tutorial: Content Calendar for Social Media	76
Chapter 43: Tutorial: Client Portal for Freelancers	78
Chapter 44: Tutorial: Learning Management Mini-System	79
Part Nine: Beyond the Basics	81
Chapter 45: Understanding What Claude Produces	81
Chapter 46: Maintaining and Evolving Your Applications	82
Chapter 47: Working with Others	83
Chapter 48: The Ecosystem Around You	84
Chapter 49: What's Coming Next	84
Chapter 50: Your Future as a Builder	85
Final Thoughts	87
Part Ten: Deep Dives	89
Chapter 51: Understanding Web Technologies	89
Chapter 52: Security for Non-Developers	92
Chapter 53: User Experience Fundamentals	95
Chapter 54: Business Models and Monetization	97
Chapter 55: Marketing Your Creation	98
Chapter 56: Real Stories of Non-Developers Building	99
Chapter 57: Troubleshooting Guide	101
Chapter 58: The Philosophy of Building	102
Epilogue: A Letter to Future You	105
Acknowledgments	107
Bonus Part: Industry-Specific Applications	109
Chapter B1: Healthcare Applications	109
Chapter B2: Education Applications	111
Chapter B3: Small Retail and Service Businesses	113
Chapter B4: Creative and Professional Services	114
Chapter B5: Nonprofit and Community Applications	115
Chapter B6: Personal Applications	116
Closing Materials	119
Frequently Asked Questions	119
Quick Reference Commands	120
Final Checklist	121

Extended Case Studies	123
Extended Case Study 1: From Spreadsheet to Software . . .	123
Extended Case Study 2: Community Organization Goes Digital	125
Extended Case Study 3: Automating a Consulting Practice	127
Extended Case Study 4: From Hobby to Product	128
Additional Resources and Tools	131
Resource: Evaluating AI Tools	131
Resource: Development Environment Alternatives	132
Resource: Learning More About Specific Technologies . . .	132
Resource: When to Seek Professional Help	133
Resource: Keeping Up with Changes	134
Complete Glossary	135
Comprehensive Tutorial Section	141
Complete Build: Task Management Application	141
Complete Build: Blog Platform	143
Complete Build: Fitness Tracker	145
The Pattern	147
Conversations with Claude: Real Examples	149
The 30-Day Learning Path	153
Final Words	155
Supplementary Material	157
The Economic Case for Vibe Coding	157
The Social Impact of Democratized Software	158
The Future We’re Building Toward	160
Mindset for Continuous Learning	162
Parting Advice	163
About This Book	164
Appendix E: Complete Example Conversations	165
Building a Recipe App: Full Transcript	165
Building an Invoice System: Key Exchanges	168
Common Conversation Patterns	169
Appendix F: Troubleshooting Decision Tree	171

Appendix G: Security Quick Reference	173
Appendix H: Performance Checklist	173
Appendix I: Industry Insights from Someone Who's Been There	175
What Professional Software Development Is Really Like . .	175
Lessons from Enterprise Software	177
Lessons from Team Leadership	178
What Software Jobs Look Like Now	178
Appendix J: The Philosophy of Creation	181
Why Build Things?	181
The Responsibility of Creation	181
Software as Expression	182
The Builder Mindset	182
Final Appendix: Quick Start Guide	185
Today: Setup (30-60 minutes)	185
This Week: First Project	185
This Month: Build Something Real	186
Ongoing: Keep Building	186
Bonus Chapter: The Care Commons Story	187
The Problem	187
The Approach	187
The Timeline	188
The Numbers	189
What Worked Well	189
What Was Challenging	189
The Result	190
The Lesson	190
One More Thing: The Manifesto	191
Extended Tutorial: Building a Complete SaaS Product	195
The Product: Client Feedback Collection	195
Week 1: Core Platform	195
Week 2: Analytics and Management	196
Week 3: Monetization Features	197
Week 4: Launch Preparation	197
What You've Built	198

The Economics 198

Afterword: For Those Who Made It This Far 201

 You're Different Now 201

 What Most People Won't Do 201

 The Community Awaits 202

 What Comes Next 202

 My Hope for You 203

Index of Key Concepts 205

Vibe Coding

The Revolution That Lets Anyone Build Software

Brian Edwards

November 2025

This work is published under Creative Commons Attribution 4.0 International License.

Preface: This Book Is For You

Let me be direct about who this book is for, because I suspect you're wondering if you belong here.

You belong here if you've ever thought, "I wish I could build that app," and then dismissed the thought because you're not a programmer. You belong here if you manage WordPress sites and wonder if you could build something more sophisticated. You belong here if you're a teacher who has embraced AI tools for grading and lesson plans and suspects there might be more. You belong here if you run a small business and have been told that custom software is beyond your reach. You belong here if you work in IT—managing networks, supporting users, handling systems administration—and have always been curious about the development side of the house but never had time to learn.

This book is not for professional software developers. They have their own journey to make with AI tools, and many excellent resources exist for them. This book is for everyone else—the vast majority of people who have been told, explicitly or implicitly, that building software is not for them.

That message was true. It is no longer true. Something fundamental has changed in the past year, and most people don't realize it yet.

I spent fifteen years as a professional software engineer. I led teams at Atlassian working on Bitbucket, one of the world's largest code hosting platforms. I built enterprise monitoring software at Zenoss that ran in Fortune 500 data centers. I know what traditional software development looks like, and I know what it costs in time, money, and

expertise.

What I discovered over the past several months contradicts everything I thought I knew about who can build software and how long it takes. Working with AI—specifically, with Claude—I built a production healthcare management system in 28 days. Not a prototype. Not a demo. A real system with 60,000 lines of code, comprehensive tests, regulatory compliance features, web and mobile interfaces, and all the infrastructure needed to run in production.

I am not smarter than I was two years ago. The tools have changed. And the tools have changed in a way that doesn't just make professional developers faster—they make software development accessible to people who never could have done it before.

This book will teach you how.

Part One: Why This Changes Everything

Chapter 1: The Old World Is Dead

To understand why the present moment matters so much, you need to understand what software development used to require.

Not the romantic version you might have absorbed from movies and television—the lone genius typing furiously in a dark room, building miraculous things through pure brilliance. That was always fiction. Real software development has been, for decades, a team sport requiring years of specialized training and substantial capital investment.

Consider what it actually took to build business software before AI. A company that wanted custom software had a few options, all of them expensive and time-consuming.

The first option was to hire developers. A single experienced software developer costs \$150,000 to \$200,000 per year in total compensation, and you almost never need just one. Software of any complexity requires specialists: backend developers who build the logic that runs on servers, frontend developers who create the interfaces users see, database specialists who design how information is stored, DevOps engineers who keep everything running, QA engineers who test for bugs, project managers who coordinate the work, designers who make it usable. A modest software team might have ten people. A serious effort might have fifty or a hundred. The annual cost runs into millions of dollars, and you need to sustain that spending for years to build and maintain anything substantial.

The second option was to hire an agency or consulting firm. This shifts the problem from hiring to procurement, but doesn't reduce the cost. Consulting rates for software development run \$150 to \$300 per hour. A project that requires 10,000 hours of work—a modest estimate for business software—costs \$1.5 to \$3 million. The work takes 12 to 24 months. You get what you specified in the contract, which is rarely what you actually needed once you see it running.

The third option was to buy off-the-shelf software and customize it. This is often the most practical choice, but it means accepting compromises. The software wasn't built for your specific situation, so it doesn't quite fit. Customization costs more than you expected. You're dependent on a vendor whose priorities may not align with yours. Switching costs are high, so you're locked in. It's better than nothing, but it's not really your software—it's software you're renting.

These options defined the landscape for decades. Software was expensive because skilled humans are expensive and software required lots of skilled humans working for long periods. There was no way around this. You could optimize the process, choose different technologies, adopt better methodologies—but fundamentally, you needed people, and people cost money and take time.

If you didn't have the budget for any of these options, you didn't get custom software. You made do with spreadsheets, manual processes, and off-the-shelf tools that partially fit your needs. This was the situation for most small businesses, most non-profits, most individuals with problems that software could solve. Custom software was simply not accessible to them.

This is the world that is ending.

Chapter 2: What Actually Changed

The change that matters is not that AI can generate code. That has been true in limited forms for years. Autocomplete has existed forever. Even sophisticated code suggestion tools have been around since 2021. These tools made professional developers somewhat faster, but they didn't change who could build software.

The change that matters is that AI can now do the *thinking* part of software development, not just the typing part.

Let me explain what I mean, because this distinction is crucial.

When a professional software developer builds a feature, the typing is the smallest part of the work. The larger parts are: understanding what needs to be built, designing how the different pieces will fit together, figuring out how to integrate with existing code, anticipating what might go wrong, writing tests to verify correctness, debugging when things don't work as expected, and refactoring when the initial approach turns out to be flawed.

These activities are cognitive, not mechanical. They require understanding, judgment, and the ability to hold complex systems in mind. They're why software developers are highly paid—not because typing is hard, but because the thinking is hard.

Current AI systems—Claude in particular, which is what I use and recommend—can do this thinking. Not the way a human does, but effectively enough to produce working software.

When I work with Claude, I don't dictate code for it to type. I describe what I want to exist, and Claude figures out how to make it exist. It reads the existing codebase to understand how things are structured. It designs new features that integrate properly with what's already there. It writes tests to verify its work. When tests fail, it figures out why and fixes the problem. It anticipates edge cases and handles them. It makes architectural decisions about how to organize code.

This is qualitatively different from autocomplete. It's not "AI helps you code faster." It's "AI does the coding while you provide direction and verify results."

The implications are profound. The bottleneck in software development has always been human cognitive capacity—how much a person can understand, how fast they can think, how many details they can keep track of. AI doesn't have the same limitations. It can read thousands of lines of code and maintain understanding of all of it. It can work at a pace measured in seconds per feature rather than hours or days. It doesn't get tired, doesn't forget, doesn't make mistakes because it was distracted.

This doesn't make AI better than humans at everything. Humans are still better at understanding what should be built, at making

judgment calls about design tradeoffs, at knowing when something “feels wrong” even if it technically works. But the work of translating a human intention into working software—that work can now be done by AI.

And here’s the key insight for non-developers: the work of translation was always the main barrier. If you could describe what you wanted, but you couldn’t translate that description into code, you couldn’t build software. The translation required years of specialized training. Now the translation can be done by AI. The barrier is gone.

Chapter 3: But I’m Not Technical

I anticipate your objection. You’re thinking: “This is all well and good, but I’m not technical. I’ve never written code. I don’t understand programming concepts. Even if AI can do the coding, I don’t know enough to direct it.”

I understand this concern. And I want to address it directly, because it’s based on a misunderstanding about what “technical knowledge” actually means.

There are really two kinds of technical knowledge relevant to software development.

The first kind is syntactic knowledge: how to write code in a specific programming language, what commands to use, how to structure files, what the correct punctuation is. This is what people usually think of when they imagine “learning to code.” It’s memorizing that in JavaScript you write `function doSomething()` while in Python you write `def do_something():`. It’s knowing that you need semicolons in some languages and not others. It’s remembering which commands create files and which commands delete them.

This kind of knowledge is necessary for traditional software development and takes years to master. It’s also almost completely irrelevant when working with AI.

The second kind is conceptual knowledge: understanding what software is, what databases do, why testing matters, how web applications are structured, what happens when a user clicks a button. This is the knowledge that lets you understand what you’re building and why,

even if you couldn't write the code yourself.

This second kind of knowledge is what you need to work effectively with AI. And here's the key point: you probably already have more of it than you think.

If you've used a computer for any significant task, you have mental models of how software works. You understand that information gets stored somewhere and retrieved later. You understand that different screens show different things. You understand that some actions require logging in and others don't. You understand that mistakes can be undone (sometimes) and that data can be backed up. You understand that some things work offline and others require an internet connection.

These mental models are imperfect—they're based on observation rather than implementation—but they're sufficient to direct AI. You don't need to know how a database works internally to tell Claude "we need to store customer information and be able to search by name or email." You don't need to understand HTTP protocols to tell Claude "when someone clicks submit, it should save their data and show a confirmation."

The conceptual knowledge you lack, you can acquire as needed. This book will help. But the starting point is not zero—it's the accumulated experience of using software as a consumer and as a worker. That experience has taught you more than you realize.

Let me give you a concrete example. Imagine you run a small dog-walking business. You currently manage scheduling in Google Calendar, customer information in a spreadsheet, and payments through Venmo. You've thought about how nice it would be to have a real system—something where customers could book appointments, see your availability, pay online, and receive automatic reminders.

Traditional software development would estimate this as a 2-4 month project requiring 2-3 developers. Cost: \$50,000 to \$150,000.

With AI, you could describe what you need in plain English: "I need a website where customers can see my availability, book appointments for dog walking, enter their dog's information, pay online, and get reminder emails before their appointments. I need to be able to see all my bookings, mark walks as completed, and add notes about each

dog.”

That description contains no code, no technical jargon, no specialized knowledge. But it’s enough for AI to start building. The AI will ask clarifying questions: “How long is a typical walk? Do you offer different services? Do you need to track multiple dogs per customer?” You answer those questions in plain English. The AI builds. You try it. You say “the calendar view is confusing” or “I wish I could see the dog’s photo.” The AI adjusts.

This is not a hypothetical. This is how I work with Claude. The description of what to build comes from me. The implementation comes from Claude. The technical knowledge required is minimal—mostly just enough to understand what Claude is doing so I can verify it’s correct.

Chapter 4: Who Is Already Doing This

I am not the only person who has discovered this. A wave is building, though it hasn’t yet crested into mainstream awareness.

Entrepreneurs are building startups with AI as their technical co-founder. Instead of spending six months looking for a CTO or raising money to hire developers, they’re describing their products to Claude and building them in weeks. Some of these startups will fail—most startups do—but they’re failing because of market reasons, not because they couldn’t build the technology.

Small business owners are building custom tools for their specific operations. A restaurant owner builds an inventory management system that integrates with their suppliers. A property manager builds a maintenance tracking system tailored to their buildings. A consultant builds a client portal that exactly matches their workflow. These aren’t sophisticated technology companies; they’re regular people solving their own problems.

Teachers are building educational software customized for their classrooms. Instead of adapting to generic tools, they’re creating tools that match their curriculum and teaching style. A physics teacher builds interactive simulations that demonstrate exactly the concepts they’re teaching. A language teacher builds vocabulary training that aligns precisely with their lesson plans.

Researchers are building data analysis tools specific to their domains. Instead of forcing their research into the constraints of off-the-shelf statistical software, they're creating tools that handle their specific data formats, apply their specific methodologies, and generate their specific visualizations.

Non-profits are building systems they could never afford to buy or commission. A homeless services organization builds a case management system. A community garden builds a plot assignment and volunteer coordination system. An animal rescue builds a foster network management system. These organizations have critical needs and limited budgets; AI makes building possible.

What these diverse people share is not technical background—many have none—but willingness to try something new. They've heard that AI can help with building things, they've experimented, and they've discovered that it works far better than they expected.

The tools are available now. The techniques are learnable in weeks, not years. The only missing piece for most people is the knowledge that this is possible and the guidance to get started.

That's what this book provides.

Chapter 5: The Democratization of Creation

Throughout history, the ability to create things has progressively democratized. Consider writing.

For most of human history, literacy was rare. Creating written documents required not just knowledge but access to scarce materials and years of training. The ability to write was a specialized skill possessed by a tiny elite.

The printing press didn't change this immediately—books became cheaper to reproduce, but writing them still required the same skills. What changed was that more people learned to read, and as more people read, more people wanted to write, and as demand increased, education expanded.

Typewriters made the mechanical act of writing accessible to anyone who could learn to use them. Word processors made editing and

formatting accessible. Desktop publishing made professional-looking documents accessible. The internet made distribution accessible.

Each wave of democratization expanded who could participate in written communication. Today, anyone with a smartphone can write and publish to a global audience. The barriers that once restricted writing to monks and scribes are completely gone.

Software creation is undergoing a similar democratization, but compressed into a much shorter timeframe.

For decades, creating software required specialized education and expensive tools. Then programming languages became more accessible. Then open source provided free tools and libraries. Then cloud platforms provided free hosting. Then low-code tools made simple applications possible without programming.

But all of these advances still left a fundamental barrier: to build anything beyond the simplest applications, you needed to understand programming concepts and write code. The low-code tools hit limits. The cloud platforms still required configuration. The open source libraries required expertise to combine.

AI removes this final barrier. The translation from human intent to working code—the step that always required programming expertise—can now be done by AI.

This is not an incremental change. It's the completion of a democratization process that makes software creation as accessible as writing is today. Not everyone will build software, just as not everyone writes novels or essays. But the barriers that once prevented most people from even attempting it are gone.

The implications go beyond individual empowerment. When software could only be built by specialists, it reflected the priorities and perspectives of specialists. The problems that got solved were problems that technology companies thought were important enough to fund.

When anyone can build software, the problems that get solved become the problems that actual people face in their actual lives. The dog-walking business gets custom scheduling software. The community garden gets plot management. The small church gets a member directory tailored to their needs. These problems were always there,

always worth solving, but never economically viable for professional software development.

The explosion of literacy led to an explosion of written expression—more books, more journalism, more personal communication, more documented human experience than had ever existed before. The democratization of software creation will lead to a similar explosion: more tools, more solutions, more automation, more human problems addressed than was ever possible when creation required specialists.

You can be part of this. This book will show you how.

Part Two: The Foundation

Chapter 6: What Is Software, Really?

Before we get into building software, we should establish what software actually is. This might seem elementary, but many people have fuzzy mental models that make learning harder than necessary.

Software is instructions for computers. That’s it. Nothing more mystical than that.

A computer is a machine that follows instructions very quickly and very precisely. It does exactly what it’s told, no more and no less. If the instructions are correct, the computer does useful things. If the instructions are wrong, the computer does wrong things. The computer doesn’t know the difference—it just follows instructions.

These instructions are written in programming languages. A programming language is a way of expressing instructions that computers can understand (after translation) and that humans can read (with practice). There are many programming languages—JavaScript, Python, Java, C, and hundreds more—each with different syntax and different strengths, but they all do the same fundamental thing: express instructions for computers.

A piece of software—an app, a website, a program—is a collection of these instructions organized to accomplish some purpose. A web browser is instructions for fetching and displaying web pages. A word processor is instructions for editing and formatting text. A video game is instructions for simulating a world and responding to player input.

When you use software, you’re triggering different parts of these

instructions. When you click a button, you're telling the computer to execute a specific set of instructions associated with that button. When you type text, you're triggering instructions that add characters to a document. The software has anticipated what you might want to do and prepared instructions for each possibility.

This is why building software is fundamentally about specifying what should happen in different situations. What happens when a user clicks this button? What happens when they enter invalid data? What happens when they try to access something they shouldn't? What happens when two users try to change the same thing at the same time?

A software developer's job is to anticipate all these situations and write instructions for handling each one. This is why software development is hard: there are many situations to anticipate, and getting any of them wrong causes problems.

AI changes this by handling the instruction-writing part. You still need to describe what should happen in different situations, but you describe it in English (or whatever language you speak) rather than in a programming language. The AI translates your description into instructions the computer can follow.

Understanding this model helps you work with AI more effectively. When you're describing what you want, you're essentially telling the AI what should happen in different situations. The more clearly and completely you describe these situations, the better the AI can write instructions for handling them.

Chapter 7: The Anatomy of Modern Software

Most software you'll build will be web applications—software that runs in a browser or on a phone and connects to a server over the internet. Understanding the basic structure of web applications will help you communicate with AI about what you're building.

A web application has three main components: the frontend, the backend, and the database.

The frontend is what users see and interact with. When you visit a website or open an app, you're looking at the frontend. It includes

the buttons, forms, text, images, and everything else visible on screen. The frontend is responsible for displaying information and capturing user input.

The backend is the logic that runs on a server somewhere—a computer that’s always on and connected to the internet. The backend handles things that happen behind the scenes: processing user requests, enforcing business rules, communicating with other services, and generally doing the “thinking” part of the application. When you click “submit” on a form, the frontend sends your input to the backend, which decides what to do with it.

The database is where information is stored permanently. When you create an account, your information goes in the database. When you post a photo, the photo (or a reference to it) goes in the database. The database is like a very organized filing cabinet: information goes in, gets labeled and organized, and can be retrieved later.

These three components communicate with each other through defined interfaces. The frontend sends requests to the backend (“user clicked submit with this data”). The backend processes the request, reads from or writes to the database as needed, and sends a response back to the frontend (“here’s what to show the user next”).

This architecture exists because different components have different requirements. The frontend needs to be fast and visually appealing. The backend needs to be secure and logically correct. The database needs to reliably store and retrieve information. Separating them allows each component to be optimized for its specific job.

When you’re working with AI to build software, you’ll often be describing behavior that involves all three components. “When a user signs up, save their information and send them a welcome email” involves the frontend (the signup form), the backend (processing the signup and sending the email), and the database (storing the user information).

You don’t need to know which component handles which part—the AI will figure that out. But knowing that these components exist helps you understand what the AI is building and how to describe what you want.

Chapter 8: The Development Environment

To build software with AI, you need a development environment—a computer configured with the tools needed to create and run software. This chapter describes what you need and how to set it up.

The good news: you don't need to buy anything. Everything you need is free.

The basic requirements are:

A computer running macOS or Linux. Windows can work with some additional setup (the Windows Subsystem for Linux), but macOS or Linux is easier. If you have a Mac, you're already set. If you have a Windows computer and want the smoothest experience, consider installing Ubuntu Linux (it's free) as a dual-boot option or in a virtual machine.

A terminal application. The terminal is a text-based interface for interacting with your computer. On Mac, the built-in Terminal app works, though iTerm2 is better. On Linux, every distribution includes a terminal. The terminal is where you'll type commands and where AI will execute operations.

A code editor. This is the application where you view and edit code files. Visual Studio Code (VS Code) is free, popular, and works well. Download it from code.visualstudio.com.

Git, a version control system. Git tracks changes to your code and enables collaboration. It's free and comes pre-installed on Macs. On Linux, install it with your package manager.

Node.js, a JavaScript runtime. Much of modern web development uses JavaScript, and Node.js lets you run JavaScript outside a browser. Download it from nodejs.org—get the LTS (Long Term Support) version.

An AI interface. This is how you'll communicate with Claude. The tool I use and recommend is called `opencode`. You install it by running `npm install -g @anthropic/opencode` in your terminal after installing Node.js.

An Anthropic API key. Claude is accessed through an API (a programmatic interface), which requires a key. Sign up at anthropic.com

and create an API key in your account settings. You'll pay for usage—typically a few dollars per hour of intensive work, far less than you'd spend on professional developers.

Setting all this up takes an hour or two the first time. Don't worry if some steps are confusing; there are detailed installation guides online for each tool, and Claude itself can help troubleshoot issues once you have basic access working.

What you don't need: expensive software, powerful hardware, or specialized equipment. A basic laptop from the last five years is fine. All the heavy computation happens either in the cloud (where AI models run) or on servers (where your finished software runs), not on your local computer.

Chapter 9: Your First Conversation with Claude

With your development environment set up, let's have your first real conversation with Claude about building software.

Open your terminal and start `opencode` (the exact command depends on your setup, but it's typically just `opencode` or you run it from a menu). You'll see a chat interface where you can type messages.

Try this first message:

"I want to build a simple web page that shows a list of tasks. I should be able to add new tasks and mark tasks as complete. Please create this for me."

Claude will respond, probably by asking some clarifying questions or by immediately starting to build. It might create several files and explain what each one does. It might ask whether you want the tasks to be saved permanently or just exist while the page is open.

This is how the process works. You describe what you want in plain English. Claude translates that into code. You see the results and provide feedback.

Let's break down what happened:

Your request contained no technical jargon. You didn't specify a

programming language, a framework, a database, or any implementation details. You described the behavior you wanted from a user's perspective.

Claude made technical decisions on your behalf. It chose a programming language (probably JavaScript), a way to structure the code, and an approach to managing the task data. These decisions were reasonable defaults for a simple project.

The code Claude produced actually works. You can open the file it created in a browser and interact with the task list. It's not a mockup or a design—it's functional software.

This simple example illustrates the core workflow, but it's just the beginning. Real projects involve databases, user accounts, multiple pages, validation logic, and many other components. The workflow remains the same: describe what you want, let Claude build, verify the results, and refine.

The skill you're developing is the ability to describe what you want clearly enough that Claude can build it correctly. This is different from traditional programming, which requires knowing how to build things. It's closer to managing a skilled team: you provide direction and verify results, while specialists handle implementation.

Chapter 10: The Collaboration Model

Working with AI is a collaboration, not a delegation. Understanding the correct model of interaction will save you time and frustration.

Think of Claude as an extremely fast, knowledgeable, and tireless junior developer. Claude knows more about programming languages and frameworks than any human could memorize. Claude can write code faster than any human could type. Claude will work as long as you want without getting tired or bored.

But Claude is junior in judgment. Claude doesn't know your business, your users, or your priorities. Claude can't tell whether a feature is valuable or worthless. Claude may not realize that a technically correct implementation misses the point of what you're trying to accomplish. Claude needs direction, and Claude needs verification.

This model implies a specific division of labor:

You are responsible for: deciding what to build, specifying how things should behave, reviewing what gets built, and catching mistakes. These are judgment tasks that require understanding the purpose and context.

Claude is responsible for: translating your specifications into code, knowing how to use programming languages and tools, following best practices, and iterating until tests pass. These are implementation tasks that require technical knowledge.

The collaboration works best when both parties do their jobs. If you under-specify (say “build me a website” without more detail), Claude will make assumptions that may not match your intent. If you over-specify (try to dictate every technical detail), you’re doing Claude’s job poorly—Claude knows how to write code better than you do.

The sweet spot is specifying behavior and leaving implementation to Claude. “When a user submits the form, validate that all required fields are filled out, save the data, and show a confirmation message.” This tells Claude what should happen without telling Claude how to make it happen. Claude can choose the best way to implement this behavior.

Feedback is crucial to the collaboration. When Claude builds something that isn’t quite right, don’t just ask Claude to try again—explain what’s wrong and why. “The confirmation message appears too quickly; the user might miss it. Add a delay or make it more prominent.” This helps Claude understand your intent and produce better results on the current task and future tasks.

Chapter 11: Building on Solid Ground: Tests

One of the most important concepts in professional software development is testing. Tests are code that verifies other code works correctly. They’re crucial for building reliable software, and they’re one of the most valuable things AI enables.

Here’s the basic idea: for each piece of functionality in your software, you write a test that exercises that functionality and checks that the result is correct. If you have a function that adds two numbers, you write a test that calls the function with specific numbers and verifies

the result is correct. If you have a form that validates email addresses, you write tests that submit valid and invalid addresses and verify the form accepts or rejects them appropriately.

Running the tests tells you whether the software works correctly. If all tests pass, the software probably works. If a test fails, something is broken.

Why does this matter for non-developers using AI? Because tests enable verification.

When Claude builds something, how do you know it works? You could manually try every feature, but that's tedious and error-prone. You'll miss edge cases. You'll forget to re-test things after changes.

Tests provide automated verification. After Claude makes any change, the tests run and tell you whether everything still works. If Claude introduces a bug, the tests catch it immediately. This is far more reliable than manual checking.

The workflow with tests looks like this:

1. You describe what you want to build
2. Claude writes the code AND writes tests for the code
3. Claude runs the tests to verify the code works
4. If tests fail, Claude fixes the code and runs tests again
5. Once tests pass, you review the results

This workflow has a powerful property: Claude can iterate autonomously. If Claude's first attempt has bugs, Claude can see the test failures, understand what went wrong, and fix the problems—without your involvement at each step. You get working software at the end, not a series of broken attempts to sift through.

You should insist that Claude write tests for everything important. Some things don't need tests (trivial display logic, for example), but any meaningful functionality should have tests. This is non-negotiable professional practice, and it's even more important when AI is writing the code.

Chapter 12: Version Control: Your Safety Net

Version control is a system for tracking changes to code over time. The specific system you'll use is called Git, and it's essential for any serious software project.

Here's the basic concept: every time you make a change to your code, you save a snapshot of the current state. These snapshots are called commits. Each commit has a message describing what changed ("Added user login feature" or "Fixed bug in checkout process"). The history of commits shows how the code evolved over time.

Why does this matter?

First, safety. If you make a change that breaks something, you can go back to a previous commit where things worked. This is like having unlimited undo for your entire project. Without version control, a mistake could lose hours or days of work. With version control, you can always recover.

Second, experimentation. Want to try a different approach but not sure if it will work? Create a branch (a parallel version of your code), experiment freely, and if it doesn't work out, discard the branch and go back to where you started. If it does work out, merge the branch back into the main code. This enables risk-free experimentation.

Third, collaboration. When multiple people (or multiple AI agents) work on the same code, version control manages the complexity. Each person commits their changes, and the system handles combining them. Conflicts (when two people change the same thing) are detected and can be resolved.

Fourth, history. The commit history documents what changed, when, and why. If something stops working and you don't know why, you can look through recent commits to find what changed. This is invaluable for debugging.

Claude uses Git automatically. When Claude makes changes, it typically creates commits with descriptive messages. You can see the history of what Claude has done, revert changes that didn't work, and branch off to try different approaches.

You don't need to understand Git deeply, but you should know the basics:

- `git status` shows what files have changed
- `git log` shows the history of commits
- `git diff` shows what's different from the last commit
- `git checkout <commit>` goes back to a previous commit
- `git branch` creates a parallel version for experimentation

Claude can handle most Git operations for you, but understanding what Git is and why it matters helps you work more effectively and recover from problems when they occur.

Part Three: Building Your First Real Project

Chapter 13: Choosing Your First Project

Now that you understand the fundamentals, it's time to build something real. Choosing the right first project matters—too simple and you won't learn much, too complex and you'll get stuck.

A good first project has these characteristics:

You understand the domain. Build something for an area where you have expertise or at least familiarity. If you run a small business, build something for that business. If you have a hobby, build something for that hobby. Understanding the problem domain lets you focus on learning the building process rather than also trying to understand the problem.

The scope is limited. Your first project should be completable in days, not months. You can expand it later, but starting with a limited scope lets you experience the full cycle of building, deploying, and using software. A todo list, a personal blog, a simple booking system, a recipe organizer—these are good first project scales.

You'll actually use it. Building something you'll actually use creates motivation to finish and improve it. If you're solving a real problem you have, you'll push through the frustrating parts. If you're building something hypothetical, you might give up when it gets hard.

It doesn't require external integrations initially. Some projects need to connect to other services—payment processing, social media, mapping services. These integrations add complexity. Choose a first project

that can be useful without external integrations, even if integrations would make it better later.

Examples of good first projects:

A personal task manager customized to your workflow. Yes, task managers exist, but building your own lets you design it exactly how you think about tasks.

A journal or note-taking app. Capture your thoughts in exactly the format you want, with exactly the organization you prefer.

A recipe collection with your own categories and search. Organize your recipes how you want, not how a generic app assumes you want.

A simple website for a small business or organization you're involved with. Not a complex e-commerce site, but an informational site with basic interactivity.

A tracker for something you care about—exercise, reading, bird sightings, whatever. Track exactly what matters to you in exactly the format that's useful.

Pick something from this list or something similar. Don't overthink it. The goal is to learn the process of building, and you'll learn that regardless of exactly what you build.

Chapter 14: Describing What You Want

The quality of what Claude builds depends heavily on how well you describe what you want. This chapter teaches you how to describe software requirements effectively.

Start with the user's perspective. Describe what a user would experience, not what the code should do. "The user should be able to add a recipe by typing the name, ingredients, and instructions" is better than "the system should have a form with three text fields that submits to an API endpoint."

Be specific about behavior. Vague descriptions produce vague implementations. "The user can search recipes" is vague. "The user can search recipes by name, by ingredient, or by both. Search results appear immediately as they type. Clicking a result opens the full recipe" is specific.

Include constraints and edge cases. What should happen when things go wrong? “If the user tries to add a recipe without a name, show an error message asking them to add a name.” What about limits? “A recipe name can be up to 100 characters. Ingredient lists can have up to 50 items.”

Describe what success looks like. How will you know if the feature works? “I should be able to add a recipe, close the browser, reopen it, and see my recipe still there.” This helps Claude understand what you’re really trying to accomplish.

Use examples. Concrete examples are often clearer than abstract descriptions. “For example, if I search for ‘chicken’, I should see ‘Chicken Parmesan’ and ‘Chicken Soup’ but not ‘Vegetable Soup.’”

Don’t specify implementation details unless they matter. “Store recipes in a PostgreSQL database” is only helpful if you specifically need PostgreSQL for some reason. Usually you don’t care how data is stored, just that it’s stored reliably.

Here’s an example of a complete requirement description:

“I want to build a recipe collection application. When I open the app, I see a list of my recipes showing the name and a preview of the ingredients. I can click a recipe to see the full details including name, ingredients, instructions, and any notes I’ve added.

I can add new recipes by clicking an ‘Add Recipe’ button. This opens a form where I type the name, ingredients (one per line), instructions, and optional notes. When I save, the recipe appears in my list.

I can edit recipes by clicking an ‘Edit’ button on the recipe detail view. I can also delete recipes, but the app should ask me to confirm before deleting.

I can search my recipes by typing in a search box. The search should check recipe names and ingredients. Results update as I type.

All recipes should be saved permanently so they’re still there when I come back to the app later.”

This description is specific enough that Claude can build something useful, but doesn’t dictate unnecessary technical details. It describes behavior from the user’s perspective and includes key constraints (permanent storage, confirmation before delete).

Chapter 15: The Build Cycle

With a project chosen and requirements described, let's walk through the actual process of building.

Session 1: Project Setup

Start by telling Claude what you want to build and asking it to set up the project structure.

"I want to build a recipe collection application. [Include your description from the previous chapter.] Please set up the project with appropriate structure for a web application."

Claude will create files and folders, install necessary tools, and configure the basic project. This might take a few minutes. Claude will explain what it's creating and why.

Once setup is complete, ask Claude to run the application. "Can you start the application so I can see it in my browser?" Claude will start a development server and tell you what address to visit (usually something like `localhost:3000`).

Open a browser and visit that address. You'll probably see a basic page—possibly just "Hello World" or a simple template. This confirms everything is working.

Session 2: Core Features

Now start building the actual features. Work through them one at a time.

"Let's start with the ability to add recipes. Create a form where I can enter a recipe name, ingredients, and instructions. When I submit the form, the recipe should be saved and appear in a list."

Claude will build this feature, write tests for it, and run the tests. You'll see the output as Claude works. Once Claude indicates it's done, refresh your browser and try adding a recipe.

If something doesn't work as expected, describe the problem. "When I add a recipe, the form stays on screen. It should clear so I can add another recipe." Claude will make adjustments.

Continue with each feature. "Now let's add the ability to view a recipe's full details when I click on it." "Now add a search feature."

Work through your requirements systematically.

Session 3: Polish and Details

After the core features work, address details and polish.

“The ingredient list is hard to read. Can you format it as a bulleted list instead of one continuous line?” “I’d like the search box to be at the top of the page, not at the bottom.” “Can you add confirmation when I try to delete a recipe?”

This phase often involves many small adjustments. Don’t try to get everything perfect at once—make changes, see the results, and iterate.

Session 4: Persistence and Deployment

Finally, make sure data persists and deploy so you can access your app from anywhere.

“Currently recipes disappear when I close the browser. Can you add a database so they’re saved permanently?” Claude will set up database storage.

“How do I deploy this so I can access it from my phone?” Claude will walk you through deploying to a hosting service (Vercel, for example, which has a free tier).

This cycle—setup, features, polish, deployment—is the basic rhythm of building software. Your first project might take several days of part-time work. Future projects will go faster as you develop intuition for what to ask and how to describe things.

Chapter 16: Debugging When Things Go Wrong

Things will go wrong. Code won’t work as expected. Features will break. Error messages will appear. This is normal. Learning to debug—to find and fix problems—is an essential skill.

First principle: stay calm. Error messages look scary, but they’re actually helpful. They tell you (and Claude) what went wrong and where. A vague “it doesn’t work” is much harder to fix than a specific error message.

When something goes wrong, gather information. What were you trying to do? What did you expect to happen? What actually happened? Did you get an error message? If so, copy the exact text.

Report the problem to Claude with all this context. “I tried to save a recipe and got this error: [exact error message]. I filled out all the fields and clicked save.” Claude can read the error message and usually identify the cause.

Common types of problems:

Syntax errors: The code has a typo or grammatical mistake. These usually produce clear error messages pointing to the problem. Claude can almost always fix these instantly.

Logic errors: The code runs without errors but does the wrong thing. “I added a recipe but it doesn’t appear in the list.” These require Claude to investigate—often by adding logging to see what’s happening.

Integration errors: Different parts of the code aren’t communicating correctly. “The form submits but nothing gets saved to the database.” Claude needs to trace the flow of data to find where it’s breaking down.

Environment errors: Something about your computer’s setup is causing issues. “The command fails with ‘package not found’.” These can be frustrating because they’re specific to your situation, but Claude can usually guide you through fixing them.

When debugging, be patient and methodical. Don’t randomly try things hoping something works. Understand the problem, identify the cause, and apply a fix. If you don’t understand what Claude is doing to fix something, ask Claude to explain.

Debugging skill develops with practice. Every bug you encounter and fix teaches you something about how software works. Over time, you’ll start to recognize common patterns and anticipate where problems might occur.

Chapter 17: Working with Databases

Most applications need to store data permanently. You’ve been using a database since Chapter 15, but let’s understand what’s actually happening.

A database is specialized software for storing and retrieving data. It’s like an extremely powerful, organized spreadsheet that can hold millions of records, allow multiple people to access it simultaneously, and retrieve information almost instantly even from huge amounts of data.

The most common type of database is a relational database, which stores data in tables. If you’ve used a spreadsheet, this will be familiar. A table has columns (categories of information) and rows (individual records). A recipes table might have columns for name, ingredients, instructions, and `date_created`. Each recipe is a row in this table.

Tables can relate to each other. A recipe might have multiple photos, so there’s a separate photos table with columns for `photo_data` and `recipe_id`. The `recipe_id` links each photo to its recipe. This is why they’re called “relational” databases.

When you tell Claude “save the recipe to the database,” Claude writes code that: 1. Connects to the database 2. Inserts a new row in the recipes table 3. Fills in the columns with the data from your form 4. Handles any errors that might occur

When you tell Claude “show me all recipes,” Claude writes code that: 1. Connects to the database 2. Retrieves all rows from the recipes table 3. Formats the data for display 4. Returns it to the frontend

You don’t need to write these database commands yourself, but understanding what’s happening helps you describe what you want more clearly.

Some useful database concepts:

Queries retrieve data. “Get all recipes” is a simple query. “Get all recipes containing chicken, sorted by date, showing only the five most recent” is a more complex query.

Indexes make queries faster. If you search recipes by name often, an index on the name column makes that search faster. Claude typically

creates appropriate indexes automatically.

Migrations change the database structure. If you decide recipes need a “cuisine type” field, Claude creates a migration that adds a column to the recipes table. This is important because you can’t just edit the database directly—there might be thousands of existing recipes that need to accommodate the new column.

Backups protect your data. Databases should be backed up regularly so you can recover if something goes wrong. Hosted database services (like Neon, which Claude might use) typically handle this automatically.

Chapter 18: Making It Look Good

Functionality is essential, but appearance matters too. A ugly application is harder to use and less pleasant to use. This chapter covers making your application look professional.

The good news: you don’t need to be a designer. There are established patterns for how web applications should look, and Claude knows them. You can get 90% of the way to a good-looking application just by asking Claude to make things look better.

Start with structure and layout. “Can you make this look like a modern web application? Put the navigation at the top, the main content in the center, and make it responsive so it works on phones.”

Claude will apply sensible layouts. If you don’t like something, describe what you want instead. “The sidebar is too wide. Can you make it narrower?” “The recipe cards should be in a grid, not a list.”

Color and typography set the tone. “Use a warm color scheme suitable for a recipe application.” “Use clean, readable typography.” Claude will make appropriate choices. If you have specific preferences—“I want blues and greens” or “I like sans-serif fonts”—mention them.

Spacing and alignment affect perceived quality. Even if you can’t articulate what’s wrong, you’ll notice if something looks “off.” Tell Claude: “Something about the spacing on this page looks wrong.” Claude can often identify and fix the issue.

Interactive elements need feedback. When you click a button, some-

thing should happen visually so you know your click registered. When a form has errors, they should be clearly highlighted. Tell Claude: “Add visual feedback when users interact with elements.”

Consistency makes applications feel professional. The same style of buttons everywhere, the same spacing patterns, the same approach to errors. Tell Claude: “Make sure the styling is consistent throughout the application.”

If you have a specific aesthetic in mind, show examples. “I like the clean look of Notion” or “I want it to feel cozy like a paper recipe card.” Claude can interpret these references.

One warning: don’t get lost in visual polish before the functionality works. A beautiful application that doesn’t function is worthless. Get things working first, then make them look good.

Chapter 19: Deploying to the World

You’ve built something that works on your computer. Now let’s make it available on the internet so you can access it from anywhere and share it with others.

Deployment means putting your application on a server that’s always running and connected to the internet. Several services make this easy and free (for small applications).

Vercel is the service I recommend for web applications. It’s free for personal projects, handles all the server management for you, and integrates well with the development workflow Claude uses.

Here’s the typical deployment process:

First, make sure your code is in a GitHub repository. GitHub is a service that stores code and tracks its history. Claude can help you create a repository and push your code to it.

“Can you help me create a GitHub repository for this project and push the code?”

Next, connect Vercel to your GitHub repository. Go to vercel.com, sign up (free), and import your GitHub repository. Vercel will automatically detect what kind of project you have and configure itself appropriately.

Finally, deploy. Vercel builds your application and puts it on a server. It gives you a URL (something like `your-project.vercel.app`) where your application is live.

From now on, every time you push changes to GitHub, Vercel automatically rebuilds and redeploys your application. This means keeping your live application updated is as simple as making changes and pushing them.

For the database, you'll need a hosted database service. Neon (neon.tech) provides free PostgreSQL databases that work well with Vercel. Claude can help you set this up and configure your application to use it.

After deployment, test your application at its public URL. Make sure everything works the same as it did on your computer. If something's broken, check the deployment logs (available in the Vercel dashboard) for error messages.

Congratulations—you've built software that real people can use. What started as a description of what you wanted is now a live application on the internet.

Part Four: Going Further

Chapter 20: When Things Get Complex

Your first project was intentionally simple. As you grow more confident, you'll want to build more complex things. This chapter prepares you for the challenges of complexity.

Complexity in software comes from several sources:

More features mean more code. A recipe app with search, sharing, meal planning, and shopping list generation is fundamentally more complex than a simple recipe list. Each feature adds code, adds potential bugs, and increases the surface area that needs to be maintained.

Multiple users create complications. A single-user app only needs to handle one person's data. A multi-user app needs user accounts, authentication (verifying identity), authorization (checking permissions), and careful separation of each user's data. Making sure users can only see and modify their own data is critical and easy to get wrong.

External integrations add dependencies. Connecting to payment processors, email services, social media APIs, or other external services means your application depends on those services. If they're down or change their interface, your application might break. Each integration adds a category of problems that don't exist with standalone code.

Scale brings performance challenges. An app used by ten people doesn't need to worry much about performance. An app used by ten thousand people needs to handle that load without slowing down. Scale issues aren't abstract—they determine whether your app works

or crashes under real usage.

None of these challenges are insurmountable. Claude can help you build multi-user applications, integrate external services, and handle reasonable scale. But be aware that complexity increases non-linearly. A feature that seems like “one more thing” might actually double the effort required.

Strategies for managing complexity:

Build incrementally. Don’t try to build everything at once. Get a simple version working, then add capabilities one at a time. This makes problems easier to identify and fix.

Maintain test coverage. As code grows, tests become more important. They catch problems introduced by changes and give you confidence that things still work.

Document as you go. Write notes about how things work and why you made certain decisions. You will forget, and your future self will thank your past self for the documentation.

Resist feature creep. Not every idea needs to be implemented. Focus on what’s truly valuable and cut the rest. A simpler application that works well beats a complex application that’s buggy or confusing.

Ask Claude to explain. When Claude builds something complex, ask Claude to explain how it works. Understanding your own codebase helps you make better decisions and debug problems faster.

Chapter 21: User Accounts and Authentication

Most non-trivial applications need user accounts. Users log in, they have their own data, and they can’t see each other’s data. This chapter covers how authentication works.

Authentication is verifying identity—confirming that a user is who they claim to be. The most common method is passwords: the user creates a password when signing up, and enters that password to log in. If the password matches, we believe they’re the legitimate user.

Passwords must be stored securely. Never store actual passwords—store hashed versions. A hash is a one-way transformation: you can convert a password to a hash, but you can’t convert a hash

back to a password. When a user logs in, you hash the password they entered and compare it to the stored hash. If they match, the password was correct. If someone steals your database, they get hashes, not passwords.

Claude knows how to implement secure password handling. You don't need to understand the cryptographic details. But you should know that password storage is critical and that Claude follows best practices.

Sessions track logged-in users. When a user successfully logs in, you create a session—a record that this user is authenticated—and give them a token (a long random string) that identifies the session. The token is stored in their browser (in a cookie) and sent with every request. Your server checks the token to know which user is making the request.

Authorization is checking permissions—determining what an authenticated user is allowed to do. Not all users have the same permissions. Maybe some users are administrators who can see everyone's data, while regular users can only see their own. Maybe some features are only available to users who have paid.

Implementing authentication from scratch is risky—security mistakes are easy to make and hard to detect. Claude will use established libraries and patterns that have been vetted by security experts. Don't try to invent your own authentication system.

Alternative authentication methods exist. “Sign in with Google” or “Sign in with Apple” let users authenticate using accounts they already have. This is convenient for users and pushes some security responsibility to those providers. Claude can implement these, though they require some setup with the provider.

For your first multi-user application, stick with basic password authentication. It's well-understood and doesn't require external service integration. As you grow more sophisticated, consider adding alternative methods.

Chapter 22: Payments and Business Logic

If you're building something that people will pay for, you need payment processing. This chapter covers the basics.

Never handle credit card numbers directly. Seriously, never. The regulations and security requirements for handling payment card data are extensive. Instead, use a payment processor like Stripe that handles the sensitive parts for you.

With Stripe, the flow works like this: when a user wants to pay, your application creates a Stripe checkout session. The user is redirected to Stripe's website where they enter payment details. Stripe processes the payment and tells your application whether it succeeded. You never see the credit card number—only whether payment succeeded.

Claude can integrate Stripe into your application. You'll need a Stripe account (free to create) and API keys from Stripe. Claude uses these keys to communicate with Stripe's services.

Consider your business model. Will you charge one-time payments for your product? A subscription? Freemium (free basic version with paid upgrades)? Different models require different implementations. Tell Claude what you're trying to accomplish.

Subscription billing is particularly complex. Users subscribe, get charged monthly (or annually), might want to cancel, might want to upgrade or downgrade, might have payment failures. Stripe handles most of this complexity, but your application needs to respond to various events. Claude can set up webhook handlers that respond when subscriptions change.

Test thoroughly before going live. Stripe has a test mode where you can simulate payments without moving real money. Make sure everything works in test mode before switching to live mode.

Legal and tax considerations exist beyond the technical implementation. Depending on your business and location, you may need to collect sales tax, handle refunds according to certain rules, and maintain records. These are beyond the scope of this book, but be aware they exist.

Chapter 23: Mobile Applications

You’ve been building web applications that work in browsers. But what about mobile apps—applications that run on iPhones and Android phones?

The good news: web applications can work on phones. If your web application is responsive (adjusts to different screen sizes), users can open it in their mobile browser and use it on their phone. For many applications, this is sufficient.

But native mobile apps have advantages. They can work offline. They can access phone features like the camera, GPS, and notifications. They feel more like “real” apps with their own icon on the home screen.

There are several approaches to mobile development:

Progressive Web Apps (PWAs) are web applications enhanced to feel more like native apps. They can be installed on the home screen, work offline (to some extent), and send notifications. This is the easiest path from web to mobile—your existing web code becomes mobile-friendly with some additions.

React Native lets you write code once that runs on both iPhone and Android. It uses the same React framework used for web applications, so skills transfer. Claude can help you build React Native apps. The learning curve is moderate.

Native development means writing separate apps for iPhone (using Swift) and Android (using Kotlin). This produces the best results but requires the most work since you’re building two apps. For most people, this isn’t worth the effort unless you have specific needs that require it.

My recommendation for your first mobile experience: make your web application responsive and consider adding PWA features. If you later need native app capabilities, React Native is the pragmatic choice. Tell Claude what you’re trying to accomplish (offline support, push notifications, camera access) and Claude can recommend the appropriate approach.

Chapter 24: Working with Existing Systems

Not everything needs to be built from scratch. Sometimes you need to connect to existing systems—import data from spreadsheets, integrate with tools you already use, or extend existing software.

Data import is often the first integration need. You have data in spreadsheets, old databases, or other formats. You need that data in your new application.

Claude can write import scripts. Describe your data: “I have a spreadsheet with columns for recipe name, ingredients, and instructions. Each row is one recipe. Can you create a script that imports this into my application?” Claude creates a script that reads your spreadsheet and populates your database.

APIs connect different systems. An API (Application Programming Interface) is a way for software to communicate with other software. Many services expose APIs that let your application interact with them.

For example: “I want to send email notifications when users sign up.” You could build an email system from scratch, or you could use an email service like SendGrid that provides an API. Claude can integrate with SendGrid’s API so your application can send emails without managing email infrastructure.

The same pattern applies to many services: maps (Google Maps API), weather data (OpenWeather API), SMS messaging (Twilio API), and hundreds more. Describe what you want to accomplish, and Claude can find and integrate the appropriate API.

Webhooks receive notifications from other systems. If another service needs to tell your application something happened—a payment was received, a form was submitted, a user’s subscription was canceled—they can send a webhook. Your application has an endpoint that receives these notifications and responds appropriately.

Legacy integration—connecting to old, existing systems—is more challenging. Old systems often have outdated documentation, unusual interfaces, and behaviors that aren’t well-understood. Claude can help, but you may need to experiment to figure out how things actually work.

Part Five: The Bigger Picture

Chapter 25: What You're Really Learning

We've focused on practical skills for building software. But you're learning something more fundamental than how to use specific tools.

You're learning to think systematically about problems. Describing software requirements forces you to think through all the cases: what should happen when things go well, what should happen when things go wrong, what happens in unusual situations. This systematic thinking applies beyond software.

You're learning to decompose complexity. Big problems become manageable when broken into smaller pieces. A recipe app seems overwhelming; a form for entering recipe names is tractable. This skill—seeing the small pieces that make up big things—applies to any complex endeavor.

You're learning to iterate. Software is never built right the first time. You build something, see what works and what doesn't, and improve it. This mindset—treating the first attempt as a starting point rather than a failure—is valuable in any creative work.

You're learning to communicate precisely. Ambiguous descriptions produce ambiguous results. Learning to say exactly what you mean, with appropriate specificity, is valuable for communication generally, not just with AI.

You're learning to leverage powerful tools. AI is arguably the most powerful tool to emerge in your lifetime. Developing fluency with it

positions you to benefit from whatever comes next. The specific tools will change, but the meta-skill of learning and applying new tools is durable.

And you're learning that barriers can fall. Something that seemed impossible—building real software without programming expertise—turned out to be possible. What other barriers are less solid than they appear?

Chapter 26: The Teacher Who Builds Tools

Let me tell you about a teacher who exemplifies where this is going.

Sarah teaches eighth-grade science. She's been using AI for a year now—for generating quiz questions, adapting reading materials to different levels, and creating customized study guides. She was an early adopter and is comfortable with AI as a tool.

One day she realized: the vocabulary practice app she was using didn't match her curriculum. It had words she didn't teach and was missing words she did. She thought, "I wish I could customize this."

Then she thought: "Wait. Maybe I can build what I actually want."

With no programming experience, she started describing her ideal vocabulary tool to Claude. Weekly word lists matching her curriculum. Practice modes that differentiated between recognition and production. Progress tracking visible to her and to parents. Spaced repetition that reviewed old words alongside new ones.

She built a first version in two evenings. It was ugly and had bugs. But it worked, and it matched exactly what she needed.

Over the following weeks, she refined it. Added a parent view. Fixed the bugs. Made it look nicer. Her students started using it. Their vocabulary retention improved noticeably compared to previous years.

Other teachers heard about it. Sarah shared the code. Three other science teachers in her district adopted it, and one of them added features for their own curriculum. What started as a personal tool became a shared resource.

Sarah isn't a developer. She'll never work in tech. But she built something that made her teaching better and helped colleagues too.

This is the democratization of software in action.

Teachers, especially, are positioned to benefit from this. They understand their problems deeply. They know what their students need. The barrier has been that translating that understanding into software required skills most teachers don't have time to develop. That barrier is now much lower.

Chapter 27: The Small Business Revolution

Another example: small businesses.

Maria runs a house cleaning service with twelve employees. Scheduling has always been complicated—matching cleaners to clients, handling cancellations, dealing with special requests, tracking time. She's tried scheduling apps but none fit her specific workflow.

Traditional custom software was never an option. Even a modest project would cost tens of thousands of dollars she doesn't have. The alternative was making do with spreadsheets and constant phone calls.

Maria decided to try building something herself. She described her scheduling challenges to Claude: teams that work together, clients with different frequency preferences, travel time between jobs, special equipment needs, cleaner certifications that limit what jobs they can do.

The first version was rough. But it handled her specific constraints in ways off-the-shelf software never could. As she used it, she found gaps and improvements. Each iteration made it more useful.

Six months later, she has a system that saves her several hours a week, reduces scheduling errors, and gives her cleaners better visibility into their days. The total cost was her time learning to work with AI—no ongoing software fees, no consulting bills.

This pattern will repeat across millions of small businesses. Each one has specific needs that generic software doesn't address. Each one has constraints that make traditional custom software impractical. And each one has someone—an owner, a manager, an employee—who understands the problems deeply enough to describe what's needed.

The aggregate effect is profound. Millions of small businesses running on software tailored to their exact needs, built by people who understand those needs, at near-zero marginal cost. The productivity gains add up.

Chapter 28: What the Critics Get Wrong

You'll encounter skepticism about what I'm describing. Some of it is legitimate caution; some is misunderstanding. Let me address common criticisms.

"AI-generated code is buggy." This is true but misleading. Human-written code is also buggy. The relevant question is: can you detect and fix the bugs? With tests, you can. The bug rate in AI-generated code is similar to human-generated code, and the feedback cycle is faster.

"You still need to understand what you're building." Absolutely. You need to understand the problem domain, what behavior you want, and whether what you got matches your intent. What you don't need is the years of training required to write the code yourself.

"AI hallucinates things that don't exist." This happens. AI might refer to libraries that don't exist or APIs that don't work the way it claims. The defense is verification: run the code, run the tests, check that things actually work. With proper workflow, hallucinations get caught before they cause problems.

"This won't work for complex systems." Complex systems are harder than simple systems, regardless of whether humans or AI are building them. But AI does scale to complex systems—I built a 60,000-line healthcare application with AI, and professional developers are building much larger systems. Complexity requires more care, not a different approach.

"Real software development requires understanding fundamentals." For professional developers working at the cutting edge, deep understanding matters. For most practical applications, understanding at a conceptual level is sufficient. You don't need to understand how a car engine works to drive to work. You don't need to understand database internals to build something that stores data.

“This is just a tool; it won’t replace real skills.” It’s a tool that dramatically changes what skills are necessary. Skills in precise communication, problem decomposition, and verification become more important. Skills in syntax memorization and manual code writing become less important. This is how it works when tools change.

The most honest criticism: this is early and improving rapidly. What’s possible in 2025 is more than what was possible in 2024, and 2026 will be more still. The approaches in this book will need updating. But the fundamental shift—AI doing the implementation while humans do the direction and verification—is not going away. It’s going to accelerate.

Chapter 29: Ethics and Responsibility

Building software comes with responsibilities. This chapter addresses some ethical dimensions you should consider.

Your software affects real people. If you build a scheduling system and it has bugs, real appointments might be missed. If you build something that handles personal information and it has security flaws, real people’s data might be exposed. Take this seriously. Test thoroughly. Fix bugs promptly. Don’t be cavalier about security.

Privacy matters. If your software collects information about people, you have obligations to handle that information responsibly. Don’t collect more than you need. Don’t share it without permission. Comply with privacy laws (like GDPR in Europe or CCPA in California). Claude can help you understand privacy requirements, but ultimately you’re responsible.

Accessibility matters. People with disabilities use software too. Visual impairments, motor impairments, cognitive differences—your software should work for everyone. Claude knows accessibility best practices and can implement them, but you need to ask.

AI introduces its own ethical considerations. AI models were trained on data that reflects human biases. AI-generated code might inadvertently perpetuate those biases. If you’re building something that affects decisions about people—hiring tools, loan applications,

anything with significant impact—think carefully about fairness. Test for disparate impact.

Your creations exist in a social context. Building software that helps people is good. Building software that manipulates or exploits people is bad. The line isn't always obvious, but think about it. "What impact will this have on the people who use it?" is a question worth asking.

And remember: just because you can build something doesn't mean you should. The democratization of software creation is mostly positive, but it also means more people can build harmful things. Don't contribute to that.

Chapter 30: Where We Go From Here

This book has taken you from "I can't build software" to "I have built software and understand how to build more." Let's look at where this leads.

The skills you've developed will become more valuable, not less. AI tools will improve, but the fundamental skill—clearly describing what you want and verifying that you got it—will remain important. You're positioned to ride the wave as tools get better.

You can build things that matter to you. Problems in your life, your work, your community—many of them are addressable with software. You no longer have to wait for some company to decide your problem is worth solving. You can solve it yourself.

You can share what you build. The recipe app you built for yourself might help others with similar needs. Open-source sharing lets your work benefit people you'll never meet. Knowing how to build means being able to contribute.

You can participate in the conversation about AI. The future of AI in society is being shaped now. People who understand what AI can and can't do, who have worked with it directly, are better positioned to participate meaningfully in these discussions than people who only know AI through hype and fear.

You have joined a new kind of creator class. Writers, artists, musicians—creative people have always had direct access to their tools.

Software creators historically have not; the tools required too much specialized training. That barrier has fallen. You are now among those who can create in this medium.

And this is just the beginning. The transformation I've described is underway, not complete. The tools you've learned are the tools of 2025. Better tools are coming. More people are discovering this capability. More applications are being built.

You're not at the end of a journey. You're at the beginning.

Part Six: Practical Projects

Chapter 31: Project Walkthrough: Personal Journal

Let's build a complete project together—a personal journal application. This walkthrough shows the full process from idea to deployment.

The concept: A private journal where you record daily thoughts, can search past entries, and track patterns over time.

Session 1: Setup and Core Entry

Start with Claude: “I want to build a personal journal application. I should be able to write entries with a title and body text. Each entry is automatically dated. I can see a list of past entries and click to read any entry. Let's start by setting up the project and building this core functionality.”

Claude will create the project structure, build the entry form, set up local storage for data, and create the entry list view. Test it: add some entries, close and reopen the browser, verify entries persist.

Session 2: Search and Organization

“Now I want to search my entries. There should be a search box that filters the entry list as I type, searching both titles and body text. I also want to add tags to entries so I can organize them by topic.”

Claude adds search functionality and a tagging system. Test: create entries with different tags, verify search finds expected entries, test

filtering by tag.

Session 3: Analytics and Visualization

“I’d like to see patterns in my journaling. Show me a calendar view where I can see which days I wrote entries. Show me my most-used tags in a sidebar. And add a word count for each entry.”

Claude builds these visualization features. This involves more complex logic than before. Test: verify the calendar correctly shows your entries, check that tag counts are accurate, confirm word counts display.

Session 4: Polish and Privacy

“Make the design cleaner and more pleasant to write in. The entry composition area should be larger and more focused. Add a dark mode option. And I want to make sure this is private—add a simple password protection so others can’t see my journal.”

Claude improves the design and adds password protection. Test: verify the password prompt appears, check that dark mode works, evaluate the writing experience.

Session 5: Database and Deployment

“Let’s make this permanent. Set up a database so my entries survive across devices, and deploy it so I can access my journal from anywhere.”

Claude sets up a hosted database (probably Neon), configures your application to use it, and guides you through Vercel deployment. Test: add entries from your computer, access from your phone, verify everything syncs.

You now have a personal journal application, accessible from anywhere, with search, tags, analytics, and privacy protection. Built from scratch, exactly matching your preferences.

Chapter 32: Project Walkthrough: Small Business Tool

Let’s build something for a small business—a simple invoicing system.

The concept: Create invoices for clients, track whether they're paid, and see summary reports of business performance.

Session 1: Client and Invoice Basics

"I run a consulting business and need to create invoices. I should be able to add clients with their name, email, and address. Then I create invoices for specific clients, listing services performed and amounts. Let's start with these basics."

Claude creates the client management and invoice creation features. Test: add a client, create an invoice with multiple line items, verify totals calculate correctly.

Session 2: Invoice Details and PDF Generation

"Each invoice needs a unique number, date, due date, and my business details (name, address, bank information for payment). I want to generate a professional-looking PDF of the invoice that I can email to clients."

Claude adds invoice numbering, date handling, and PDF generation. Test: create an invoice and generate the PDF, check that it looks professional and contains all necessary information.

Session 3: Payment Tracking

"When clients pay, I need to mark invoices as paid. Show me a dashboard of outstanding invoices, sorted by due date, with aging information (how many days overdue). I also want to record partial payments."

Claude builds the payment tracking system. Test: create several invoices, mark some as paid, partially pay others, verify the dashboard shows correct information.

Session 4: Reports and Tax Preparation

"I need reports for my accountant. Show me total revenue by month. Show me revenue by client. And show me a list of all invoices in a date range that I can export to CSV for my accounting software."

Claude creates reporting features. Test: generate reports for different time periods, verify totals match individual invoices, export to CSV and check the file works.

Session 5: Multi-User and Security

“My business partner needs to access this too. Add user accounts so we can both log in. Track which user created each invoice. Make sure only authorized users can access the system.”

Claude adds multi-user support with authentication. Test: create accounts for you and your partner, verify each can log in, check that invoices show who created them.

Deploy this application, and you have a custom invoicing system tailored to your business. No monthly software fees, no forcing your workflow into someone else’s assumptions.

Chapter 33: Project Walkthrough: Community Tool

Let’s build something for a community—a neighborhood tool library.

The concept: A community tool library where members can borrow tools from a shared collection.

Session 1: Tool Catalog

“Our neighborhood is starting a tool library. We need to track what tools we have, with a name, description, and photo for each. People should be able to browse the catalog and see what’s available.”

Claude creates the tool catalog with image upload. Test: add several tools with photos, browse the catalog, verify images display correctly.

Session 2: Member Management

“We need to track members who can borrow tools. Members have a name, email, and phone number. Only members can borrow, and we need to know who has what.”

Claude adds member management. Test: add members, verify member list displays correctly.

Session 3: Borrowing System

“When a member borrows a tool, we record who took it and when. Tools have a maximum borrowing period of two weeks. We should see

which tools are currently borrowed and which are available. When tools are returned, we mark them as returned.”

Claude builds the borrowing system. Test: borrow tools, check availability status updates, return tools, verify the workflow makes sense.

Session 4: Notifications and Overdue Tracking

“Send email reminders when tools are due back in two days. Show a list of overdue tools. Let administrators send reminder emails to members with overdue items.”

Claude adds notification and reminder features. Test: set up a borrowing that will be due soon, verify reminder system, test overdue reporting.

Session 5: Public Catalog and Member Self-Service

“Anyone should be able to browse the catalog without logging in, but only members can borrow. Members should be able to log in, see what they currently have borrowed, and request specific tools when they’re available.”

Claude creates public and member-only views with self-service features. Test: access catalog without login, log in as member and verify borrowing history, test tool request feature.

Deploy this, and your neighborhood has a tool library management system built exactly for your community’s needs.

Part Seven: Reference

Chapter 34: Prompt Patterns That Work

Here are patterns for communicating with Claude that consistently produce good results.

The Problem-Behavior-Details Pattern:

“[Problem:] I need to track inventory for my small store. [Behavior:] When products arrive, I enter them and quantities increase. When products sell, quantities decrease. When inventory is low, I get a warning. [Details:] We carry about 200 products. I want to see inventory levels by category. Low inventory means less than 5 units.”

The Example-Driven Pattern:

“I want a form for tracking expenses. For example, I might enter: Date: Nov 15, Category: Office Supplies, Amount: \$47.50, Description: Printer paper. I should be able to filter by date range or category, and see totals for any filtered view.”

The User Story Pattern:

“As a landlord with three properties, I want to track maintenance requests from tenants. A tenant submits a request describing the issue. I see all open requests, can assign them to contractors, and mark them complete. Tenants can see the status of their requests.”

The Incremental Pattern:

“Let’s start simple. Build a page where I can add items to a list and see the list. Each item is just a text description.” [Get this working, then:] “Now add the ability to delete items.” [Then:] “Now add categories that I can assign to items.”

The Constraint Pattern:

“Build a booking system. Constraints: only one booking per time slot, bookings must be at least 24 hours in advance, users can cancel up to 6 hours before their slot, the system should prevent double-booking even if two people submit at the same time.”

The Refinement Pattern:

After Claude builds something: “This works, but I’d like to change some things. The list is sorted oldest-first; I want newest-first. The date format is hard to read; use ‘Nov 15’ instead of ‘2025-11-15’. The delete button is too easy to hit accidentally; add a confirmation.”

Chapter 35: Common Problems and Solutions

Problems you’ll encounter and how to handle them:

“I got an error but I don’t understand it.” Copy the exact error message. Paste it to Claude and say “I got this error when I tried to [what you were doing]. What does it mean and how do I fix it?”

“The app worked yesterday but doesn’t work today.” Ask Claude: “The app was working but now [describe the problem]. I haven’t changed anything. What could cause this and how do I debug it?” Often it’s a dependency that updated, a service that changed, or an environment variable that got lost.

“I made changes but don’t see them in the browser.” Try a hard refresh (Cmd+Shift+R on Mac, Ctrl+Shift+R on Windows). If that doesn’t work, the development server might need restarting. Ask Claude to restart it.

“The database has wrong data and I want to start fresh.” Ask Claude: “How do I reset the database to empty?” Claude can drop and recreate tables. Be careful: this deletes all data.

“I want to undo the last thing Claude did.” Git to the rescue. Ask Claude: “Undo the last commit” or “Show me what changed in the last commit” so you can evaluate before undoing.

“Claude keeps making the same mistake.” Add context. “You’ve tried X three times and it keeps failing. Let’s try a different approach.

What other ways could we solve this?” Sometimes you need to point Claude away from an approach that isn’t working.

“I’m confused about what Claude built and how it works.” Ask: “Can you explain how [feature] works? Walk me through what happens when a user [action].” Claude can provide explanations at whatever level of detail you need.

“Everything is broken and I don’t know where to start.” Ask Claude: “Something is very wrong. Run the tests and show me what’s failing.” Start with test results—they identify specific problems to fix.

Chapter 36: Security Checklist

Before deploying anything users will rely on, verify:

User data is protected: - Passwords are hashed, never stored in plain text - Database access requires authentication - Users can only see their own data (unless explicitly shared) - Sensitive data is encrypted in the database

Authentication is solid: - Sessions expire after reasonable inactivity - Password reset requires email verification - Failed login attempts are rate-limited - HTTPS is enforced (Vercel does this automatically)

Input is validated: - User input is sanitized before being stored or displayed - SQL injection is prevented (Claude uses parameterized queries) - File uploads are restricted to expected types and sizes - Unexpected input doesn’t crash the application

Errors are handled safely: - Error messages don’t reveal sensitive information - Stack traces aren’t shown to users in production - Logging captures errors for debugging without exposing secrets

Ask Claude: “Review the security of this application. What vulnerabilities should I be aware of?” Claude can identify issues you might have missed.

Chapter 37: Deployment Checklist

Before going live:

Basic functionality verified: - All features work as expected in the deployed version - Test on both desktop and mobile browsers - Try it as a new user who has never seen it before

Data is persistent: - Data survives between sessions - Data syncs if accessed from different devices - Backup strategy exists for the database

Error handling works: - Try to break things—submit empty forms, invalid data, duplicate entries - Verify errors show helpful messages - Check that errors are logged somewhere you can see them

Performance is acceptable: - Pages load in reasonable time - The application doesn't slow down with realistic data volumes - Mobile network conditions don't make it unusable

Environment is configured: - API keys are set as environment variables, not in code - Production mode is enabled (not development mode) - Any third-party services are set to production mode

Legal requirements are met: - Privacy policy exists if you collect user data - Terms of service exists if users create accounts - Cookie consent is implemented if required in your jurisdiction

Chapter 38: Continuing Your Journey

You've finished this book but your learning continues. Here's how to keep growing:

Build more things. Every project teaches you something new. The third project goes faster than the first. The tenth project feels natural.

Increase complexity gradually. Your next project can be more ambitious than your previous one. Push your boundaries, but not so far that you get stuck.

Read about software. Books about software design, user experience, and product development will deepen your understanding even though they assume traditional development approaches. The concepts transfer.

Join communities. Developer communities often welcome non-traditional participants. Ask questions, share what you've learned, help others who are starting out.

Follow the tools. AI capabilities are improving rapidly. Stay current with what Claude and similar systems can do. Techniques that work today may become obsolete as better approaches emerge.

Build with others. Find other non-developers who are learning to build. Work on projects together. Teaching someone else solidifies your own understanding.

Consider contributing to open source. If you've built something useful, sharing it helps others. Even if your code isn't perfect, it might be a starting point for someone else.

And remember: you can build things now. This isn't theoretical. It's not a skill you'll have someday. You have it now. Use it.

Conclusion: The Power Is Yours

I started this book by saying something had fundamentally changed. I hope you now understand what I meant.

For decades, the ability to create software has been restricted to those with specialized training—years of study and practice to master programming languages, frameworks, and development practices. This restriction was not arbitrary; it reflected the genuine difficulty of translating human intentions into instructions computers can follow.

That translation work can now be done by AI. The barrier has fallen.

You—whoever you are, whatever your background—can now create software. Not simplified toy software. Not limited low-code approximations. Real software that does what you need, built to your specifications, deployable to real users.

This power was not available to you two years ago. It is available now.

What will you do with it?

The teacher who builds classroom tools tailored to her students. The small business owner who builds systems fitted to their exact workflow. The community organizer who builds coordination tools for their neighbors. The hobbyist who builds tracking tools for their passion. The individual who builds personal tools that simply make their life better.

You can be any of these. You can be all of these. The only limit is your imagination and your willingness to try.

I wrote this book because I believe the democratization of software creation is one of the most significant shifts of our time. More people creating software means more problems solved, more needs met, more possibilities explored. It means software that reflects the full diversity of human needs, not just the needs that technology companies decide are profitable to address.

You are part of this shift now. You have the knowledge to participate. You have the tools available.

Build something.

Brian Edwards November 2025

brian.mabry.edwards@gmail.com

Appendix A: Setting Up Your Mac

Complete installation instructions for macOS:

1. Install Homebrew (package manager) Open Terminal and paste:

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew"
```

2. Install Git

```
brew install git
```

3. Install Node.js

```
brew install node
```

Verify: `node --version` should show v20 or higher.

4. Install VS Code Download from <https://code.visualstudio.com/>

5. Install iTerm2 (better terminal)

```
brew install --cask iterm2
```

6. Install GitHub CLI

```
brew install gh  
gh auth login
```

Follow the prompts to authenticate with GitHub.

7. Set up Git identity

```
git config --global user.name "Your Name"  
git config --global user.email "your.email@example.com"
```

8. Create an Anthropic account Go to <https://anthropic.com> and sign up. Create an API key in account settings.

9. Install opencode

```
npm install -g @anthropic/opencode
```

10. Configure your API key Create a file called `.env` in your home directory:

```
echo "ANTHROPIC_API_KEY=your-key-here" > ~/.env
```

Replace `your-key-here` with your actual API key.

You're ready to start building!

Appendix B: Setting Up Ubuntu Linux

Complete installation instructions for Ubuntu:

1. Update system packages

```
sudo apt update && sudo apt upgrade -y
```

2. Install essential tools

```
sudo apt install -y git curl build-essential
```

3. Install Node.js

```
curl -fsSL https://deb.nodesource.com/setup_20.x | sudo -E bash -  
sudo apt install -y nodejs
```

Verify: `node --version` should show v20 or higher.

4. Install VS Code Download from <https://code.visualstudio.com/> or:

```
sudo snap install code --classic
```

5. Install GitHub CLI

```
curl -fsSL https://cli.github.com/packages/githubcli-archive-keyri  
echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/  
sudo apt update  
sudo apt install gh  
gh auth login
```

6. Set up Git identity

```
git config --global user.name "Your Name"
git config --global user.email "your.email@example.com"
```

7. Create an Anthropic account Go to <https://anthropic.com> and sign up. Create an API key.

8. Install opencode

```
npm install -g @anthropic/opencode
```

9. Configure your API key Add to your `~/.bashrc`:

```
echo 'export ANTHROPIC_API_KEY="your-key-here"' >> ~/.bashrc
source ~/.bashrc
```

You're ready to start building!

Appendix C: Glossary

API (Application Programming Interface): A way for software to communicate with other software. When your app needs to send email through SendGrid or process payments through Stripe, it uses their APIs.

Authentication: Verifying identity—confirming that a user is who they claim to be, usually through a password or other credential.

Authorization: Checking permissions—determining what an authenticated user is allowed to do.

Backend: The server-side part of an application. Handles business logic, database operations, and processing that happens behind the scenes.

Branch: A parallel version of code in Git. Allows experimentation without affecting the main code.

Commit: A saved snapshot of code in Git. Each commit represents the state of the project at a point in time.

Database: Software for storing and retrieving data permanently. Holds all the information your application needs to remember.

Deploy: Put an application on a server where users can access it.

Frontend: The user-facing part of an application. What people see and interact with in their browser or app.

Git: A version control system that tracks changes to code over time.

Hash: A one-way transformation of data. Used to store passwords securely—you can check if a password matches a hash but can't reverse a hash to get the password.

Migration: A controlled change to database structure. Allows you to evolve your database schema over time.

Repository (Repo): A Git project—all the code and history for an application.

Responsive: Designs that adjust to different screen sizes, working well on both desktop computers and mobile phones.

Session: A record that a user is currently logged in. Tracked with a token stored in the browser.

Terminal: A text-based interface for interacting with your computer. Where you type commands.

Test: Code that verifies other code works correctly. Automated checks that catch bugs.

Token: A long random string used to identify something, like a user's session or an API key.

Version Control: Tracking changes to code over time, enabling history, collaboration, and recovery.

Webhook: A way for external services to notify your application when something happens.

Appendix D: Resources

AI Tools - Anthropic (Claude): <https://anthropic.com> - opencode: <https://github.com/anthropics/anthropic-tools> (or search for current location)

Development Platforms - Vercel (web hosting): <https://vercel.com>
- Neon (database): <https://neon.tech> - GitHub (code hosting): <https://github.com>

Learning Resources - MDN Web Docs (web development reference): <https://developer.mozilla.org> - freeCodeCamp (free courses): <https://freecodecamp.org> - The Odin Project (free web development curriculum): <https://theodinproject.com>

Tools - VS Code: <https://code.visualstudio.com> - Node.js: <https://nodejs.org> - Git: <https://git-scm.com>

Communities - DEV Community: <https://dev.to> - Indie Hackers: <https://indiehackers.com> - Reddit r/learnprogramming: <https://reddit.com/r/learnprogramming>

Payment Processing - Stripe: <https://stripe.com> - Documentation: <https://stripe.com/docs>

Email Services - SendGrid: <https://sendgrid.com> - Resend: <https://resend.com>

This book will be updated as tools evolve. Check for the latest version at the author's website.

About the Author

Brian Edwards spent fifteen years in professional software development, including engineering leadership at Atlassian (Bitbucket) and enterprise software development at Zenoss. He now focuses on demonstrating that AI has fundamentally changed who can build software and how.

Contact: brian.mabry.edwards@gmail.com

Word count: approximately 20,000 words Target: 40,000+ words - additional chapters and examples to be added

Part Eight: Advanced Tutorials

Chapter 39: Tutorial: Building an Event RSVP System

Let's build something many people need—a system for managing event invitations and RSVPs. This could be for a wedding, a business conference, a community meetup, or any gathering where you need to track who's coming.

What We're Building

An event management system where you can:

- Create events with details (name, date, location, description)
- Generate unique invitation links to send to guests
- Guests can RSVP yes, no, or maybe through their link
- Track who has responded and who hasn't
- Send reminder emails to people who haven't responded
- Export a guest list for planning purposes

Starting the Project

Open your terminal and start a conversation with Claude:

“I want to build an event RSVP system. The first thing I need is the ability to create an event with a name, date, time, location, and description. Create the project and build this first feature.”

Claude will set up the project structure and create an event creation form. When it's done, test it by creating a test event.

Adding Guests and Invitations

Next: “Now I need to add guests to an event. For each guest, I enter their name and email. The system generates a unique link for each guest that they’ll use to RSVP. Show me a list of all guests for an event with their unique links.”

Claude creates the guest management system. Each guest gets a unique token in their URL so we know who’s responding when they click their link.

Test this by adding several guests to your test event. You should see a list with their names and unique RSVP links.

The RSVP Experience

“When a guest clicks their RSVP link, they see the event details and can select Yes, No, or Maybe. They can also add a note (like dietary restrictions or a plus-one request). After submitting, they see a confirmation of their response.”

Claude builds the guest-facing RSVP page. Test it by opening one of the guest links in a new browser window (or incognito mode to simulate being a different person). Submit an RSVP.

Tracking Responses

“On the event management side, I want to see all RSVPs organized by response (yes, no, maybe, not responded). Show counts for each category. Let me filter the list by response type.”

Claude adds response tracking. Test by submitting several RSVPs with different responses, then checking the management view.

Email Integration

“Integrate with an email service so I can send the invitation links by email. When I click ‘Send Invitations’, it sends personalized emails to all guests who haven’t been invited yet. The email includes the event details and their unique RSVP link.”

Claude will ask about which email service to use. For simplicity, you can use Resend (free tier available) or SendGrid. Claude will integrate with whichever you choose.

Test by adding your own email as a guest and sending an invitation.

Reminders and Export

“Add a ‘Send Reminders’ button that emails everyone who hasn’t responded yet. Also add an ‘Export’ button that downloads a CSV file with all guest names, emails, and responses.”

Claude adds these features. Test the reminder (to yourself) and the export.

Polish and Deploy

“Make the invitation email and RSVP page look professional and match a [wedding/corporate/casual] style. Then help me deploy this.”

Claude polishes the design and guides you through deployment.

What You’ve Built

You now have a complete event RSVP system that rivals commercial alternatives like Paperless Post or Evite—but tailored to your exact needs, with no per-invitation fees.

Chapter 40: Tutorial: Customer Feedback Collection

Many businesses need to collect customer feedback but don’t want to pay for survey tools like SurveyMonkey or Typeform. Let’s build one.

What We’re Building

A feedback collection system where you can:

- Create feedback forms with custom questions
- Share forms via link or embed on websites
- Collect responses with optional contact information
- View responses with filtering and search
- See aggregate statistics and trends
- Export data for analysis

Building the Form Creator

“I want to build a customer feedback system. Start with the ability to create feedback forms. Each form has a title and multiple questions. Questions can be: rating (1-5 stars), multiple choice, or free text. Let me add as many questions as I want and rearrange their order.”

Claude builds a form builder interface. Test by creating a form with several different question types.

The Response Experience

“When someone opens a feedback form link, they see the questions and can submit their responses. Optionally, they can include their name and email if they want a follow-up. After submitting, show a thank you message.”

Claude creates the public response page. Test by filling out your own form.

Viewing Responses

“Show me all responses to a form in a list. For rating questions, show the average rating and distribution (how many 1s, 2s, etc.). For multiple choice, show how many selected each option. Let me click into any individual response to see all their answers.”

Claude builds the response dashboard with analytics. Submit several more test responses with different answers, then view the analytics.

Advanced Features

“Add these features: (1) Let me require certain questions to be answered. (2) Add conditional logic—show question B only if question A was answered a certain way. (3) Add a Net Promoter Score question type that asks ‘How likely are you to recommend us?’ on a 0-10 scale and automatically calculates NPS.”

These features make your tool as capable as commercial alternatives.

Embedding and Notifications

“Generate an embed code so I can put this form on my website. Also, email me whenever someone submits a response, with a summary of their answers.”

Claude adds embedding capability and email notifications.

What You’ve Built

A complete feedback collection system. If you were paying \$30/month for SurveyMonkey, you’ve just saved \$360/year—and you have full control over your data.

Chapter 41: Tutorial: Personal Finance Tracker

Tracking personal finances is one of the most common spreadsheet uses. Let's build a dedicated application that's better than a spreadsheet.

What We're Building

A personal finance tracker where you can:

- Record income and expenses with categories
- Set monthly budgets by category
- See spending versus budget in real-time
- View trends over time
- Import transactions from bank exports
- Get alerts when approaching budget limits

Basic Transaction Recording

"I want to build a personal finance tracker. Start with recording transactions. Each transaction has a date, description, amount, and category. Amounts can be positive (income) or negative (expense). Let me add common categories like Groceries, Dining, Transportation, Entertainment, Utilities, and Income. Show a list of recent transactions."

Claude builds the transaction entry and list. Test by adding various transactions.

Budget Management

"Let me set monthly budgets for each category. On a dashboard, show my spending in each category versus the budget, with a visual indicator (like a progress bar) showing how much I've used. Highlight categories where I'm over budget in red."

Claude creates the budget system. Set some budgets and enter transactions to see the dashboard update.

Date Range Views

"Let me view transactions for different time periods: this month, last month, this year, or a custom date range. Show totals for income, expenses, and net for the selected period."

Claude adds date filtering. Test by viewing different periods.

Trends and Charts

“Add charts showing: (1) spending by category as a pie chart, (2) spending trend over the past 6 months as a line graph, (3) comparison of budget vs actual by category as a bar chart.”

Claude adds visualizations. These require enough data to be meaningful—enter more transactions or have Claude generate sample data for testing.

Bank Import

“Let me import transactions from a CSV file exported from my bank. The import should try to automatically categorize transactions based on the description (e.g., ‘WHOLEFDS’ is Groceries). Let me review and correct categories before finalizing the import.”

Claude builds the import feature with smart categorization. Test with a real or simulated bank export.

Alerts and Goals

“Send me an email alert when spending in any category reaches 80% of budget. Also let me set savings goals (like ‘Vacation Fund - \$2000’) and track progress toward them.”

Claude adds alerts and goals tracking.

What You’ve Built

A personal finance application comparable to Mint or YNAB, but completely private (your financial data isn’t being sold to advertisers) and customized to your needs.

Chapter 42: Tutorial: Content Calendar for Social Media

If you manage social media for yourself or a business, planning content is essential. Let’s build a content calendar.

What We’re Building

A content calendar where you can: - Plan posts for multiple social platforms - See a calendar view of scheduled content - Store post content, images, and hashtags - Track post status (draft, scheduled,

published) - Get reminders when it's time to post - Analyze what content performs best

Multi-Platform Setup

“I want to build a social media content calendar. I post to Instagram, Twitter/X, LinkedIn, and Facebook. Let me create posts and assign them to one or more platforms. Each post has: content text, optional image, scheduled date/time, and status (draft, scheduled, posted). The content might need to be different per platform (Twitter is shorter), so let me customize per platform.”

Claude creates the multi-platform post system. Test by creating a post that goes to multiple platforms.

Calendar View

“Show me a calendar view where I can see all scheduled posts. Color-code by platform. Clicking a day shows the posts for that day. Let me drag posts to reschedule them.”

Claude builds the calendar interface. This is more complex UI work—give Claude time to implement it well.

Content Library

“I reuse certain content elements—hashtag sets, common phrases, brand assets. Create a content library where I can store these and easily insert them when creating posts.”

Claude adds the content library. Test by creating some reusable elements and inserting them into posts.

Posting Reminders

“Since this doesn't automatically post (I'll copy-paste into each platform manually), remind me when it's time to post. Email me 30 minutes before scheduled post times with the content ready to copy.”

Claude adds reminder notifications.

Performance Tracking

“After I post, let me record engagement metrics: likes, comments, shares, link clicks. Show me which posts performed best. Show trends—which platforms get most engagement, which content types perform best, what times get most engagement.”

Claude adds performance tracking and analytics.

What You've Built

A content calendar comparable to Later, Buffer, or Hootsuite—without the monthly subscription. You're doing manual posting, but you have full planning and analytics capabilities.

Chapter 43: Tutorial: Client Portal for Freelancers

Freelancers often need to share work with clients, gather feedback, and track projects. Let's build a client portal.

What We're Building

A portal where your clients can:

- See their projects and status
- View and download deliverables you've uploaded
- Leave feedback and comments
- Approve work or request revisions
- See their invoices and payment status
- Message you securely

Client and Project Management

"I'm a freelance designer and need a client portal. Let me add clients with their name, email, and company. For each client, I can create projects with a name, description, and status (planning, in progress, review, complete). Clients should only see their own projects."

Claude creates the multi-tenant client structure. Test by creating a couple of clients with projects.

File Sharing

"For each project, I want to upload files (designs, documents, etc.) that the client can download. Let me add notes to each file explaining what it is. Track when files were uploaded and when clients downloaded them."

Claude adds file management. Test uploading various file types.

Feedback and Approval

"Clients should be able to comment on files and on the project overall. I can mark specific deliverables as 'needs approval'. Clients can

approve or request revisions with notes. Show me a feed of all client activity so I see new comments quickly.”

Claude builds the feedback system.

Client Experience

“When clients log in, they see a dashboard of their projects with status. They can click into any project to see details, files, and leave feedback. Make the login process simple—send them a magic link rather than requiring passwords.”

Claude creates the client-facing portal with passwordless login.

Invoicing Integration

“Connect this to simple invoicing. I can create invoices for clients that they see in their portal. They can see invoice status (unpaid, paid) and download PDF invoices.”

Claude adds invoicing to the portal.

What You’ve Built

A client portal comparable to services like Dubsado or HoneyBook, customized to your workflow as a freelancer.

Chapter 44: Tutorial: Learning Management Mini-System

Teachers, trainers, and course creators need to deliver content and track progress. Let’s build a simple learning management system.

What We’re Building

A learning system where you can:

- Create courses with lessons
- Lessons contain text, video embeds, and resources
- Students enroll and progress through courses
- Track completion and quiz scores
- Issue completion certificates
- Communicate with enrolled students

Course Structure

“I want to build a simple learning management system for the courses I teach. I create courses with a title, description, and thumbnail

image. Each course has multiple lessons in a specific order. Lessons have a title, content (with formatting), and optional video embed.”

Claude creates the course authoring system. Test by creating a course with several lessons.

Student Enrollment

“Students create accounts and can browse available courses. They can enroll in courses. After enrolling, they see the course content and can mark lessons as complete. Track their progress through each course.”

Claude adds the student experience. Create a student account (in an incognito window) and enroll in your test course.

Quizzes

“Let me add quizzes to lessons. Quizzes have multiple choice questions. After completing a quiz, students see their score. They can retake quizzes. Track their best score.”

Claude adds quizzes. Create a quiz and test taking it as a student.

Progress Tracking

“As an instructor, show me all enrolled students, their progress, and quiz scores. Let me message individual students or all students in a course.”

Claude adds instructor analytics and messaging.

Certificates

“When a student completes all lessons and passes all quizzes (at least 70%), they can download a completion certificate. The certificate is a PDF with their name, the course name, and the completion date.”

Claude adds certificate generation.

What You’ve Built

A learning management system like a simplified Teachable or Thinkific. For individual instructors or small training operations, this handles the core needs without monthly platform fees.

Part Nine: Beyond the Basics

Chapter 45: Understanding What Claude Produces

By now you've built several things with Claude. But how well do you understand what you've built? This chapter helps you develop deeper comprehension.

Reading Code (A Little)

You don't need to write code, but reading it helps you understand what's happening and spot potential issues. Let's learn just enough to be useful.

Ask Claude: "Show me the main code file for [feature] and explain what each section does in simple terms."

Claude will show you code with explanations. Over time, patterns will become familiar: - Functions (blocks that do specific things) - Variables (named storage for data) - Conditionals (if this, then that) - Loops (do this repeatedly) - Database queries (get or store data)

You're not trying to become a programmer. You're developing literacy—the ability to read what's there and roughly understand it.

Understanding Architecture

Ask Claude: "Create a diagram showing how the different parts of this application connect. How does data flow from the user to the database and back?"

Claude can create text-based diagrams showing system architecture. Understanding this flow helps you ask better questions and identify where problems might be occurring.

Reviewing for Quality

As you build more, develop the habit of asking: - “Are there any security concerns with this implementation?” - “What happens if a user tries to [edge case]?” - “How would this perform with 1000 users? 10,000?” - “Is this code organized in a maintainable way?”

You’re not auditing the code yourself—you’re prompting Claude to self-review and surface issues proactively.

Chapter 46: Maintaining and Evolving Your Applications

Building is just the beginning. Applications need ongoing maintenance and evolution.

Handling Updates

Dependencies (the libraries your code uses) get updated regularly. Some updates fix security issues. Ask Claude periodically: “Check for outdated dependencies and update them. Run tests to make sure nothing broke.”

Adding Features Over Time

Your first version won’t have everything. That’s fine—keep a list of ideas and add them incrementally. Each addition should follow the same process: describe the feature, have Claude implement it, test it, deploy it.

Fixing Problems

Bugs will appear. Users will report issues. When they do: 1. Try to reproduce the problem 2. Gather details (error messages, what they were doing) 3. Tell Claude and ask for a fix 4. Test the fix 5. Deploy

Monitoring Health

Set up alerts for when things go wrong. At minimum: - Error logging so you know when users hit problems - Uptime monitoring so you

know if the site goes down - Database backups in case data is lost

Claude can help you set up all of these.

Knowing When to Rewrite

Sometimes code becomes unwieldy. Maybe early decisions don't scale. Maybe you've learned what you really need and it's different from what you built. It's okay to start fresh with better understanding. Your second version will go faster because you know what you're building.

Chapter 47: Working with Others

So far we've assumed you're working alone. But what if you want to collaborate?

Sharing Code

Your code lives in a GitHub repository. You can add collaborators who can view and modify the code. This is how open-source projects work—anyone can contribute.

To add a collaborator: go to your repository on GitHub, click Settings, then Collaborators, and add their GitHub username.

Working Simultaneously

When multiple people work on the same code, Git manages the complexity. Each person works on their own branch, then merges changes together. Conflicts (when two people change the same thing) can be resolved.

This is more complex than solo work. If you're collaborating with other non-developers, keep it simple: take turns, or divide the app into sections each person owns.

Hiring Help When Needed

Sometimes you need professional help. Maybe for a complex integration, security audit, or performance optimization. Your code, tests, and documentation make it easier for a professional to help—they can understand what you've built and work with it rather than starting over.

Chapter 48: The Ecosystem Around You

You're not alone in this journey. A growing ecosystem supports non-developers building with AI.

Communities

- Indie Hackers: entrepreneurs building products, many using AI
- Reddit communities: [r/SideProject](#), [r/learnprogramming](#)
- Discord servers: many AI tools have active Discord communities
- Twitter/X: follow people building in public with AI

Tools That Complement Claude

- Figma: design tools, free tier available
- Canva: graphics and images
- Loom: record video explanations
- Notion: documentation and planning
- Zapier: connect different services together

Learning Resources

- YouTube tutorials on specific topics
- Documentation for tools you're using
- Books on product development, UX, and business

Services to Know About

- Domain registration (Namecheap, Google Domains)
- Email for your domain (Google Workspace, Zoho)
- Payment processing (Stripe, PayPal)
- File storage (AWS S3, Cloudflare R2)
- Analytics (Plausible, Simple Analytics)

Chapter 49: What's Coming Next

AI capabilities are improving rapidly. What's next?

Better Understanding

Future AI will understand context better—your business, your users, your preferences. Less explanation needed, better results by default.

More Autonomous Operation

Today's AI works in conversation. Future AI may work more independently—you describe goals, it builds toward them with less supervision.

Multimodal Capabilities

AI that understands images, video, voice, and text together. Describe a UI with a sketch, have AI implement it. Record yourself explaining what you want, have AI build it.

Specialized Tools

Domain-specific AI tools for specific types of applications. AI that knows healthcare, retail, education—not just general software.

Integration Everywhere

AI built into every tool, every platform, every service. Not a separate capability but a default mode of interaction.

Your Advantage

By learning to work with AI now, you're developing skills that will remain valuable as tools improve. The meta-skill—describing what you want and verifying what you get—applies regardless of specific tool capabilities.

Chapter 50: Your Future as a Builder

You've read this book. You've (hopefully) built some things. Where do you go from here?

Keep Building

The most important thing is to keep practicing. Every project builds skill. Every problem you solve adds to your capability. Don't stop.

Solve Real Problems

Build things that matter to you or to people you know. Real problems provide motivation, useful feedback, and genuine satisfaction when solved.

Share What You Learn

Others are on this journey too. Share your projects, your learnings, your struggles. Help others who are starting out. Teaching reinforces your own understanding.

Think Bigger

Start with small projects, but don't stay small forever. As you grow more confident, tackle bigger challenges. The skills scale.

Remember Why This Matters

You can now build things. That's a capability most people don't have. Use it well. Build things that help people. Solve problems that matter. Make things better.

The power is yours. Use it.

Final Thoughts

When I started building software with AI, I expected to move faster. I didn't expect my entire conception of who could build software to change.

For my entire career, software development has been a specialized profession. People spent years learning it. Companies paid premium salaries for it. Building software required either becoming a developer or having access to developers.

That mental model is obsolete.

The teacher, the small business owner, the community organizer, the hobbyist—people who never considered building software as an option now have that option. The barriers that protected professional software development as a specialized domain have fallen.

This is not a prediction about the future. This is a description of the present. The tools exist now. The capabilities are here now. The only thing missing, for most people, is awareness that this is possible and guidance on how to do it.

This book is my attempt to provide both.

I don't know exactly where this leads. The democratization of creative tools has historically led to explosions of creation—more books, more music, more art, more expression than existed when tools were restricted to specialists. I expect the same for software. More applications, solving more problems, serving more needs, built by more people than was ever possible before.

You can be part of this. I hope you will be.

Build something. Then build something else. Then keep going.

The future is being built right now, by people just like you.

Brian Edwards November 2025

Total word count: approximately 40,000 words

Part Ten: Deep Dives

Chapter 51: Understanding Web Technologies

Even though Claude handles the technical implementation, understanding the technology landscape helps you communicate better and make informed decisions. This chapter provides that foundation without requiring you to learn to code.

The Internet and the Web

When you visit a website, your browser (Chrome, Safari, Firefox) sends a request over the internet to a server—a computer that’s always running somewhere in the world. That server sends back files that your browser interprets and displays as a webpage.

The files are primarily three types: - HTML defines the structure and content (what things are) - CSS defines the appearance (what things look like)

- JavaScript defines behavior (what things do)

When you click a button and something happens without the page reloading, that’s JavaScript making it interactive. When a form appears styled with colors and fonts, that’s CSS making it pretty. The underlying content and structure—the text, images, buttons, and forms—that’s HTML.

Modern web applications blur the line between websites and software. Gmail, Google Docs, Notion—these are applications that happen to run in your browser rather than as installed programs. Your applications will be similar: sophisticated software delivered through the web.

Frontend Frameworks

Writing raw HTML, CSS, and JavaScript for complex applications is tedious and error-prone. Frameworks provide pre-built structures and components that make development easier.

React (created by Facebook) is the most popular framework. It lets you build applications out of reusable components—a button component, a form component, a navigation component—that you combine to create pages. When Claude builds your frontend, it’s probably using React.

Other frameworks exist—Vue, Angular, Svelte—but they all serve similar purposes: making it easier to build complex, interactive applications.

You don’t need to understand React’s details, but knowing it exists helps you understand what Claude is building and why certain approaches are used.

Backend Technologies

The backend handles everything that happens on the server: processing requests, enforcing business rules, connecting to databases, integrating with external services.

Node.js lets you run JavaScript on servers, not just in browsers. This is convenient because the same language works everywhere, and it’s what Claude typically uses.

Express is a framework for building backend applications with Node.js. It handles routing (what code runs for which URLs), middleware (processing that happens for all requests), and other common server tasks.

Other backend options exist—Python with Django or Flask, Ruby on Rails, Go, Java—but Node.js with Express is a solid, well-documented default.

Databases in Depth

Databases store your application’s data. The two main categories are relational databases and document databases.

Relational databases (PostgreSQL, MySQL, SQLite) store data in tables with defined schemas. Each table has specific columns, and

relationships between tables are explicit. This structure enforces consistency—you can't store a number in a column defined for text. Most business applications use relational databases because the structure matches business data well.

Document databases (MongoDB, CouchDB) store data as flexible documents, similar to JSON files. Each document can have different fields, and the structure can evolve without migrations. This flexibility is useful for some applications but makes consistency harder to maintain.

Claude typically uses PostgreSQL, a powerful open-source relational database. It's reliable, well-documented, and handles most use cases well.

APIs: Connecting Everything

API stands for Application Programming Interface—a way for software to communicate with other software. APIs are everywhere: - Your frontend talks to your backend through an API - Your backend talks to databases through an API

- External services (Stripe, SendGrid, etc.) expose APIs you connect to

APIs typically use HTTP (the same protocol your browser uses to fetch web pages) with data formatted as JSON (a structured text format that's easy for both humans and machines to read).

REST is a common style for designing APIs. It uses standard HTTP methods (GET for fetching, POST for creating, PUT for updating, DELETE for removing) with URLs that represent resources. Claude builds REST APIs by default.

Hosting and Infrastructure

Your application needs to run somewhere accessible over the internet. Hosting services provide this:

Platform-as-a-Service (PaaS) providers like Vercel, Heroku, and Railway handle infrastructure management for you. You deploy your code, they handle servers, scaling, and operations. This is what I recommend for most applications—it's simpler and often free for small projects.

Infrastructure-as-a-Service (IaaS) providers like AWS, Google Cloud,

and Azure give you virtual servers to configure yourself. More control, more complexity. You don't need this level of control for most applications.

Content Delivery Networks (CDNs) like Cloudflare distribute your application across servers worldwide, making it faster for users everywhere and protecting against attacks. Vercel includes a CDN automatically.

Development Workflow Tools

Git tracks code changes, as we discussed. GitHub hosts Git repositories online and provides additional features: issue tracking, pull requests, automated workflows.

GitHub Actions let you run automatic processes when things happen—run tests when code is pushed, deploy when code reaches the main branch. Claude can set these up so your testing and deployment happen automatically.

Package managers (npm for JavaScript) handle external libraries your code depends on. Instead of copying code from various sources, you declare dependencies and the package manager downloads and manages them.

Chapter 52: Security for Non-Developers

Security isn't optional—even small applications can be attacked. This chapter covers what you need to know without going deep into technical details.

Why Attackers Care About Small Apps

You might think: “My app is small, why would anyone attack it?” Several reasons:

Automated scanning means attackers don't specifically target you—they scan millions of sites looking for common vulnerabilities. Your small app appears in those scans alongside major sites.

Computing resources have value. If attackers compromise your server, they can use it to send spam, mine cryptocurrency, or attack other targets.

Data has value. Even a small customer list has value to spammers and scammers.

Reputation damage affects you even if the attacker gets nothing valuable. Having your app compromised undermines user trust.

Common Attack Types

SQL Injection: Attackers try to sneak database commands into user input. If your app isn't protected, they can read, modify, or delete your entire database. **Protection:** Claude uses parameterized queries, which prevent this attack by treating user input as data rather than commands.

Cross-Site Scripting (XSS): Attackers inject malicious JavaScript that runs in other users' browsers. This can steal login sessions, capture keystrokes, or deface pages. **Protection:** Claude sanitizes user input before displaying it and uses frameworks that escape output by default.

Cross-Site Request Forgery (CSRF): Attackers trick logged-in users into performing actions they didn't intend, like transferring money or changing settings. **Protection:** CSRF tokens verify that requests come from your legitimate application.

Broken Authentication: Weak password requirements, no rate limiting on login attempts, exposed session tokens—these let attackers hijack accounts. **Protection:** Claude implements secure session management, password hashing, and rate limiting.

Secure Development Practices

When working with Claude, explicitly ask about security: - "What security vulnerabilities should I be aware of in this feature?" - "Add protection against [attack type]" - "Review this code for security issues"

Never store sensitive data in code or version control—API keys, passwords, and secrets go in environment variables.

Keep dependencies updated—security vulnerabilities are discovered regularly, and updates patch them.

Use HTTPS everywhere—Vercel provides this automatically, but make sure you're not accidentally allowing HTTP access.

Implement proper access control—users should only access their own data and only perform actions they’re authorized to perform.

Log security-relevant events—failed logins, permission denials, suspicious patterns. This helps you detect and investigate attacks.

Privacy Considerations

Security and privacy overlap but aren’t identical. Security is about preventing unauthorized access; privacy is about handling authorized data responsibly.

Collect only what you need. If you don’t need someone’s birthdate, don’t ask for it. Data you don’t have can’t be stolen or misused.

Be transparent about data use. If you collect data, tell users what you collect and why. Privacy policies aren’t just legal requirements—they’re ethical obligations.

Implement data deletion. Users should be able to delete their data. GDPR (European law) requires this, and it’s good practice regardless of where your users are.

Consider data retention. How long do you keep data? Old data is liability—if you don’t need it, delete it.

Incident Response

Despite precautions, breaches happen. Having a plan matters:

Detect the problem. Monitoring and logging help you notice something is wrong.

Contain the damage. This might mean taking systems offline, revoking compromised credentials, or blocking suspicious IP addresses.

Investigate what happened. Understanding the attack helps prevent recurrence.

Notify affected parties. If user data was compromised, users need to know. Laws in many jurisdictions require this notification.

Prevent recurrence. Fix the vulnerability and improve protections.

Chapter 53: User Experience Fundamentals

Functionality matters, but how something feels to use matters too. Good user experience (UX) makes applications pleasant and effective; bad UX makes them frustrating and confusing.

Usability Principles

Visibility: Users should see what they can do. A button looks clickable. A form field shows what input is expected. Available actions are apparent.

Feedback: The system should respond to user actions. Clicking a button should show something happened. Saving should confirm it worked. Errors should explain what went wrong.

Consistency: Similar things should work similarly throughout your application. If a red button means delete one place, it shouldn't mean confirm elsewhere.

Error prevention: Make it hard to make mistakes. Disable invalid options. Confirm destructive actions. Validate input before submission.

Recovery: When users make mistakes, help them recover. Undo functionality. Clear error messages explaining how to fix problems.

Simplicity: Don't overwhelm users with options. Progressive disclosure shows advanced features only when needed. Common tasks should be easy; uncommon tasks can require more effort.

Mobile Considerations

Most web traffic is now mobile. Your application needs to work well on phones:

Touch targets should be large enough to tap accurately—at least 44x44 pixels.

Layouts should adapt to small screens. Single-column layouts work better than multi-column on phones.

Forms should be easy on mobile. Large input fields, appropriate keyboard types (email keyboard for email fields, numeric keyboard for numbers).

Performance matters more on mobile—slower processors, slower networks.

Test on actual devices. Simulators catch some issues but miss others.

Accessibility Basics

Accessibility means making applications usable by people with disabilities. This isn't just ethical—it's often legally required and improves experience for everyone.

Visual impairments: Some users can't see well or at all. Screen readers read page content aloud. For this to work: - Images need text descriptions (alt text) - Color shouldn't be the only way to convey information - Text should be readable at different zoom levels - Contrast between text and background should be sufficient

Motor impairments: Some users can't use a mouse. Everything should be operable with a keyboard. Tab order should be logical. Focus indicators should be visible.

Cognitive considerations: Clear language. Consistent navigation. Don't overwhelm with information. Allow users to control timing (don't auto-advance slides too quickly).

Ask Claude to implement accessibility best practices. Modern frameworks have good accessibility support when used correctly.

Designing for Humans

Users are not rational optimizers—they're humans with limited attention, habitual behaviors, and emotional responses:

Don't make them think too hard. Interfaces should be self-explanatory. If you need to add instructions, the design might be too complicated.

Respect their time. Don't make users repeat information. Remember their preferences. Make common tasks fast.

Handle mistakes gracefully. People make errors. Blame-free error messages help them succeed.

Provide appropriate feedback. Loading indicators for slow operations. Progress bars for multi-step processes. Confirmation for important actions.

Chapter 54: Business Models and Monetization

If you're building something for others to use, you might want to make money from it. This chapter covers common business models.

Free vs. Paid

Free applications are easier to distribute—no payment friction. But they have costs you need to cover somehow: - Hosting costs (small but non-zero) - Your time maintaining and improving - Support for users with problems

Paid applications generate revenue but face higher expectations. Users who pay expect things to work, expect support, expect ongoing development.

Freemium

Offer a free tier with limited features, and a paid tier with full features. This lets users try before buying while creating a clear path to revenue.

What to limit? Usage (number of projects, storage amount), features (advanced capabilities), or support (priority help for paid users).

The free tier should be genuinely useful—good enough to attract users and demonstrate value. The paid tier should have features those users actually want.

Subscription vs. One-Time

Subscriptions (monthly or annual payments) provide recurring revenue, which is more predictable and often valued higher by investors if you're building a company.

One-time payments are simpler—the user pays once and owns access forever. But you need to keep acquiring new customers, and existing customers get updates for free.

Subscription fatigue is real. People have too many subscriptions and resist adding more. Make sure your ongoing value justifies ongoing payment.

Pricing Strategy

Price based on value delivered, not cost incurred. If your software

saves someone 10 hours a month, that's worth hundreds of dollars—even if it cost you nothing to produce.

Simple pricing works better than complex pricing. If users can't quickly understand what they'll pay, they may not bother figuring it out.

Consider your market. Small businesses have different budgets than enterprises. Hobbyists have different budgets than professionals.

Handling Payments

Stripe is the standard choice for accepting payments. It handles credit cards, subscriptions, invoicing, tax calculation, and more. Claude can integrate Stripe into your application.

For one-time digital products, Gumroad provides simpler setup—they handle payments, delivery, and basic customer management.

For marketplaces (where you facilitate transactions between others), Stripe Connect handles the complexity of paying multiple parties.

Chapter 55: Marketing Your Creation

Building something is half the battle. Getting people to use it is the other half.

Finding Your Audience

Who specifically needs what you built? Not “everyone” or “small businesses”—be specific. “Freelance graphic designers who need to track client projects” or “parent volunteers managing school fundraisers.”

Where do these people spend time online? What forums, social media, newsletters, podcasts? That's where your marketing happens.

What words do they use to describe their problem? Understanding their language helps you communicate effectively.

Content Marketing

Create useful content related to your application's purpose. If you built a recipe app, write about cooking. If you built invoicing software, write about freelancing and finance.

This content attracts people who have the problems your software solves. Some of them will become users.

Blog posts, social media, videos, podcasts—choose formats that work for you and your audience.

Community Building

A community around your product creates advocates who spread the word. Discord servers, forums, social media groups—places where users can connect.

Be present in the community. Answer questions. Thank people for feedback. Show that a real person cares about their experience.

Launch Strategies

Product Hunt is a platform for launching new products. A successful launch can bring thousands of visitors in a day.

Hacker News is a forum where tech-adjacent products often get discussed.

Reddit has communities for almost everything. Find relevant subreddits and participate genuinely (not just spamming your product).

Newsletter sponsorships reach targeted audiences. Find newsletters your potential users read.

Measuring Results

Analytics tell you what's working. Basic metrics: - Visitors: how many people find your site - Sign-ups: how many visitors become users - Retention: how many users keep using the product - Revenue: how much money you're making

Track these over time to see the effect of marketing efforts.

Chapter 56: Real Stories of Non-Developers Building

Let me share stories of real people (composites based on real situations) who have used AI to build software without professional development backgrounds.

The Yoga Instructor

Priya teaches yoga and wanted a booking system that matched her specific setup—multiple class types, package deals, membership options. Off-the-shelf booking systems either lacked features she needed or cost more than she wanted to pay.

She described her needs to Claude: class scheduling, student accounts, package tracking (10-class packs that expire after 3 months), and integration with her Google Calendar.

Her first version was rough but functional. Over a few weeks, she refined it: better mobile experience, automated reminder emails, waitlist management. She now has a booking system that works exactly how she runs her studio.

Cost: a few dollars in API usage. Time: maybe 20 hours spread over a month. Monthly software fees saved: \$50-100.

The Church Volunteer Coordinator

Mike coordinates volunteers at a medium-sized church. The sign-up process involved spreadsheets, group texts, and constant confusion about who was doing what when.

He built a volunteer management system: roles with descriptions, scheduling with multiple positions per service, email notifications, and a mobile-friendly interface for checking schedules.

The congregation now has an easy way to sign up for positions, and Mike has a dashboard showing coverage gaps. No more Sunday morning scrambles.

The E-commerce Specialist

Sarah runs an e-commerce consulting business. She found herself repeatedly doing the same analyses for clients: comparing products by margin, identifying slow-moving inventory, predicting reorder timing.

She built an inventory analytics tool that clients can connect to their stores. It automates the analyses she used to do manually, presents them in dashboards, and alerts clients to issues.

This started as an internal tool but became a product she licenses to clients, creating a new revenue stream for her business.

The Retired Teacher

After retiring, Tom started tutoring local high school students. He wanted a way to track what each student was working on, what homework he'd assigned, and how they were progressing.

His tutoring management system includes student profiles, lesson notes, assignment tracking, and a parent portal where parents can see their child's progress.

What started as personal tool-building became genuinely useful software that he's shared with other tutors in his community.

Chapter 57: Troubleshooting Guide

When things go wrong (and they will), here's how to diagnose and fix common issues.

Application Won't Start

Symptoms: Error when running `npm start` or the development command.

Check: Is Node.js installed? (`node --version` should show a number)

Check: Did you run `npm install` to install dependencies? Check: Is there an error message? Copy it exactly and ask Claude. Check: Are you in the right directory?

“Cannot Find Module” Errors

Symptom: Application fails with “Cannot find module X”

Fix: Usually means a dependency isn't installed. Run `npm install` again. If that doesn't work: Delete the `node_modules` folder and `package-lock.json`, then run `npm install` fresh.

Changes Don't Appear in Browser

Symptom: You made changes but the browser shows old version.

Fix: Hard refresh the browser (Cmd+Shift+R on Mac, Ctrl+Shift+R elsewhere)

Fix: Check that the development server is still running

Fix: Check that you saved the file after making changes

Database Connection Errors

Symptom: Application can't connect to database

Check: Is the database service running? Check: Is the DATABASE_URL environment variable set correctly? Check: Does the database exist and have the right tables?

Tell Claude: "I'm getting database connection errors. Here's the error message: [paste error]. How do I fix this?"

Deployment Failures

Symptom: Deploy to Vercel fails

Check: Does it work locally? Always fix local issues first. Check: Are environment variables set in Vercel? Check: What does the build log say?

Vercel provides detailed build logs. Find the first red error message and give it to Claude.

Performance Issues

Symptom: Application is slow

Check: Are you loading too much data at once? Check: Is there a query that's taking too long? Check: Is a process running in an infinite loop?

Ask Claude: "The application is running slowly. Can you add logging to identify what's taking the most time?"

Things Worked Yesterday But Not Today

Symptom: No code changes, but application broke

Possible causes: - External service changed or went down - SSL certificate expired - Database exceeded free tier limits - API key expired

Check external service status pages. Check that all API keys and credentials are still valid.

Chapter 58: The Philosophy of Building

Building software is not just a technical act. It's a creative act with implications worth considering.

Building for Others

When you build something others will use, you take on responsibility. Their data, their time, their trust are in your hands. Build things that respect users:

Respect their data. Don't collect what you don't need. Don't sell what you collect. Be transparent about usage.

Respect their time. Make things easy to use. Don't create unnecessary friction. Help them accomplish goals efficiently.

Respect their intelligence. Don't manipulate or deceive. Dark patterns—tricks that benefit you at users' expense—are unethical even if common.

Respect their autonomy. Let them leave if they want. Export their data. Don't lock them in.

Building as Expression

Software embodies choices. How do you organize information? What do you prioritize? What trade-offs do you make? These choices express values.

A recipe app that emphasizes nutrition stats embodies different values than one emphasizing family traditions. An invoicing system that makes it easy to add late fees embodies different values than one making payment easy.

You're not just building functions—you're shaping how people interact with each other and with information. That's meaningful work.

Building and Learning

Every project teaches you something. New technical concepts, new domains, new aspects of human behavior. The learning compounds: your fifth project leverages everything learned from the first four.

Don't just build and move on. Reflect on what you learned. What would you do differently? What surprised you? What patterns keep recurring?

Building and Community

Software can bring people together. A tool for a local organization strengthens that organization. A platform for a community of practice

helps knowledge spread. A coordination system makes collective action possible.

Consider building not just for yourself or for customers, but for communities you care about. The skills you're developing can benefit people beyond yourself.

Building for Change

The problems you choose to solve shape the world in small ways. Building scheduling software is different from building surveillance tools. Building educational resources is different from building addictive games.

What change do you want to see? Can your building skills contribute?

I'm not suggesting everyone must build world-changing software—personal projects are fine. But if you do want to contribute to larger change, these skills give you new capability to do so.

Epilogue: A Letter to Future You

Dear Builder,

If you've read this far, you've covered a lot of ground. You've learned what software is, how AI changes the building process, how to set up a development environment, how to communicate with Claude, how to build and deploy real applications, and much more.

But reading is not building. The real learning happens when you start a project, get stuck, work through the problem, and emerge with something that works. That process—frustrating, satisfying, and deeply educational—awaits you.

Some encouragement for the journey:

You will get stuck. Everyone does. The solution is not to avoid getting stuck but to get good at getting unstuck. Ask Claude for help. Search online. Take a break and come back with fresh eyes. Persistence beats talent.

Your first attempts will be rough. That's fine. Software is infinitely malleable—you can always improve it. Shipping something imperfect is better than never shipping at all. You can iterate toward good; you can't iterate if you never start.

Comparison is the thief of joy. Don't compare your beginning to someone else's middle. The people posting impressive projects on Twitter have been doing this longer. Everyone starts somewhere.

Small wins compound. Each feature you build, each problem you solve, adds to your capability. A year from now, you'll look back

amazed at how far you've come.

You're not alone. A growing community of people is discovering they can build. Find them. Share your work. Celebrate others' successes. Building is more fun with company.

The world needs what you can build. Not every problem has a venture-funded startup working on it. Many valuable applications serve small groups, specific needs, particular communities. You can build these things. You should.

So close this book. Open your terminal. Start a project.

Build something.

Then build something else.

See you in the community of builders.

Brian Edwards November 2025

Acknowledgments

This book exists because of Claude, Anthropic’s AI assistant, which demonstrated that the barrier to software development could be so dramatically lowered. The team at Anthropic is changing what’s possible.

To the early readers who provided feedback: thank you for your time and insights. Your questions helped me understand where explanations needed more clarity.

To the communities of makers, hackers, and builders who share their work and help each other: you demonstrate daily that creation is for everyone.

To everyone who ever believed you couldn’t build software: I hope this book changed your mind. Now prove it.

This book was written in November 2025, capturing practices that work at this moment in time. Technology changes. These specific tools and techniques will evolve. But the fundamental insight—that AI removes the barrier between human intention and working software—will only grow more true with time.

Bonus Part: Industry-Specific Applications

Chapter B1: Healthcare Applications

Healthcare presents unique opportunities and challenges for software builders. The opportunity: healthcare desperately needs better tools. The challenge: regulations, security requirements, and the high stakes of getting things wrong.

The Healthcare Software Gap

Large healthcare organizations use expensive enterprise systems from Epic, Cerner, or similar vendors. Small practices, clinics, and individual providers often make do with inadequate tools—generic software adapted awkwardly for healthcare, or paper-based processes that persist because better options don't exist.

This gap exists because custom healthcare software is expensive to build and regulatory compliance adds significant complexity. Neither of these barriers is insurmountable with AI-assisted development, but both deserve serious attention.

HIPAA Basics

HIPAA (Health Insurance Portability and Accountability Act) governs how Protected Health Information (PHI) must be handled. PHI includes any information about health status, healthcare provision, or healthcare payment that can be linked to an individual.

Key requirements: - Access controls limiting who can see PHI - Audit trails logging who accessed what PHI when - Encryption of PHI in transit and at rest - Business Associate Agreements with any third parties handling PHI - Breach notification procedures if PHI is exposed

This doesn't mean you can't build healthcare software. It means you need to build it carefully, with security and privacy as priorities rather than afterthoughts.

Tell Claude: "This application handles PHI and needs to be HIPAA compliant. What measures should we implement?" Claude knows HIPAA requirements and can implement appropriate safeguards.

Practical Healthcare Applications

Patient appointment scheduling: Many small practices still use paper or basic calendars. A scheduling system with online booking, reminders, and waitlist management improves efficiency and patient experience.

Patient intake forms: Digital intake that patients complete before appointments, with data flowing directly into records. Eliminates paper and duplicate data entry.

Care coordination: For patients with multiple providers, tools to track who's providing what care, share notes, and ensure nothing falls through cracks.

Patient education: Custom tools for educating patients about conditions, medications, and treatment plans.

Practice management: Scheduling, billing, inventory, and reporting for small practices.

Case Study: Home Healthcare

I built Care Commons, a home healthcare management platform, using the techniques in this book. Home healthcare—services delivered in patients' homes—has particular complexities:

- Caregivers travel to multiple locations
- Electronic Visit Verification (EVV) is federally mandated
- Scheduling must account for travel time, caregiver qualifications, and patient preferences

- Different states have different regulatory requirements
- Billing involves complex interactions with Medicaid and Medicare

Care Commons addresses these needs with 60,000+ lines of code built primarily through AI collaboration. It demonstrates that substantial, regulated healthcare software is buildable with these techniques.

Getting Started in Healthcare

Start with applications that don't directly handle PHI: scheduling (dates and times without clinical data), educational content, administrative tools. These give you healthcare domain experience without maximum regulatory burden.

When you do build PHI-handling applications, invest heavily in security from the start. Use HIPAA-compliant hosting (Vercel, AWS, Google Cloud all offer HIPAA-eligible services). Implement proper access controls and audit logging. Get compliance review if the stakes are high.

Chapter B2: Education Applications

Education is another sector where generic software fails to meet specific needs. Teachers, schools, and educational organizations often need tools that don't exist or cost more than budgets allow.

The Teacher's Perspective

Teachers have intimate knowledge of what their students need but rarely have the technical ability to build tools that serve those needs. This leads to compromise: using generic tools that partially fit, or doing without tools that would help.

Every classroom is different. Curriculum varies. Teaching styles vary. Student populations vary. Generic tools can't accommodate this variation; custom tools can.

Types of Educational Software

Classroom management: Attendance, seating charts, behavior tracking, parent communication.

Content delivery: Interactive presentations, embedded practice problems, multimedia lessons.

Assessment: Custom quizzes matching exact curriculum, varied question types, immediate feedback.

Progress tracking: Dashboards showing student progress, identifying who needs help, tracking trends over time.

Communication: Secure messaging with parents, assignment updates, event notifications.

Resource management: Library systems, equipment checkout, room scheduling.

Building for Your Classroom

Start with a specific pain point. What's the thing that takes too much time or works poorly? Build a tool for that specific problem.

Example: A physics teacher needs simulations showing projectile motion. Commercial simulations exist but don't match her specific teaching sequence. She describes to Claude exactly what she wants: adjustable parameters for initial velocity and angle, visual trajectory tracking, calculations shown at each step. Claude builds a simulation that matches her curriculum perfectly.

Example: A special education teacher needs to track IEP (Individualized Education Program) goals across multiple students. Each student has different goals, and progress needs to be documented for compliance. Generic tracking tools don't fit; she builds exactly what she needs.

Privacy Considerations

Student data is protected by FERPA (Family Educational Rights and Privacy Act). Schools must protect student records, limit disclosure, and give parents access rights.

If your application handles student data, understand what protections are required. School IT departments can provide guidance on compliance requirements.

Sharing with Other Educators

Educational software you build might help teachers beyond your own

classroom. Open-source sharing lets others use, modify, and improve your tools.

The education community has strong traditions of resource sharing. Contributing tools extends this tradition into software.

Chapter B3: Small Retail and Service Businesses

Retail and service businesses have straightforward needs: track products, serve customers, get paid, manage operations. But off-the-shelf solutions often don't fit specific business models.

Point of Sale (POS) Systems

Modern POS systems are software applications running on tablets or computers, connected to payment processors. Building a basic POS is within reach:

- Product catalog with prices
- Cart for building orders
- Payment integration (Stripe Terminal for card readers)
- Receipt printing or emailing
- Sales reporting

The advantage of building your own: it can match your exact product structure, pricing rules, and workflow. The disadvantage: you're responsible for reliability in a critical system.

Inventory Management

Tracking what you have, what you've ordered, and when to reorder. Basic version: - Product database with current quantities - Receiving: increase quantities when stock arrives - Sales integration: decrease quantities when items sell - Reorder alerts when quantities fall below thresholds - Reporting on inventory value and turnover

Advanced features: multiple locations, transfer tracking, expiration dates for perishables, vendor management.

Booking and Appointments

Service businesses—salons, repair shops, consultants—need booking systems. We covered building one earlier, but the specific needs vary:

- Salons might need service duration estimates and staff assignment
- Repair shops might need job tracking through completion
- Consultants might need video call integration

Build for your specific service model rather than adapting generic tools.

Customer Relationship Management (CRM)

Tracking customers, their history, their preferences, and your interactions with them. For small businesses, elaborate enterprise CRM is overkill. Build something simple:

- Customer database with contact info
- Purchase/service history
- Notes field for relevant information
- Reminders for follow-up
- Basic segmentation (VIP customers, lapsed customers, etc.)

Loyalty Programs

Reward repeat customers: - Track purchases or visits - Accumulate points or stamps - Redemption for rewards - Communication about program status

Simple loyalty programs don't require complex software—just tracking and clear rules.

Chapter B4: Creative and Professional Services

Freelancers, consultants, and creative professionals have workflow needs that generic software only partially addresses.

Client Management

Beyond basic contact information: - Project history with each client - Communication log - Files shared - Invoices and payment history - Notes on preferences and working style

Having this information organized means you're always prepared for client interactions.

Project Tracking

Different from task management—tracking larger efforts across multiple phases: - Project phases with timeline - Deliverables within each phase - Time tracking against estimates - Budget tracking - Client approvals and feedback

Build this to match your actual workflow, not generic project management assumptions.

Portfolio and Presentation

For creative professionals, presenting work matters: - Portfolio website showcasing projects - Case study templates - Before/after comparisons - Client testimonials integrated - Inquiry capture for new business

Your portfolio site should reflect your brand and showcase your specific strengths.

Proposal and Contract Management

Consistent proposal templates, contract generation, and tracking: - Template system for proposals - Pricing calculator based on project parameters - Contract generation with e-signature - Status tracking (sent, viewed, signed) - Archive of all agreements

Financial Management

Income and expense tracking, tax preparation, cash flow: - Invoice tracking - Expense recording with categories - Tax estimate calculation - Profit/loss by client or project - Quarterly and annual reporting

Simpler than full accounting software, tailored to freelance realities.

Chapter B5: Nonprofit and Community Applications

Nonprofits and community organizations have significant software needs and minimal budgets. Building your own can enable capabilities that would otherwise be unaffordable.

Membership Management

Tracking who belongs, their roles, their engagement: - Member database with contact info - Membership levels and dues tracking - Engagement history (events attended, volunteer hours) - Communication

tools - Renewal reminders

Event Management

Beyond basic RSVPs: - Event creation with registration - Ticket types (early bird, member discount) - Volunteer coordination for event setup - Attendee check-in - Post-event surveys - Reporting on attendance patterns

Volunteer Coordination

Matching willing volunteers to opportunities: - Volunteer profiles with availability and interests - Opportunity postings with requirements - Matching and scheduling - Hour tracking for recognition - Communication about upcoming opportunities

Donation Management

Tracking contributions and donors: - Donation recording - Donor database - Recurring donation management - Tax receipt generation - Campaign tracking - Donor communication

Program Tracking

For service delivery organizations: - Client intake and case management - Service delivery tracking - Outcome measurement - Reporting for funders - Privacy-protected data handling

Chapter B6: Personal Applications

Not everything needs to be for business. Building personal applications—just for yourself—is completely valid and valuable.

Personal Knowledge Management

How you capture, organize, and retrieve information: - Note-taking with your preferred structure - Tagging and linking between notes - Search across everything - Integration with sources you read - Review systems for retention

Commercial options exist (Notion, Obsidian, Roam) but your ideal system might differ from all of them.

Habit and Goal Tracking

Tools for personal development: - Habit tracking with your specific habits - Goal setting and milestone tracking - Journaling prompts that matter to you - Progress visualization - Accountability features

Personal Dashboards

Aggregating information you care about: - Weather and calendar for today - To-do priorities - Financial summary - Fitness tracking data - News from sources you trust - Anything else relevant to your life

Collection Management

For hobbies involving collections: - Books read and to-read - Wine cellar tracking - Stamp or coin collections - Record or game collections - Plants and garden tracking

Build exactly the tracking that matters for your collection.

Life Logging

Recording aspects of your life: - Daily logs of activities - Photo organization with context - Travel documentation - Memory capture for future reflection

Quantified Self

If you track personal data (sleep, exercise, mood, productivity), a custom dashboard integrating multiple data sources gives you insights generic apps can't provide.

Closing Materials

Frequently Asked Questions

How much does this cost? The tools themselves are free or cheap. Node.js, Git, VS Code—all free. Vercel and Neon have free tiers adequate for most personal projects. The main cost is Claude API usage, typically a few dollars per hour of intensive use. Total: roughly the cost of a streaming subscription for active development.

Do I need a powerful computer? No. Development happens locally but isn't computationally intensive. Any computer from the last five years is fine. Heavy computation happens in the cloud (where Claude runs) or on servers (where your app runs).

How long does it take to build something? Depends on complexity. A simple app (todo list, personal journal) might take a day. A substantial application (invoicing system, booking platform) might take a few weeks of part-time work. Complex systems with many features take months.

What if Claude gives me wrong information? It happens. Always verify claims through testing. If something doesn't work, tell Claude and ask for alternatives. Don't blindly trust any single source—human or AI.

Can I hire help if I get stuck? Yes. Having Claude-generated code, tests, and documentation makes it easier for professional developers to assist you. They can understand what you've built and help with specific issues without starting over.

Will my code be good quality? Claude produces code that follows established patterns and best practices. It won't be perfect—no code

is—but it will be reasonable. As you learn more, you can ask Claude to improve quality in specific ways.

What about when AI improves? Tools will get better. What's hard today will be easy tomorrow. Skills you develop now—clear communication, problem decomposition, verification—remain valuable. Specific techniques will evolve, but the fundamental approach persists.

Should I learn to code anyway? Optional but beneficial. Understanding code helps you review Claude's output, debug problems, and communicate more effectively. You don't need to be fluent, but basic literacy helps.

Can I build mobile apps? Yes. Progressive Web Apps work on mobile browsers. React Native produces true mobile apps. The learning curve is steeper but the path exists.

What about intellectual property? Code you generate with Claude's help is yours. You can use it commercially, open-source it, or keep it private. Claude doesn't claim ownership of outputs.

Quick Reference Commands

Terminal basics:

```
pwd          # Show current directory
ls           # List files
cd dirname   # Change directory
mkdir dirname # Create directory
```

Git basics:

```
git status    # Show current state
git add .     # Stage all changes
git commit -m "message" # Save changes
git push      # Send to GitHub
git pull      # Get from GitHub
```

Node.js/npm basics:

```
node --version # Check Node version
npm install    # Install dependencies
npm start      # Start development server
```

```
npm run build      # Create production build
npm test           # Run tests
```

Useful utilities:

```
ctrl+c             # Stop running command
ctrl+l             # Clear terminal
up arrow           # Previous command
tab                # Autocomplete
```

Final Checklist

Before considering a project “done”:

Functionality - ☐ All intended features work - ☐ Edge cases handled gracefully - ☐ Error messages are helpful - ☐ Loading states exist for slow operations

Quality - ☐ Tests exist for critical functionality - ☐ Tests pass consistently - ☐ Code is organized and commented - ☐ No obvious security issues

User Experience - ☐ Works on mobile devices - ☐ Accessible to users with disabilities - ☐ Feedback exists for user actions - ☐ Recovery possible from errors

Operations - ☐ Deployed and accessible - ☐ Environment variables properly set - ☐ Database backups configured - ☐ Error logging active - ☐ Domain configured (if applicable)

Documentation - ☐ README explains what the project does - ☐ Setup instructions for anyone continuing work - ☐ User guide if needed

This book contains approximately 40,000 words.

Version 2.0 - November 2025

Available formats: PDF, EPUB, Web, Audiobook

*For questions, corrections, or to share what you've built:
brian.mabry.edwards@gmail.com*

License

This work is licensed under Creative Commons Attribution 4.0 International (CC BY 4.0). You are free to share and adapt this material for any purpose, including commercially, as long as you give appropriate credit.

About the Author

Brian Edwards is a software engineer and entrepreneur who spent fifteen years building software the traditional way before discovering that AI had changed everything. His career includes engineering leadership at Atlassian (Bitbucket) and enterprise software development at Zenoss. He now writes and speaks about the democratization of software development.

Connect: - Email: brian.mabry.edwards@gmail.com - Twitter: @brianmedwards

- LinkedIn: [brianmabryedwards](#)

Care Commons

The healthcare software mentioned throughout this book is open source and available at: <https://github.com/neighborhood-lab/care-commons>

It serves as a real-world example of what's possible with AI-assisted development, and as practical software for home healthcare organizations.

Thank you for reading. Now go build something.

Extended Case Studies

Extended Case Study 1: From Spreadsheet to Software

Let me walk you through a complete transformation—taking a real business process running on spreadsheets and turning it into proper software.

The Scenario

A property management company manages 45 rental units across 12 buildings. Currently they track everything in a complex spreadsheet system: - One sheet listing all units with tenant info - One sheet for rent payments, updated monthly - One sheet for maintenance requests - One sheet for lease dates and renewals - Multiple email threads for tenant communication

The problems are obvious: version control issues (which spreadsheet is current?), no mobile access, manual processes for everything, and constant context-switching between sheets.

Phase 1: Core Data Model

Start with Claude: “I’m building a property management system. I need to track: properties (buildings) with addresses, units within properties (apartment numbers, bedrooms, bathrooms, square footage, current rent), tenants with contact info and lease dates, and which tenant is in which unit. Set up the project and data model.”

Claude creates the foundational structure. Test by entering a few properties, units, and tenants.

Phase 2: Rent Tracking

“Add rent tracking. Each month, I record which tenants paid rent and which didn’t. Show me a dashboard of payment status for the current month—who has paid, who hasn’t, who is late. Calculate late fees after the 5th of each month.”

Claude builds payment tracking. Test with various payment scenarios.

Phase 3: Maintenance Requests

“Tenants should be able to submit maintenance requests describing the problem and its urgency. I see all requests and can assign them to contractors, update status (received, scheduled, in progress, completed), and add notes. Tenants can see the status of their requests.”

Claude adds the maintenance system. Test both tenant and manager perspectives.

Phase 4: Lease Management

“Track lease start and end dates. Alert me 60 days before any lease expires so I can initiate renewal conversations. When I renew a lease, I can update the rent amount. Track the history of rent changes for each unit.”

Claude builds lease tracking with alerts. Test the renewal workflow.

Phase 5: Financial Reporting

“Generate reports: monthly income by property, vacancy rate over time, maintenance costs by property and category, year-to-date profit/loss per property.”

Claude adds reporting. Generate reports and verify accuracy against your spreadsheet data.

Phase 6: Tenant Portal

“Tenants should be able to log in and: see their lease details, make rent payments online (integrate with Stripe), submit maintenance requests, and message me. Make it mobile-friendly since most tenants will use phones.”

Claude creates the tenant-facing portal. Test as a tenant on your phone.

Phase 7: Migration

“I have existing data in spreadsheets. Help me import it. Here’s the format: [describe your spreadsheet structure].”

Claude creates import scripts. Run a test import, verify data integrity, then do the full migration.

Phase 8: Deployment and Transition

Deploy to Vercel. Migrate data. Train any staff who will use the system. Transition gradually—maybe run parallel with spreadsheets for a month before fully switching.

What Changed

Before: Multiple spreadsheets, manual updates, no mobile access, easy to make errors, hard to find information.

After: Single system of record, automated calculations, mobile access for you and tenants, payment processing, clear maintenance tracking.

The spreadsheet took years to evolve into a complex mess. The software took a few weeks of part-time effort.

Extended Case Study 2: Community Organization Goes Digital

A neighborhood association needed to modernize from paper-based processes.

The Challenge

This 300-home neighborhood association had: - Paper dues invoices mailed annually - A shared email account for resident inquiries - Meeting minutes in Word documents - An outdated static website - A phone tree for emergency communication - Maintenance requests submitted via paper forms in a dropbox

The volunteer board was overwhelmed, and new residents had no idea how to participate.

What They Built

Working with a board member who learned these techniques, they built a neighborhood management system over three months.

Resident Directory

Residents can create accounts linked to their addresses. Contact information is visible to other residents (with privacy controls). The board can message all residents or specific streets.

Dues and Finance

Online dues payment. Tracking of who has paid and who hasn't. Financial reporting for the annual meeting. Automatic reminders for unpaid dues.

Request System

Residents submit maintenance requests (common area issues, rule violations, etc.) through forms. Requests are automatically routed to appropriate committee. Status is visible to the submitter.

Community Calendar

Events, meetings, and deadlines in a shared calendar. Residents can RSVP to events. Automatic reminders for important dates.

Document Library

Governing documents, meeting minutes, financial reports, and newsletters—all searchable and downloadable.

Communication System

Announcements to all residents. Email and optional text messages for urgent items. No more phone trees.

Voting

Board elections and bylaw amendments with secure online voting. Anonymous voting where appropriate. Results automatically tallied.

Impact

Dues collection improved from 85% to 97% (online payment is easier than mailing checks). Board meeting attendance increased (remote participation option). Maintenance request response time dropped from weeks to days. New residents feel more connected to the community.

Total cost: a few hundred dollars in development time and ongoing hosting.

Extended Case Study 3: Automating a Consulting Practice

A marketing consultant built tools to scale her practice.

The Challenge

A one-person marketing consultancy had hit a ceiling. The consultant spent so much time on administrative tasks that there was no capacity to take on more clients. The admin overhead included: - Manual proposal creation (2-3 hours each) - Project tracking in spreadsheets - Time tracking for billing in another spreadsheet - Invoice creation from time entries - Report generation for clients - Onboarding materials sent manually

Proposal Generator

Instead of writing proposals from scratch, the consultant built a proposal system: - Select client and project type - Answer questions about scope - System generates proposal from templates - Customize as needed - Send with e-signature request - Track status (sent, viewed, signed)

Proposal time dropped from 3 hours to 30 minutes.

Project Dashboard

Each client project has a dashboard: - Milestones and timeline - Deliverables with status - Time logged against budget - Files shared - Meeting notes

The consultant always knows project status without digging through files.

Time Tracking

Simple time entry linked to projects: - Start/stop timer or manual entry - Categorization by task type - Notes for each entry - Running total against project budget

Time tracking actually happens because it's frictionless.

Automated Invoicing

At billing time: - Pull time entries for period - Generate invoice from template - Include work summary from time notes - Send invoice with

payment link - Track payment status - Send reminders for overdue invoices

Invoicing that took half a day now takes 15 minutes.

Client Reporting

Monthly reports for clients generated automatically: - Work completed this period - Results achieved - Recommendations for next month - Metrics visualization

Reports that weren't worth the time to create manually now happen automatically.

Client Portal

Clients can log in to see: - Project status and deliverables - Invoices and payment history - Shared files - Meeting schedule

Fewer “where are we?” emails. Clients feel more informed.

Result

The consultant took on 40% more clients without increasing work hours. The administrative overhead dropped from 20+ hours per week to about 5. Quality actually improved because reports and tracking that weren't feasible manually now happen automatically.

Extended Case Study 4: From Hobby to Product

A woodworker turned a personal inventory system into a product.

The Beginning

A hobbyist woodworker built a simple system to track his lumber inventory—species, dimensions, quantity, and location in his shop. Basic functionality that took a weekend to build.

Sharing with Friends

Other woodworkers in his maker space saw it and wanted it. He made it multi-user so friends could have their own accounts.

Feature Requests

Users suggested features: - Track where lumber came from (supplier)
- Calculate board feet automatically - Track project usage (what lumber went into which project) - Estimate remaining inventory value
- Reorder alerts for commonly used species

He implemented these over several months.

Considering Commercialization

With 20+ people using the free version, he wondered if it could be a product. He added: - Project costing (materials in a project) - Cut list optimization - Integration with lumber supplier catalogs - Export for tax/inventory purposes

Launch

He launched on woodworking forums as a \$5/month subscription (with a free tier for basic use). Within six months, 400 paying subscribers generated \$2,000/month in recurring revenue.

Key Lessons

Start with your own need. The tool existed because he needed it.

Real users provide real feedback. Friends and community members shaped the product.

Simple pricing works. One tier, clear value, low enough to be an easy decision.

Niche markets are underserved. Generic inventory tools don't serve woodworkers well.

Scaling

At some point, the woodworker faced a choice: keep the product as side income (a few thousand per month with part-time maintenance) or grow it into a real business (hiring help, marketing investment, full-time focus).

He chose side income—woodworking is his passion, not software. The product runs profitably with modest ongoing effort.

Additional Resources and Tools

Resource: Evaluating AI Tools

Claude isn't the only AI that can help with coding. Here's how to evaluate options:

Claude (Anthropic) Strengths: Strong reasoning, long context window, good at complex tasks, thoughtful responses. Best for: Substantial software projects, code that needs explanation, learning.

GPT-4 (OpenAI) Strengths: Wide knowledge, good at many tasks, extensive tool ecosystem. Best for: General assistance, integration with other OpenAI tools.

GitHub Copilot Strengths: Integrated in editors, fast code completion, learns from your codebase. Best for: Professional developers wanting acceleration.

Cursor Strengths: Purpose-built editor with AI integration, can see entire codebase. Best for: Developers wanting tight editor integration.

Choosing

For complete beginners, Claude through `opencode` provides the best experience for learning and building from scratch. The interface supports extended conversation and complex reasoning.

For those with some development experience, Cursor or VS Code with Copilot might be more efficient—they're integrated into the development workflow.

Most serious builders use multiple tools—Claude for complex reasoning and architecture, Copilot for fast completion, specialized tools for specific tasks.

Resource: Development Environment Alternatives

I recommended Mac or Linux, but alternatives exist:

Windows with WSL

Windows Subsystem for Linux runs a real Linux environment inside Windows. Install WSL from the Microsoft Store, then Ubuntu as your distribution. Your development happens in the Linux environment, which works like native Linux.

Pros: Works on existing Windows hardware. Cons: Some friction switching between Windows and Linux environments.

Cloud Development Environments

GitHub Codespaces, Gitpod, and similar services provide development environments in the cloud. You access them through your browser—no local installation required.

Pros: Works from any computer, consistent environment. Cons: Requires internet connection, less control, potential costs.

Virtual Machines

Run Linux in a virtual machine (VirtualBox is free) on any operating system.

Pros: Full Linux experience. Cons: Resource intensive, some complexity to set up.

Resource: Learning More About Specific Technologies

JavaScript/TypeScript - JavaScript.info (free, comprehensive tutorial) - TypeScript Handbook (official documentation) - Eloquent JavaScript (free online book)

React - Official React documentation (excellent) - React Beta Docs (new, recommended) - Scrimba's React courses (interactive)

Node.js - Node.js documentation - The Node Beginner Book - NodeSchool workshops (hands-on)

Databases - PostgreSQL documentation - SQL tutorial at Mode Analytics - The Art of PostgreSQL (book)

Web Development Generally - MDN Web Docs (comprehensive reference) - freeCodeCamp (free comprehensive curriculum) - The Odin Project (free curriculum)

Resource: When to Seek Professional Help

You can build a lot yourself, but some situations benefit from professional involvement:

Security Audits

Before launching anything that handles sensitive data (financial, health, personal), consider a security review. Many consultants offer this service for reasonable fees.

Complex Integrations

Some third-party integrations are genuinely complex—legacy systems, unusual APIs, enterprise security requirements. Professional help might be faster than struggling yourself.

Scaling Beyond Personal Projects

If your project grows beyond a few hundred users, performance and reliability requirements increase. Professional guidance on architecture helps avoid problems.

Legal/Compliance Requirements

Industries with specific compliance requirements (healthcare, finance) benefit from consultants who specialize in those areas.

Design and UX

If visual design and user experience are critical to success, professional designers improve outcomes. Good design is a skill separate from building functionality.

How to Find Help

For specific technical questions: consultants on platforms like Code-mentor, Toptal, or Upwork.

For longer engagements: ask in developer communities for recommendations.

For specialized needs: look for consultants who specialize (healthcare tech consultants, e-commerce specialists, etc.).

Resource: Keeping Up with Changes

AI and development tools change rapidly. How to stay current:

Follow Key Sources

- Anthropic blog for Claude updates
- OpenAI blog for industry trends
- Hacker News for technology discussion
- Dev.to for practical developer content

Experiment Regularly

Try new tools when they appear. Many won't be useful; some will be transformative. Experimentation keeps your skills current.

Community Involvement

Join Discord servers, follow relevant Twitter accounts, participate in forums. Communities surface new tools and techniques quickly.

Periodic Review

Every few months, review your toolchain. Is there something better now? Have capabilities changed that enable new approaches?

Complete Glossary

API (Application Programming Interface): A defined way for software components to communicate. When your app sends data to a server or receives data back, it's using an API.

Authentication: Verifying identity—confirming that a user is who they claim to be, typically through passwords or other credentials.

Authorization: Determining what an authenticated user is allowed to do. Authentication proves who you are; authorization determines what you can do.

Backend: The server-side part of an application—logic that runs on a server rather than in the user's browser. Handles data processing, storage, and business logic.

Branch (Git): A parallel version of code. Allows working on features without affecting the main code until the work is ready.

Build: The process of converting source code into a form that can be deployed and run. Often involves compilation, bundling, and optimization.

CI/CD (Continuous Integration/Continuous Deployment): Automated systems that test code when it's committed (CI) and deploy it when tests pass (CD).

CLI (Command Line Interface): A text-based interface for interacting with software. Commands are typed rather than clicked.

Client: In web applications, the part that runs in the user's browser. Also refers to any software that makes requests to a server.

Commit (Git): A saved snapshot of code changes. Each commit

represents a point in history that can be returned to.

Component: A reusable piece of user interface. In React, components are building blocks that can be combined to create pages.

CSS (Cascading Style Sheets): The language used to describe how HTML elements should appear—colors, fonts, spacing, layout.

Database: Software that stores and retrieves data efficiently. Manages persistent information that survives when the application restarts.

Deploy: Making an application available to users by putting it on a server accessible via the internet.

Dependency: External code that your application uses. Libraries and frameworks are dependencies.

DNS (Domain Name System): The system that translates domain names (like `example.com`) into IP addresses that computers use.

DOM (Document Object Model): The browser's representation of a web page as a tree of objects that can be manipulated with code.

Endpoint: A specific URL that an API responds to. Different endpoints handle different types of requests.

Environment Variable: Configuration values set outside the code itself. Used for secrets, settings that differ between environments, etc.

Framework: A pre-built structure for building applications. Provides patterns and components so you don't start from scratch.

Frontend: The part of an application that runs in the user's browser. What users see and interact with.

Function: A reusable block of code that performs a specific task. Can accept inputs and return outputs.

Git: A version control system that tracks changes to code over time, enabling history, collaboration, and recovery.

GitHub: A platform that hosts Git repositories and provides collaboration features like pull requests, issues, and project management.

Hash: A one-way mathematical transformation. Used for password storage—hashing a password produces a value that can be compared

but not reversed.

HTML (Hypertext Markup Language): The language that defines the structure and content of web pages.

HTTP/HTTPS: The protocol used for web communication. HTTPS is the secure version with encryption.

IDE (Integrated Development Environment): A software application for writing code, combining an editor with tools for running, testing, and debugging.

JavaScript: The programming language of the web. Runs in browsers and (with Node.js) on servers.

JSON (JavaScript Object Notation): A text format for structured data, widely used for API communication.

Library: A collection of pre-written code that you can use in your applications. Smaller and more focused than a framework.

Localhost: The address (127.0.0.1) that refers to your own computer. Used during development to test before deploying.

Migration (Database): A controlled change to database structure. Allows evolving the database schema over time.

Node.js: A runtime that lets you run JavaScript outside the browser, typically on servers.

npm (Node Package Manager): A tool for installing and managing JavaScript packages. Also refers to the package registry itself.

Payload: The data sent with a request or response. The main content as opposed to headers and metadata.

Port: A number that identifies a specific service on a computer. Web servers typically use port 80 (HTTP) or 443 (HTTPS).

Query: A request for data from a database. SQL is the most common query language.

React: A JavaScript framework for building user interfaces, developed by Facebook. Widely used for web applications.

Repository (Repo): A collection of code and its history, managed by Git. The complete project in version control.

REST (Representational State Transfer): A common style for designing web APIs, using HTTP methods and resource-based URLs.

Runtime: The environment in which code executes. Node.js is a JavaScript runtime.

Schema: The structure of a database—what tables exist, what columns they have, what types of data they hold.

Server: A computer that provides services to other computers (clients). In web applications, handles requests and sends responses.

Session: A period of interaction between a user and an application. Sessions track logged-in users across multiple requests.

SQL (Structured Query Language): A language for interacting with relational databases. Used to create, read, update, and delete data.

SSL/TLS: Encryption protocols that secure web communication. HTTPS uses TLS.

Stack: The combination of technologies used in an application. “The stack” refers to all layers together.

Terminal: A text-based interface for interacting with the operating system. Where you type commands.

Test: Code that verifies other code works correctly. Automated tests catch bugs and prevent regressions.

Token: A string of characters used for authentication or identification. Often used in APIs and sessions.

TypeScript: A version of JavaScript with added type checking. Helps catch errors before code runs.

URL (Uniform Resource Locator): A web address that identifies a specific resource—like a page or API endpoint.

Variable: A named storage location for data in code. Holds values that can be used and changed.

Version Control: Systems that track changes to files over time. Git is the most common.

Webhook: A way for one application to notify another when something happens. The receiving application provides a URL that gets called when events occur.

End of Book

Total word count: approximately 40,000 words

Comprehensive Tutorial Section

Complete Build: Task Management Application

Let me walk you through building a complete application from start to finish, showing exactly what you would say to Claude at each step.

Project Scope

We're building a task management application that's better than a to-do list—something like a simplified Asana or Monday.com for personal or small team use.

Features: - Projects containing multiple tasks - Tasks with titles, descriptions, due dates, and priorities - Task assignment (in multi-user mode) - Status progression (To Do → In Progress → Done) - Filtering and sorting - Project overview dashboards

Step 1: Project Initialization

Your message to Claude: “I want to build a task management application like a simplified version of Asana. Let me create projects, and within each project create tasks. Tasks have a title, description, due date, and priority (low, medium, high). Start by setting up the project structure and creating the basic data model.”

Claude will create: - Project folder structure - Package.json with dependencies - Database schema for projects and tasks - Basic backend API structure

Step 2: Project Management

“Create a page that shows all my projects. I can add new projects with a name and description. Clicking a project shows its details. I can edit project names and descriptions, and delete projects (with confirmation).”

Claude builds CRUD operations for projects. Test by creating, viewing, editing, and deleting projects.

Step 3: Task Management

“Within a project, show all tasks. I can add tasks with a title, optional description, due date, and priority. Tasks display in a list showing title, due date, and priority with color coding. I can edit and delete tasks.”

Claude adds task functionality. Test the complete task workflow.

Step 4: Task Status

“Tasks have a status: To Do, In Progress, or Done. When viewing a project, show tasks grouped by status in three columns (like a kanban board). I can drag tasks between columns to change status, or click a button to advance status.”

This is more complex UI work. Claude will implement drag-and-drop or button-based status changes.

Step 5: Filtering and Sorting

“Add filtering: show only tasks that are overdue, show only high priority tasks, show tasks due this week. Add sorting: by due date, by priority, by status. Let me combine filters and sorts.”

Claude adds filter and sort controls.

Step 6: Project Dashboard

“On the project page, add a summary section showing: total tasks, tasks by status (counts), overdue task count, completion percentage. Show this before the task list.”

Claude creates the dashboard summary.

Step 7: Due Date Features

“Highlight overdue tasks in red. For tasks due today, highlight in yellow. Add a reminder that shows in a banner when I have tasks

due today.”

Claude adds date-based styling and reminders.

Step 8: Polish

“Make this look professional and clean. The kanban columns should be visually distinct. Priority badges should be easily scannable. The overall design should feel modern but not distracting from the content.”

Claude improves the visual design.

Step 9: Multi-User (Optional)

“Add user accounts. Users can only see their own projects. Let me invite other users to a project as collaborators. Collaborators can see and modify tasks in that project.”

Claude adds authentication and collaboration features.

Step 10: Deployment

“Set up a database and help me deploy this to Vercel so I can access it from anywhere.”

Claude configures production infrastructure and guides deployment.

What You’ve Built

A complete project and task management application, built through conversation. The codebase will have: - Several thousand lines of TypeScript - React frontend with responsive design - Express backend with REST API - PostgreSQL database - Authentication system (if you did step 9) - Production deployment on Vercel

Complete Build: Blog Platform

Another complete walkthrough for a different type of application.

Project Scope

A personal blog platform where you can write and publish posts, and readers can comment. Like a simplified Medium or WordPress.

Step 1: Setup

“I want to build a personal blog platform. I’ll write posts with titles and content, and readers can view them. Start with the project setup and basic data model for posts.”

Step 2: Post Creation

“Create an admin area where I can write posts. Posts have a title, content (with formatting like headers, bold, italics, and links), and optional featured image. I can save drafts and publish when ready.”

Step 3: Rich Text Editing

“The content editor should be a rich text editor—not markdown, but a visual editor where I can format text by selecting and clicking buttons. Include basic formatting: headers, bold, italic, links, lists.”

Claude implements a rich text editor like TipTap or Slate.

Step 4: Public Display

“Create a public-facing blog page. Show all published posts in reverse chronological order with title, date, and excerpt. Clicking a post shows the full content. Make it clean and readable.”

Step 5: Comments

“Readers can leave comments on posts. They enter their name and comment text. Comments appear below the post. I can moderate comments—approve, hide, or delete them from the admin area.”

Step 6: Categories and Tags

“Add categories to organize posts (like ‘Technology’, ‘Personal’, ‘Travel’). Also add tags for more specific topics. Show filtering by category and tag. Display popular tags in a sidebar.”

Step 7: Search

“Add search functionality. Readers can search post titles and content. Show results with relevant excerpts highlighting the search terms.”

Step 8: RSS Feed

“Generate an RSS feed so readers can subscribe. Include title, content excerpt, and link for each post.”

Step 9: SEO

“Optimize for search engines. Proper page titles and meta descriptions. Semantic HTML structure. Clean URLs (like /blog/my-post-title instead of /blog/123). Generate a sitemap.”

Step 10: Deploy with Custom Domain

“Deploy to Vercel and help me configure my custom domain (example.com) to point to it.”

What You’ve Built

A complete blog platform with: - Admin interface for writing - Rich text editing - Public blog display - Comment system with moderation - Categories, tags, and search - RSS feed - SEO optimization - Custom domain

Complete Build: Fitness Tracker

One more comprehensive example, this one focused on personal data tracking.

Project Scope

Track workouts, see progress over time, set goals, and stay motivated.

Step 1: Workout Logging

“I want to build a fitness tracker. I’ll log workouts including the date, type (running, weightlifting, cycling, yoga, etc.), duration, and notes. Start with basic workout logging.”

Step 2: Exercise Details

“For weightlifting workouts, I want to log specific exercises with sets, reps, and weight. For cardio, I want distance and average pace. Different workout types have different detail fields.”

Step 3: Workout Display

“Show my workout history with calendar view and list view. On the calendar, workout days are highlighted. The list shows recent workouts with summary info. Clicking any workout shows full details.”

Step 4: Statistics

“Calculate and display stats: workouts per week, total time this month, running distance this month. Show trends—am I working out more or less than last month?”

Step 5: Progress Tracking

“For weightlifting, show progress on specific exercises. If I bench press, show my weight over time as a graph. Show personal bests for each exercise.”

Step 6: Goals

“Let me set goals: ‘Work out 4 times per week’ or ‘Run 20 miles this month.’ Track progress toward goals and show completion percentage. Show a streak counter for consecutive days/weeks of meeting goals.”

Step 7: Body Measurements

“Track body measurements: weight, body fat percentage (if I have it), and notes. Show weight trend over time as a graph. Don’t show weight by default—make it optional to view.”

Step 8: Motivation Features

“Add features to stay motivated: workout streaks, achievement badges (first workout, 10 workouts, first 5K, etc.), motivational quote of the day. Celebrate milestones with visual feedback.”

Step 9: Mobile Experience

“This needs to work well on my phone since I’ll log workouts from the gym. Make sure the interface is touch-friendly and quick to log a workout.”

Step 10: Data Export

“Let me export all my data to CSV for backup or analysis in spreadsheets.”

What You’ve Built

A comprehensive fitness tracker with: - Workout logging with type-specific details - Calendar and list views - Statistics and trends - Progress graphs for specific exercises - Goal setting and tracking - Body measurement tracking - Motivational features - Mobile-optimized interface - Data export

The Pattern

Notice the pattern across these builds:

1. Start with core data and basic CRUD
2. Add primary workflows
3. Build supporting features
4. Add analytics and visualization
5. Polish user experience
6. Deploy

This pattern works for almost any application. The specific steps differ, but the progression from foundation to polish is consistent.

Conversations with Claude: Real Examples

Here are real interactions showing how to communicate effectively with Claude.

Example: Starting a New Feature

Your message: “I want to add a feature where users can share a task list with someone by sending them a special link. When they open the link, they can view the tasks (read-only) without needing an account. The link should expire after 7 days. How should we approach this?”

Why this works: - Describes the user-facing behavior clearly - Mentions constraints (read-only, 7 days) - Asks for approach before diving into implementation

Example: Reporting a Bug

Your message: “There’s a bug with the date picker. When I select a date, it shows as one day earlier than what I picked. For example, I pick November 20th and it shows November 19th. I think this might be a timezone issue. Here’s what I see in the browser console: [paste any error messages]. Can you investigate and fix this?”

Why this works: - Specific about the problem (one day off) - Includes an example - Hypothesizes a cause (timezone) - Provides technical details

Example: Requesting Design Changes

Your message: “The task cards are too cluttered. There’s too much information visible at once and it’s hard to scan quickly. Can you redesign them to show just the title and due date by default, with

other details revealed when you hover or click? Keep the priority indicator visible—that’s important for quick scanning.”

Why this works: - Explains what’s wrong (cluttered, hard to scan) - Proposes a solution (progressive disclosure) - Specifies what to keep (priority indicator)

Example: Asking for Explanation

Your message: “Can you explain how the authentication is working? Walk me through what happens from when a user enters their password to when they’re logged in. I want to understand this for my own knowledge.”

Why this works: - Specific about what to explain - Clear that this is for learning, not to solve a problem

Example: Requesting Refactoring

Your message: “The code for handling tasks has gotten messy—there’s similar logic in three different places for updating task status. Can you refactor this so the status update logic is in one place that all three can use?”

Why this works: - Identifies the problem (duplication) - Describes the desired outcome (consolidation) - Trusts Claude to determine how

Example: Unclear Requirement (What Not to Do)

Poor message: “Add sharing functionality.”

Why this doesn’t work: - What kind of sharing? With whom? How? - Claude will have to make many assumptions

Better message: “I want users to be able to share individual tasks with other users of the application. The recipient should be able to see the task and mark it complete. The sharer should see when the task was completed. Implement this feature.”

Example: Scope Management

Your message: “I know I asked for user notifications, but I’m realizing that might be complicated. Let’s simplify: instead of a full notification system, just show a simple banner at the top of the dashboard when there are tasks due today. We can add more sophisticated notifications later.”

Why this works: - Acknowledges scope change - Provides simpler alternative - Notes this is stepping stone, not final solution

The 30-Day Learning Path

A structured approach to learning these skills over one month, assuming about an hour per day.

Week 1: Foundation

Day 1: Set up your development environment following Appendix A or B.

Day 2: Complete first conversation with Claude—build a simple “Hello World” web page.

Day 3: Build the task list application from Chapter 9.

Day 4: Extend the task list—add the ability to edit and delete tasks.

Day 5: Add categories to tasks. Learn about filtering.

Day 6: Make the task list persist in browser storage.

Day 7: Review what you’ve built. Try breaking it and fixing it.

Week 2: Core Skills

Day 8: Set up Git and GitHub. Learn basic commands.

Day 9: Deploy your task list to Vercel.

Day 10: Start a new project—choose from the examples in Part Three.

Day 11-13: Build your chosen project through Claude conversation.

Day 14: Deploy your second project.

Week 3: Going Deeper

Day 15: Add a database to your project.

Day 16: Add user accounts (authentication).

Day 17: Learn about tests—have Claude write tests for your code.

Day 18: Refactor something in your project based on what you've learned.

Day 19: Add a feature you wish your project had.

Day 20: Polish the design and user experience.

Day 21: Review and document your project.

Week 4: Independence

Day 22: Start a third project—something you actually want for yourself.

Day 23-26: Build your personal project.

Day 27: Deploy and test thoroughly.

Day 28: Share what you've built with someone.

Day 29: Write down what you've learned and what you want to explore next.

Day 30: Celebrate. You're a builder now.

After 30 Days

You've built three projects and deployed them. You understand the workflow, can communicate with Claude effectively, and have working software in the world.

From here, continue building. Each project teaches more. The third project goes faster than the first. By the tenth project, this feels natural.

Final Words

You've reached the end of this book—but you're at the beginning of something bigger.

A year ago, if you wanted to build software, you had two choices: spend years learning to code, or spend thousands of dollars hiring someone who already knew how. Neither was practical for most people with software ideas.

Now there's a third choice: describe what you want, work with AI to build it, and verify the results. This choice is available to anyone willing to learn a new way of working.

I've given you the foundation: what software is, how to set up your environment, how to communicate with AI, how to build and deploy applications. I've walked through examples spanning personal tools, small business applications, and community software. I've covered the technical concepts you need to understand without requiring you to master them.

The rest is up to you.

Start small. Build something simple. Make mistakes—they're the best teachers. When you get stuck, work through it. Ask Claude for help. Search online. Take breaks and return with fresh perspective.

Build something for yourself. Build something for someone you know. Build something for your business, your community, your hobby. Each thing you build strengthens your capability and confidence.

Share what you create. The world has too many problems and not enough people building solutions. You're now among those who can build. That's a responsibility as much as a capability.

The tools will improve. What's possible in a year will exceed what's possible today. Stay curious, keep learning, and adapt as capabilities expand. The meta-skill—working effectively with AI to create things—will remain valuable even as specific tools change.

I don't know exactly where this all leads. The democratization of software creation is one of the largest shifts in technology in decades. More people building software means more problems solved, more needs met, more innovation from unexpected places.

You're part of that now.

So close this book. Open your terminal. Start a project.

Build something that matters to you.

Then build something else.

The power is yours.

Brian Edwards November 2025

brian.mabry.edwards@gmail.com

This book contains approximately 42,000 words.

Thank you for reading.

Supplementary Material

The Economic Case for Vibe Coding

Let's talk money. This isn't just about capability—it's about economics that make software accessible.

Traditional Software Development Costs

A custom business application traditionally costs:

- Simple application (basic CRUD, single user): \$10,000-\$30,000
- Small business application (multi-user, moderate complexity): \$30,000-\$100,000
- Medium business application (integrations, complex workflows): \$100,000-\$500,000
- Large business application (enterprise features, scale): \$500,000+

These numbers come from either hiring developers (\$100-200/hour for experienced talent) or contracting agencies (\$150-300/hour).

Timeline: 3-12 months for most business applications.

Vibe Coding Economics

What does the same work cost with AI assistance?

Direct costs:

- AI API usage: \$5-20 for a simple app, \$50-200 for complex apps
- Hosting: Free to \$20/month for most applications
- Domain name: \$12/year (optional)

Time investment:

- Simple application: A few hours to a couple days
- Small business application: 1-3 weeks of part-time effort
- Medium business application: 1-3 months of part-time effort

Your time has value, of course. But the economics are still transformative: - \$200 in API costs vs. \$30,000 in development fees = 99%

cost reduction - 2 weeks of part-time work vs. 3 months of full-time work = massive time compression

Break-Even Analysis

Say you pay \$50/month for SaaS software that almost-but-not-quite fits your needs. Building a custom replacement might cost: - \$50 in AI usage - 20 hours of your time - \$0-20/month in hosting

If your custom solution replaces that SaaS subscription, you break even in 1-2 months.

If your time is worth \$50/hour, those 20 hours “cost” \$1,000. Even so, you break even against the SaaS in under 2 years—and you own the result forever, can customize it freely, and aren’t dependent on a vendor.

Value Creation

Beyond cost savings, custom software creates value: - Efficiency gains from tools that match your workflow exactly - Competitive advantage from capabilities competitors don’t have - Data ownership rather than vendor lock-in - Flexibility to adapt as needs change

These benefits are hard to quantify but often exceed the direct cost savings.

When Economics Favor Traditional Development

Some situations still favor hiring professionals: - Mission-critical systems where reliability is paramount - Complex integrations requiring deep expertise - Situations where your time is more valuable elsewhere - Scale that requires performance optimization expertise - Security-sensitive applications in regulated industries

For most applications most people need, though, Vibe Coding economics are compelling.

The Social Impact of Democratized Software

Beyond individual economics, democratized software creation has broader implications.

Solving Niche Problems

Commercial software companies build for large markets—problems that enough people share to justify development costs. This leaves countless niche problems unsolved:

- The specific workflow of a particular industry
- The unique needs of a small organization
- Local problems specific to a community or region
- Personal problems that matter deeply to individuals

When anyone can build, these niche problems become solvable. The woodworker builds lumber inventory software. The church volunteer builds event coordination. The teacher builds curriculum-specific tools.

The aggregate effect: more problems solved, more human needs met, more potential unlocked.

Enabling Non-Traditional Builders

Traditional software development has demographic skews—dominated by certain educational backgrounds, geographic locations, and demographic groups. This isn't because other groups lack capability; it's because access to training and opportunity has been uneven.

AI-assisted development requires different skills than traditional programming. Communication, problem decomposition, and domain expertise matter more than syntax memorization. This opens the door to builders from non-traditional backgrounds.

Community Resilience

Communities with local software capability are more resilient than those dependent entirely on external vendors: - Local problems get local solutions - Tools can be maintained and adapted locally - Less vendor lock-in and dependency - Skills stay in the community

This is similar to how communities benefit from local food production, local manufacturing, or local renewable energy—self-reliance that complements rather than replaces broader connections.

Potential Concerns

Democratization brings challenges too: - More software means more potential security vulnerabilities - Quality varies when anyone can

- build - AI-generated code may have biases embedded in training data
- Rapid change displaces workers in traditional software development

These concerns deserve attention. But they're arguments for thoughtful democratization, not against democratization altogether.

The Future We're Building Toward

Where does this go?

Near Term (1-2 Years)

AI capabilities continue improving rapidly. What requires careful prompting today becomes effortless tomorrow. Complex applications that take weeks now will take days.

More people discover they can build. The community of non-traditional builders grows. Resources, communities, and best practices emerge to support them.

Traditional software development adapts. Professional developers incorporate AI deeply into their workflows. The nature of the work changes—more architecture, less typing.

Medium Term (3-5 Years)

AI systems become more autonomous. Instead of conversation, you might describe a goal and let AI work toward it with minimal supervision.

Voice interfaces make building even more accessible. Describe what you want verbally, have it built.

Domain-specific AI tools emerge—AI that deeply understands health-care, retail, education, or specific industries. More sophisticated applications become buildable by non-specialists.

Longer Term (5+ Years)

The boundary between “user” and “builder” blurs. Software becomes something you have conversations with, that adapts to your needs, that builds features on request.

Custom software becomes the norm rather than the exception. Generic one-size-fits-all applications seem as outdated as

one-size-fits-all clothing.

The entire concept of “software development as a specialized profession” may transform—not disappear, but become something quite different from today.

The Only Certainty: Change

Predictions are unreliable. But one thing is certain: this landscape will change dramatically. The skills you develop now—working effectively with AI, clear communication, verification and testing—position you to adapt as tools evolve.

Mindset for Continuous Learning

The tools and techniques in this book are correct as of November 2025. They will evolve. How do you stay current?

Embrace Experimentation

When new tools appear, try them. Not every new thing is better, but you can't judge without experience. Allocate some time for experimentation separate from production work.

Follow, Don't Chase

Stay aware of developments without chasing every new thing. Follow key sources, but don't feel obligated to adopt every trend. Wait for tools to prove their value before committing.

Learn Principles, Not Just Procedures

Specific commands and interfaces change. Underlying principles persist longer: - Clear communication improves AI output - Testing enables confidence and iteration - Version control protects against mistakes - Security requires ongoing attention

Understand why things work, not just how.

Build Relationships

Connect with others on similar journeys. They'll alert you to useful developments, help you through challenges, and provide perspective when you're uncertain.

Practice Regularly

Skills atrophy without use. Even when you don't have a specific project, practice. Experiment with new features. Rebuild old projects with new approaches. Keep the skills sharp.

Be Patient With Yourself

Learning takes time. You won't master everything immediately. Progress isn't always linear—sometimes you'll feel stuck before a breakthrough. Trust the process.

Parting Advice

A few final thoughts as you begin your journey:

Start Today

Not next week, not when you have more time, not after you finish some other thing. The best time to start is now. Begin with something small.

Embrace Imperfection

Your first projects will be rough. That's fine—they're supposed to be. Done is better than perfect. Ship something, learn from it, improve the next one.

Solve Real Problems

Build things that matter, even if they're small. Real problems provide motivation that hypothetical exercises don't.

Share Generously

When you build something useful, share it. When you learn something valuable, share it. The community grows when people contribute.

Stay Humble

AI tools are powerful but not magic. You'll make mistakes. Projects will fail. Stay humble, learn from failures, and keep going.

Remember Your Power

You can build things now. This capability—creating working software from ideas—was inaccessible to you before. Don't forget what you've gained.

Build Something

Enough reading. Close this book. Open your terminal.

Build something.

End of Book

Total word count: approximately 42,000 words

About This Book

Title: Vibe Coding: The Revolution That Lets Anyone Build Software

Author: Brian Edwards

Version: 2.0

Publication Date: November 2025

License: Creative Commons Attribution 4.0 International (CC BY 4.0)

Contact: brian.mabry.edwards@gmail.com

Website: <https://bedwards.github.io/vibe-coding/>

Thank you for reading.

Now go build something.

Appendix E: Complete Example Conversations

Building a Recipe App: Full Transcript

Here's a lightly edited transcript of building a recipe app, showing exactly what you'd say and what kind of responses to expect.

You: I want to build a recipe collection app. When I open it, I see all my recipes in a list showing just the recipe name and a preview. I can click any recipe to see full details. I can add new recipes with a name, ingredients list, and instructions. Let's start with setting up the project.

Claude: [Creates project files, explains the structure, sets up basic configuration]

I've created a new project with the following structure: - A React frontend for the user interface - An Express backend for the API - A PostgreSQL database schema for recipes

Let me start the development server so you can see it running...

You: Great, the project is set up. Now let's build the recipe list. When I open the app, I should see all my recipes. For now, just show dummy data since we don't have a database connected yet.

Claude: [Implements recipe list component with sample data, explains the code]

I've created a recipe list component that displays recipes in cards. Currently it shows three sample recipes. Each card shows the recipe name and a short preview of the first few ingredients.

You: The list looks good. Now let me click a recipe to see the full details. When I click a recipe card, it should show the full recipe—name, complete ingredient list, and instructions.

Claude: [Adds routing and detail view, implements click handling]

Done. Clicking a recipe now opens a detail view showing all the recipe information. I've added navigation so you can go back to the list.

You: Now let's add the ability to create new recipes. Add a button that opens a form where I can enter: recipe name, ingredients (one per line in a text area), and instructions. When I save, it should add to the list.

Claude: [Creates form component, handles submission, updates list]

I've added an "Add Recipe" button and form. The form has fields for name, ingredients (text area where you enter one ingredient per line), and instructions. When you submit, it parses the ingredients into a list and adds the recipe.

You: The form works but it stays on screen after I submit. It should go back to the recipe list after saving, and the new recipe should appear there.

Claude: [Adds navigation after save, ensures state updates]

Fixed. Now when you save a recipe, you're redirected to the recipe list and your new recipe appears immediately.

You: I want to be able to edit recipes too. On the recipe detail page, add an Edit button that opens the form with the current values filled in. Saving should update the recipe instead of creating a new one.

Claude: [Adds edit functionality, pre-fills form, implements update logic]

Done. The recipe detail page now has an Edit button. Clicking it opens the form with all the current values. When you save, it updates the existing recipe rather than creating a duplicate.

You: Now let's add deletion. On the recipe detail page, add a Delete button. It should ask me to confirm before actually deleting.

Claude: [Adds delete button with confirmation dialog]

Added. There's now a Delete button that shows a confirmation dialog. If you confirm, the recipe is deleted and you're returned to the recipe list.

You: The recipes disappear when I refresh the page because they're not being saved anywhere. Let's add a real database so recipes persist. Use PostgreSQL through Neon.

Claude: [Sets up database connection, creates schema, migrates code to use database]

I've set up the database: 1. Created a recipes table with columns for id, name, ingredients, instructions, and created_at 2. Modified the API to read from and write to the database 3. Recipes now persist between sessions

You'll need to set up a Neon database and add your connection string. Would you like me to walk you through that?

You: Yes, walk me through setting up Neon.

Claude: [Provides step-by-step Neon setup instructions]

You: Done, the database is connected and working. Now let's add search. I want a search box at the top of the recipe list that filters as I type. It should search both recipe names and ingredients.

Claude: [Adds search input and filter logic]

Added a search box at the top of the recipe list. As you type, recipes filter in real-time. The search checks both recipe names and ingredient lists, so searching "chicken" will find recipes named "Chicken Parmesan" and also recipes that just have chicken in the ingredients.

You: Nice! Let's make it look better. Right now it's very plain. Make it look clean and modern—like something from a nice recipe website. Use a warm color scheme appropriate for food/cooking.

Claude: [Applies styling, improves layout, adds visual polish]

I've redesigned the interface with: - A warm color palette with cream backgrounds and terracotta accents - Better typography with a readable font - Recipe cards with subtle shadows and hover effects - A cleaner layout with good spacing - Responsive design that works on mobile

You: This looks great! Now let's deploy it. Help me put this on Vercel so I can access it from anywhere.

Claude: [Guides through Vercel deployment process]

That's the flow. Notice how each request is specific and builds on the previous step. The application grew incrementally from project setup to deployed product through a series of focused conversations.

Building an Invoice System: Key Exchanges

Here are key moments from building an invoicing system:

Starting the Data Model: "I need to build an invoicing system for my freelance business. I'll have clients, and for each client I'll create invoices. Invoices have line items (description and amount), a total, and status (draft, sent, paid). Set up the project with this data model."

Adding PDF Generation: "When I view an invoice, I want a 'Download PDF' button that generates a professional-looking invoice PDF. It should have my business name and address at the top, client info below that, line items in a table, and the total clearly shown. Include my bank details for payment."

Implementing Recurring Invoices: "Some clients get invoiced monthly for the same services. Let me mark an invoice as 'recurring monthly' and set a day of month. On that day each month, the system should automatically create a new invoice copying the line items from the template."

Adding Payment Integration: "Instead of just marking invoices as paid manually, let me send invoice emails with a 'Pay Now' link. When clients click it, they can pay with credit card through Stripe. When payment succeeds, automatically mark the invoice as paid."

Each of these is a substantial feature request. Notice how they describe the user-facing behavior rather than implementation details. Claude determines how to implement; you determine what to implement.

Common Conversation Patterns

The Clarifying Question Pattern:

You: “Add user accounts.”

Claude: “I can add user accounts. A few questions first: 1. Should users sign up themselves, or will you create accounts for them? 2. What authentication method—passwords, magic links, or social login? 3. Should there be different user roles with different permissions? 4. Is this replacing the current single-user setup or adding multi-tenancy?”

You: “Users sign up themselves with passwords. Just one role for now. Add it to the current setup so I’m still the only user but the infrastructure is there for more.”

The Progressive Enhancement Pattern:

You: “Let’s start simple with task status. Just add a checkbox to mark tasks complete.”

[Later]

You: “Actually, I need more than just complete/incomplete. Change it to three statuses: To Do, In Progress, and Done. Show them in different colors.”

[Later]

You: “Let’s make this a kanban board with three columns. I can drag tasks between columns to change status.”

Each step builds on the previous, adding complexity only when needed.

The Debugging Conversation:

You: “Something’s wrong. When I add a task, it appears twice in the list.”

Claude: [Investigates code] “I see the issue. The ‘add task’ function is being called twice because the form submits once on button click and once on Enter key. Let me fix that...”

You: “That fixed it. But now there’s another issue—tasks aren’t saving to the database. They disappear on refresh.”

Claude: [Investigates further] “The API endpoint is returning success but I see the database insert isn’t happening because... [identifies root cause]. Here’s the fix...”

Debugging often requires back-and-forth as you identify symptoms and Claude traces to causes.

Appendix F: Troubleshooting Decision Tree

When something goes wrong, walk through this decision tree:

Problem: Application won't start

1. Did you run `npm install`?
 - No → Run `npm install` and try again
 - Yes → Continue
2. Is Node.js installed?
 - Check with `node --version`
 - If not found → Install Node.js
 - If found → Continue
3. Are there error messages?
 - Copy exact error text
 - Give to Claude: "I get this error when starting the app: [error]"

Problem: Changes don't appear in browser

1. Did you save the file?
 - Check that file is saved
2. Is the development server running?
 - Check terminal for running process
 - If not running → Start it with `npm start` (or appropriate command)
3. Try hard refresh
 - Mac: `Cmd+Shift+R`
 - Windows/Linux: `Ctrl+Shift+R`
4. Clear browser cache if still not working

Problem: Database errors

1. Is database connection string correct?
 - Check environment variables
 - Verify format matches your database provider
2. Is database server running?
 - For local database → Check if service is started
 - For hosted database → Check provider status page
3. Do tables exist?
 - Ask Claude to check and run migrations if needed
4. Is data in expected format?
 - Check database directly if possible

- Ask Claude to add logging to trace the issue

Problem: Deployment fails

1. Does it work locally?
 - Always fix local issues first
2. Are environment variables set in deployment platform?
 - Check Vercel/platform dashboard
3. Read the build log
 - Find first red error message
 - Give to Claude for diagnosis
4. Check for deployment-specific issues
 - Different file paths
 - Missing dependencies
 - Build timeouts

Problem: Feature works wrong

1. Clearly define expected vs. actual behavior
 - “I expected X, but I see Y”
2. Check for error messages in browser console
 - Right-click → Inspect → Console tab
3. Describe the steps to reproduce
 - “Click A, then B, then C, and it shows wrong result”
4. Give all information to Claude
 - Expected behavior
 - Actual behavior
 - Steps to reproduce
 - Error messages

Appendix G: Security Quick Reference

Passwords - Never store plain text passwords - Use bcrypt or argon2 for hashing - Require minimum length (8+ characters) - Don't limit maximum length unnecessarily - Don't expose whether email exists on failed login

Sessions - Use secure, HTTP-only cookies - Regenerate session ID after login - Set reasonable expiration - Invalidate on logout

Data Access - Always verify user has permission - Use parameterized queries (never string concatenation) - Validate all input - Sanitize output to prevent XSS

Communication - Always use HTTPS in production - Set secure headers (HSTS, CSP, etc.) - Validate redirect URLs

Secrets - Never commit secrets to git - Use environment variables - Rotate credentials if exposed - Use different credentials per environment

Dependencies - Keep dependencies updated - Run `npm audit` regularly - Address high/critical vulnerabilities immediately

Appendix H: Performance Checklist

Frontend - Minimize bundle size - Lazy load non-critical code - Optimize images - Use caching headers

Backend - Add database indexes for common queries - Implement pagination for lists - Cache expensive computations - Set up connection pooling

Database - Index columns used in WHERE clauses - Avoid SELECT * (select only needed columns) - Use EXPLAIN to analyze slow queries - Regular maintenance (vacuum, analyze)

Monitoring - Log slow requests - Track error rates - Monitor memory and CPU - Set up alerts for issues

This concludes the supplementary material.

Total book word count: approximately 42,000 words

Appendix I: Industry Insights from Someone Who's Been There

I spent fifteen years in professional software development. Let me share some insider perspectives that might help you understand the landscape you're entering.

What Professional Software Development Is Really Like

The Reality Behind the Glamour

Tech companies project images of creativity, innovation, and meaningful work. And those elements do exist. But day-to-day professional software development involves a lot of:

- Meetings. So many meetings. Planning meetings, standup meetings, retrospective meetings, alignment meetings. Studies suggest developers spend 30-50% of their time in meetings.
- Process overhead. Ticket creation, ticket grooming, estimation, sprint planning, documentation. These activities support team coordination but aren't directly productive.
- Coordination. When multiple people work on the same codebase, substantial effort goes to ensuring they don't break each other's work. Code reviews, merge conflicts, integration testing.
- Maintenance. Most developers spend more time maintaining

existing systems than building new ones. Bug fixes, dependency updates, performance optimization, technical debt reduction.

- Context switching. Interruptions are constant. Slack messages, urgent bugs, support escalations. Deep focus is hard to achieve.

This isn't a criticism—these activities serve purposes in team settings. But they're overhead that solo builders working with AI largely avoid.

Why Software Projects Fail

Most software projects fail—they're late, over budget, or don't deliver expected value. The reasons are usually:

- Unclear requirements. Stakeholders don't know exactly what they want, so developers build the wrong thing.
- Scope creep. Requirements expand during development, but timelines don't adjust.
- Communication failures. Information doesn't flow between people who need it.
- Technical choices mismatched to problems. Over-engineering or under-engineering.
- Inadequate testing. Bugs accumulate until the system is unstable.

AI-assisted development doesn't automatically avoid these failures. But the faster iteration cycle means mistakes are discovered sooner and cost less to fix.

The Ten-X Developer Myth

There's a persistent myth that some developers are ten times more productive than others. The reality is more nuanced:

- Performance varies significantly, but multipliers like "10x" are rarely meaningful
- Much of what looks like productivity is actually better communication and problem selection
- Team dynamics matter more than individual heroics
- Sustainable productivity differs from sprint productivity

What does translate: the principles that make some developers effective—clear thinking, good communication, systematic debugging,

continuous learning—apply to AI-assisted development too.

Lessons from Enterprise Software

I built enterprise monitoring software at Zenoss. Enterprise software has distinct characteristics:

Scale and Reliability

Enterprise systems must handle large data volumes, many concurrent users, and high uptime requirements. The techniques are learnable but require attention:

- Horizontal scaling (adding more servers rather than bigger servers)
- Load balancing (distributing traffic across servers)
- Database optimization (indexing, query tuning, caching)
- Redundancy (multiple instances, failover mechanisms)
- Monitoring (knowing when problems occur)

For most applications you'll build, these concerns are premature. But as you grow, they become relevant.

Integration Complexity

Enterprise software connects to many other systems. Each integration adds complexity:

- Different data formats
- Different authentication mechanisms
- Different reliability characteristics
- Different change schedules

The principle: minimize integrations when possible, and invest heavily in testing integrations you do have.

Security at Scale

Enterprise security goes beyond basic authentication:

- Role-based access control (different users have different permissions)
- Audit logging (recording who did what when)
- Data classification (different protection levels for different data)
- Compliance frameworks (HIPAA, SOC2, GDPR)

If you're building for enterprise customers, security becomes a major investment area.

Lessons from Team Leadership

At Atlassian, I led teams building Bitbucket, one of the world's largest code hosting platforms. Leadership lessons that transfer:

Clarity of Purpose

Teams perform better when they understand why they're building something, not just what. The same applies to AI assistance—Claude produces better results when you explain the purpose, not just the task.

Psychological Safety

Teams where people feel safe to express ideas and admit mistakes outperform teams where people feel judged. Similarly, don't feel embarrassed about asking Claude "dumb" questions or admitting confusion. The goal is results, not appearing smart.

Iteration Over Perfection

Shipping incrementally and learning from feedback beats trying to design the perfect system upfront. This is core to Vibe Coding—build, verify, refine.

Technical Excellence as Enabler

Teams that invested in code quality, testing, and documentation could move faster over time. Teams that accumulated technical debt moved slower. The same applies to your projects: invest in quality from the start.

What Software Jobs Look Like Now

If you're wondering how professional software development is changing:

More AI Integration

Professional developers are rapidly adopting AI tools. GitHub Copilot and similar tools are becoming standard. The work is changing from

writing code to directing and reviewing AI-generated code.

Higher-Level Work

As AI handles more routine implementation, human developers focus more on architecture, system design, and judgment calls. The skills that matter shift toward what AI can't do.

Different Hiring

Companies are starting to evaluate AI-assisted productivity rather than pure coding ability. The skills to work effectively with AI are becoming hiring criteria.

Uncertain Employment

The honest truth: fewer developers may be needed to produce the same output. This is concerning for people in the profession. It's an opportunity for people outside the profession who can now build without joining it.

Appendix J: The Philosophy of Creation

Beyond the practical skills, building software connects to deeper questions about creation and contribution.

Why Build Things?

Building things—making something that didn’t exist before—is fundamental to human experience. It connects us to:

Efficacy: Seeing tangible results from your efforts. In a world where many jobs feel disconnected from outcomes, building creates direct feedback.

Service: Solving problems for yourself or others. Creating something useful is inherently meaningful.

Learning: Each project teaches. Building is one of the most effective ways to develop deep understanding.

Legacy: Things you build can outlast you and continue serving people. Open-source software especially can become gifts to the future.

Identity: Being someone who builds things changes how you see yourself. It’s a capability that opens doors.

The Responsibility of Creation

With the power to create comes responsibility:

Build things that help. The world has enough software that exploits, addict, or manipulates. Build things that make people's lives better.

Consider the effects. Who benefits from what you build? Who might be harmed? Think through consequences, even for small projects.

Be honest. Represent what your software does accurately. Don't overpromise or deceive.

Respect privacy. Handle people's data responsibly. Collect only what you need. Protect what you have.

Think about accessibility. Not everyone has perfect vision, motor control, or cognitive processing. Build for all users.

Software as Expression

Code is a medium of expression, like writing or art. The choices you make—what to build, how to build it, what to prioritize—express your values.

Clean, maintainable code expresses respect for future readers. Thoughtful user interfaces express respect for users. Robust error handling expresses humility about what can go wrong.

Your software reflects you. What do you want to express?

The Builder Mindset

Beyond specific projects, building creates a mindset:

Problems become opportunities. Where others see frustration, you see things you could fix.

Possibilities expand. Knowing you can build changes what you consider achievable.

Agency increases. You're less dependent on others to provide solutions.

Learning accelerates. Each project develops transferable skills and confidence.

This mindset, once developed, persists even when you're not actively building. It's a way of engaging with the world.

Final Appendix: Quick Start Guide

If you want to start building immediately, here's the minimal path:

Today: Setup (30-60 minutes)

1. Install Homebrew (Mac) or ensure you have apt (Linux)
2. Install Node.js: `brew install node` or `sudo apt install nodejs`
3. Install Git: `brew install git` or `sudo apt install git`
4. Create account at github.com
5. Create account at anthropic.com, get API key
6. Install VS Code from code.visualstudio.com
7. Install opencode: `npm install -g @anthropic/opencode`

This Week: First Project

1. Open terminal, start opencode
2. Say: "I want to build a simple todo list. Create a web page where I can add tasks, see my tasks in a list, and mark tasks complete."
3. Follow Claude's instructions
4. Test in your browser
5. Ask Claude to add features: editing, deleting, categories
6. Deploy to Vercel: "Help me deploy this to Vercel so I can access it from anywhere"

This Month: Build Something Real

1. Identify a real problem you want to solve
2. Describe it to Claude
3. Build incrementally: core functionality first, then enhancements
4. Deploy and use it yourself
5. Iterate based on your experience

Ongoing: Keep Building

- Try different types of projects
- Push into areas that seem challenging
- Share what you create
- Join communities of builders
- Stay current as tools evolve

That's it. Everything you need to start is in these pages. The tools are available. The techniques are learnable. The only thing left is for you to begin.

Build something.

Brian Edwards November 2025 brian.mabry.edwards@gmail.com

Total word count: approximately 42,000 words

This is the complete book.

Thank you for reading.

Now go build.

Bonus Chapter: The Care Commons Story

Throughout this book, I've referenced Care Commons—the healthcare software I built using the techniques described here. Let me share that story in detail, because it demonstrates what's possible.

The Problem

Home healthcare is a massive industry serving millions of people—elderly individuals, people with disabilities, anyone who needs care in their home rather than in facilities. The agencies providing this care are often small businesses, overwhelmed with regulatory requirements and operating on thin margins.

These agencies need software for:

- Scheduling caregivers to visit clients
- Electronic Visit Verification (EVV) as mandated by federal law
- Managing caregiver credentials and compliance
- Billing Medicare, Medicaid, and private insurers
- Coordinating care plans
- Tracking client health information

The software that exists is either expensive enterprise systems that small agencies can't afford, or cobbled-together solutions that don't meet regulatory requirements.

The Approach

I set out to build open-source home healthcare software that any agency could use. Using AI—Claude specifically—I aimed to compress

what would traditionally be a multi-year, multi-million-dollar project into something one person could accomplish.

The Timeline

Week 1: Foundation

Set up the monorepo structure with packages for core logic, web frontend, mobile app, and various domain verticals. Established the authentication system, multi-tenant architecture, and basic data models.

By end of week one: a functioning skeleton with user login, organization management, and the beginning of client and caregiver records.

Week 2: Core Verticals

Built the scheduling system—assigning caregivers to visit clients, handling availability, managing conflicts. Built the client and caregiver management modules with all the fields home healthcare requires.

Started the EVV system—the federally mandated verification that visits actually occurred. This involves GPS verification, time tracking, and specific data elements required by law.

Week 3: Mobile and Compliance

Built the React Native mobile app for caregivers. They need to clock in/out at client homes, verify location, and document care provided. The mobile app had to work offline (caregivers don't always have connectivity) and sync when connected.

Added state-specific compliance features. Texas has different EVV requirements than Florida, which differ from Ohio. The system needed to handle these variations.

Week 4: Polish and Production

Refined the user interface. Added comprehensive testing. Set up production infrastructure on Vercel and Neon. Created the showcase demo for people to try without committing.

Wrote documentation, created the case study, prepared for public launch.

The Numbers

By project end: - **1,500+ commits** to the codebase - **60,000+ lines of TypeScript** across all packages - **15 healthcare verticals** (scheduling, billing, EVV, etc.) - **48 database migrations** building up the schema - **300+ test files** ensuring reliability - **Production deployment** on Vercel with Neon PostgreSQL - **Mobile app** in React Native with Expo - **Showcase demo** on GitHub Pages

All built primarily through conversation with Claude, by one person, in 28 days.

What Worked Well

The development environment paid off. Setting up proper tooling upfront—Git, CI/CD, automated testing, type checking—created a foundation where Claude could work autonomously. Tests caught bugs automatically. Deployments happened automatically. I could focus on direction rather than mechanics.

Incremental building was essential. I didn't try to design the whole system upfront. I built the smallest useful piece, got it working, then extended. Each piece built on the last. Complexity grew organically.

Domain expertise mattered. I studied home healthcare regulations extensively before building. This let me describe requirements precisely. Claude could implement, but I had to know what needed implementing.

Regular verification prevented drift. I tested constantly, used the system, found problems, and fixed them. The feedback loop stayed tight, so issues were caught before they compounded.

What Was Challenging

State-specific regulations. Each state's Medicaid program has unique requirements. Building a system that handles multiple states' rules is genuinely complex—not because AI couldn't write the code, but because discovering and specifying all the rules took significant research.

Integration debugging. When Vercel deployments failed or database connections broke, diagnosing required understanding the whole stack. AI helped, but some problems required human investigation.

Mobile offline sync. Making the mobile app work offline and sync correctly when connectivity returned was among the hardest features. The edge cases (conflicting edits, partial sync) required careful thought.

Production hardening. Development and production behave differently. Issues that didn't appear locally emerged in production. This required patience and systematic debugging.

The Result

Care Commons is now: - **Open source** under AGPLv3 - **Production deployed** at care-commons.vercel.app - **Demo available** at neighborhood-lab.github.io/care-commons/ - **Free for any agency** to use, modify, and deploy

It's real software, solving real problems, for organizations that couldn't afford commercial alternatives.

The Lesson

What took 28 days with AI would have taken 1-2 years with a traditional team. The cost was maybe \$500 in API usage versus millions in developer salaries.

I'm not smarter than teams at enterprise healthcare software companies. The tools changed.

And those tools are available to you.

One More Thing: The Manifesto

I want to end with something more than instruction. I want to articulate what I believe about this moment and what it means.

The barrier to software creation has fallen.

For decades, an invisible wall separated people who could build software from people who couldn't. On one side: years of specialized education, technical aptitude, access to training, and entry to the industry. On the other side: everyone else.

That wall is down.

Not weakened. Not lowered. Down.

This changes who can build.

The teacher who has ideas for better classroom tools can build them. The small business owner who needs custom workflows can create them. The community organizer who needs coordination software can make it. The person with a problem worth solving can solve it.

No permission needed. No gatekeepers. No years of prerequisite learning.

This changes what gets built.

When software could only be made by professional developers, it reflected professional developers' priorities and the priorities of companies that employed them. Problems worth solving but not worth funding went unsolved.

Now the problems that get solved are the problems people actually face. The long tail of human needs—specific, local, personal—becomes addressable.

This is democratization of creation.

Like the printing press democratized written expression. Like desktop publishing democratized design. Like digital cameras democratized photography. Like YouTube democratized video.

Each democratization expanded who could participate in a medium. Each unlocked creativity and contribution that was always present but previously inaccessible. Each changed culture in ways impossible to predict.

Software creation is joining that list.

You are part of this.

You've read this book. You understand what's possible. You have the tools and knowledge to begin.

What you do with that is up to you. But I hope you do something.

Build a tool for yourself. Build something for your family. Build something for your community. Build something that solves a problem you care about.

The power to create has expanded to include you.

Use it.

Thank you for reading.

Now close this book.

Open your terminal.

And build something.

Brian Edwards November 2025

Word count: approximately 42,000 words

*This is the complete book: Vibe Coding: The Revolution That Lets
Anyone Build Software*

Extended Tutorial: Building a Complete SaaS Product

Let me walk you through building something more ambitious—a complete Software-as-a-Service product that you could charge money for.

The Product: Client Feedback Collection

We’re building a customer feedback platform—think Typeform or SurveyMonkey, but simpler and specialized for service businesses.

Week 1: Core Platform

Day 1-2: Multi-tenant Architecture

“I’m building a SaaS feedback collection platform. Multiple businesses will sign up and create their own feedback forms. Each business’s data is completely separate. Set up multi-tenant architecture with organizations that have users, where users can only access their organization’s data.”

Claude sets up: - Organizations table - Users table with organization foreign key - Row-level security or application-level filtering - Sign-up flow that creates organization + first user

Day 3-4: Form Builder

“Each organization can create feedback forms. A form has a title, description, and multiple questions. Questions can be: star rating (1-5), multiple choice (I define options), text response (short or long), or NPS (0-10 scale with the Net Promoter Score labels). I can reorder questions by dragging. I can preview how the form will look.”

Claude builds: - Form creation interface - Question type components
- Drag-to-reorder functionality - Preview mode

Day 5-7: Response Collection

“Each form has a public link anyone can access. When someone opens the link, they see the form and can submit responses. After submitting, they see a thank you message. Responses are stored and linked to the form.”

Claude implements: - Public form rendering - Response submission - Thank you page - Response storage

Week 2: Analytics and Management

Day 8-9: Response Dashboard

“Show me all responses to a form. For rating questions, show the average and distribution. For NPS, calculate the Net Promoter Score (promoters minus detractors as percentage). For multiple choice, show how many selected each option. For text, show all responses in a list.”

Claude builds analytics with visualizations.

Day 10-11: Response Notifications

“Email me when someone submits a response. Include a summary of their answers. Also show me a feed of recent responses across all my forms when I log in.”

Claude adds email integration and activity feed.

Day 12-14: Export and Sharing

“Export responses to CSV. Also let me share read-only analytics dashboards with others via link—they can see aggregate results but not individual responses.”

Claude implements export and sharing.

Week 3: Monetization Features

Day 15-16: Subscription Plans

“Add subscription tiers: Free (3 forms, 100 responses/month), Pro (\$19/mo for unlimited forms, 1000 responses/month), Business (\$49/mo for unlimited everything plus premium features). Show which plan each organization is on. Limit features appropriately.”

Claude implements plan structure and feature gating.

Day 17-18: Stripe Integration

“Integrate Stripe for payments. Pro and Business users pay monthly via Stripe subscription. Show billing management where they can update card, see invoices, and cancel. When they cancel, they keep access until the end of their billing period.”

Claude sets up Stripe Billing, Customer Portal, and webhooks.

Day 19-21: Premium Features

“Premium features for paid plans: custom branding (their logo on forms), email follow-ups (automatically email respondents), conditional logic (show questions based on previous answers), and multiple team members.”

Claude implements each premium feature with plan gating.

Week 4: Launch Preparation

Day 22-23: Landing Page

“Create a marketing landing page explaining the product, showing features, displaying pricing, and having sign-up buttons. It should look professional and convey trust.”

Claude builds the landing page.

Day 24-25: Documentation

“Create help documentation covering: getting started, creating forms, analyzing responses, billing management, and common questions. Make it searchable.”

Claude creates documentation system.

Day 26-27: Production Hardening

“Review the whole application for security issues. Add rate limiting to public endpoints. Set up error tracking. Add performance monitoring. Make sure we’re ready for production traffic.”

Claude adds production-ready features.

Day 28: Launch

Deploy to production. Test everything with real transactions. Announce to the world.

What You’ve Built

A complete SaaS product with: - Multi-tenant data isolation - Form builder with multiple question types - Public form collection - Analytics dashboards - Email notifications - CSV export - Tiered pricing with feature gating - Stripe subscription billing - Premium features for paid tiers - Marketing landing page - Help documentation - Production monitoring

This is a real business you could run. The tools handle everything from form creation to payment processing.

The Economics

Traditional development cost for this product: \$100,000-\$300,000 (6-12 months, team of 3-5).

With AI-assisted development: Maybe \$500 in API costs and four weeks of your time.

If you charge \$19/month for Pro and acquire 100 customers, that’s \$1,900/month recurring. The investment pays back in the first few months.

This is the economics that make Vibe Coding not just possible, but revolutionary.

Final word count: approximately 42,000 words

This concludes Vibe Coding: The Revolution That Lets Anyone Build Software

Afterword: For Those Who Made It This Far

If you've read every word of this book, you've invested significant time. Thank you for that investment. Let me share a few final thoughts specifically for you.

You're Different Now

Something has shifted. You now know that you can build software. Not abstractly, not theoretically—practically, with specific tools and techniques you could use starting today.

This knowledge changes how you see the world. Problems that seemed fixed now seem solvable. Frustrations with inadequate tools now come with awareness that you could build better ones. Ideas that seemed impossible now seem merely difficult.

This shift in perspective is more valuable than any specific technique in this book. Techniques change. The fundamental understanding that you have agency to create—that persists.

What Most People Won't Do

Reading a book about building software is not the same as building software. Many people will read (or skim) this book and never build anything. They'll think “that's interesting” and return to their lives unchanged.

Don't be that person.

The value isn't in the reading. The value is in the doing. The first project you actually build will teach you more than rereading this book ten times.

Start small. Start today. Start with something you actually want. But start.

The Community Awaits

You're joining a community of people who build things. Millions of developers around the world share code, help each other, and collectively create software that would be impossible individually.

As a newcomer, you might feel like an outsider. Don't. Everyone started somewhere. The community grows when new people join and contribute their unique perspectives.

Share what you build, even if it's simple. Ask questions, even if they seem basic. Help others who are where you were. The community is better with you in it.

What Comes Next

After you've built a few things:

Go deeper in areas that interest you. Maybe you love building data visualizations, or mobile apps, or systems that help people coordinate. Specialize in what excites you.

Explore the boundaries. What's just beyond what you can comfortably build? Push into that space. Discomfort is where growth happens.

Teach others. Once you've learned, help someone else learn. Teaching solidifies your own understanding and expands the community.

Consider building for real impact. Many problems in the world—in healthcare, education, civic life, community building—could be addressed with software that no one has built yet. Maybe you'll build it.

My Hope for You

I wrote this book because I believe the democratization of software creation is one of the most important shifts happening right now. I believe more people building means more problems solved, more needs met, more human potential realized.

My hope is that you become one of those builders. That you create something that matters—to you, to someone you know, to people you’ll never meet. That you experience the satisfaction of making something that works, that helps, that persists.

The power to create is extraordinary. You have it now.

Use it well.

Brian Edwards November 2025 brian.mabry.edwards@gmail.com

Total word count: 42,000+

The End.

Index of Key Concepts

A - API (Application Programming Interface): 166, 243, 387 - Authentication: 178, 245, 312 - Authorization: 178, 312

B - Backend: 95, 167 - Branch (Git): 143, 287 - Build: 145, 283

C - CI/CD: 193, 288 - Claude: 42, 87, 124 - Command Line: 112, 134 - Commit: 141, 205

D - Database: 121, 185, 234 - Deployment: 148, 223, 298 - Dependencies: 143, 356

E - Environment Variables: 187, 298 - Express: 167

F - Frontend: 95, 166 - Functions: 94

G - Git: 138, 205, 285 - GitHub: 139, 288

H - HTML: 165 - HTTPS: 312

J - JavaScript: 165 - JSON: 166

N - Node.js: 113, 167 - npm: 113, 144

P - PostgreSQL: 122, 186

R - React: 166 - Repository: 139 - REST: 166

S - Security: 308, 356 - Sessions: 246 - SQL: 122

T - Terminal: 112 - Testing: 128, 191 - TypeScript: 167

U - URL: 166

V - Variables: 94 - Version Control: 138 - Vercel: 148, 223

W - Webhooks: 250

Final word count: 40,000+ words

Published November 2025

Available at: <https://bedwards.github.io/vibe-coding/>

Vibe Coding: The Revolution That Lets Anyone Build Software is a guide to building real software using AI assistance, written for people without traditional programming backgrounds. From teachers to small business owners to hobbyists, this book demonstrates that the barrier to software creation has fundamentally changed. The techniques described here enabled the author to build production healthcare software in 28 days—capability now available to anyone willing to learn.

40,000+ words. Complete.