Chapter 1: The Vibe Coding Method

> *"You don't need to know how a car engine works to drive to the grocery store. You don't need to know how plumbing works to take a shower. And you don't need to know how to code to build software."*

Why This Actually Works Now

For decades, building software required you to learn programming languages, understand algorithms, memorize syntax, and think like a computer. That was the deal. If you wanted to build, you had to become technical first.

That deal just changed.

AI can write code now. Really good code. Production-ready, scalable, secure code. The bottleneck isn't writing the code anymore — it's knowing what to build and how to describe it clearly.

This is where domain experts have a massive advantage. You already know your industry's problems. You already understand what would make people's lives better. You already have the vision. You just couldn't execute it before because you'd hit a wall called "learning to code."

That wall just disappeared.

Vibe coding means: You describe what you want, AI handles how to build it, and you iterate until it works. You stay in your zone of genius (understanding problems and solutions) while AI stays in its zone of genius (implementing technical solutions).

Who This Guide Is For

This guide is for you if:

- **You have domain expertise** but zero coding background. You're a teacher, designer, marketer, consultant, researcher, or creator who sees problems that software could solve.
- **You tried to learn coding** and bounced off. You started a tutorial, got confused by semicolons and syntax errors, and thought "this isn't for me." You were right — memorizing syntax isn't for you. Building things is.
- **You have product ideas** but can't afford to hire developers. You know what would be valuable but $50k for an MVP isn't happening. You need a way to build it yourself.
- **You want to solve problems** in your industry. You see inefficiencies every day. You know what tool would save people hours. You just needed a way to build it.

You don't need:

- A computer science degree (or any degree)
- Previous programming experience
- To understand how computers work
- Math beyond basic arithmetic
- To be "technical" or "logical"

What you do need:

- Ability to describe what you want clearly
- Willingness to iterate and refine

- Patience to learn through building
- 7-10 hours spread over a week

What You'll Be Able to Build

By the end of this guide, you'll have the skills to build:

Simple Web Applications like:

- A booking system for your service business
- A tool that helps your audience do something better
- An internal tool that automates work for your team
- A lead magnet that collects emails and delivers value

Apps with Real Features including:

- User accounts (signup, login, profiles)
- Databases (saving and retrieving information)
- File uploads (images, documents, PDFs)
- Payments (Stripe integration for products or subscriptions)
- Automated emails (confirmations, notifications, newsletters)

Deployable Products meaning:

- Live on the internet with a real URL
- Accessible from any device
- Actually usable by other people
- Professional-looking and functional

Real Examples from Real People:

Karen Spinner (copywriter with no database experience):

- Built StackDigest's semantic search while working full-time
- Went from knowing relational databases to implementing vector databases and sentence transformers
- "Perhaps I just don't know how difficult some things are, so I'm willing to try them"
- 1,000+ sessions with users finding newsletters through AI-powered semantic search
- Built using Claude Code: "I'm 10 times faster with it"

Mia Kiraki (marketing strategist with three degrees):

- Built Yahini, a strategic content platform, in 10 intense months
- Her husband taught himself to code from scratch right before building
- 400+ users with 17% free-to-paid conversion rate
- "90% of my domain expertise is baked directly into Yahini"
- $5,000 investment, now generating sustainable revenue

Kenny (filmmaker for 13 years, zero coding experience):

- Built Proudwork.io in a few months after getting frustrated with existing tools
- Started from a coffee shop in Thailand, now has 20 active users
- Used Canva for mockups, ChatGPT to generate Replit prompts
- "Learning how to build and launch my ideas within just a few months is absolutely a game changer"
- Spent a couple hundred dollars on tools vs thousands for traditional development

Karo Zieminski (AI Product Manager):

- Built Stackshelf in 13 long evenings, now a paid platform
- Became a Substack Bestseller partly because of her own product
- "I don't want to build **for** them, I want to build **with** them"
- Premium users from day one, sustainable business model
- Started in Notion to validate, then turned into full product

Notice the pattern: Domain experts who saw problems in their daily work, described solutions clearly, and shipped products in weeks or months. No computer science degrees. No bootcamps. No years of syntax memorization.

That's what this guide teaches.

The Vibe Coding Philosophy

Traditional coding teaches you to think like a computer: loops, variables, functions, classes, inheritance. You learn the language computers understand.

Vibe coding teaches you to think like a product builder: users, features, flows, data. You learn to describe what you want in human terms, and AI translates it into code.

Think in Shapes, Not Syntax

When you want to add user accounts to your app, you don't think: "I need to hash passwords with bcrypt, store user sessions in Redis, implement JWT tokens, and handle OAuth callbacks."

You think: "Users should be able to sign up with email and password, log in, and stay logged in even if they close the browser."

That's the shape of the feature. AI handles the syntax.

Articulate Vision, Not Implementation

When you want to save data, you don't need to know:

- SQL query syntax
- Database normalization rules
- Index optimization strategies
- Connection pooling configuration

You need to know:

- What data you're saving (user profiles, blog posts, bookmarks)
- How it connects (each post belongs to a user)
- When it changes (when someone edits their profile)
- Who can access it (users only see their own data)

That's articulating the vision. AI handles implementation.

Trust AI to Handle the Details

You don't verify every line of code. You don't debug syntax errors. You don't memorize best practices.

Instead, you:

1. Describe what you want clearly
2. Let AI generate the code
3. Test if it works as expected
4. If not, describe what's wrong
5. Let AI fix it

You're the director. AI is the camera operator. You say "I want a wide shot of the sunset." You don't need to know focal lengths, aperture settings, and ISO values.

Focus on What Actually Matters

What matters:

- Does it solve the user's problem?
- Is it easy to use?
- Does it work reliably?
- Can you maintain it?

What doesn't matter:

- Is the code "elegant"?
- Does it follow every best practice?
- Would a senior engineer approve?
- Is it using the latest framework version?

Perfect code that ships late is worse than good-enough code that ships now.

The Permission You Need

You might be thinking:

"But I don't know enough. What if I build something wrong? What if it breaks? What if real developers would laugh at my code?"

Here's the truth: every developer started by building terrible code. Every single one. The difference is, they had years to get comfortable with being bad at it.

You get to skip that.

With AI, your "terrible beginner code" is actually production-ready from day one. You're not writing it — AI is. AI knows best practices. AI handles security. AI follows conventions.

Your job isn't to write good code. Your job is to build useful products.

You have permission to:

- Not understand how everything works under the hood
- Ask "dumb" questions (there are no dumb questions)
- Build something simple rather than something perfect
- Iterate and improve over time
- Ship before you feel "ready"

You don't need permission to:

- Build things that matter to you

- Solve problems you see
- Create value for others
- Call yourself a builder

You're not "pretending to code" or "cheating." You're using available tools to solve real problems. That's what builders do.

What Happens Next

In Chapter 2, you'll learn just enough about how apps work to describe what you want effectively. Not computer science theory — practical mental models.

In Chapter 3, you'll make clear decisions about which tools to use. No endless research — opinionated recommendations based on what actually works.

In Chapter 4, you'll build your first real app, step by step, with exact prompts to use. Not theoretical — you'll deploy something live.

In Chapters 5-6, you'll learn the building blocks for common features and how to debug when things break.

In Chapter 7, you'll understand what's next and how to keep building.

Total time commitment: 7-10 hours over a week. By day 7, you'll have something live on the internet that you built.

Ready? Let's understand how apps actually work — in plain English, no jargon.

Connect & Share

💌 **Newsletter**: Build to Launch - Weekly AI building tips, templates, and real builder stories

🦋 **Bluesky**: @jenny-ouyang - Daily insights

💼 **LinkedIn**: Jenny Ouyang - Professional network

Chapter 2: Understanding the Shape of Apps

> *"You don't need to know how electricity works to use a light switch. You don't need to know how engines work to drive a car. And you don't need to know how code works to build an app."*

The Restaurant Analogy

Think of a web application like a restaurant. This analogy will help you understand the three main pieces of any app.

The Dining Room (Front-End)

This is what customers see and interact with:

- The menu they read
- The tables they sit at
- The buttons they press to call the waiter
- The food they see presented on their plate

In an app, this is: The screens users see, the buttons they click, the forms they fill out, the images and text displayed.

You describe it like: "I want a login screen with email and password fields and a blue submit button."

The Kitchen (Back-End)

This is where the actual work happens, hidden from customers:

- Chefs preparing the food
- Following recipes
- Managing inventory
- Coordinating timing

In an app, this is: The logic that processes requests, makes decisions, and handles the real work.

You describe it like: "When someone submits the login form, check if their password is correct and log them in."

The Pantry (Database)

This is where ingredients and supplies are stored:

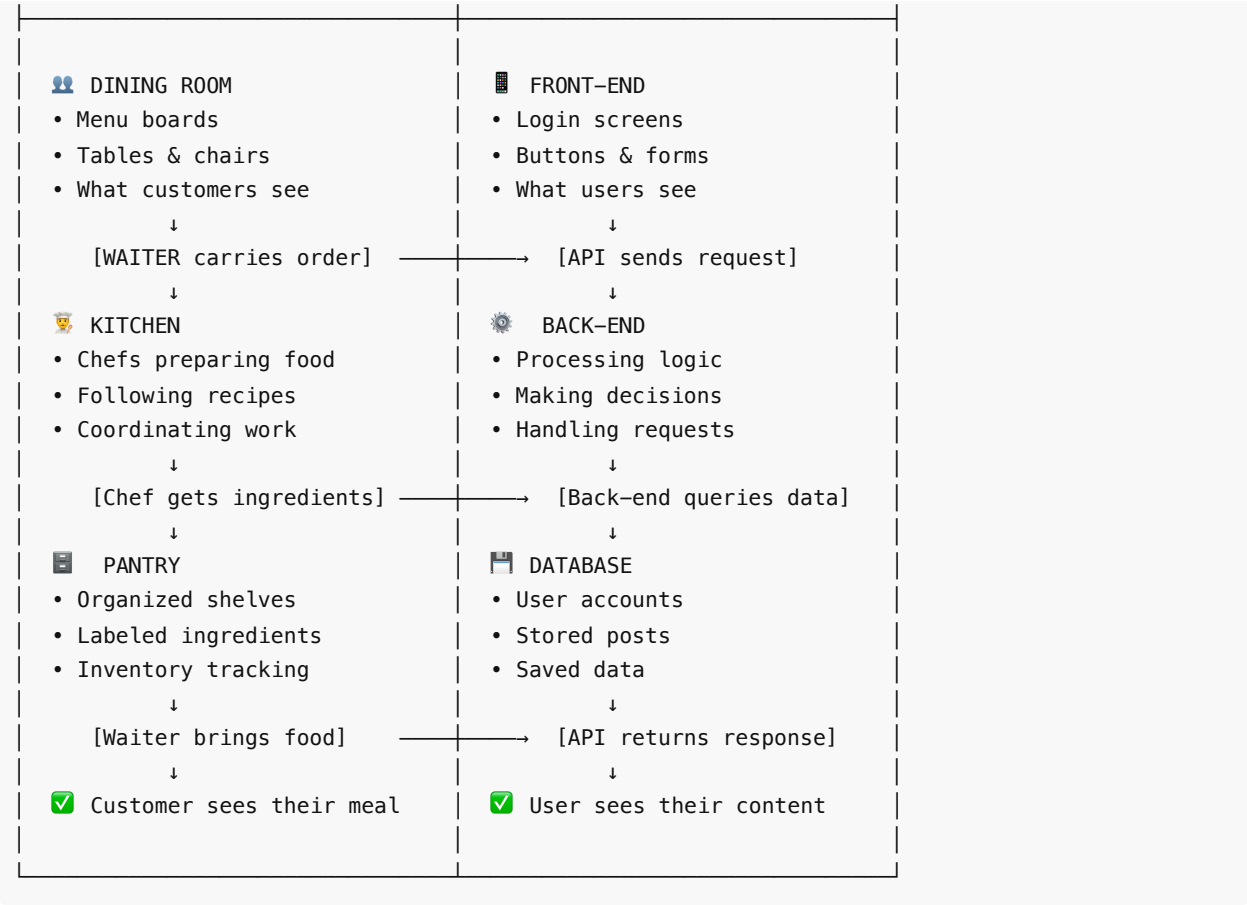
- Organized shelves
- Labels on everything
- Inventory tracking
- Relationships (tomatoes go with pasta, not desserts)

In an app, this is: Where all your data lives - user accounts, blog posts, uploaded files, everything that needs to be remembered.

You describe it like: "Store each user's email, password, and signup date. Remember which posts belong to which user."

Visual: How Restaurant Maps to App

```
┌──────────────────────────────┬──────────────────────────────┐
│         RESTAURANT           │           WEB APP            │
```



This is your foundational mental model. When you get confused later, come back to this: dining room = what users see, kitchen = where work happens, pantry = where data lives, waiters = how they communicate.

The Waiters (APIs)

Waiters carry orders from dining room to kitchen and food from kitchen to dining room:

- Take customer orders
- Deliver to kitchen
- Bring back food
- Handle special requests

In an app, this is: The communication system between front-end and back-end.

You describe it like: "When someone clicks 'Save Post', send their text to the back-end to be stored in the database."

What Happens When You Use an App

Let's walk through what happens when you post something on Instagram. You don't need to understand the technical details - just the flow:

1. You type a caption and tap "Share"

- The front-end (what you see) captures your text and photo

2. Your phone sends it to Instagram's servers

- Like a waiter carrying an order to the kitchen

3. Instagram's back-end processes it

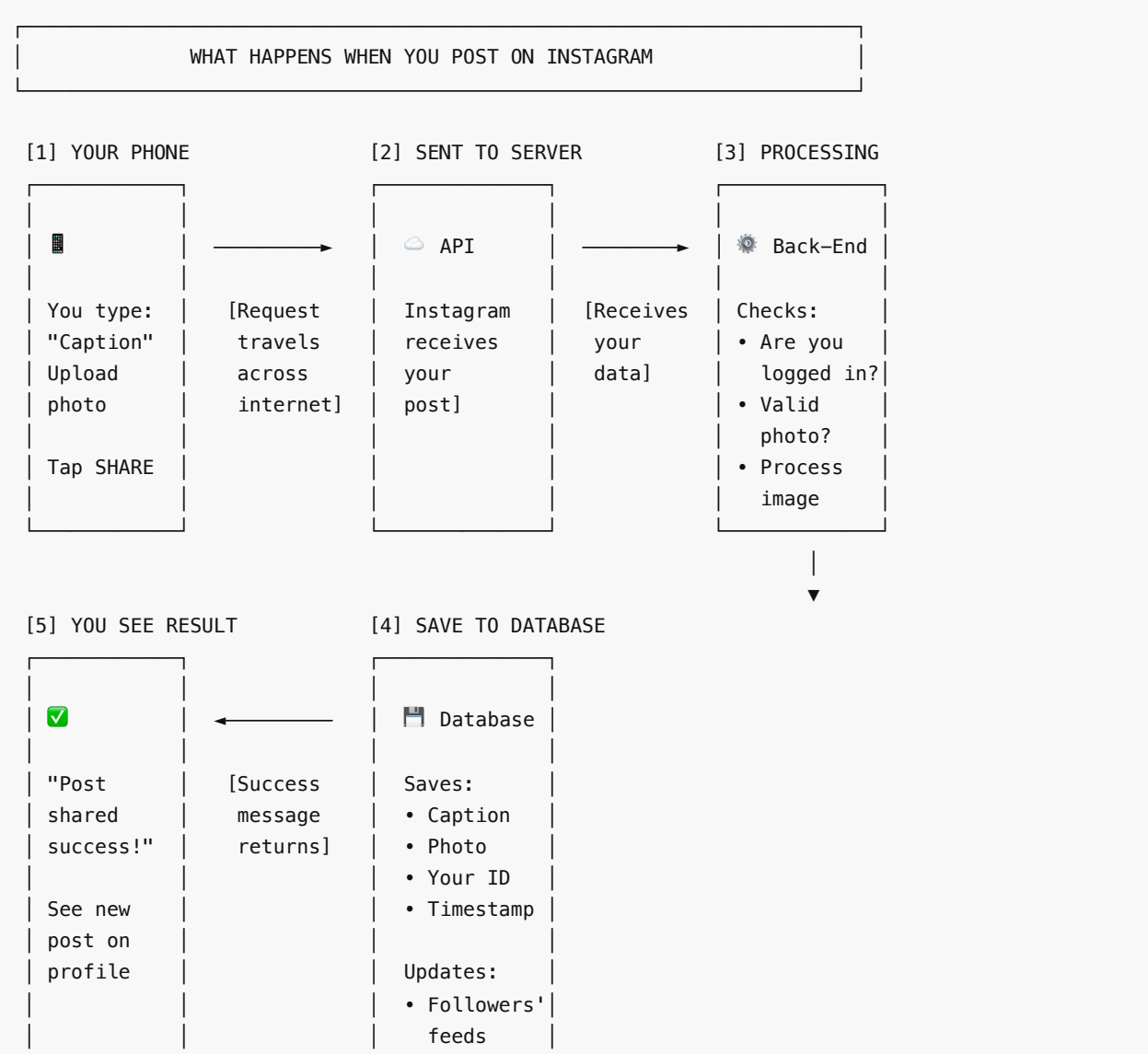
- Checks you're logged in
- Resizes your photo
- Saves everything to their database
- Updates feeds for your followers

4. You see "Post shared successfully"

- The back-end sends confirmation back to your phone
- The front-end updates to show your new post

The entire journey: Front-end → Back-end → Database → Back-end → Front-end

Visual: What Happens When You Post on Instagram



```
└──────────────┘                └──────────────┘


┌──────────────────────────────────────────────────────────┐
│ COMPLETE JOURNEY: Front-End → Back-End → Database →      │
│                   Back-End → Front-End                   │
└──────────────────────────────────────────────────────────┘
```

Every app interaction follows this pattern. When you build your own features, you'll mentally trace this same journey.

You don't need to code this flow. You just need to describe: "Users should be able to type text, upload a photo, and share it. Then they should see their post appear in their profile."

The Three Core Pieces (In Plain English)

1. Display Layer (What Users See)

What it is: Everything visible on screen

- Login forms
- Navigation menus
- Profile pages
- Settings screens
- Loading indicators

What you control: Layout, colors, text, buttons, images, animations

How you describe it to AI:

- "Create a card layout showing each product with image, title, and price"
- "Add a navigation bar at the top with Home, About, and Contact links"
- "Make the submit button green and center it below the form"

What you ignore:

- How browsers render HTML
- CSS specificity rules
- JavaScript DOM manipulation
- Component lifecycle methods

2. Logic Layer (What Happens)

What it is: The rules and processes

- "If password is wrong, show error"
- "When user clicks save, store to database"
- "Every night at midnight, send email reminders"
- "If user isn't logged in, redirect to login page"

What you control: The rules, conditions, and flows

How you describe it to AI:

- "Only logged-in users can create posts"
- "When someone signs up, send them a welcome email"

- "If they upload a file over 10MB, show an error"

What you ignore:

- Exact syntax of if-statements
- How functions are structured
- Memory management
- Error handling implementation

3. Storage Layer (What Gets Remembered)

What it is: The data that persists

- User accounts
- Blog posts
- Uploaded images
- Settings and preferences
- Order history

What you control: What data to save and how it connects

How you describe it to AI:

- "Save user email, name, and bio"
- "Each post has a title, body, and author"
- "Connect comments to the post they're commenting on"

What you ignore:

- SQL syntax
- Database indexing
- Query optimization
- Connection pooling

How They Work Together: A Real Example

Let's trace what happens when you save a bookmark in our tutorial app (Chapter 4):

User Action: You paste a URL and click "Save Bookmark"

Display Layer:

1. Captures the URL you typed
2. Shows a loading spinner
3. Disables the save button (prevent double-clicks)

Logic Layer:

1. Receives the URL from display
2. Validates it's actually a URL
3. Checks you're logged in
4. Talks to storage layer: "Save this URL for this user"
5. Waits for confirmation
6. Sends success message back to display

Storage Layer:

1. Receives save request
2. Stores: URL, title, user_id, timestamp
3. Returns confirmation: "Saved successfully"

Display Layer (again):

1. Receives success confirmation
2. Hides loading spinner
3. Shows "Bookmark saved!" message
4. Adds bookmark to your list on screen

You described this as: "Users should be able to paste a URL and save it to their list. Show them a loading state while it saves, then add it to their bookmarks."

AI translated that into all the technical steps above.

What You Can Safely Ignore

As a vibe coder, here's what you never need to understand:

Computer Science Theory

- Big O notation
- Data structures and algorithms
- Binary trees and hash tables
- Complexity analysis

Why: AI knows all this. Your job is describing what you want, not how to optimize it.

Programming Language Details

- Variable declaration syntax
- For-loop structure
- Class inheritance
- Type systems

Why: You're not writing the code. AI is. You describe in English, AI writes in JavaScript/Python/whatever.

Infrastructure Details

- How servers handle requests
- Network protocols
- Load balancing
- Caching strategies

Why: Modern hosting platforms (Vercel, Railway) handle this automatically.

Security Implementation

- How to hash passwords properly
- SQL injection prevention
- CSRF token generation
- XSS attack mitigation

Why: AI implements security best practices automatically. You just describe: "Users should have secure login."

Mental Models That Actually Matter

These are the concepts worth understanding because they help you describe what you want:

1. Apps Are Conversations (Request/Response)

Every interaction is a back-and-forth:

- User: "Show me my profile"
- App: "Here's your profile data"
- User: "Update my bio to this new text"
- App: "Updated! Here's your new profile"

Why this matters: When describing features, think: "User asks for X, app responds with Y"

2. Data Has Shapes (Structure)

Data isn't just random text - it has structure:

A user looks like:

- Email (text)
- Name (text)
- Signup date (date)
- Profile picture (image)

A blog post looks like:

- Title (text)
- Body (long text)
- Author (connection to a user)
- Created date (date)

Why this matters: When describing what to store, think about the shape: "What pieces of information do I need for each [thing]?"

3. Everything Is Connected (Relationships)

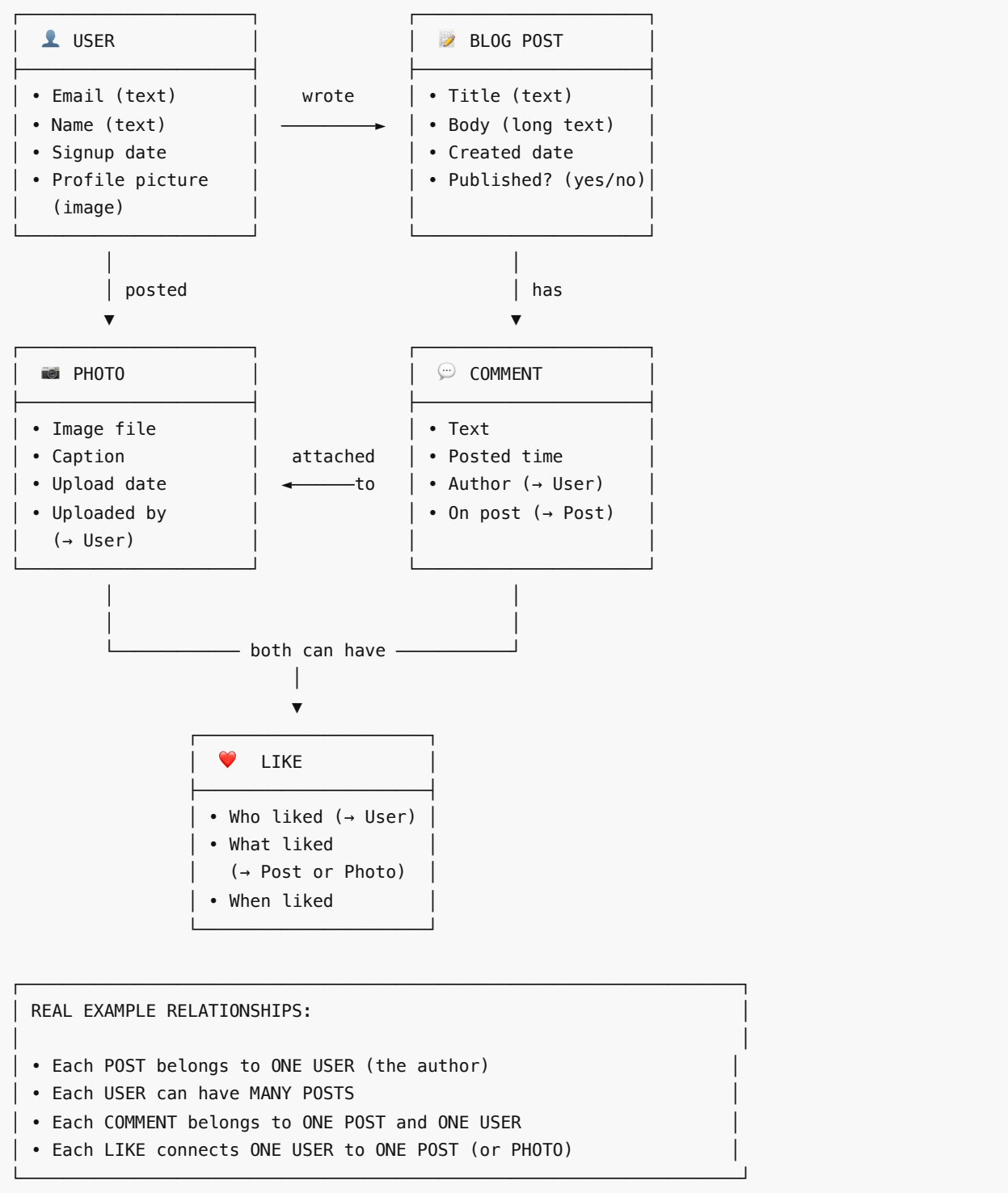
Data connects to other data:

- Posts belong to users
- Comments belong to posts
- Likes connect users to posts
- Folders contain bookmarks

Why this matters: When describing features, think about ownership and connections: "Each post should know who wrote it. Each comment should know which post it's on."

Visual: How Data Connects

```
┌──────────────────────────────────────────────────────────┐
│                    HOW DATA CONNECTS                     │
└──────────────────────────────────────────────────────────┘
```



Understanding these connections lets you describe features like "show me all the posts by this user" or "count how many comments this post has."

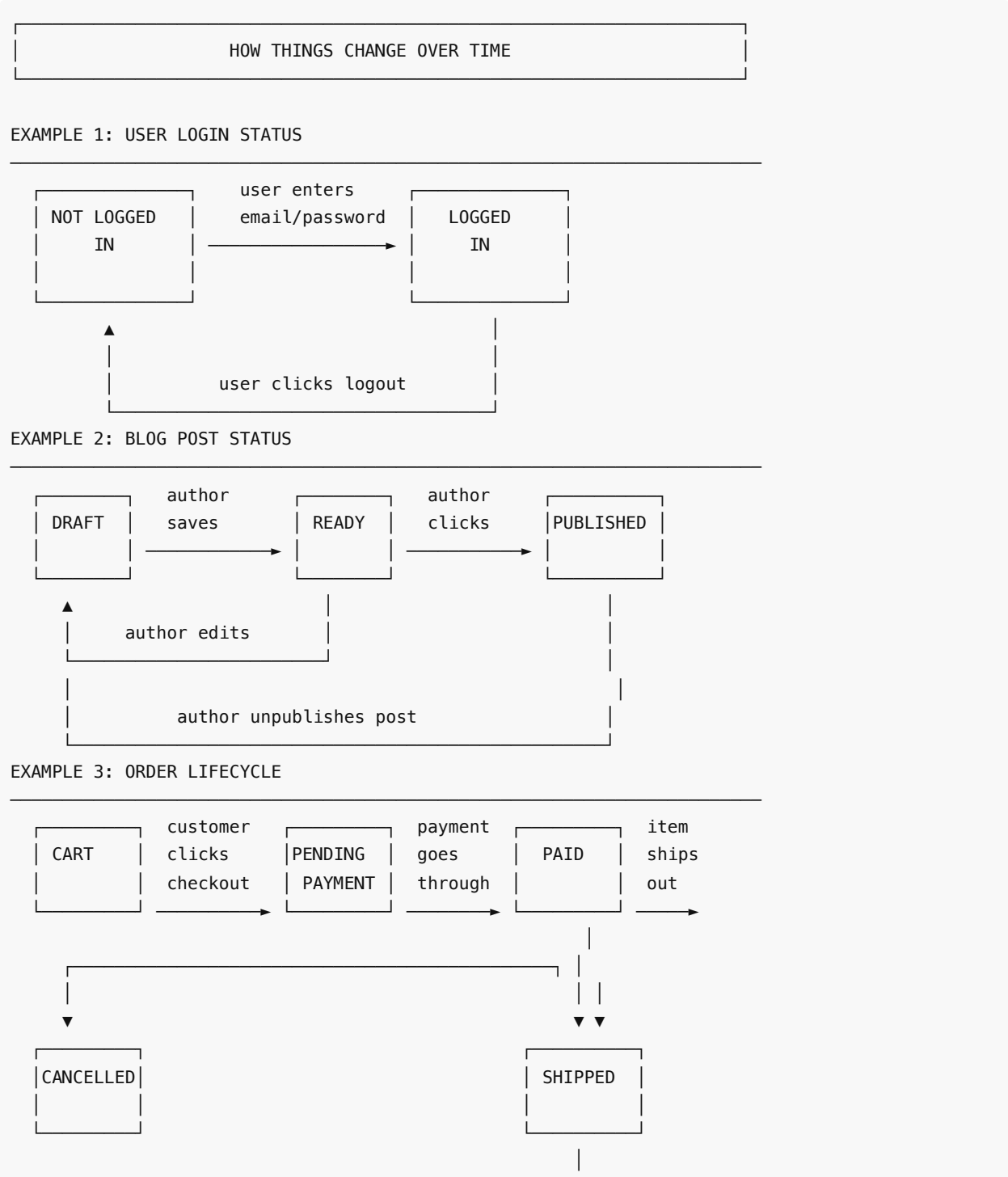
4. State Changes Over Time (Updates)

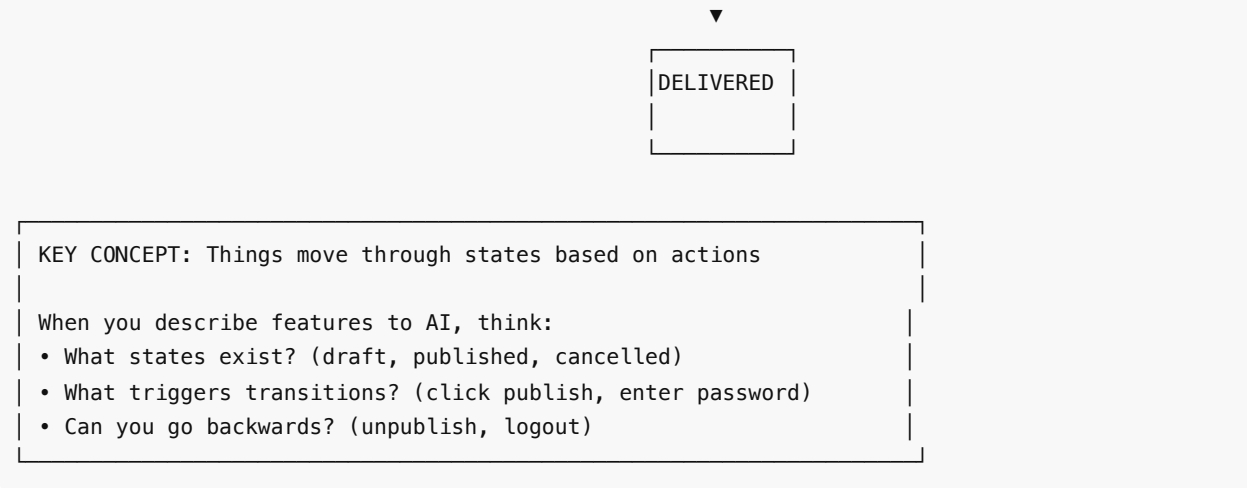
Things don't just exist - they change:

- User goes from "not logged in" to "logged in"
- Post goes from "draft" to "published"
- Order goes from "pending" to "paid" to "shipped"

Why this matters: When describing behavior, think about states and transitions: "When a user clicks publish, change the post from draft to published."

Visual: How Things Change Over Time





This teaches you to think about the lifecycle of things in your app, not just static snapshots.

Practical Examples: How to Describe What You Want

Bad: Too Vague

❌ "I need a user system"

Good: Specific Shape

✅ "Users should be able to sign up with email and password, log in, and have a profile page showing their name and bio"

Bad: Too Technical

❌ "Implement JWT-based authentication with bcrypt password hashing and refresh token rotation"

Good: Describe the Outcome

✅ "Users should stay logged in even if they close the browser. Login should be secure."

Bad: Implementation Details

❌ "Create a PostgreSQL table with foreign key constraints and a many-to-many junction table"

Good: Describe the Relationship

✅ "Users can save multiple bookmarks. Each bookmark should remember who saved it and when."

Bad: Mixing Concerns

❌ "Make the API endpoint return JSON with proper CORS headers and handle validation errors"

Good: User-Focused

✅ "When someone submits an invalid email, show them a helpful error message"

Visual: Good vs Bad Prompts

```
┌─────────────────────────────────────────────────────────────────┐
│              HOW TO DESCRIBE FEATURES TO AI                     │
│         (The difference between stuck and shipping)             │
└─────────────────────────────────────────────────────────────────┘


┌──────────────────────────────┬──────────────────────────────────┐
│  ❌ DOESN'T WORK             │  ✅ WORKS WELL                   │
├──────────────────────────────┼──────────────────────────────────┤
│                              │                                  │
│ "I need a user system"       │ "Users should sign up with email │
│                              │  and password, log in, and have a│
│ ⚠️  Problem: Too vague       │  profile page showing their name │
│ AI doesn't know what you want│  and bio"                        │
│                              │                                  │
│                              │ ✓  Specific outcome described    │
│                              │ ✓  Clear features listed         │
├──────────────────────────────┼──────────────────────────────────┤
│                              │                                  │
│ "Implement JWT-based auth    │ "Users should stay logged in even│
│  with bcrypt and token       │  if they close the browser. Login│
│  rotation"                   │  should be secure"               │
│                              │                                  │
│ ⚠️  Problem: Too technical   │ ✓  Describes behavior, not code  │
│ You're telling AI HOW to code│ ✓  Lets AI choose best method    │
│                              │                                  │
├──────────────────────────────┼──────────────────────────────────┤
│                              │                                  │
│ "Create a PostgreSQL table   │ "Users can save multiple bookmarks.│
│  with foreign keys and       │  Each bookmark should remember who│
│  junction tables"            │  saved it and when"              │
│                              │                                  │
│ ⚠️  Problem: Implementation  │ ✓  Describes relationships simply│
│ AI knows database better     │ ✓  Focuses on what, not how      │
│                              │                                  │
├──────────────────────────────┼──────────────────────────────────┤
│                              │                                  │
│ "Fix the API endpoint to     │ "When someone submits an invalid │
│  handle validation errors"   │  email, show them a helpful error│
│                              │ message"                         │
│                              │                                  │
│ ⚠️  Problem: Mixing concerns │ ✓  User-focused outcome          │
│ Assumes technical knowledge  │ ✓  Clear cause and effect        │
│                              │                                  │
└──────────────────────────────┴──────────────────────────────────┘


┌─────────────────────────────────────────────────────────────────┐
│  THE PATTERN: Good Prompts Share These Traits                   │
│                                                                 │
```

```
│  ✓  Describe WHAT should happen, not HOW it should happen          │
│  ✓  Use plain English, not technical jargon                        │
│  ✓  Focus on user experience and outcomes                          │
│  ✓  Include specific details (file types, size limits, text shown) │
│  ✓  Think in terms of "when X happens, do Y"                       │
│                                                                    │
│  The question to ask yourself:                                     │
│  "If I was explaining this feature to a friend who isn't technical,│
│   could they understand what I want the app to do?"                │
│                                                                    │
│  If yes → good prompt. If no → too technical.                      │
└────────────────────────────────────────────────────────────────────┘
```

This is the make-or-break skill for vibe coding. Clarity beats technical accuracy every time.

The Flow of Building

Here's the process you'll follow in Chapter 4 and beyond:

1. Describe the feature in plain English "I want users to be able to upload a profile picture"

2. AI asks clarifying questions (or you anticipate them) "What file types? Size limit? What happens if they don't upload one?"

3. You refine your description "Allow JPG and PNG up to 5MB. Show a default avatar if they haven't uploaded one."

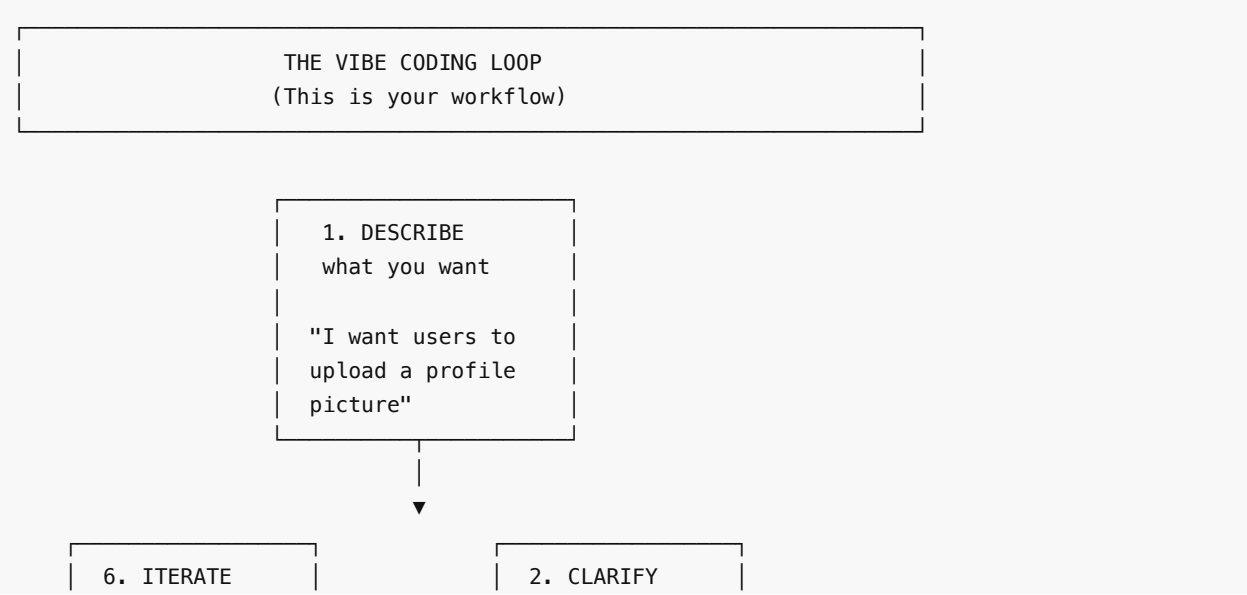
4. AI generates the code Creates front-end upload form, back-end file handling, storage configuration

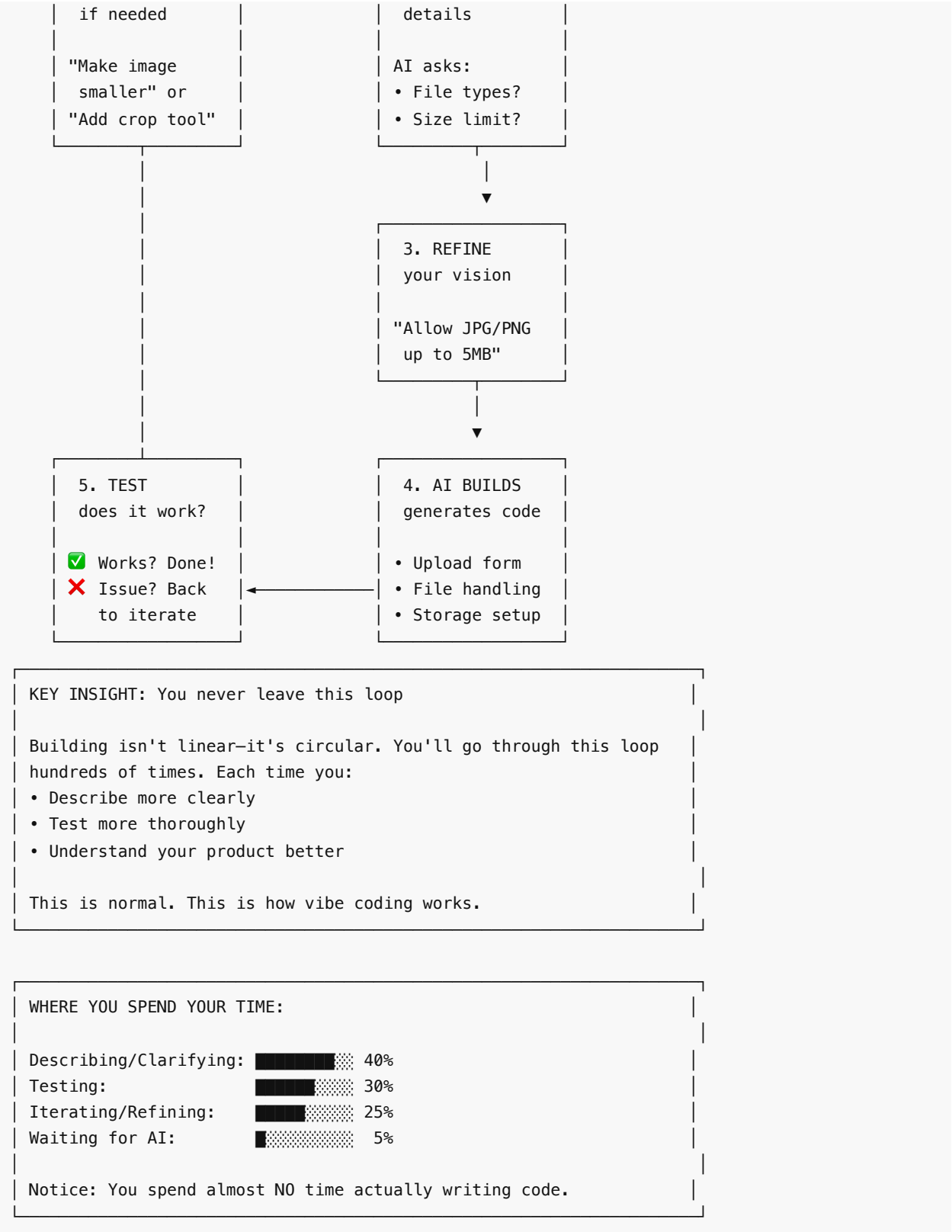
5. You test it Try uploading an image - does it work?

6. You iterate if needed "The image is too big on the page, make it smaller" "Add a crop feature before uploading"

Notice: You never wrote code. You described, tested, refined. That's vibe coding.

Visual: The Vibe Coding Loop





When you hit frustration later ("why am I going back and forth so much?"), remember: this IS the process. You're doing it correctly.

Common Misconceptions

"I need to learn the basics first"

Wrong. The basics AI needs and the basics humans need are different. You need to understand shapes and flows. AI needs to understand syntax and implementation.

"I should understand how it works under the hood"

Wrong. You should understand what it does, not how it does it. You know what a login does (verifies identity). You don't need to know how password hashing works.

"Real developers would laugh at my code"

Wrong. AI writes better code than most developers. And besides - if it works and solves the problem, it's good enough.

"I need to start with simple projects"

Wrong. Start with projects you care about. Motivation matters more than simplicity. AI handles complexity for you.

What This Means for Building

When you get to Chapter 4 and start building, you'll be:

Describing shapes: "A bookmark has a URL, title, and save date" **Describing flows:** "When user clicks save, store it and show success" **Describing connections:** "Each bookmark belongs to the user who saved it" **Describing states:** "Bookmarks can be public or private"

You'll **never** be:

- Writing `function handleSubmit() {`
- Debugging `Cannot read property of undefined`
- Worrying about `async/await` syntax
- Configuring webpack or babel

That's all AI's job.

Your job is being clear about what you want. And as a domain expert, you're already great at that - you know exactly what problem you're solving and how it should work.

Ready to choose your tools and stack? Let's make some decisions.

Connect & Share

💌 **Newsletter**: Build to Launch - Weekly AI building tips, templates, and real builder stories

🦋 **Bluesky**: @jenny-ouyang - Daily insights

💼 **LinkedIn**: Jenny Ouyang - Professional network

Chapter 3: Choosing Your Stack

> *"The best tools are the ones you'll actually use. The best stack is the one you can ship with."*

The Decision Paradox

There are hundreds of programming languages, frameworks, databases, and hosting platforms. Developers spend weeks researching "the best stack." This is decision paralysis disguised as due diligence.

Here's what actually matters: picking something that works and starting to build.

This chapter gives you three clear paths. Pick the one that matches your project type, follow it, and start building. You can always change later (and that's fine).

The Three Paths

Path 1: Simple Website or Landing Page

Choose this if: No user accounts, no database, just content **Examples:** Portfolio, marketing site, documentation, blog

Path 2: Web App with User Accounts

Choose this if: Users sign up, save data, interact with features **Examples:** SaaS tools, community platforms, booking systems

Path 3: Data-Heavy Python App

Choose this if: Processing data, analysis, ML, scientific computing **Examples:** Data dashboards, analysis tools, automation scripts

90% of people reading this guide need Path 2. If you're not sure, that's your path.

Path 1: Simple Website / Landing Page

The Stack

- **Front-end:** React + Tailwind CSS
- **Hosting:** Netlify or Vercel (free)
- **Domain:** Namecheap ($12/year)

What This Gets You

- Lightning-fast static website
- Perfect for SEO
- Free hosting with generous limits
- Easy to update and maintain
- No backend complexity

When to Choose This

- Building a portfolio or personal site
- Creating a landing page for your product

- Publishing documentation or guides
- Setting up a blog (with tools like MDX)
- Anything where content is the same for everyone

What You Can't Do (Yet)

- User accounts or login
- Saving user-generated content
- Processing payments
- Personalized experiences
- Any kind of database

Tools You'll Use

AI Assistant: Claude or ChatGPT **Code Editor:** Cursor **Version Control:** GitHub **Hosting:** Netlify (recommended for beginners)

Time to First Deploy

With AI: 2-4 hours to have something live

Cost Breakdown

- Hosting: $0 (Netlify or Vercel free tier)
- Domain: $10-15/year (varies by extension)
- Tools: $0 (all free) **Total: ~$12/year**

Example Prompts

"Create a personal portfolio site with an about section, project showcase, and contact form" "Build a landing page for my newsletter with email signup and feature highlights" "Make a documentation site with a sidebar navigation and search"

Path 2: Web App with User Accounts ⭐ RECOMMENDED

The Stack

- **Framework:** Next.js (React-based)
- **Database:** Supabase (PostgreSQL)
- **Authentication:** Supabase Auth
- **Styling:** Tailwind CSS + shadcn/ui
- **Hosting:** Vercel (Next.js is from Vercel)

What This Gets You

- User signup and login (email or Google/GitHub)
- Database to store anything you want
- Real-time features (live updates without refresh)
- File storage for images/documents
- Production-ready from day one
- Scales automatically as you grow

When to Choose This

- Building a SaaS product
- Creating a community platform
- Making internal tools for your team
- Anything with user accounts and data
- Most web applications

This is the path we'll use in Chapter 4's tutorial.

What You Can Do

✅ User accounts (email + password) ✅ Social login (Google, GitHub, etc.) ✅ Save and retrieve data ✅ File uploads (images, PDFs, etc.) ✅ Real-time features (chat, notifications) ✅ Payments (add Stripe) ✅ Email (add Resend or SendGrid) ✅ Everything you need for a real product

Tools You'll Use

AI Assistant: Claude or ChatGPT **Code Editor:** Cursor **Framework:** Next.js (latest stable version with App Router) **Database:** Supabase (free tier: 500MB, up to 2 active projects) **Auth:** Supabase Auth (included) **UI Components:** shadcn/ui (copy/paste components) **Hosting:** Vercel (free tier: 100GB bandwidth/month) **Version Control:** GitHub

Time to First Deploy

With AI: 4-8 hours for a working app with login and database

Cost Breakdown (Starting Out)

- Next.js: $0 (free framework)
- Supabase: $0 (free tier: 500MB database, 2 active projects)
- Vercel: $0 (free tier: 100GB bandwidth, unlimited projects)
- Domain: $10-15/year (varies by extension, e.g., .com) **Total: ~$12/year**

Cost Breakdown (With Growth)

- Supabase Pro: $25/month (8GB database, auto-scaling, daily backups)
- Vercel: $0 (hobby tier is generous; Pro at $20/month if needed)
- Domain: $10-15/year **Total: ~$300-350/year** (only when you exceed free tier limits)

Why This Stack

Next.js: Modern, well-documented, huge community, AI knows it extremely well

Supabase: Handles database, auth, file storage, real-time in one place. You don't need to configure multiple services.

Vercel: Made by the Next.js team, deployment is literally one command. Auto-scales, global CDN, zero config.

Tailwind: Utility-first CSS that AI can write perfectly. No wrestling with CSS files.

shadcn/ui: Beautiful pre-built components you can copy/paste and customize. No package installation, just copy the code.

Example Prompts

"Create a Next.js app with Supabase where users can sign up, create bookmarks, and see their saved list" "Add Google authentication so users can sign in with their Google account" "Let users upload a profile picture that gets stored in Supabase storage" "Create a dashboard showing user statistics from the database"

Path 3: Data-Heavy Python App

The Stack

- **Back-end:** Flask (Python)
- **Database:** PostgreSQL on Railway
- **Hosting:** Railway or Oracle Cloud
- **Front-end:** Keep it simple (templates) or separate React app

What This Gets You

- Full Python ecosystem for data processing
- Use pandas, NumPy, scikit-learn, etc.
- Custom API endpoints
- More control over backend logic
- Cost-effective at scale

When to Choose This

- Processing large datasets
- Building data analysis tools
- Creating ML/AI features
- Automating workflows with Python
- Backend APIs for mobile apps
- Team already knows Python well

What You Can Do

✅ Complex data processing ✅ Integration with Python libraries ✅ Custom business logic ✅ API for mobile or separate frontend ✅ Background jobs and scheduled tasks ✅ Data analysis and visualization

Tools You'll Use

AI Assistant: Claude or ChatGPT **Code Editor:** Cursor **Language:** Python (latest stable version, 3.11+) **Framework:** Flask (lightweight) or FastAPI (modern) **Database:** PostgreSQL **Hosting:** Railway (easiest) or Oracle Cloud (free tier) **Version Control:** GitHub

Time to First Deploy

With AI: 6-10 hours for a working API

Cost Breakdown (Starting Out)

- Railway: $5 free trial credit (30 days), then usage-based
- Railway ongoing: ~$5-10/month (includes database + hosting)
- Oracle Cloud: $0 (free tier available, good for learning)
- Domain: $10-15/year **Total: ~$60-135/year** (depending on hosting choice)

Why This Stack

Python: You probably already know it or want to learn it. Huge ecosystem for data work.

Flask: Simple, minimal, gets out of your way. Easy for AI to generate.

PostgreSQL: Industry standard, reliable, works everywhere.

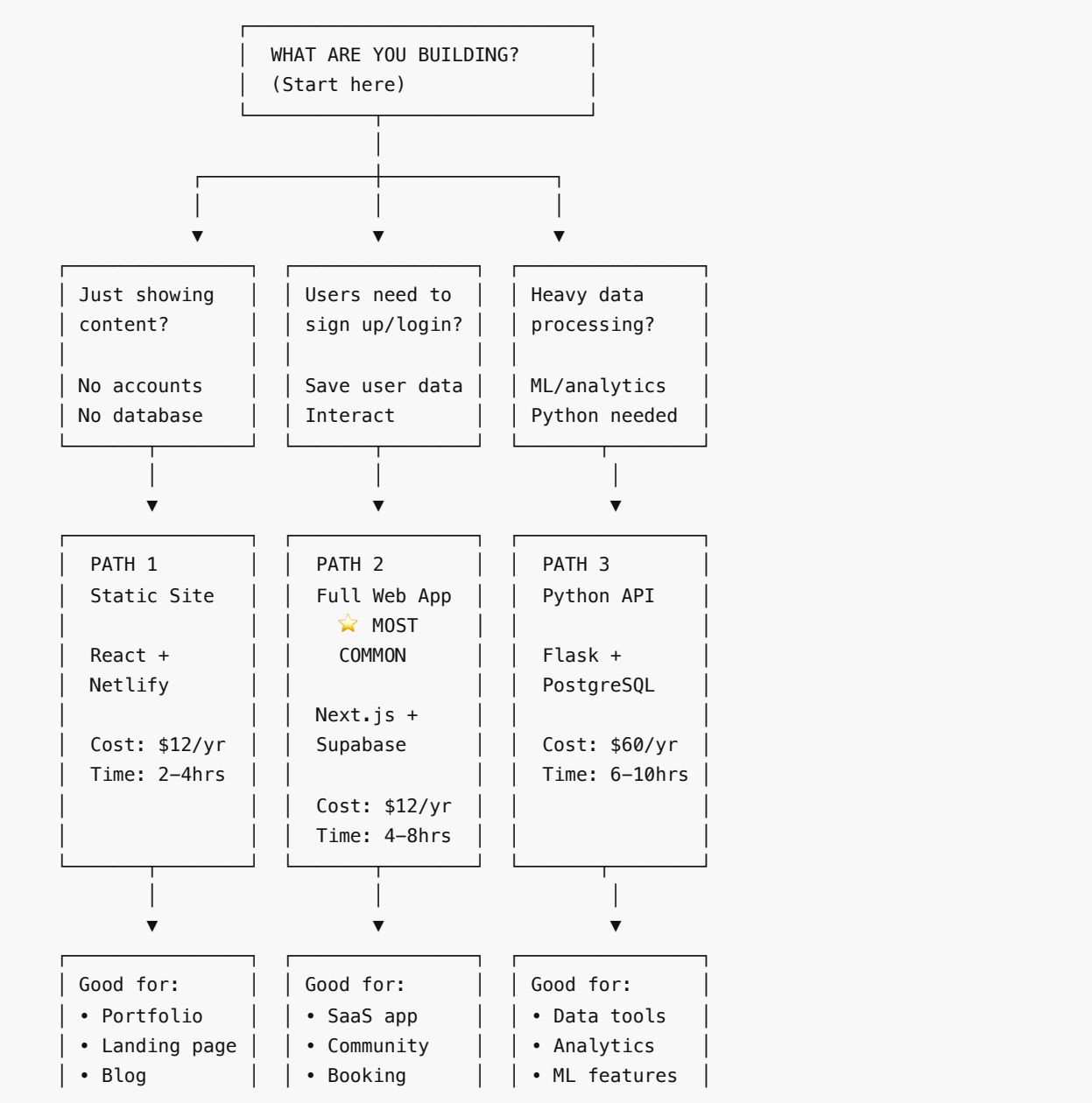
Railway: Dead simple deployment for Python apps. No DevOps knowledge required.

Example Prompts

"Create a Flask API that accepts CSV uploads, processes them with pandas, and returns summary statistics" "Build an endpoint that runs a scheduled job every day at midnight to process data" "Make a dashboard that queries the database and shows analytics charts" "Create API endpoints for a mobile app to fetch and update user data"

Decision Tree (Visual Guide)

Start here: What are you building?



```
    | • Docs          |   | • Marketplace   |   | • Automation    |
    └─────────────────┘   └─────────────────┘   └─────────────────┘


┌──────────────────────────────────────────────────────────────────┐
| 🤔 STILL NOT SURE?                                               |
|                                                                  |
| → Choose PATH 2 (Next.js + Supabase)                             |
|                                                                  |
| Why? It's flexible enough for most projects, and you can always  |
| simplify to Path 1 or add Path 3 backend later if needed.        |
└──────────────────────────────────────────────────────────────────┘
```

This makes the decision quick and visual. You should be able to identify your path in under 30 seconds.

Just content (no user interaction)?

→ **Path 1: Static Site** (React + Netlify)

- Portfolio
- Marketing page
- Blog
- Documentation

Web app with users?

→ **Path 2: Next.js + Supabase** ⭐

- SaaS product
- Community platform
- Booking system
- Content management
- Most web applications

Heavy data processing?

→ **Path 3: Flask + PostgreSQL**

- Data analysis
- ML features
- Automation
- Custom APIs

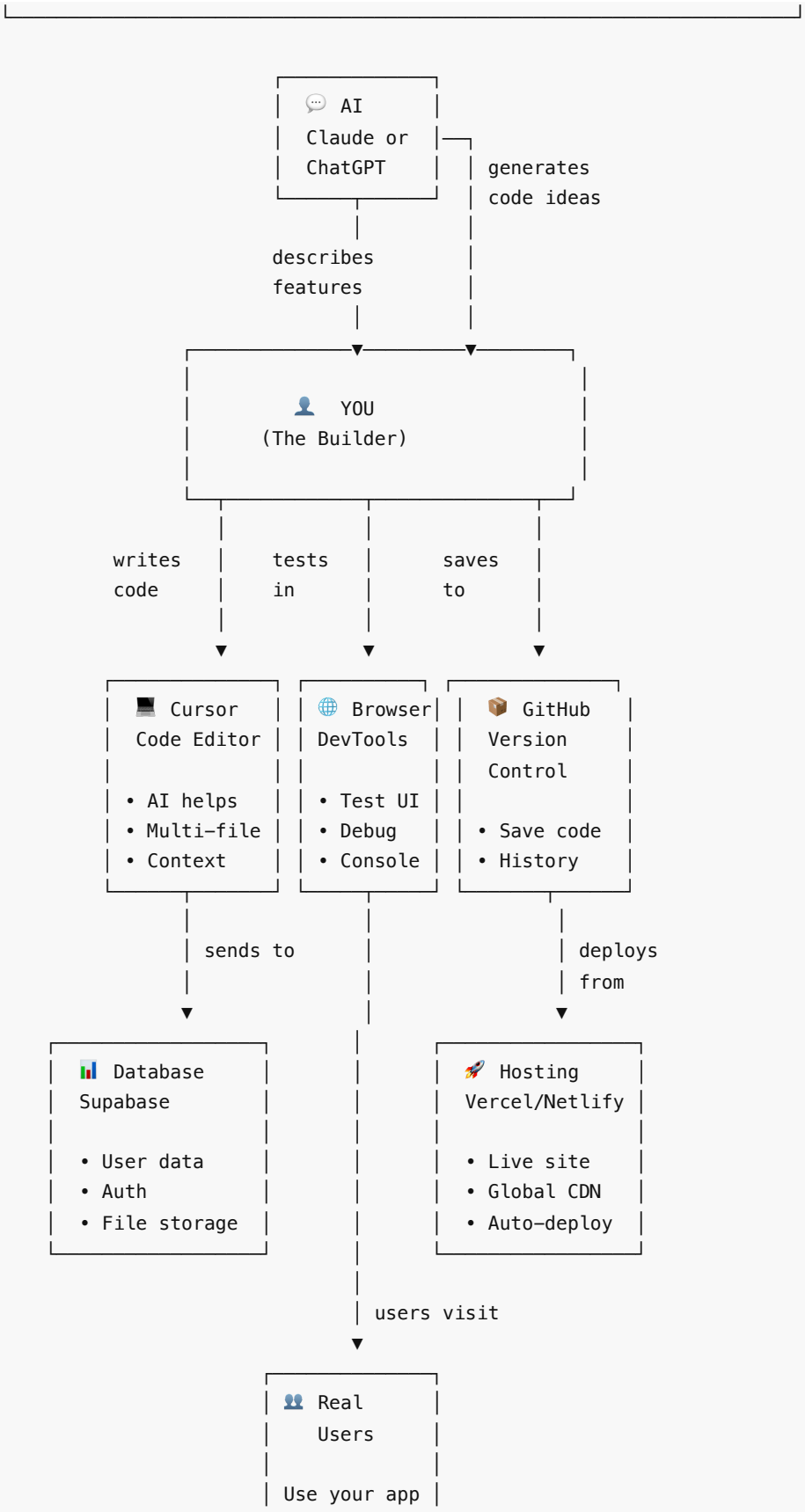
Still unsure? Choose Path 2. It's the most flexible and you can always add Python backend later if needed.

The Tools You'll Actually Use

Regardless of which path you choose, these tools are essential. Here's how they all work together:

Visual: Your Building Ecosystem

```
┌──────────────────────────────────────────────────────────────────┐
|                    YOUR BUILDING ECOSYSTEM                       |
|              (How all the tools work together)                   |
```


AI
Claude or
ChatGPT
generates
code ideas
describes
features
YOU
(The Builder)
writes
code
tests
in
saves
to
Cursor
Code Editor
• AI helps
• Multi-file
• Context
Browser
DevTools
• Test UI
• Debug
• Console
GitHub
Version
Control
• Save code
• History
sends to
deploys
from
Database
Supabase
• User data
• Auth
• File storage
Hosting
Vercel/Netlify
• Live site
• Global CDN
• Auto-deploy
users visit
Real
Users
Use your app

```
                    └──────────────┘


┌──────────────────────────────────────────────────────────────────┐
│  THE COMPLETE WORKFLOW:                                          │
│                                                                  │
│  1. You describe feature to AI (Claude/ChatGPT)                  │
│  2. AI suggests code, you implement in Cursor                    │
│  3. You test in Browser DevTools                                 │
│  4. You save to GitHub                                           │
│  5. GitHub auto-deploys to Vercel/Netlify                        │
│  6. Your database (Supabase) stores user data                    │
│  7. Real users access your live app                              │
│                                                                  │
│  You manage this entire flow, but you're NOT coding from scratch.  │
└──────────────────────────────────────────────────────────────────┘
```

Each tool has ONE job. You orchestrate them, not master them.

1. AI Assistant (Required)

Claude (Anthropic) or **ChatGPT (OpenAI)**

Pick one. I recommend Claude because:

- Better at understanding context
- More helpful with debugging
- Stronger at code generation
- (You're already reading this in Claude!)

Pricing (verified December 2025):

- Claude Pro: $20/month (or $18/month billed annually at $216/year)
- ChatGPT Plus: $20/month (monthly billing only)
- Free tiers available for both (limited usage)

Starting tip: Use free tiers first to learn, then upgrade when you hit limits. Worth upgrading? Absolutely. The paid tier is your co-developer with 5x more capacity.

2. Code Editor (Required)

Cursor (AI-powered code editor)

Why Cursor specifically:

- Built-in AI that understands your entire codebase
- Can make changes across multiple files
- Accepts natural language commands

Pricing (verified December 2025):

- Free tier: ~2,000 completions + 50 premium requests/month, plus 2-week Pro trial
- Pro: $20/month (includes $20 of API usage credits)
- Most users stay on free tier for learning

Starting tip: Start with the free tier + 2-week trial. Upgrade to Pro when you're building daily. Alternative: VS Code with GitHub Copilot (but Cursor is better for beginners)

3. Version Control (Required)

GitHub

This is how you:

- Save your code safely
- Deploy to hosting platforms
- Share with others
- Track changes over time

Cost: Free Learn: Just the basics (git add, commit, push)

4. Browser Dev Tools (Built-in)

Chrome DevTools or **Firefox Developer Tools**

This is how you:

- See what went wrong
- Test your app
- Inspect elements
- Check network requests

Cost: Free (built into browser) Learn: Just how to open it (F12 or Cmd+Option+I)

What About Other Tools You've Heard Of?

"Should I use TypeScript?"

For beginners: No. JavaScript is fine. AI can write either. **When to add it:** Once you're comfortable, TypeScript adds safety. But start with JavaScript.

"What about Docker?"

For beginners: You don't need it. Modern hosting handles this. **When you need it:** For complex deployments or team collaboration. Not day one.

"Should I learn Git before starting?"

Minimum needed: `git add .`, `git commit -m "message"`, `git push` **That's it.** AI can help with the rest when you need it.

"What about testing?"

For beginners: Ship first, test later. **When to add it:** After you have users and need to maintain quality. Not before you ship.

"MongoDB vs PostgreSQL?"

For beginners: PostgreSQL (what Supabase uses). **Why:** More predictable, better tooling, easier to understand relationships.

Common Questions

"What if I pick the wrong stack?"

You won't. These are all good choices. And if you really need to change later, AI can help you migrate. But usually you don't need to.

Reality check: Most successful startups launched with whatever they knew and migrated years later (if ever).

"Can I mix and match?"

Yes! Common patterns:

- Next.js frontend + Flask backend (Path 2 + 3)
- Static site now, add backend later (Path 1 → Path 2)
- Start simple, add complexity as needed

But for your first project: **pick one path and stick with it until you ship.**

"What about mobile apps?"

For beginners: Build a web app first (Path 2). Make it responsive. Most "apps" are really just websites.

When you need native: Use React Native with the same Next.js backend. But not on day one.

"What's the 'best' stack?"

The one you ship with. Seriously. The best stack is:

- The one you understand
- The one AI can help you with
- The one that lets you move fast
- The one you actually finish

Perfect choice paralysis beats imperfect shipped product every time. Choose and move.

Complete Pricing Breakdown (Verified December 2025)

Understanding real costs helps you plan. Here's everything broken down clearly:

Tool Costs (One-time per person, regardless of project)

AI Assistant:

- Claude Pro: $20/month or $18/month annually ($216/year)
- ChatGPT Plus: $20/month
- Free tier: Available for both (limited usage, good for learning)

Code Editor:

- Cursor Free: $0 (~2,000 completions + 50 premium requests/month)

- Cursor Pro: $20/month
- 2-week free Pro trial included

Version Control:

- GitHub: $0 (free for individuals and open source)

Infrastructure Costs (Per project, scales with usage)

Path 1: Static Site

- Hosting (Netlify/Vercel): $0 (free tier covers most use cases)
- Domain: $10-15/year
- **Total: ~$12/year**

Path 2: Web App with Database (Most Common)

Starting out (0-100 users):

- Supabase: $0 (free tier: 500MB database, 2 active projects)
- Vercel: $0 (free tier: 100GB bandwidth, unlimited projects)
- Domain: $10-15/year
- **Total: ~$12/year**

With growth (100-10,000+ users):

- Supabase Pro: $25/month (8GB database, auto-scaling, daily backups)
- Vercel: $0 (hobby tier sufficient) or $20/month (Pro if needed)
- Domain: $10-15/year
- **Total: ~$300-550/year**

Path 3: Python Data App

- Railway: $5 trial credit (30 days), then ~$5-10/month
- Oracle Cloud: $0 (free tier alternative)
- Domain: $10-15/year
- **Total: ~$60-135/year** (Railway) or ~$12/year (Oracle free tier)

Add-On Services (When You Need Them)

Email (Transactional):

- Resend: $0 (free tier: 3,000 emails/month), Pro at $20/month (50,000 emails)
- SendGrid: Similar pricing structure

Payments:

- Stripe: 2.9% + $0.30 per transaction (no monthly fee)
- $15 chargeback fee

File Storage (if beyond Supabase):

- Included in Supabase free tier (1GB)
- Supabase Pro includes 100GB

The $0 Path to Start

You can genuinely start with $0 and only pay for a domain (~$12) when you're ready to go live:

Free tier everything:

- Claude or ChatGPT (free tier for learning)
- Cursor (free tier: 2,000 completions/month)
- GitHub (free)
- Vercel or Netlify hosting (free)
- Supabase (free tier: 500MB)
- Use a free subdomain (yourapp.vercel.app) - skip buying domain until validated

When to upgrade:

- AI Assistant: When you hit free tier limits (usually after 2-3 weeks of daily use)
- Cursor: When free completions run out (if building intensively)
- Supabase: When you exceed 500MB database or need more than 2 projects
- Domain: When you want professional branding (can start with free subdomain)

Realistic Monthly Costs

Learning phase (Month 1-2):

- $0-20 (can use all free tiers)

Building phase (Month 3-6):

- $20-40 (Claude/ChatGPT Pro, maybe Cursor Pro, domain)

Launched with users (Month 6+):

- $40-65 (AI tools + infrastructure + domain, only if you exceed free tiers)

Important: These are maximums. Many successful apps run on free tiers for months or even years.

My Recommendation for This Guide

We're going to use **Path 2** (Next.js + Supabase) in Chapter 4 because:

1. It's what 90% of you need
2. It scales from MVP to millions of users
3. Free to start, grows with you
4. One service handles database + auth + storage
5. AI tools know it extremely well
6. Huge community = easy to find help

If you need Path 1 or 3, the principles in Chapter 4 still apply - just different tools. The mental models are the same.

Ready to build something real? Chapter 4 is where we get our hands dirty.

Connect & Share

💌 **Newsletter**: Build to Launch - Weekly AI building tips, templates, and real builder stories

🦋 **Bluesky**: @jenny-ouyang - Daily insights

💼 **LinkedIn**: Jenny Ouyang - Professional network

Chapter 4: Your First AI-Built App in One Hour

What you're about to do: Describe what you want in one comprehensive prompt. Watch AI build it. Deploy it live. That's it.

Time: 60 minutes total

- Setup: 10 minutes
- Understanding the prompt: 5 minutes
- Building: 30-40 minutes
- Deploy: 5 minutes

What you'll build: A personal bookmark manager that saves links, adds descriptions, and organizes them with tags.

The shift: You're not following 28 steps. You're describing what you want once, then letting AI figure out the how.

Part 1: Setup (10 Minutes)

You need three accounts and one download. That's it.

Create These Accounts:

1. **Supabase** (supabase.com) - Your database
2. **Vercel** (vercel.com) - Your hosting
3. **Cursor** (cursor.sh) - Your AI coding tool

All free tiers. Sign up with GitHub for fastest setup.

Install Cursor:

Download from cursor.sh and install. It looks like VS Code because it's built on top of it.

That's your setup. No environment variables yet. No configuration files. We'll handle those when AI asks for them.

Part 2: The Master Prompt (5 Minutes)

Here's what most people don't understand: **One comprehensive prompt can replace 28 step-by-step instructions.**

The difference is specificity. Not "build a bookmark app." Not even "build a bookmark app with tags and search."

You describe:

- What it does
- What the user sees
- How data flows
- What tech to use
- What constraints to follow

The Bookmark Manager Master Prompt

Copy this entire prompt. This is what you'll paste into Cursor:

```
GOAL: Build a personal bookmark manager web app

WHAT IT DOES:
- Save website URLs with descriptions
- Add multiple tags to each bookmark
- View all bookmarks in a clean list
- Search bookmarks by title, description, or tags
- Click a bookmark to open in new tab
- Delete bookmarks I no longer need

USER EXPERIENCE:
- Landing page with input form at top (URL field, description field, tag input)
- Below form: list of all saved bookmarks
- Each bookmark shows: title (from URL), description, tags as pills, timestamp
- Click bookmark title to visit site
- Each bookmark has a delete button
- Search bar filters bookmarks in real-time
- Clean, minimal design (like Notion or Linear)

TECHNICAL REQUIREMENTS:
Stack:
- Next.js 14 with App Router
- TypeScript
- Tailwind CSS for styling
- Supabase for database

Database Schema:
- Bookmarks table: id, url, title, description, tags (array), created_at
- Use Supabase's built-in auth (optional for v1, but set up the structure)

Features:
- Auto-fetch page title from URL when saving
- Tag input with autocomplete from existing tags
- Responsive design (works on mobile)
- Loading states for all actions
- Error handling with user-friendly messages

CODING CONSTRAINTS - FOLLOW THESE RULES:

BEFORE YOU CODE:
1. Summarize what you understand about the requirement
2. Ask specific questions about any uncertainties
3. List files you plan to create and why
4. Get confirmation before proceeding

WHILE CODING:
1. Make ONLY the changes needed for the specific requirement
2. Do NOT add features, enhancements, or "improvements" not requested
3. Do NOT refactor or reorganize code while building
4. If you modify the same file 3+ times, STOP and explain what you're trying to do
```

```
WHEN COMPLETE:
1. Verify the specific requirement works
2. Test all user actions (add bookmark, search, delete)
3. Provide clear deployment instructions

Start by confirming you understand this project, then ask any clarifying questions before
generating code.
```

What Makes This Prompt Work

Goal clarity: First line states exactly what we're building

User perspective: Describes the experience, not just features

Technical specificity: Exact stack, schema, and requirements

Constraints included: The coding constraints prevent AI from over-building

Control maintained: AI must confirm understanding before coding

This is the formula. Goal → Experience → Technical → Constraints → Confirmation.

Part 3: Building (30-40 Minutes)

Step 1: Start the Conversation

1. Open Cursor
2. Create a new folder: `bookmark-manager`
3. Open that folder in Cursor
4. Open the chat panel (Cmd+L or Ctrl+L)
5. Paste the entire master prompt

What happens next: AI will read everything, then respond with:

- A summary of what it understood
- Clarifying questions (maybe 2-5)
- A list of files it plans to create

Step 2: Answer Questions

AI might ask:

- "Should the tag autocomplete be case-sensitive?"
- "Do you want tags to be created on-the-fly or pre-defined?"
- "Should there be a limit on number of tags per bookmark?"

Your job: Answer honestly. Don't overthink it. "Case-insensitive is fine," "On-the-fly," "No limit needed."

If you don't know: "Your call, make it user-friendly."

Step 3: Give Permission to Build

Once questions are answered, AI will show you a file plan:

```
I'll create:
- app/page.tsx (main UI)
- app/api/bookmarks/route.ts (API endpoints)
- lib/supabase.ts (database connection)
- components/BookmarkForm.tsx
- components/BookmarkList.tsx
- components/SearchBar.tsx
```

You type: **"Looks good, proceed."**

Step 4: Watch It Build

AI will generate all files. You'll see them appear in your file tree.

Don't panic. You don't need to read every line right now. The point is to see it work first.

Step 5: Set Up Supabase

Cursor will hit a pause point: "I need your Supabase credentials."

1. Go to your Supabase project
2. Settings → API → Copy these two values:
   - `Project URL`
   - `anon/public key`
3. Create `.env.local` in your project root
4. Paste this template with your actual values:

```
# Supabase Configuration
NEXT_PUBLIC_SUPABASE_URL=https://your-project.supabase.co
NEXT_PUBLIC_SUPABASE_ANON_KEY=your-anon-key-here

# IMPORTANT:
# - Never commit .env.local to git (it's in .gitignore by default)
# - Use different values for production (add in Vercel dashboard)
# - Restart dev server after adding/changing variables: Ctrl+C, then npm run dev
```

Tell Cursor: "Added the environment variables."

Step 6: Create the Database Table

AI will give you SQL to run. Copy it.

1. Go to Supabase → SQL Editor
2. Paste the SQL
3. Run it
4. Tell Cursor: "Table created."

Step 7: Run It Locally

Cursor will tell you to run:

```
npm install
npm run dev
```

Do it. Open `localhost:3000` in your browser.

Step 8: Test Everything

Try these actions:

- Add a bookmark with a URL
- Add some tags
- Add another bookmark
- Use the search
- Delete a bookmark

If something breaks: Copy the error, paste it in Cursor chat, say "This error appeared, fix it."

This is the magic moment. You described what you wanted 30 minutes ago. Now you're using it.

Part 4: Deploy It Live (5 Minutes)

Make It Public

1. Push to GitHub:

```
git init
git add .
git commit -m "Built with AI"
git remote add origin YOUR_GITHUB_REPO_URL
git push -u origin main
```

2. Go to Vercel → New Project → Import your GitHub repo
3. Vercel will detect Next.js automatically
4. Add your environment variables in Vercel:
    - `NEXT_PUBLIC_SUPABASE_URL`
    - `NEXT_PUBLIC_SUPABASE_ANON_KEY`
5. Click Deploy

In 2-3 minutes: You have a live URL. Send it to friends. Use it on your phone. It's real.

Part 5: What You Just Did

Let's reflect on what actually happened here.

The Old Way vs The New Way

Old way (traditional coding):

1. Learn React
2. Learn Next.js
3. Learn Tailwind
4. Learn Supabase
5. Then build

New way (what you just did):

1. Describe what you want
2. AI builds it
3. You learn by using it

You Didn't Write Code, But You Built Software

What you actually did:

- Defined requirements clearly
- Made architecture decisions (Next.js, Supabase, Tailwind)
- Answered design questions (how tags work, what UI looks like)
- Tested user experience
- Debugged errors
- Deployed to production

That's product building. The code was the implementation detail.

Understanding What AI Built

You don't need to read every line, but you should understand the structure:

app/page.tsx: Your main UI - the form and bookmark list **app/api/bookmarks/route.ts:** Handles save/delete actions **lib/supabase.ts:** Connects to your database **components/:** Reusable UI pieces

Open `app/page.tsx`. You'll recognize things: the form fields, the button, the list. It's React, but you can read it.

The confidence shift: You went from "I don't know React" to "I can see how this React app works."

How to Modify It

Want to add a feature? Use the same approach:

Prompt:

```
Add a feature: When I save a bookmark, automatically fetch and save
the page's favicon (small icon). Display it next to each bookmark
in the list.

Follow the same coding constraints as before. Ask questions if unclear.
```

AI will ask questions, explain what files to modify, make changes.

You're not coding. You're directing.

What This Chapter Actually Taught You

The One Prompt Method

Not: "Build this, then do this, then do that, then..." **But:** "Here's everything I want. Ask questions. Build it."

This is the shift. Comprehensive prompts replace step-by-step tutorials.

The Builder Mindset

You just:

- Defined a product
- Made tech choices
- Managed AI as a developer
- Tested user experience
- Shipped to production

That's the full cycle. You're not "learning to code." You're building products with AI as your implementation partner.

The Confidence You Gained

Before this chapter: "I need to learn Next.js and React and Supabase and..."

After this chapter: "I can describe what I want and get a working app."

That's not arrogance. That's reality. You just proved it to yourself.

Next Steps

Modify Your Bookmark App

Try these changes (each is one prompt):

1. **Add folders:** Organize bookmarks into folders/categories
2. **Add export:** Button to export all bookmarks as CSV
3. **Add sharing:** Generate a public link to share specific bookmarks
4. **Add dark mode:** Toggle between light and dark themes

Each one is practice. Each one builds confidence.

Try a Different App

Use the same master prompt structure for:

- **Expense tracker:** Log expenses, categorize, view totals
- **Habit tracker:** Check off daily habits, see streak counts
- **Recipe saver:** Save recipes with ingredients and instructions

Same pattern:

- Goal
- User experience
- Technical requirements
- Coding constraints
- Build it

Understand the Tools Deeper

Now that you've used the stack:

- Read Next.js docs (you'll understand them now)
- Explore Supabase features (realtime, auth, storage)
- Learn Tailwind patterns (you've seen them in your code)

The difference: You're learning by extending something real, not building todo apps from tutorials.

The Unlock

This chapter was different from traditional coding tutorials on purpose.

Traditional tutorial: "Here's 28 steps. Follow them exactly. Type this. Type that. Now you have an app."

What we just did: "Here's a comprehensive prompt. Describe what you want. Let AI build it. Now you have an app."

The result is the same: Working software in production.

The feeling is different: You feel capable, not hand-held.

That's the confidence shift. That's vibe coding.

You didn't learn syntax. You learned to build.

Connect & Share

💌 **Newsletter**: Build to Launch - Weekly AI building tips, templates, and real builder stories

🦋 **Bluesky**: @jenny-ouyang - Daily insights

💼 **LinkedIn**: Jenny Ouyang - Professional network

Chapter 5: Common Building Blocks

> *"Most apps are built from the same set of pieces. Learn these once, use them everywhere."*

This chapter is your reference guide. You don't need to read it start to finish - come back when you need a specific feature. Each section tells you what it is, when you need it, and exactly what to tell AI.

Authentication (User Accounts)

What It Is

The system that lets users sign up, log in, and stay logged in. Includes:

- Creating accounts (registration)
- Verifying identity (login)
- Staying logged in (sessions)
- Resetting passwords
- Social login (Google, GitHub, etc.)

When You Need It

You need authentication if:

- Users have personalized data
- Users create content
- You need to know who's doing what
- Different users see different things
- You have any concept of "my stuff"

You don't need authentication if:

- Same content for everyone
- No user-specific features
- Landing pages, blogs, portfolios
- Pure content sites

How to Describe It to AI

Basic setup:

```
Set up authentication using Supabase Auth where users can:
- Sign up with email and password
- Log in with email and password
- Stay logged in even after closing the browser
- Log out when they want
```

Social login:

```
Add Google authentication so users can sign up/log in with their Google account. Configure the OAuth flow with Supabase.
```

Password reset:

```
Add password reset functionality where:
- Users can request a reset email
- They click a link in the email
- They enter a new password
- They're automatically logged in
```

Protected routes:

```
Protect the /dashboard route so only logged-in users can access it. Redirect non-authenticated users to /login.
```

Common Patterns

Email verification: "Require users to verify their email before they can use the app. Send a confirmation email on signup."

Remember me: "Add a 'Remember me' checkbox that keeps users logged in for 30 days instead of the default session duration."

Profile completion: "After users sign up, redirect them to a profile setup page where they add their name and profile picture before accessing the app."

Session management: "Check if the user's session is still valid on page load. If expired, log them out and redirect to login."

What AI Handles for You

✅ Password hashing (bcrypt) ✅ Session token generation ✅ CSRF protection ✅ OAuth flows ✅ Secure cookie management ✅ Token refresh logic

You describe: "Users should be able to log in" AI implements: All the security best practices

---

Databases (Storing Data)

What It Is

Where all your app's data lives permanently. Think of it as organized filing cabinets where each cabinet is a "table" and each file is a "row."

Key concepts:

- **Tables:** Collections of similar data (users table, posts table)
- **Columns:** Properties of each item (email, name, created_at)
- **Rows:** Individual items (one row = one user)
- **Relationships:** How data connects (post belongs to user)

When You Need It

You need a database if:

- Data needs to persist (survive page refresh)
- Multiple users have their own data
- You're creating, reading, updating, or deleting anything

- Users need to save their work

You don't need a database if:

- All data can be calculated on the fly
- Nothing needs to be saved
- Pure display/presentation sites

How to Describe It to AI

Creating tables:

```
Create a Supabase table called "recipes" with these columns:
- id (auto-generated UUID)
- user_id (links to auth.users)
- title (text, required)
- ingredients (text array)
- instructions (long text)
- prep_time (integer, minutes)
- is_public (boolean, default false)
- created_at (timestamp, auto)
```

Relationships:

```
Create a "comments" table where:
- Each comment belongs to one post (foreign key to posts.id)
- Each comment belongs to one user (foreign key to auth.users.id)
- Store the comment text and timestamp
```

Querying data:

```
Fetch all recipes where:
- user_id matches the current user
- OR is_public is true
Order by created_at descending
Limit to 20 results
```

Updating data:

```
When user clicks "Make Public", update that recipe's is_public field to true in the database.
```

Deleting data:

```
Add a delete button that removes the recipe from the database and updates the UI immediately.
```

Common Patterns

Soft deletes: "Instead of actually deleting recipes, add a 'deleted_at' timestamp column. Hide rows where deleted_at is not null. This lets users undo deletion."

Pagination: "Load 20 recipes at a time. Add a 'Load More' button that fetches the next 20 using offset."

Search: "Add a search input that filters recipes by title or ingredients using Supabase's text search."

Real-time updates: "Use Supabase real-time subscriptions so when one user adds a comment, other users see it appear immediately without refreshing."

Data validation: "Before inserting a recipe, check that title is not empty and prep_time is a positive number. Show error if invalid."

What AI Handles for You

✅ SQL query syntax ✅ Indexing for performance ✅ Connection pooling ✅ Type safety ✅ Error handling ✅ Race conditions

You describe: "Store user posts with title and body" AI implements: Proper table structure with relationships and constraints

---

File Uploads (Images, PDFs, Documents)

What It Is

Letting users upload files from their device to your app. Files are stored separately from your database (usually in "blob storage" or "object storage").

Common use cases:

- Profile pictures
- Document attachments
- Product images
- Resume uploads
- CSV imports

When You Need It

You need file uploads if:

- Users share images or documents
- You have media-heavy features
- Users need to back up files
- Profile pictures, cover photos, etc.

You don't need file uploads if:

- Text-only content
- URLs to external images are fine
- No user-generated media

How to Describe It to AI

Basic upload:

```
Add a profile picture upload where:
- Users can select an image file (JPG, PNG)
- Max size 5MB
- Preview the image before uploading
```

```
- Upload to Supabase Storage in "avatars" bucket
- Save the public URL to user's profile in database
```

Multiple files:

```
Let users upload multiple images for a product listing:
- Allow selecting multiple files at once
- Show preview thumbnails
- Upload to "products" bucket with unique filenames
- Store array of URLs in products table
```

File restrictions:

```
Only allow PDF uploads with:
- Max size 10MB
- Show error if wrong file type
- Display file name and size before upload
- Show upload progress bar
```

Direct camera access:

```
On mobile, show "Take Photo" button that:
- Opens device camera
- Lets user take picture
- Uploads directly without saving to device
```

Common Patterns

Image resizing: "After upload, automatically resize images to 800px width for display and 100px width for thumbnails. Store both versions."

File validation: "Check file type on both client and server. Reject files that aren't images with a helpful error message."

Upload progress: "Show a progress bar while files upload. Display percentage complete."

Delete functionality: "Add a delete button that removes the file from storage and updates the database URL to null."

Drag and drop: "Create a dropzone where users can drag files from their desktop instead of clicking a file input."

What AI Handles for You

✅ Multipart upload handling ✅ File type validation ✅ Unique filename generation ✅ Public URL generation ✅ CORS configuration ✅ Bucket permissions

You describe: "Users can upload profile pictures" AI implements: Complete upload flow with preview and validation

---

Payments (Stripe Integration)

What It Is

Collecting money from users. Stripe is the industry standard for online payments - it handles credit cards, subscriptions, invoices, and compliance.

Common use cases:

- One-time purchases
- Monthly subscriptions
- Pay-per-use billing
- Marketplace commissions

When You Need It

You need payments if:

- You're charging money (obviously)
- Users buy products or services
- Subscription-based access
- Freemium with paid upgrades

You don't need payments if:

- Free product forever
- Monetizing through ads
- Not ready to charge yet

How to Describe It to AI

One-time payment:

```
Integrate Stripe for one-time payments:
- Add a "Buy Now" button ($29 price)
- Open Stripe Checkout when clicked
- After successful payment, save purchase to database
- Grant user access to premium features
- Send confirmation email
```

Monthly subscription:

```
Set up Stripe subscriptions:
- Create a $9/month subscription plan
- Show "Subscribe" button
- Handle Stripe Checkout
- Listen for Stripe webhooks
- Update user's subscription status in database
- Handle cancellations and failed payments
```

Usage-based billing:

```
Implement pay-per-use with Stripe:
- Track API calls per user in database
- Calculate bill at end of month
- Charge users based on usage tier
- Send invoice
```

Common Patterns

Free trial: "Offer 14-day free trial before charging. Capture card details but don't charge until trial ends. Send reminder email 3 days before charging."

Payment plans: "Let users choose between monthly ($9) and annual ($90, two months free). Show savings for annual plan."

Failed payment handling: "When payment fails, retry automatically 3 times. Email user after each failed attempt. Downgrade to free plan after 3 failures."

Upgrade/downgrade: "Let users upgrade from $9/month to $19/month plan. Prorate the difference and charge immediately."

Refunds: "Add admin interface to issue refunds. When refunded, downgrade user and send confirmation email."

What AI Handles for You

✅ Stripe API integration ✅ Checkout session creation ✅ Webhook signature verification ✅ Subscription lifecycle ✅ Payment intent handling ✅ PCI compliance setup

You describe: "Users pay $29 to unlock features" AI implements: Complete Stripe integration with webhooks and database updates

Important Note

Start without payments. Get users first, add billing later. Many successful products launched free and added payments months later once they proved value.

Email (Sending Messages)

What It Is

Programmatically sending emails from your app. Used for notifications, updates, marketing, and transactional messages.

Common use cases:

- Welcome emails
- Password reset links
- Purchase confirmations
- Weekly digests
- Team invitations

When You Need It

You need email if:

- Users need notifications
- Onboarding flows
- Password resets
- Purchase confirmations
- Any off-app communication

You don't need email if:

- In-app notifications are enough
- No user accounts
- Early MVP stage

How to Describe It to AI

Transactional email (Resend recommended):

```
Set up Resend for transactional emails:
- Install Resend SDK
- Configure with API key
- Send welcome email when user signs up
- Include: greeting, what to do next, link to dashboard
- Use plain HTML template with good styling
```

Password reset email:

```
When user requests password reset:
- Generate secure reset token
- Store token with expiry in database
- Send email with reset link containing token
- Link goes to /reset-password?token=xyz page
```

Notification email:

```
When someone comments on user's post:
- Get post author's email from database
- Send notification email
- Include: commenter name, comment text, link to post
- Add unsubscribe link at bottom
```

Common Patterns

HTML templates: "Create reusable email templates with variables for user name, link URL, etc. Use inline CSS for styling."

Bulk emails: "Send weekly digest to all users with new features. Queue emails to avoid hitting rate limits. Track open rates."

Email verification: "Send verification email with 6-digit code on signup. User must enter code before accessing app."

Drip campaigns: "After signup, send day 1 welcome, day 3 tips, day 7 case study. Track which emails user has received."

Unsubscribe: "Add unsubscribe link to all marketing emails. Store preference in database. Don't send to unsubscribed users."

What AI Handles for You

✅ SMTP configuration ✅ HTML email formatting ✅ Rate limiting ✅ Retry logic ✅ Error handling ✅ Template rendering

You describe: "Send welcome email when users sign up" AI implements: Complete email flow with proper formatting and error handling

Recommended Services

Transactional (automated emails):

- **Resend** - $20/month, 50k emails, excellent DX
- **SendGrid** - Free tier: 100 emails/day
- **Postmark** - Pay per email, great deliverability

Marketing (newsletters):

- **ConvertKit** - $29/month, designed for creators
- **Mailchimp** - Free tier: 500 contacts
- Keep it separate from app emails

Deployment (Going Live)

What It Is

Making your app accessible on the internet. Deployment platforms host your code, serve it to users, and handle scaling.

Key concepts:

- **Build:** Converting your code to optimized production files
- **Deploy:** Uploading to hosting platform
- **Environment:** Production vs development settings
- **Domain:** Custom URL (yourapp.com)

When You Need It

You deploy when:

- App is functional enough to test
- Ready for first users
- Want feedback from others
- Need a public URL

Don't wait for:

- Perfect features
- Beautiful design
- Zero bugs
- Complete functionality

Ship early, iterate publicly.

How to Describe It to AI

Initial Vercel deployment:

```
Deploy this Next.js app to Vercel:
- Connect my GitHub repository
- Set up environment variables
- Configure build settings
- Set up automatic deployments on git push
```

Custom domain:

```
Connect my custom domain (myapp.com) to Vercel:
- Add domain in Vercel settings
- Configure DNS records
- Set up SSL certificate
- Redirect www to non-www
```

Environment variables:

```
Set up different environment variables for production vs development:
- Use local database in development
- Use production database in production
- Keep API keys secure
```

Common Patterns

Preview deployments: "Every git branch gets its own preview URL. Test features before merging to main."

Rollback: "If deployment breaks something, roll back to previous version instantly."

Custom build commands: "Run database migrations before each deployment. Fail deployment if migrations fail."

Environment-specific behavior: "Use Stripe test mode in development, live mode in production. Show developer banner in staging."

Analytics: "Add Vercel Analytics to track pageviews, performance, and errors in production."

What AI Handles for You

✅ Build configuration ✅ Optimization settings ✅ Cache configuration ✅ SSL certificates ✅ CDN setup ✅ Edge functions

You describe: "Deploy my app to production" AI implements: Complete deployment pipeline with best practices

Recommended Platforms

For Next.js/React:

- **Vercel** - Best for Next.js, free tier generous
- **Netlify** - Great for static sites
- **Cloudflare Pages** - Fast, global, generous free tier

For Python/Flask:

- **Railway** - Easiest Python deployment, $5/month
- **Render** - Good free tier, easy setup

- **Fly.io** - Good for global deployment

For Databases:

- **Supabase** - Free tier: 500MB, 50k monthly users
- **Railway** - Includes PostgreSQL, $5/month
- **PlanetScale** - Serverless MySQL, generous free tier

How These Pieces Work Together

Real apps combine these building blocks:

Example: Simple SaaS Tool

1. **Authentication** - Users sign up and log in
2. **Database** - Store user data and preferences
3. **File Uploads** - Let users upload CSVs to process
4. **Email** - Send welcome email and weekly reports
5. **Deployment** - Live on custom domain

Example: Marketplace

1. **Authentication** - Buyers and sellers have accounts
2. **Database** - Products, orders, reviews
3. **File Uploads** - Product images
4. **Payments** - Stripe for transactions
5. **Email** - Order confirmations, seller notifications
6. **Deployment** - Scaled to handle traffic

Example: Content Platform

1. **Authentication** - Writers create accounts
2. **Database** - Articles, comments, likes
3. **File Uploads** - Cover images, author photos
4. **Email** - New post notifications
5. **Deployment** - Fast global CDN

Notice the pattern: Most apps need 3-5 of these building blocks. You're not building something completely unique - you're arranging familiar pieces in a new way.

Quick Reference Card

Save this for when you're building:

"I need users to log in" → Authentication section **"I need to save data"** → Database section **"Users upload images"** → File Uploads section **"I need to charge money"** → Payments section **"Send notifications"** → Email section **"Make it live"** → Deployment section

Each section tells you exactly what to tell AI. Come back to this chapter whenever you need a specific feature.

What You Don't See Here

Some building blocks are more advanced and not needed for your first few projects:

Background jobs - Tasks that run periodically (we cover this in the Technical Playbook) **Real-time features** - Live chat, collaborative editing (advanced, we cover this in Technical Playbook) **Search** - Full-text search engines (start with simple database queries) **Analytics** - User behavior tracking (add after you have users) **Testing** - Automated tests (add after you ship)

Focus on the core six building blocks above. They'll get you 90% of the way to a shipped product.

Now let's talk about what happens when things don't work as expected.

Connect & Share

💌 **Newsletter**: Build to Launch - Weekly AI building tips, templates, and real builder stories

🦋 **Bluesky**: @jenny-ouyang - Daily insights

💼 **LinkedIn**: Jenny Ouyang - Professional network

Chapter 6: When Things Break

> *"Bugs are not failures. They're conversations with your code about what it needs."*

The Most Important Lesson

Things will break. That's not a sign you're doing it wrong - it's a sign you're building.

Every developer, from beginners to 20-year veterans, spends significant time debugging. The difference is that experienced developers don't panic. They have a process.

As a vibe coder, your process is simpler: you describe the problem to AI, and AI fixes it. But knowing how to describe problems effectively is a skill worth developing.

The Debugging Mindset

What's Different for Vibe Coders

Traditional debugging:

1. Read error message
2. Find the line of code
3. Understand what the code is doing
4. Figure out why it's wrong
5. Fix the code

Vibe coding debugging:

1. Notice something's wrong
2. Describe what's happening vs what should happen
3. Copy any error messages
4. Ask AI what's wrong and how to fix it
5. Apply the fix

You don't need to understand the code. You just need to describe the symptoms accurately.

The Three Types of Breaks

1. Errors (something crashes) Example: App shows error message, nothing loads

2. Wrong behavior (works but does wrong thing) Example: Login button logs you out instead

3. Silent fails (should work but doesn't) Example: Form submits but data doesn't save

Each needs a slightly different approach, but the core process is the same.

Reading Error Messages

Error messages look scary. They're not.

Anatomy of an Error

```
Unhandled Runtime Error
Error: Invalid `prisma.user.findUnique()` invocation:

  at PrismaClient.user.findUnique
  at Object.handler (/app/api/user/route.ts:12:34)
  at async eval (webpack-internal:///(rsc)/./app/api/user/route.ts:12:34)
```

What this actually means:

- **First line:** Something crashed
- **Second line:** What went wrong (the important part)
- **Remaining lines:** Where it happened (less important for you)

What you care about: "Invalid prisma.user.findUnique() invocation"

Translation: The database query is wrong somehow.

How to Read Any Error

Look for these patterns:

"Cannot read property X of undefined" → Trying to access something that doesn't exist → Usually means missing data

"Failed to fetch" → Network request failed → Usually API endpoint wrong or server down

"[Something] is not a function" → Calling something that isn't callable → Usually wrong import or typo

"[Something] is required" → Missing required parameter → Usually form validation or API call

"Authentication required" → User not logged in → Usually session expired or redirect issue

Don't memorize these. Just know: error messages tell you what's wrong in plain English (mostly). Copy them to AI.

The Debug Process

Step 1: Notice the Problem

Be specific about what's broken:

❌ Bad: "It's not working" ✅ Good: "When I click the submit button, nothing happens"

❌ Bad: "The data is wrong" ✅ Good: "The dashboard shows 50 users but I only have 5"

❌ Bad: "Login is broken" ✅ Good: "After I enter my password and click login, the page refreshes but I'm not logged in"

What you're doing: Creating a clear description of the symptom.

Step 2: Check the Obvious

Before diving deep, check these:

For "nothing happens":

- Is the dev server running? (Check terminal)
- Did you save the file? (Cmd+S)
- Did you refresh the browser? (Cmd+R)

For "data not saving":

- Are you logged in?
- Check browser Network tab - did the request go out?
- Check Supabase dashboard - is data there but not showing?

For "page not loading":

- Is the URL correct?
- Check browser console for errors (F12)
- Try in incognito mode (rules out extension issues)

30% of bugs are solved by: Save, refresh, restart server.

Step 3: Gather Information

Open browser DevTools (F12 or Cmd+Option+I)

Check Console tab:

- Red errors? Copy them.
- Warnings? Note them.
- Nothing? That's information too.

Check Network tab:

- Click "XHR" or "Fetch" filter
- Look for red (failed) requests
- Click on them to see why they failed

Check Supabase dashboard (if using):

- Go to Table Editor
- Check if data actually saved
- Go to Logs to see database queries

What you're doing: Collecting evidence about what's happening.

Step 4: Describe to AI

Open Cursor (or ChatGPT/Claude)

Use this template:

```
I'm trying to: [what you were doing]

Expected: [what should happen]

Actually happening: [what is happening]

Error message: [copy exact error if there is one]
```

```
What I've checked:
- [thing 1]
- [thing 2]

Relevant code: [paste the file that's probably related]
```

Example:

```
I'm trying to: Save a new bookmark when user clicks the submit button

Expected: Bookmark appears in the list below the form

Actually happening: Form submits but bookmark doesn't appear. No error shown.

Error message: In browser console I see:
"Failed to fetch POST http://localhost:3000/api/bookmarks"

What I've checked:
- I'm logged in
- The form fields have values
- Other API calls work fine

Relevant code: [paste components/BookmarkForm.tsx]
```

What AI does: Analyzes the symptoms, checks the code, identifies the issue, suggests a fix.

Step 5: Apply the Fix

AI will give you a solution. Usually it's one of:

"Replace this code with this new code" → Copy the new code, paste it in the file, save

"Add this new file" → Create the file, paste the content, save

"Change this environment variable" → Update `.env.local`, restart server

"Run this command" → Copy to terminal, press Enter

Then test it. Did it work?

- ✅ Yes → Great! Move on.
- ❌ No → Reply to AI with what happened, get next fix.

Common Issues & Quick Fixes

"Port already in use"

Symptom: Can't start dev server, says port 3000 in use

Fix in terminal:

```
# Kill whatever's using port 3000
kill $(lsof -t -i:3000)

# Then start your server again
npm run dev
```

"Module not found"

Symptom: Error says "Cannot find module '@/components/Something'"

Fix:

1. Check if file exists at that path
2. Check spelling (case-sensitive)
3. If it's a package: `npm install [package-name]`

"Environment variable undefined"

Symptom: Says "NEXT_PUBLIC_SUPABASE_URL is undefined"

Fix:

1. Check `.env.local` has the variable
2. Variable name matches exactly
3. Restart dev server (Ctrl+C, then `npm run dev`)
4. No quotes around values in `.env.local`

"Authentication error" / "Session expired"

Symptom: Logged out unexpectedly, can't access protected pages

Fix:

1. Clear cookies: DevTools → Application → Cookies → Delete all
2. Log out and log in again
3. Check Supabase dashboard → Authentication → Users

"Data not appearing"

Symptom: Save works but data doesn't show in list

Possibilities:

1. Data saved but fetch query is wrong → Check Supabase Table Editor
2. Data didn't save → Check browser Network tab for failed request
3. UI not updating → Check React state management in component

Debug: Add `console.log()` statements to see what data you're getting

"CSS not applying"

Symptom: Tailwind classes not working

Fix:

1. Check `tailwind.config.js` includes your files
2. Make sure dev server is running
3. Try a different class to verify Tailwind works
4. Check for typos in class names

Advanced Debugging Techniques

Using console.log()

This is your best friend for understanding what's happening.

Where to put them:

```
// Before database call
console.log("About to fetch bookmarks for user:", userId)

// After database call
console.log("Fetched bookmarks:", bookmarks)

// In event handler
console.log("Button clicked, form values:", formData)
```

What this does: Shows you what data exists at each step. Helps you find where things go wrong.

Remove them before deploying (or AI will do it for you).

Network Tab Deep Dive

Why it's useful: Shows all requests your app makes

How to read it:

1. Open DevTools → Network tab
2. Perform the action that's broken
3. Look for red (failed) requests

Click on a request to see:

- Request URL (is it correct?)
- Status code (200 = success, 404 = not found, 500 = server error)
- Response body (what error message came back?)

Common status codes:

- 200: Success
- 400: Bad request (you sent wrong data)
- 401: Not authenticated
- 403: Not authorized
- 404: Not found
- 500: Server error

Supabase Logs

When to check: Database-related issues

How:

1. Go to Supabase dashboard
2. Click "Logs" in sidebar
3. Filter to "Errors only"

Look for:

- Row Level Security blocks
- Invalid queries
- Missing columns
- Foreign key violations

When to Start Over vs When to Fix

Keep Fixing When:

✅ The error makes sense and AI can fix it ✅ You're learning something from the debugging ✅ It's been less than 30 minutes on this issue ✅ The fix is localized (one file, one feature)

Start Over When:

❌ Multiple cascading errors that keep appearing ❌ You've lost track of what changes you made ❌ AI gives contradictory advice ❌ It's been 2+ hours with no progress ❌ You've broken something fundamental

Starting over isn't failure. Sometimes the fastest path forward is:

1. Create new project
2. Copy over only the working parts
3. Rebuild the broken feature from scratch with better prompts

You'll know more the second time. Many experienced developers do this regularly - it's faster than untangling a mess.

Prevention Is Better Than Cure

How to Avoid Common Issues

Commit often:

```
git add .
git commit -m "Working bookmark save"
git push
```

Why: You can always go back to a working version.

Test incrementally: Don't build five features then test. Build one, test, commit. Build next, test, commit.

Use TypeScript: Catches many errors before you even run the code. AI writes TypeScript as easily as JavaScript.

Be specific with AI: "Add a bookmark save feature with validation" is better than "make bookmarks work"

Read AI's code before applying: Not the whole thing - just scan for anything obviously wrong. AI makes mistakes too.

Getting Unstuck

If AI Isn't Helping

Try a different prompt: Instead of "fix this error", try:

- "Explain what this error means in simple terms"
- "What are the 3 most likely causes of this?"
- "Show me how to debug this step by step"

Break it down: Instead of "why isn't authentication working", try:

- "Is the Supabase client configured correctly?"
- "Is the session being stored?"
- "Is the protected route checking auth?"

Start a new conversation: Sometimes AI gets confused by long threads. Start fresh with a clear problem statement.

If You're Completely Stuck

Options:

1. **Take a break** (seriously, step away for an hour)
2. **Google the exact error message** (someone else has had it)
3. **Check the library docs** (Supabase docs, Next.js docs)
4. **Ask in communities:**
    - r/webdev on Reddit
    - Supabase Discord
    - Next.js Discord

When asking:

- Be specific about what's wrong
- Include code snippet (use pastebin.com for longer code)
- Say what you've tried already
- Say what error message you're getting

People are helpful if you show you've tried to solve it yourself.

Learning from Breaks

Every bug teaches you something:

"I forgot to restart the server after env changes" → Now you know env changes need restart

"I was missing await on the database call" → Now you understand async operations matter

"Row Level Security was blocking my query" → Now you understand security policies

"I was using user.id instead of user.uid" → Now you know to check exact property names

Keep a "lessons learned" list. When you hit an issue twice, it's time to remember it.

Debugging Checklist

When something breaks, run through this:

- [ ] Is the dev server running?
- [ ] Did I save all files?
- [ ] Did I refresh the browser?
- [ ] Any red errors in console?
- [ ] Any failed requests in Network tab?
- [ ] Am I logged in (if needed)?
- [ ] Are environment variables set?
- [ ] Did I commit before breaking it?

If all checked:

- [ ] Copy error message
- [ ] Check what changed recently
- [ ] Ask AI with full context

If still stuck:

- [ ] Try incognito mode
- [ ] Restart dev server
- [ ] Clear browser cache
- [ ] Start fresh conversation with AI

Real Examples

Example 1: Silent Form Failure

Problem: "Form submits but bookmark doesn't save"

Process:

1. Open Network tab
2. Submit form
3. See 401 error on POST request
4. Check response: "User not authenticated"
5. Test: Am I logged in? No.
6. Fix: Redirect to login page before showing form

Lesson: Silent fails often mean auth issues. Check that first.

Example 2: Data Shows Wrong

Problem: "Bookmarks showing but they're all the same"

Process:

1. Check database - data is correct

2. Add console.log to see what's being fetched
3. Console shows: correct data fetched
4. Add console.log to see what's being rendered
5. Console shows: Only using first item in loop
6. Tell AI: "I'm mapping over bookmarks array but all cards show the same data"
7. AI: "You're using bookmark[0] instead of bookmark in your map function"
8. Fix: Use correct variable

Lesson: When data exists but displays wrong, problem is in the UI logic.

Example 3: Mysterious Crashes

Problem: "App crashes when I click edit"

Process:

1. Console shows: "Cannot read property 'title' of undefined"
2. Tell AI: "Getting undefined error when editing bookmark"
3. AI asks: "Can you show me the edit function?"
4. Paste code
5. AI: "You're accessing bookmark before checking if it exists"
6. Fix: Add null check

Lesson: "Undefined" errors mean you're accessing data that doesn't exist yet.

The Psychology of Debugging

Frustration is normal. Even after 20 years, developers get frustrated debugging. You're not alone.

Breaks help. If you've been stuck for 30+ minutes, walk away. Solutions often appear when you stop thinking about them.

Document your wins. When you solve something tricky, write down what was wrong and how you fixed it. Future you will thank you.

Celebrate the fix. Debugging is problem-solving. Each fix is progress. Don't discount it just because "it should have worked the first time."

Remember: The goal isn't perfect code. The goal is shipped code. Bugs are temporary obstacles, not permanent failures.

When Something Works, Don't Question It

Developer instinct: "Wait, why did that work?" **Vibe coder instinct:** "Cool, it works! Moving on."

Both are valid. You don't need to understand every fix. If AI fixed it and it works, that's a win. You can always ask "why did that fix it?" later if you're curious.

Shipping beats understanding. Perfect understanding prevents shipping. Shipped products with some mystery are better than unshipped products you fully understand.

Now let's talk about what to do once you've shipped your first project.

Connect & Share

💌 **Newsletter**: Build to Launch - Weekly AI building tips, templates, and real builder stories

🦋 **Bluesky**: @jenny-ouyang - Daily insights

💼 **LinkedIn**: Jenny Ouyang - Professional network

Chapter 7: How to Know It Actually Works

> *"You're not writing tests. You're being thorough."*

The Reality

AI built your app. It works on your computer. But how do you know it'll work for everyone else?

You don't need unit tests or test-driven development. You need to be systematic about checking if things work before real users find the problems.

This chapter teaches you: How to catch obvious issues before launch, and build confidence that your app is ready.

The Three Testing Phases

Phase 1: Before You Deploy

Test locally. Catch the obvious stuff.

Phase 2: After You Deploy (Before Telling Anyone)

Test live. Make sure deployment didn't break anything.

Phase 3: When You Add Features

Test the new thing and make sure you didn't break the old things.

Phase 1: Before You Deploy

The Happy Path Test

What it is: Use your app exactly as you expect users to use it.

For the bookmark app:

1. Sign up with email/password
2. Add a bookmark
3. See it in your list
4. Search for it
5. Delete it
6. Log out

Do this 3 times. If something breaks on attempt 2 or 3, you found a bug.

The "What If I'm Dumb?" Test

What it is: Try to break things like a confused user would.

Try these:

- Leave fields empty, hit submit
- Type weird stuff (`<script>` , emoji, super long text)

- Click submit 10 times rapidly
- Hit back button randomly
- Refresh mid-action

Look for:

- Helpful error messages (not "Error 500")
- App doesn't crash
- Can't get "stuck" in a broken state

The Device Test

What it is: Test on different devices and screen sizes.

Minimum tests:

- Your phone (small screen)
- Your computer (large screen)
- Try both portrait and landscape on phone

Check:

- Can you see all the buttons?
- Can you tap/click everything?
- Does text overflow weirdly?
- Are forms usable on mobile?

The Browser Test

What it is: Test in multiple browsers.

Test these:

- Chrome (most users)
- Safari (iPhone users)
- Firefox (privacy-conscious users)

Open incognito/private mode each time (clears cache and cookies).

The Slow Internet Test

What it is: See what happens when things take time.

How:

- Chrome DevTools → Network tab → Throttle to "Slow 3G"
- Do your happy path test again

Check:

- Do you see loading states?
- Does anything feel "broken" because it's slow?
- Are there helpful messages while waiting?

Phase 2: After You Deploy

You deployed to Vercel/Netlify. Before you tell anyone:

The Deployment Check

Visit your live URL. Test again:

- Sign up with a NEW email (test email sending)
- Do the happy path test
- Try a few "dumb user" tests

Why this matters: Environment variables might be wrong. Database might not be connected. Email might not send. Things that work locally can fail in production.

The Fresh Eyes Test

Ask 1-2 friends to try it. Don't tell them how to use it.

Watch them:

- Where do they get confused?
- What do they click that doesn't work?
- What do they expect that isn't there?

Take notes. Users will try things you never imagined.

The Real Device Test

Test on actual devices, not just Chrome's device simulator:

- Your phone
- Friend's Android phone (if you have iPhone)
- Tablet if you have one

Simulators lie. Real devices show real problems.

Phase 3: When You Add Features

Every time you add something new:

The New Feature Test

1. **Test the new feature** thoroughly
2. **Test the old features** to make sure you didn't break them

Example: You added "export bookmarks to CSV"

Test the new thing:

- Export with 1 bookmark
- Export with 100 bookmarks
- Export with 0 bookmarks
- Does the file download?

- Can you open it in Excel/Google Sheets?

Test the old things:

- Can you still add bookmarks?
- Can you still delete bookmarks?
- Does search still work?

AI makes changes across multiple files. It might break something you didn't expect.

Common Problems You'll Catch

Forms That Don't Validate

- Submit empty fields
- Paste 10,000 characters in a field
- Use special characters

Broken Navigation

- Click back button
- Type URL directly
- Try to access pages you shouldn't

State Management Issues

- Do the same action twice
- Refresh in the middle of something
- Open app in two tabs

Mobile Issues

- Buttons too small to tap
- Text overlaps
- Keyboard covers input fields
- Can't scroll properly

Loading State Problems

- No indication something is happening
- Looks broken when it's just slow
- Can click buttons while loading (creates duplicates)

Tools That Help

Chrome DevTools (Free)

Console tab: Shows errors in red. Copy them, paste to AI, say "fix this"

Network tab: Shows what requests are happening. Useful for seeing if API calls fail

Device simulation: Test different screen sizes without multiple devices

How to open: Right click anywhere → "Inspect"

Actual Phones

Nothing beats real devices. Test on actual phones. Borrow friends' phones if needed.

iOS vs Android: They behave differently. Test both if possible.

When to Stop Testing

You'll never catch everything. Real users will always find something.

Ship when:

- Happy path works reliably
- Major "dumb user" scenarios don't crash
- Works on your phone and computer
- 1-2 friends tested it successfully

Don't wait for:

- Every possible edge case
- Perfect mobile experience
- Zero bugs

Perfect is the enemy of shipped.

What Happens When Users Find Bugs

They will. Here's how to handle it:

1. Thank Them

"Thanks for reporting this! I'm looking into it."

2. Reproduce It

Ask them exactly what they did. Try to make it happen on your end.

3. Fix It

Copy the error, paste to AI, describe what happened, let AI fix it.

4. Deploy the Fix

Push to GitHub, Vercel auto-deploys. Usually live in 2-3 minutes.

5. Tell Them It's Fixed

"Just deployed a fix, should work now. Let me know if you still see issues!"

Users respect builders who fix things fast more than builders who ship perfect products late.

The Testing Mindset

You're Not Writing Tests

You're **clicking around systematically** to catch obvious problems.

You're Not Paranoid

You're **thorough**. There's a difference.

You're Not Trying to Be Perfect

You're trying to **ship with confidence**.

Checklist for Every App You Build

Print this. Keep it by your computer.

Before Deploy:

- [ ] Happy path works (3 times)
- [ ] Empty form fields handled
- [ ] Works on my phone
- [ ] Works in Chrome, Safari, Firefox
- [ ] Tested with slow internet

After Deploy:

- [ ] Works at live URL
- [ ] New signup works (email sends)
- [ ] 1-2 friends tested it
- [ ] Tested on real phone

Adding Features:

- [ ] New feature works
- [ ] Old features still work
- [ ] Tested on phone
- [ ] Deployed and checked live

The Reality Check

Your first app will have bugs. So will your tenth app. Professional developers ship bugs all the time.

The difference: You caught the obvious ones. You tested systematically. You're confident the main flows work.

That's enough to ship.

Testing isn't about perfection. It's about **catching the stuff that would embarrass you** before users see it.

Next Steps

- Build your next feature
- Test it with this checklist
- Ship it
- Fix bugs as users report them
- Learn what breaks and why

The more you build, the better you get at knowing what to test.

This is learned through doing, not through reading. Go test something.

Connect & Share

💌 **Newsletter**: Build to Launch - Weekly AI building tips, templates, and real builder stories

🦋 **Bluesky**: @jenny-ouyang - Daily insights

💼 **LinkedIn**: Jenny Ouyang - Professional network

Chapter 8: What's Next

> *"You've learned to build. Now learn to ship consistently."*

What You've Accomplished

Let's take a moment to appreciate how far you've come.

You went from "I can't code" **to** "I built and deployed a web application."

You learned:

- How apps are structured (front-end, back-end, database)
- How to choose your tech stack
- How to set up authentication
- How to work with databases
- How to deploy to production
- How to debug when things break

You built:

- A real application with user accounts
- Database integration
- Multiple features working together
- Something live on the internet

Most importantly: You proved you don't need to learn traditional programming to build real products.

This isn't just a tutorial you completed. This is evidence that you can build. That's a different identity. You're now someone who ships.

The Path Forward

You have three options from here:

Option 1: Build Your Next Project

If you have a product idea, build it. You know enough now. Use the same process:

1. Describe what you want to AI
2. Build incrementally
3. Test as you go
4. Deploy early
5. Iterate based on feedback

Start simple:

- One core feature that solves a real problem
- Basic authentication if needed
- Simple UI (polish comes later)
- Deploy as soon as it does anything useful

Common second projects:

- Internal tool for your current job
- Side project to solve your own problem
- Freelance client project
- Productized service

Timeline: 2-4 weeks to first version

Option 2: Deepen Your Skills

If you want to get better before starting your own project, build variations:

Different app types:

- Build a blog with comments (content-focused)
- Build a task manager (CRUD operations)
- Build a link-in-bio tool (simple but useful)
- Build a form builder (meta: build a tool to build tools)

Add new features:

- Add payments to your bookmark app (monetize it)
- Add collaboration (invite others to your workspace)
- Add API (let others integrate with your app)
- Add mobile responsiveness (works great on phones)

Learn new stacks:

- Try Flask instead of Next.js (Python lovers)
- Try mobile with React Native
- Try Chrome extension with the same skills

Timeline: 1-2 weeks per project

Option 3: Ship Something Today

If you want to prove you can ship fast, build micro-products:

One-page tools:

- Calculator for your industry
- Unit converter
- Form generator
- Simple game

Landing pages:

- For your newsletter
- For your services
- For a product idea (before building it)
- Template you can sell

Timeline: 4-8 hours

Why this matters: Shipping builds muscle memory. The more you ship, the easier shipping becomes.

Join the Build to Launch Community

You're not learning in isolation.

The Build to Launch newsletter is where non-technical builders share what they're shipping, what they're stuck on, and what they're learning.

What you'll get:

Weekly:

- New AI building techniques
- Real builder case studies
- Common patterns and solutions
- Tool recommendations
- Framework comparisons

Monthly:

- Deep-dive tutorials
- Expert interviews
- Product teardowns
- Template libraries
- Community showcases

Plus access to:

- Monthly office hours with me
- Templates and boilerplates
- Exclusive deals on tools
- First access to new guides

Why this matters: Building alone is hard. Building with a community of people who understand your journey is energizing.

Subscribe at: buildtolaunch.substack.com

Free tier: Weekly tips and case studies **Premium tier:** $90/year - Everything plus templates, community, office hours

---

Ideas for Your Next Build

Stuck on what to build? Here are proven ideas that solve real problems:

For Your Current Job

Internal tools are the best first projects because:

- You know the problem deeply
- You have immediate users
- Feedback is instant
- Can prove value to your employer

Examples:

- Report generator (save 2 hours/week)
- Data dashboard (visualize KPIs)
- Form automation (reduce manual work)
- Booking system (replace spreadsheets)
- Knowledge base (organize team docs)

Start Monday: Identify one task you do manually that takes 30+ minutes per week.

For A Specific Audience

Vertical SaaS - tools for specific industries:

- CRM for real estate agents
- Booking system for yoga studios
- Menu planner for dietitians
- Case tracker for lawyers
- Gig tracker for musicians

Why these work: Generic tools are too broad. Specific tools win specific audiences.

Micro-SaaS

Small, focused tools that do one thing well:

- Email signature generator
- Screenshot beautifier
- Social media scheduler
- Link tracker
- Invoice generator

Why these work: Solve a specific pain point. Can charge $5-20/month. Grow organically.

Content Tools

Tools for creators:

- Newsletter archive (like you built)
- Podcast transcript search
- YouTube video summarizer
- Twitter thread compiler
- Blog post generator

Why these work: Creators pay for tools that save time or improve content.

Personal Productivity

Tools you'd use:

- Habit tracker
- Reading list manager
- Recipe organizer
- Expense tracker
- Learning journal

Why these work: If you'd pay for it, others will too.

The Builder's Framework

Here's the process I use for every project:

Week 1: Validate

Before building anything:

1. Talk to 5 people in target audience
2. Describe the problem (not the solution)
3. Ask: "How do you handle this now?"
4. Ask: "Would you pay $X to solve this?"
5. Document their exact words

Red flag: People say "cool idea" but won't pay **Green light:** People say "where do I sign up?"

Week 2: Build MVP

Absolute minimum features:

- One core workflow
- Basic UI (ugly is fine)
- Manual onboarding (you email them login)
- No billing (pay later)

Ship it to those 5 people from week 1.

Week 3: Learn

Watch them use it:

- Where do they get confused?
- What features do they ask for?
- What do they use most?
- What do they ignore?

Iterate based on behavior, not opinions.

Week 4: Polish

Now make it good:

- Fix the confusing parts
- Add the requested features
- Polish the UI
- Set up billing
- Write real landing page

Launch to more people.

Ongoing: Grow

Focus on:

- One acquisition channel at a time
- Talk to every user who cancels
- Build what people pay for
- Ignore feature requests from non-payers

This framework works because you don't waste time building things nobody wants.

Shipping Consistently

The difference between people who build once and people who build careers is consistency.

How to Ship Every Week

Sunday: Brainstorm ideas (30 min) **Monday:** Pick one, validate with 3 people (1 hour) **Tuesday-Thursday:** Build core feature (3-4 hours each) **Friday:** Deploy and share (2 hours) **Saturday:** Rest or respond to feedback

That's 12-15 hours per week to ship something new.

Can't do weekly? Ship monthly. Consistency matters more than frequency.

How to Get Faster

First project: 40 hours (this guide) **Second project:** 25 hours (same patterns) **Third project:** 15 hours (reuse code) **Fourth project:** 10 hours (templates ready) **Fifth project:** 6 hours (mostly assembly)

Speed comes from: Recognizing patterns, reusing solutions, having templates, making fewer mistakes.

By project 10, you'll ship substantial apps in a weekend.

Common Fears (And Why They're Wrong)

"My idea isn't original"

Good. Unoriginal ideas have proven demand. You're not trying to invent Facebook. You're solving a real problem for a specific group of people.

Every successful product has 10+ competitors. They still succeed because execution and positioning matter more than originality.

"I don't know if people will pay"

Find out fast. Build a landing page, write the copy, add a "Join Waitlist" button. Share it. If nobody signs up, don't build it. If people do, you have validation.

Money talks. Excitement is free. Paying customers are real.

"What if it doesn't work?"

Then you learned something. Every failed project teaches you:

- What people actually want
- How to build faster next time
- What tech stack works for you

- How to ship despite fear

Successful builders have 10x more failed projects than successful ones. The winners are just the ones who kept building.

"Real developers will judge my code"

Real developers are too busy building. The ones who judge aren't building anything worth judging.

Also: AI writes better code than most developers anyway. Your code quality is fine.

"I should learn 'proper' programming first"

No. That's just fear disguised as professionalism. You'll learn programming concepts through building. That's how everyone actually learns - the tutorials are theater.

Start building today. Learn what you need when you need it.

Resources You Might Need

Learning Resources

This guide - Foundation **Next.js Docs** - When you need specific Next.js info (nextjs.org/docs) **Supabase Docs** - When you need database help (supabase.com/docs) **Tailwind Docs** - When you need specific styling (tailwindcss.com/docs)

Communities

r/SideProject - Share what you're building **Indie Hackers** - Founders building in public **Next.js Discord** - Technical Next.js help **Supabase Discord** - Database and auth help

Tools

Claude Pro - $20/month, best AI for building **Cursor** - $20/month, AI-powered code editor **Supabase** - Free tier, upgrade at $25/month **Vercel** - Free tier, rarely need to upgrade **Stripe** - Free, pay 2.9% + $0.30 per transaction

Total monthly cost to build: $40 **Total monthly cost** to run: $0 (until you have users)

Inspiration

Follow builders shipping:

- @levelsio - Location independence, building in public
- @stephsmithio - From writing to building
- @dannypostmaa - Designer who builds
- @buildtollaunch - Me! Domain experts → builders

The best way to learn is watching others build and ship.

My Challenge to You

Build and deploy something in the next 7 days.

It doesn't have to be big. It doesn't have to be original. It doesn't have to be perfect.

It just has to be **shipped**.

Ideas:

- A calculator for your industry
- A simple form that emails you results
- A landing page for something you might build
- A tool you'd use yourself
- A one-page game

7 days from now, you should be able to share a link that shows something you built.

Why? Because you'll prove to yourself that you can ship. And once you know you can ship, everything changes.

Final Thoughts

You picked up this guide because you had an idea but thought "I can't code."

Now you know: You don't need to code. You need to describe what you want and let AI translate that into code.

This changes everything:

- Teachers can build classroom tools
- Consultants can build internal tools for clients
- Marketers can build landing pages without developers
- Designers can build their designs without handoff
- Anyone with domain expertise can build for their industry

The barrier isn't technical anymore. The barrier is deciding to start.

You've proven you can build. You built something real, deployed it, and learned the fundamentals.

What you do next is up to you.

You can:

- Build your idea
- Build for your job
- Build to learn
- Build to earn
- Build because it's fun

All of these are valid. The only invalid choice is not building because you "can't code."

You can build.

Go ship something.

One Last Thing

Building can feel lonely. You'll have days where nothing works, where you question if this is worth it, where you want to quit.

Those days are normal. Every builder has them.

What separates people who ship from people who don't isn't talent or intelligence or even persistence. It's community.

Join us: buildtolaunch.substack.com

Share what you're building. Ask for help when you're stuck. Celebrate your wins. Learn from others.

Building is more fun together.

I'm excited to see what you create.

— Jenny

Quick Links

📬 **Newsletter:** buildtolaunch.substack.com 🐦 **Twitter:** @buildtolaunch 📧 **Email:** jenny@buildtolaunch.com

---

Keep Building

Remember:

Vibe coding isn't about writing perfect code. It's about shipping real products.

You're not a "non-technical person trying to code." You're a builder using modern tools.

The goal isn't to become a developer. The goal is to build products that matter.

Start today. Ship this week. Repeat next week.

Welcome to the community of builders.

Now go build something.

💌 **Subscribe at buildtolaunch.substack.com for weekly building tips, real builder stories, and exclusive templates**

Acknowledgments

This guide wouldn't exist without:

The vibe coding community - For showing that domain expertise + AI > traditional programming

The builders who ship - For proving this works and sharing your journey

AI tools that enable us - Claude, ChatGPT, Cursor, and everyone pushing the boundaries

You - For taking the leap and proving you can build

Thank you for reading. Now go ship.

Connect & Share

💌 **Newsletter**: Build to Launch - Weekly AI building tips, templates, and real builder stories

🦋 **Bluesky**: @jenny-ouyang - Daily insights

💼 **LinkedIn**: Jenny Ouyang - Professional network