

VIBE CODING

*The Architect's Guide to
AI-Powered Software Development*

From Prompt Engineering to Production Systems

Josh Cooper

Software Architect | 15 Years in Enterprise Systems

Table of Contents

Introduction: Why This Book Exists

Chapter 1: The Old World: How Traditional Programming Worked

- 1.1 The Software Development Lifecycle (Before AI)
- 1.2 The Time and Cost Reality
- 1.3 What Made It Hard
- 1.4 Chapter Takeaway

Chapter 2: The AI Revolution: How We Got Here

- 2.1 From Search Engines to Thinking Machines
- 2.2 The AI Agent Landscape: Know Your Tools
- 2.3 Claude (Anthropic) — The Architect's Partner
- 2.4 Gemini (Google) — The Visual Builder
- 2.5 ChatGPT (OpenAI) — The Generalist Sounding Board
- 2.6 Emerging Specialists
- 2.7 Chapter Takeaway

Chapter 3: The Art of the Prompt: Communicating with AI

- 3.1 Why Prompting Is the New Literacy
- 3.2 The Prompt Framework: CONTEXT – CONSTRAINTS – OUTPUT
- 3.3 Prompt Patterns That Work
- 3.4 Common Prompting Mistakes
- 3.5 Prompt in Action: Building a Small Business Website
- 3.6 Prompt in Action: Building a Client Management System
- 3.7 Prompt in Action: Refactoring and Extending Existing Code
- 3.8 Chapter Takeaway

Chapter 4: Think Like an Architect

- 4.1 Why Architecture Matters More Than Ever
- 4.2 The Architect's Mindset
- 4.3 Architecture Patterns Every Vibe Coder Should Know
- 4.4 Using AI as Your Architecture Partner
- 4.5 Chapter Takeaway

Chapter 5: Choosing Your Tech Stack

- 5.1 Why Stack Decisions Matter
- 5.2 Frontend Frameworks: What to Build With
- 5.3 Backend: Where Logic Lives
- 5.4 Databases: Choosing the Right Fit
- 5.5 ORMs and Data Access
- 5.6 Using AI to Evaluate Your Stack
- 5.7 Chapter Takeaway

Chapter 6: GitHub: Your Code's Home Base

- 6.1 Why Every Project Needs a Repository
- 6.2 Git Fundamentals
- 6.3 GitHub as a Collaboration Platform
- 6.4 GitHub Actions: Automation Built In
- 6.5 GitHub + AI
- 6.6 Repository Best Practices
- 6.7 Chapter Takeaway

Chapter 7: Docker: Your Development Superpower

- 7.1 Why Docker Changes Everything
- 7.2 Docker Core Concepts
- 7.3 Docker in the CI/CD Pipeline
- 7.4 A Real Docker Win
- 7.5 Getting Started with Docker (AI-Assisted)
- 7.6 Chapter Takeaway

Chapter 8: Cloud Providers: Where Your Code Lives

- 8.1 The Big Three: AWS, Azure, and GCP
- 8.2 Developer-Friendly Alternatives
- 8.3 Choosing the Right Provider
- 8.4 AI-Assisted Cloud Setup
- 8.5 Chapter Takeaway

Chapter 9: Security and Compliance: Building Trust

- 9.1 Why Security Can't Be an Afterthought
- 9.2 The OWASP Top 10: Your Security Baseline
- 9.3 Security Tools Every Developer Should Use
- 9.4 AI-Assisted Security Practices
- 9.5 Chapter Takeaway

Chapter 10: The Compound Effect: Growing with AI

- 10.1 The Beginner's Advantage
- 10.2 The Experience Multiplier
- 10.3 Building Your AI-Augmented Workflow
- 10.4 The Future of Vibe Coding
- 10.5 What to Do Right Now
- 10.6 Final Takeaway

Appendix A: Recommended Tools and Resources

Appendix B: Prompt Templates Library

Appendix C: Pre-Deployment Security Checklist

Introduction: Why This Book Exists

I've been building software for fifteen years. Enterprise systems, cloud infrastructure, APIs that serve millions of requests—the whole stack. And I'll be honest with you: what's happening right now with AI is the biggest shift I've seen in my entire career. It's not hype. It's real, and it's moving fast.

Here's the thing most people get wrong about “vibe coding” though. They think it's just about getting AI to spit out code for you. And sure, it can do that. But the developers who are actually thriving right now? They're the ones who think like architects. They understand how systems fit together. They can look at AI-generated code and know whether it's solid or whether it's going to fall apart at 2 AM on a Saturday when traffic spikes.

I wrote this book for two kinds of people, and you probably already know which one you are:

If you're already a software engineer, you've got the hard-won knowledge. You understand systems, patterns, trade-offs. This book is going to show you how to plug AI into that experience and multiply your output by 5–10x. I'm not exaggerating—I've seen it in my own work.

If you're newer to development—maybe you've been curious about building software but the learning curve always felt too steep—AI just knocked that wall down for you. But you still need to understand how the pieces connect. You don't need to memorize syntax, but you do need to think in systems. That's what this book gives you.

Every chapter comes back to one core idea: the more you understand the system you're building, the more powerful your AI partnership becomes. That's true whether you're writing your first prompt or architecting a distributed cloud application.

So grab a coffee, open your favorite AI agent, and let's build something.

Chapter 1: The Old World: How Traditional Programming Worked

Before we talk about where we're going, let's talk about where we've been

1.1 The Software Development Lifecycle (Before AI)

If you've been in the industry for any amount of time, this will feel familiar. If you're newer, this is what it used to look like—and why AI feels like such a big deal to those of us who lived through it.

- **Requirements gathering:** Weeks of meetings. Stakeholder interviews. Documentation that nobody would read but everybody demanded. And all of this happened before anyone wrote a single line of code.
- **Architecture and design:** System design documents, UML diagrams, review cycles—this phase alone could eat months on enterprise projects. I've seen teams spend longer designing a system than it would now take to build it.
- **Implementation:** Hand-coding every function, class, and module. A simple CRUD app—the kind you can now scaffold in an afternoon with AI—could take a solo developer 2–4 weeks.
- **Testing:** Manual QA cycles, writing unit tests line by line, standing up integration testing environments that took days to configure and broke constantly.
- **Deployment:** FTP uploads. Manual server configuration. And that beautiful moment where you crossed your fingers and hoped production would match staging. (It usually didn't.)

1.2 The Time and Cost Reality

Let's put some real numbers on this, because the contrast with AI-assisted development is staggering:

- **Enterprise projects:** 6–18 months from concept to MVP. Fortune 500 projects routinely blew past budgets by 50–100%. I've seen it firsthand—and it wasn't because the teams were bad. The process demanded it.
- **Startup MVPs:** Even lean startups burned 3–6 months building a first version. A lot of seed funding got spent before a single user ever touched the product.
- **Maintenance burden:** Here's the number that surprises people: for every hour of new development, teams typically spent 2–3 hours maintaining existing code. Most of the work wasn't building—it was keeping things running.
- **The knowledge bottleneck:** You needed deep expertise in specific languages, frameworks, and infrastructure. Wanted to switch from Java to Python? That's months of ramp-up. The tools were powerful but narrow.

1.3 What Made It Hard

It wasn't that developers were slow—it's that the work was inherently labor-intensive:

- **Boilerplate code:** Roughly 70% of code in most projects was structural plumbing—not the business logic that actually mattered.
- **Documentation drift:** Documentation was always outdated. Always. Because maintaining it was a completely separate, manual process that nobody had time for.
- **Tribal knowledge:** The most critical system understanding lived in people's heads. When someone left the team, they took institutional knowledge with them. We've all been there.
- **Context switching:** Frontend. Backend. Infrastructure. Security. Each one required a different mental model and different toolset. Switching between them was exhausting.

1.4 Chapter Takeaway

Look, traditional development wasn't bad—it was the best we had. And the fundamentals from that era (clean architecture, testing discipline, understanding systems) are exactly what make AI-assisted development so powerful. You're not throwing away that knowledge. You're about to supercharge it.

Chapter 2: The AI Revolution: How We Got Here

A no-hype look at what these tools actually are and what each one does best

2.1 From Search Engines to Thinking Machines

I'm not going to give you a full history of artificial intelligence—you can Google that. But there are a few key moments that directly explain why you can now ask an AI to build you a full-stack application:

- **The transformer breakthrough (2017):** Google published “Attention Is All You Need” and it quietly changed everything. This is the architecture that powers every major AI coding tool today.
- **GPT and the scaling era:** OpenAI figured out that bigger models with more data did things nobody predicted. Emergent capabilities—the model could suddenly do things it was never explicitly trained to do.
- **The ChatGPT moment (2022):** AI went from research labs to everyone's browser overnight. If you were in software at the time, you remember the week everything changed.
- **Code-native AI (2023–2025):** Coding assistants evolved from fancy autocomplete to genuine architectural partners. This is where things got really interesting for people like us.

2.2 The AI Agent Landscape: Know Your Tools

Here's something I wish someone had told me early on: not all AI agents are the same. They each have real strengths and real weaknesses. The best vibe coders don't have a favorite—they have a toolkit. Let me walk you through the ones that matter.

2.3 Claude (Anthropic) — The Architect's Partner

Full disclosure: I'm biased here because this is the tool I use most for serious development work. But there's a reason for that.

- **Where it shines:** System architecture, infrastructure design, complex multi-file codebases, API design, security analysis, and any task that requires holding a lot of context at once.
- **Reach for it when:** You're planning a project structure, writing backend services, setting up CI/CD pipelines, or thinking through edge cases that could bite you later.
- **The real advantage:** It's exceptionally good at reasoning about how components fit together. Claude Code (the CLI tool) can work directly with your files and understand your whole project. That's a game-changer.

2.4 Gemini (Google) — The Visual Builder

- **Where it shines:** Image generation, UI/UX design, asset creation, and anything where you need the AI to see or create something visual.
- **Reach for it when:** You're creating design assets, generating UI mockups, working with visual data, or building frontend interfaces where the look and feel matters.
- **The real advantage:** Native multimodal capabilities. It can look at a screenshot, understand what it sees, and generate code or assets based on that.

2.5 ChatGPT (OpenAI) — The Generalist Sounding Board

- **Where it shines:** Broad general knowledge, brainstorming, explaining unfamiliar concepts, and quick prototyping when you're exploring an idea.
- **Reach for it when:** You're rubber-ducking ideas, exploring a domain you don't know well, or need a quick answer without a lot of setup.
- **The real advantage:** Massive training data and a huge plugin ecosystem. It's the Swiss Army knife—not the best at any one thing, but capable of almost everything.

2.6 Emerging Specialists

The landscape is moving fast, and there are some newer tools worth knowing about:

- **Cursor / Windsurf:** IDE-integrated AI that understands your entire codebase in context. If you like working inside an editor, these are worth trying.
- **GitHub Copilot:** Inline code suggestions tuned for developer workflow. Great for filling in the gaps while you're coding.
- **Specialized agents:** Domain-specific tools for database design, security scanning, and automated testing are popping up constantly. Keep your eye on this space.

2.7 Chapter Takeaway

Don't marry one tool. Build a toolkit. I use Claude for architecture and backend, Gemini when I need visual work, and ChatGPT when I'm exploring something new. The developers getting the best results aren't the ones using one AI—they're the ones routing the right task to the right agent.

Chapter 3: The Art of the Prompt: Communicating with AI

This is the skill that separates “it kind of works” from “this is production-ready”

3.1 Why Prompting Is the New Literacy

Here's a truth that took me a while to fully appreciate: the quality of what AI gives you is directly proportional to the quality of what you give it. Vague prompts produce vague results. Every time. But when you learn to communicate precisely—when you give AI the same context you'd give a senior developer joining your team—it produces code that's genuinely production-ready. Prompting well isn't a nice-to-have. It's the skill.

3.2 The Prompt Framework: CONTEXT → CONSTRAINTS → OUTPUT

I use this framework for almost every prompt I write. It's simple, but it works every time:

- **Context:** What are you building? What's the tech stack? What already exists? Think of it this way: if a new developer sat down at your desk, what would you tell them before they started coding?
- **Constraints:** What are the boundaries? Performance requirements, security standards, coding conventions, file structure preferences, which dependencies to use or avoid. The AI doesn't know your preferences unless you tell it.
- **Output:** What exactly do you want back? A complete file? A code review? An architecture diagram? A list of options with trade-offs? Be explicit—AI is great at following instructions, but terrible at reading your mind.

3.3 Prompt Patterns That Work

Over time, I've found myself reaching for the same prompt structures again and again. Here are the ones I use most:

- **The Architect Prompt:** “I'm building [system]. Here's what exists: [context]. I need you to design [component] that integrates with [existing pieces]. Consider [constraints].” This is your workhorse.
- **The Debugging Prompt:** “Here's my code: [code]. Here's the error: [error]. Here's what I've tried: [attempts]. What am I missing?” The “what I've tried” part stops the AI from repeating things you already ruled out.

- **The Review Prompt:** “Review this code for [security/performance/maintainability]. Be specific about what to change and why.” Don’t just ask “is this good?”—you’ll get a generic answer.
- **The Iterative Prompt:** “That’s close, but [specific feedback]. Keep [what worked] and change [what didn’t].” Treat it like a conversation, not a one-shot request.

3.4 Common Prompting Mistakes

I see these constantly, and I’ve made every one of them myself:

- **Too vague:** “Build me a website” gives you something generic. “Build a Next.js landing page with a hero section, pricing table, and contact form using Tailwind CSS” gives you something you can actually use. Specificity is free—use it.
- **No context:** Dropping code into a prompt without explaining the project, the stack, or what you’re trying to achieve.
- **Ignoring iteration:** Expecting perfection on the first try. That’s not how it works—and honestly, it’s not how working with a human colleague works either. It’s a conversation.
- **Not verifying:** Blindly copying AI output into production without reading it, testing it, or understanding it. Never ship code you don’t understand.

3.5 Prompt in Action: Building a Small Business Website

Let me show you what a well-crafted prompt actually looks like in practice. Say you’re helping a local bakery get online:

EXAMPLE PROMPT — Multi-Page Business Website

Build a new multi-page website for a local bakery called Golden Crust Bakery. The site should include the following: Landing page: A welcoming hero section with the bakery’s name and tagline. A section displaying the bakery’s location with address, hours of operation, and an embedded map. A menu section showcasing the bakery’s core offerings organized by category (breads, pastries, cakes, seasonal specials). A schedule section highlighting limited-time bakery offerings and upcoming community events the bakery is sponsoring or attending. A footer with contact information and social media links. Blog page: Linked from the landing page navigation. Displays a list of blog posts with titles, thumbnails, and preview text. Each post is a recipe the bakery wants to share with its community. Tech stack: Use Next.js with Tailwind CSS. Structure the project with reusable components. Make the schedule and blog content easy to update (use a simple JSON or markdown-based content system so the

bakery owner can update without touching code). Ensure the site is mobile-responsive and optimized for local SEO.

What makes this work: it names the business, defines pages, specifies content structure by category, declares the tech stack, asks for maintainability (JSON content), and calls out non-functional requirements (mobile, SEO). Compare it to “build me a bakery website.” Night and day.

3.6 Prompt in Action: Building a Client Management System

Now let's level up. This next example is closer to what real business software looks like—a system with data relationships, user workflows, and actual business logic:

EXAMPLE PROMPT — CPA Client Management CRM

Create a new project for a CRM application designed for a CPA (Certified Public Accountant) practice. The system should manage the full client lifecycle. Here are the requirements: Client management: Add, edit, and archive client records. Each client should have: business name, contact name, email, phone, address, EIN/SSN (encrypted and masked in the UI), entity type (sole proprietor, LLC, S-Corp, C-Corp, partnership, nonprofit), fiscal year end date, and assigned CPA staff member. Service tracking: Track which services each client is enrolled in (tax preparation, bookkeeping, payroll, advisory, audit). Log the status of each service per tax year or billing period (not started, in progress, waiting on client, under review, complete, filed). Document management: Upload and organize client documents by tax year and category (W-2s, 1099s, receipts, prior returns, correspondence). Track which required documents have been received and which are still outstanding. Deadlines and reminders: Display upcoming filing deadlines by client. Automatic visual alerts for deadlines within 30, 14, and 7 days. Track extensions filed and their new due dates. Dashboard: Overview showing total active clients, services in progress, upcoming deadlines, and documents pending. Filterable by assigned staff member, entity type, and service status. Tech stack: Use React with TypeScript for the frontend, Node.js with Express for the API, and PostgreSQL for the database. Use Prisma as the ORM. Include authentication with role-based access (admin, CPA, staff). All sensitive data (SSN/EIN) must be encrypted at rest. Structure the project with a clean separation between API routes, business logic, and data access layers.

This works because it defines the domain model, describes workflow behavior (not just data), calls out security explicitly, and provides architecture guidance. The AI isn't guessing—you're telling it exactly what a CPA practice needs.

3.7 Prompt in Action: Refactoring and Extending Existing Code

The first two examples were “build from scratch” prompts. But most real development is working with code that already exists—debugging, refactoring, adding features. This is where a lot of people struggle with AI because they don't give enough context about what's already built. Here's how I handle it:

EXAMPLE PROMPT — Refactoring Existing Code

I have an existing Express.js API for an e-commerce platform. The codebase has grown organically and everything is in a single `routes/api.js` file—about 1,200 lines with route handlers, database queries, validation, and business logic all mixed together. Here's the current structure: - `routes/api.js` (all endpoints: products, orders, users, payments) - `models/db.js` (raw SQL queries using pg driver) - No middleware for auth or validation - No error handling pattern—each route has its own try/catch I need you to: 1. Refactor this into a clean architecture with separate route files, a service layer for business logic, and a data access layer for database queries. 2. Add a centralized error handling middleware. 3. Add input validation using Zod for the product and order endpoints. 4. Keep the existing pg driver (don't migrate to an ORM) but organize queries into repository files. 5. Maintain backward compatibility—all existing API endpoints should work identically after refactoring. Start by showing me the new folder structure, then refactor the product endpoints as an example I can follow for the rest.

Notice what's different here compared to the “build from scratch” prompts:

- **Current state described honestly:** You're telling the AI exactly what exists today—warts and all. The 1,200-line file, the mixed concerns, the missing patterns. The AI can't help fix problems it doesn't know about.
- **Constraints preserve what works:** “Keep the pg driver” and “maintain backward compatibility” tell the AI not to rewrite everything. In real projects, you can't throw away working code just because the architecture is messy.
- **Scoped deliverable:** “Refactor the product endpoints as an example” is way better than “refactor everything.” You get a pattern you can follow, not a massive output you can't review.

- **Progressive approach:** Folder structure first, then one example. This gives you a chance to course-correct before the AI does 10 more files the same way.

This is the kind of prompting that experienced developers do naturally—they're managing scope, preserving stability, and asking for patterns instead of finished products. If you're newer to this, study this example closely. It's the workflow you'll use most often. For a grab-and-go version of this and other templates, see Appendix B.

3.8 Chapter Takeaway

Prompting isn't magic and it's not a gimmick. It's a skill, and like any skill, it gets better with practice. The difference between a mediocre prompt and a great one isn't about length—it's about specificity, structure, and thinking like an architect before you type a single word. These examples aren't theoretical—they're the kind of prompts I write every day.

Chapter 4: Think Like an Architect

This is the chapter that makes everything else in this book more powerful

4.1 Why Architecture Matters More Than Ever

Here's something that might seem counterintuitive: AI can generate code faster than any human, but that actually makes architectural thinking more valuable, not less. AI can't decide what to build, how services should talk to each other, or which trade-offs are right for your business. That's still your job—and it's the highest-leverage skill you can develop.

4.2 The Architect's Mindset

Architecture isn't about being the smartest person in the room. It's about asking the right questions before you start building:

- **Systems thinking:** Every component exists in relationship to others. Before you write code (or prompt it), map the system: how does data flow? Where are the service boundaries? What happens when something fails?
- **Trade-off analysis:** There are no perfect solutions—only trade-offs. Monolith vs. microservices? SQL vs. NoSQL? The architect doesn't just know the options. They know why they're choosing one over the other for this specific project.
- **Future-proofing (without over-engineering):** Design for what's likely to change, not every possible scenario. Isolate the volatile from the stable. Keep things simple everywhere else.
- **Simplicity bias:** This one is hard-earned. The best architecture is the simplest one that meets the requirements. I've seen more projects fail from over-engineering than from under-engineering.

4.3 Architecture Patterns Every Vibe Coder Should Know

You don't need to memorize every design pattern ever invented. But these four will cover 90% of what you'll encounter:

- **Monolith-first:** Start simple. A well-structured monolith can serve most projects through their first 100K users. Don't split into microservices until you have a real reason to.
- **API-first design:** Define your contracts before your implementation. This lets frontend and backend work in parallel, and it makes AI-generated code far more predictable.
- **Event-driven architecture:** Decouple services through events. This pattern scales naturally and makes systems resilient when individual components fail.
- **The 12-Factor App:** A battle-tested methodology for building cloud-native applications. Short, practical, and still as relevant as the day it was written. Read it at 12factor.net.

4.4 Using AI as Your Architecture Partner

One of the most underrated uses of AI isn't code generation—it's thinking through architecture with a partner that never gets tired:

- **Describe your system to Claude:** Lay out your requirements, constraints, and scale expectations. Ask for two or three architectural options with trade-offs. The quality of the response will surprise you.
- **Request architecture decision records (ADRs):** Have AI help document why you made each major technical decision. Future-you will be grateful.
- **Validate with diagrams:** Ask for Mermaid or PlantUML diagrams of your proposed architecture. Seeing the system visually almost always surfaces problems you missed.
- **Stress-test your design:** Ask “What happens when [component] fails?” or “How does this handle 10x traffic?” These questions separate architectures that survive from ones that don't.

4.5 Chapter Takeaway

AI levels the playing field for code generation. But it raises the bar for architectural thinking. The developers who understand systems—how the pieces connect, where things break, what trade-offs matter—are the ones building software that lasts. That's always been true. AI just made it more obvious.

Chapter 5: Choosing Your Tech Stack

The decisions you make here ripple through everything else—so let's get them right

5.1 Why Stack Decisions Matter

Before you prompt a single line of code, you need to decide what you're building with. Framework. Language. Database. ORM. CSS approach. These choices ripple through your entire project—and once you're a few weeks in, switching becomes expensive. The good news: AI can actually help you evaluate your options before you commit.

New to this?

A “tech stack” is just the combination of tools and languages you use to build your app. Think of it like choosing whether to build a house with wood framing or steel—each has trade-offs, and the choice affects everything that comes after.

5.2 Frontend Frameworks: What to Build With

The frontend is what your users see and interact with. Here's the honest breakdown of the major options:

- **React:** The most widely used framework. Massive ecosystem, huge community, and every AI agent speaks it fluently. If you're unsure, React is the safe bet. Next.js adds server-side rendering, routing, and deployment simplicity on top.
- **Vue:** Easier learning curve than React, with excellent documentation. Great for smaller projects and teams that value simplicity. Nuxt.js is the Vue equivalent of Next.js.
- **Svelte:** Compiles to vanilla JavaScript—no runtime framework. Incredibly fast and fun to write. Smaller ecosystem, but growing quickly. SvelteKit is the full-stack framework.
- **Plain HTML/CSS/JS:** Don't overlook this. For static sites, landing pages, and simple projects, you don't need a framework at all. AI generates excellent vanilla HTML.

5.3 Backend: Where Logic Lives

The backend handles data, business logic, authentication, and integrations. Your choice here depends on what you already know and what you're building:

- **Node.js (Express / Fastify):** JavaScript everywhere—frontend and backend in one language. Express is the workhorse. Fastify is faster and more modern. Both have excellent AI support.

- **Python (FastAPI / Django):** If you're doing data science, ML, or need rapid prototyping, Python is a natural fit. FastAPI is modern and fast. Django gives you everything out of the box.
- **Go:** When performance matters. Go compiles to a single binary, starts instantly, and handles concurrency beautifully. Steeper learning curve but worth it for high-performance services.
- **Serverless functions:** For simpler backends, platforms like Vercel, Cloudflare Workers, or AWS Lambda let you write individual functions without managing servers at all.

5.4 Databases: Choosing the Right Fit

Your database choice is one of the hardest to change later, so it's worth getting right upfront:

New to this?

A database is where your app stores information—user accounts, products, orders, messages. Think of it as a really organized filing cabinet that your code can search through instantly.

- **PostgreSQL:** My default recommendation. Relational, rock-solid, handles JSON well, scales far more than most projects need. If you're unsure, start here.
- **MySQL:** Similar to PostgreSQL, widely supported, and the default for many hosting platforms. Slightly simpler, slightly less feature-rich.
- **MongoDB:** Document-based (NoSQL). Good for rapid prototyping and data that doesn't fit neatly into tables. Be careful with it at scale—relational databases handle complex queries better.
- **SQLite:** A file-based database that requires zero setup. Perfect for local development, mobile apps, and projects that don't need concurrent writes from multiple servers.

5.5 ORMs and Data Access

An ORM (Object-Relational Mapper) lets you interact with your database using your programming language instead of writing raw SQL. Here are the ones that work best with AI-generated code:

New to this?

An ORM is a translator between your code and your database. Instead of writing SQL queries by hand, you write code in JavaScript or Python and the ORM converts it into database operations for you.

- **Prisma:** My top pick for Node.js projects. Type-safe, great developer experience, and AI agents generate excellent Prisma schemas. Migrations are straightforward.
- **Drizzle:** Lighter weight than Prisma, closer to SQL. Good if you want more control over your queries without going fully raw.

- **SQLAlchemy:** The gold standard for Python. Powerful, flexible, and well-documented. Every Python AI agent knows it.
- **No ORM:** For simple projects or when you need maximum performance, writing SQL directly with a query builder (like Knex.js) is perfectly valid. Don't add complexity you don't need.

5.6 Using AI to Evaluate Your Stack

Here's a practical approach I use on every new project: describe your project's requirements to Claude and ask it to recommend a tech stack with reasoning. Tell it your constraints—team expertise, budget, timeline, scale expectations—and ask for two or three options with trade-offs. Then pick one and commit. The worst decision is no decision. And remember: for most projects, the “best” stack is the one you (and your AI agent) can work with most productively, not the one that wins benchmarks.

5.7 Chapter Takeaway

Stack decisions aren't about finding the perfect tools—they're about finding the right tools for your project, your constraints, and your timeline. AI agents are fluent in all major stacks, so the “will AI support this?” question is almost always yes. Pick what fits, commit early, and spend your energy on what makes your product unique—not on second-guessing your framework choice.

Chapter 6: GitHub: Your Code's Home Base

If your code isn't in a repository, it doesn't exist

6.1 Why Every Project Needs a Repository

I'm going to be direct about this: if you're writing code (or having AI write code for you) and it's not in a Git repository, you're flying without a safety net. A repo isn't just storage—it's a living history of every decision, every change, and every version of your project.

New to this?

Git is a system that tracks every change you make to your files. GitHub is a website that hosts your Git projects online. Think of Git as the journal, and GitHub as the library where the journal lives.

6.2 Git Fundamentals

If Git is new to you, here are the concepts that matter most:

- **Commits:** Every meaningful change gets a snapshot. Think of it like save points in a video game. If something breaks, you roll back. Seconds, not hours.
- **Branches:** Work on new features in isolation without touching your stable code. When it's ready, merge it in. When it's not, throw it away. Zero risk.
- **Pull requests:** The review process where you catch mistakes before they reach production. Even solo developers benefit from reviewing their own AI-generated code before merging.
- **History and blame:** Every line of code has an audit trail. Who changed what, when, and why. Invaluable for debugging.

6.3 GitHub as a Collaboration Platform

GitHub is more than Git hosting. It's become the collaboration layer for modern development:

- **Issues and project boards:** Track bugs, features, and tasks right next to your code. No more lost Slack messages or scattered spreadsheets.
- **Code reviews:** Pull request reviews create a culture of quality. Pro tip: even as a solo developer, reviewing your own AI-generated code in a PR is one of the best quality habits you can build.
- **Discussions and wikis:** Document your architectural decisions and project context in a searchable location tied to your codebase.
- **Your developer portfolio:** Public repositories showcase your work. Your GitHub profile is your portfolio. Hiring managers look at it.

6.4 GitHub Actions: Automation Built In

This is where GitHub goes from “nice to have” to “how did I ever work without this”:

- **Continuous integration:** Every push triggers tests, linters, and security scans. Problems get caught before they compound.
- **Continuous deployment:** Push to main and your app deploys. No manual steps, no human error.
- **Workflow automation:** Dependency updates, release notes, changelogs—all triggered by Git events. The boring stuff takes care of itself.
- **Marketplace:** Thousands of pre-built actions. Most automation you need already exists—you just plug it in.

6.5 GitHub + AI

AI-generated code comes fast—really fast. That speed is a double-edged sword without guardrails. Git is those guardrails:

- **Safety net for experimentation:** When AI writes code quickly, you need to be able to revert quickly too. Git lets you experiment aggressively.
- **Branch-per-feature:** Create a branch, prompt AI to build a feature, review carefully, merge only what passes. Main stays clean.
- **Copilot integration:** GitHub Copilot gives inline AI suggestions trained on the context of your own repository.
- **Automated PR reviews:** AI-powered review tools catch security issues and logic errors in every pull request.

6.6 Repository Best Practices

- **README as your front door:** What it does, how to set it up, how to contribute. If someone can't get running in 10 minutes, your README needs work.
- **.gitignore from day one:** Never commit secrets, API keys, or node_modules. Ask AI to generate a .gitignore for your stack—thirty seconds, prevents real disasters.
- **Branch protection rules:** Require PR reviews and passing CI before merging to main. Two minutes to configure, will save you someday.
- **Semantic commit messages:** “fixed stuff” tells you nothing. “fix: resolve race condition in payment queue” tells you everything.

6.7 Chapter Takeaway

GitHub isn't optional—it's infrastructure, right alongside your database and your hosting. Set it up first. Every time.

Chapter 7: Docker: Your Development Superpower

The end of “it works on my machine”

7.1 Why Docker Changes Everything

If you’ve ever spent a day debugging a problem that turned out to be an environment difference between your machine and the server, Docker is about to become your new best friend. Containers package your application with everything it needs—runtime, dependencies, configuration—so it behaves identically everywhere.

New to this?

Think of Docker like shipping your app in a box that includes everything it needs to run—the code, the settings, even the operating system pieces. Wherever that box goes, it works the same way.

7.2 Docker Core Concepts

- **Images:** A snapshot of your application and everything it needs to run. Like a recipe—follow the same steps, get the same result. Every time.
- **Containers:** A running instance of an image. Lightweight, isolated, and disposable. Spin one up in seconds, throw it away when you’re done.
- **Dockerfile:** The instructions for building your image. Each line is a layer, and understanding how layers work is the difference between a 50MB image and a 2GB image.
- **Docker Compose:** Run your entire stack—app, database, cache, message queue—with a single command. One YAML file, one “docker compose up.”

7.3 Docker in the CI/CD Pipeline

Docker really earns its keep when you connect it to your deployment pipeline. I learned this the hard way on a client project where staging and production had different versions of Node.js installed—spent two days tracking down a bug that didn’t exist in our code. With Docker, that entire category of problem disappears:

- **Consistent builds:** Your CI server builds the exact same image that runs in production. No environment drift. What you test is what you ship.
- **Parallel testing:** Spin up isolated containers for each test suite. No shared state, no flaky tests from leftover data.

- **Rolling deployments:** Deploy new containers alongside old ones, shift traffic gradually, roll back instantly if something goes wrong. It's how the big companies do it, and Docker makes it accessible to everyone.
- **Multi-stage builds:** Separate build dependencies from runtime dependencies. Your production image can be under 100MB while your dev environment has everything you need.

7.4 A Real Docker Win

Let me share a concrete example. Last year I had a project running Node.js, PostgreSQL, and Redis. Setting up a new developer used to mean installing three tools, configuring connection strings, and hoping their local Postgres version matched. With Docker Compose, a new developer clones the repo and runs one command: “docker compose up.” The entire environment—app, database, cache—starts in thirty seconds. Every time. That's not a minor improvement. That's a fundamentally different experience.

7.5 Getting Started with Docker (AI-Assisted)

Here's one of my favorite use cases for AI: describe your application stack to Claude and ask for a production-ready Dockerfile and docker-compose.yml. You'll get a working, best-practices configuration in minutes that would have taken hours to research and write. It's the perfect example of AI and architectural knowledge working together. See Appendix B for a ready-to-use Docker prompt template.

7.6 Chapter Takeaway

Docker is the bridge between writing code and running it reliably in the real world. For vibe coders, it's especially powerful because AI agents can generate Docker configurations that follow best practices out of the box. Enterprise-grade deployment infrastructure without needing enterprise-grade DevOps experience.

Chapter 8: Cloud Providers: Where Your Code Lives

You've built it. Now where does it run?

8.1 The Big Three: AWS, Azure, and GCP

These are the major cloud platforms. I've worked with all three extensively over fifteen years, and each one has given me both great results and genuine headaches. Here's the honest breakdown:

New to this?

Cloud providers are companies that rent you computers, storage, and networking over the internet. Instead of buying and maintaining your own servers, you pay for what you use. It's like renting an apartment instead of building a house.

- **AWS (Amazon Web Services):** The largest and most mature. If a cloud service exists, AWS probably has a version of it. The trade-off: it's complex. I once spent an entire day just configuring IAM permissions for a simple S3 bucket. Best for teams that need maximum flexibility and don't mind a steeper learning curve.
- **Azure (Microsoft):** If you're in a Microsoft shop—.NET, Active Directory, Office 365—Azure is the natural fit. The integration is genuinely seamless. I've seen Azure deployments go from concept to production in half the time of equivalent AWS setups when the team was already Microsoft-native.
- **GCP (Google Cloud):** Leading in AI/ML services and data analytics. BigQuery is outstanding, and their Kubernetes support is best-in-class. Best for AI-heavy workloads and data teams.

8.2 Developer-Friendly Alternatives

Here's what I tell people who are starting a new project and don't need the full weight of the Big Three. I actually used Vercel for a client project last year—deployed a Next.js app in under five minutes. The client couldn't believe it was live already:

- **Vercel:** Push your Next.js project to Git and you've got a live URL. Zero configuration. It almost feels like cheating.
- **Railway / Render:** Simple, affordable full-stack hosting. Database included. Great for MVPs where you want to focus on building, not configuring.
- **Cloudflare Workers:** Edge computing at global scale with pay-per-request pricing. Fast, cheap, and increasingly capable.

- **DigitalOcean / Linode:** Straightforward VPS and managed services. Predictable pricing, clean interfaces, none of the Big Three complexity.

8.3 Choosing the Right Provider

I'll save you some time—here's how I think about this decision:

- **Budget:** Start with free tiers. Don't pay for scale you don't have yet.
- **Scale expectations:** Building an MVP? Pick the simplest deployment path. You can always migrate later, and the migration is usually easier than people fear.
- **Team expertise:** The best cloud provider is the one your team can actually work with effectively. Theoretical superiority means nothing if nobody knows how to use it.
- **Regulatory requirements:** Healthcare (HIPAA), finance (SOC 2, PCI), and government projects may dictate specific providers. Know your compliance requirements early—they're hard to retrofit.

8.4 AI-Assisted Cloud Setup

One of the highest-leverage things you can do with AI: describe your application's requirements and ask for infrastructure-as-code—Terraform, CloudFormation, or Pulumi templates. You'll get production-grade infrastructure that would normally require a dedicated DevOps engineer. I've used this approach on real projects, and it works remarkably well. See Appendix B for an IaC prompt template.

8.5 Chapter Takeaway

Cloud infrastructure isn't a barrier anymore. Between developer-friendly platforms and AI-generated infrastructure code, anyone can deploy globally. Start simple. Scale when the traffic demands it. Not before.

Chapter 9: Security and Compliance: Building Trust

The chapter nobody wants to read but everybody needs to

9.1 Why Security Can't Be an Afterthought

I'm going to be straight with you: AI-generated code can introduce vulnerabilities just as easily as human-written code—sometimes more easily, because AI tends to favor simplicity over security when you don't explicitly ask for it. Always include security requirements in your prompts. "Make it work" is not enough. "Make it work securely" is the minimum.

New to this?

Security isn't just for big companies. If your app stores user emails, passwords, or payment info, you're responsible for protecting it. The basics aren't hard—they just need to be intentional.

9.2 The OWASP Top 10: Your Security Baseline

OWASP maintains a list of the most critical web application security risks. You don't need to memorize it, but you need to know these four—they're responsible for the vast majority of breaches:

- **Injection attacks:** SQL injection, XSS, command injection—still the most common vulnerabilities, decades later. Always sanitize inputs, use parameterized queries, and escape output. No exceptions.
- **Broken authentication:** Use established auth libraries—Auth0, Clerk, Supabase Auth. Please don't roll your own authentication. I've seen smart developers get this wrong. It's harder than it looks.
- **Sensitive data exposure:** Encrypt data at rest and in transit. Never log secrets, tokens, or PII. Use environment variables for configuration.
- **Security misconfiguration:** Default credentials, open ports, verbose error messages in production—these are the low-hanging fruit attackers check first. Also the easiest to fix.

9.3 Security Tools Every Developer Should Use

The good news: excellent tools automate most of this. Set them up once and they watch your back continuously. See Appendix A for the full list with links:

- **Dependabot / Snyk:** Automated dependency scanning. These tools tell you when your packages have known vulnerabilities, often before they're widely exploited.

- **SonarQube / CodeQL:** Static code analysis that catches security issues before they reach production. A security-focused reviewer that never takes a day off.
- **Trivy / Grype:** Container image scanning. If you're using Docker (Chapter 7), you need this. Trivy is free, fast, and thorough.
- **OWASP ZAP:** Dynamic application security testing—it attacks your running application to find vulnerabilities. Run it before an actual attacker does.
- **HashiCorp Vault / AWS Secrets Manager:** Centralized secrets management. Stop putting API keys in .env files and hoping nobody commits them to Git.

9.4 AI-Assisted Security Practices

Here's where AI and security knowledge combine really well:

- **Security-focused code reviews:** Ask Claude to review your code specifically for OWASP Top 10 issues. It's remarkably good at catching common mistakes that human reviewers miss.
- **Threat modeling:** Describe your architecture and ask AI to identify potential attack vectors. Used to require a dedicated security consultant.
- **Compliance documentation:** AI can generate security policies, incident response plans, and compliance docs for SOC 2, HIPAA, and GDPR. Won't replace a lawyer, but gives you a solid starting point.
- **Penetration testing prep:** Use AI to generate test cases that probe your defenses. Find the holes before someone with bad intentions does.

9.5 Chapter Takeaway

Security is a shared responsibility between you and your AI tools. AI can help you write more secure code and catch vulnerabilities faster. But it can't replace your judgment. The tools are powerful. The human in the loop is what makes them trustworthy. See Appendix C for a complete pre-deployment security checklist.

Chapter 10: The Compound Effect: Growing with AI

Whether you're writing your first prompt or your thousandth, this applies to you

10.1 The Beginner's Advantage

If you're new to software development, I want you to hear this clearly: you're entering at the best possible time. AI agents remove the steepest part of the learning curve—the syntax, the boilerplate, the configuration headaches. You get to skip straight to the interesting part: understanding systems and solving real problems.

10.2 The Experience Multiplier

Here's what I've observed working with developers at every level:

- **Beginners + AI:** You can build functional applications that would have been flat-out impossible without years of training. The AI handles syntax. You focus on what you're building and why.
- **Intermediate developers + AI:** You're moving 3–5x faster. You know enough to guide the AI effectively and spot its mistakes. This is the sweet spot where AI goes from helpful to transformative.
- **Senior developers + AI:** You become a force multiplier. Architect entire systems, review AI-generated code at expert level, manage complexity that previously required a full team.
- **Architects + AI:** This is where it gets extraordinary. Deep systems knowledge plus AI agents equals enterprise-grade applications at startup speed. The ceiling is higher than it's ever been.

10.3 Building Your AI-Augmented Workflow

After working this way for over a year, these are the habits that make the biggest difference:

- **Start every project with architecture:** Before you prompt a single line of code, sketch the system. Components, data flows, interfaces. Five minutes of thinking saves hours of rework.
- **Use the right agent for the right task:** Route architecture to Claude, visual work to Gemini, exploration to ChatGPT. Match the tool to the job.
- **Iterate aggressively:** AI makes iteration almost free. The third or fourth version is usually where the magic happens.
- **Always understand what you ship:** Never deploy code you don't understand. AI is a co-pilot, not an autopilot. Read it. Test it. Understand it. Then ship it.

- **Keep learning the fundamentals:** The more you understand about systems, networks, databases, and security, the more powerful every AI interaction becomes. Fundamentals aren't obsolete—they're amplified.

10.4 The Future of Vibe Coding

We're in the early innings. AI agents are going to keep getting more capable, more specialized, and more integrated into every phase of development. The developers who invest now in learning how to truly collaborate with AI will be the ones shaping the next generation of software. The most exciting stuff hasn't been built yet.

10.5 What to Do Right Now

Don't just close this book and think about it. Here are three things you can do today that will put everything you've read into practice:

- **Try the bakery prompt from Chapter 3:** Open Claude, paste the Golden Crust Bakery prompt, and watch a full website scaffold appear in minutes. There's no better way to feel the power of precise prompting than to experience it firsthand.
- **Set up a GitHub repository:** Create a free GitHub account if you don't have one. Initialize a repo with a README and .gitignore. Push your first commit. The whole process takes ten minutes, and now you're working like a professional from day one.
- **Run a security scan:** If you have an existing project, add Dependabot or install Trivy and scan your dependencies or Docker images. You'll probably find something—and fixing it feels great.

10.6 Final Takeaway

Vibe coding isn't about replacing skill with shortcuts. It's about amplifying whatever you bring to the table. Whether you're writing your first line of code or your millionth, AI makes you more capable than you were yesterday. But it's the architects—the ones who understand the whole system, who ask the right questions, who think before they prompt—they're the ones who will build the future. This book was my best attempt at helping you become one of them. Now go build something great.

Appendix A: Recommended Tools and Resources

A curated collection of the tools, platforms, and learning resources referenced throughout this book. Organized by category so you can quickly find what you need for each phase of development.

A.1 AI Coding Agents

These are the primary AI agents discussed in Chapter 2. Each has distinct strengths, and the most effective workflow uses multiple agents for different tasks.

Tool	Best For	Website	Pricing
Claude (Anthropic)	Architecture, infrastructure, backend, security, long-context reasoning	claude.ai	Free / Pro \$20/mo
Claude Code (CLI)	Agentic file-aware development, multi-file projects, terminal workflows	claude.ai/code	Included with Pro
Gemini (Google)	Image generation, UI/UX design, multimodal tasks, visual prototyping	gemini.google.com	Free / Advanced \$20/mo
ChatGPT (OpenAI)	Brainstorming, general knowledge, quick prototyping, broad exploration	chat.openai.com	Free / Plus \$20/mo
GitHub Copilot	Inline code suggestions, IDE integration, autocomplete	github.com/copilot	\$10/mo individual
Cursor	Full IDE with AI, codebase-aware context, multi-file editing	cursor.com	Free / Pro \$20/mo
Windsurf (Codeium)	IDE-integrated AI, code generation and understanding	windsurf.com	Free / Pro \$15/mo

A.2 Version Control and Collaboration

Tool	Description	Website
GitHub	Industry-standard code hosting with Actions CI/CD, project boards, code review, and Copilot integration	github.com
GitLab	Full DevOps platform with built-in CI/CD, container registry, and security scanning	gitlab.com
Bitbucket	Git hosting with tight Jira and Atlassian ecosystem integration	bitbucket.org

Git (CLI)	The distributed version control system that powers all of the above. Essential to understand fundamentals	git-scm.com
-----------	---	-------------

A.3 Containerization and CI/CD

Tool	Description	Website
Docker	Container platform for packaging and running applications consistently across environments	docker.com
Docker Compose	Multi-container orchestration for local development (included with Docker Desktop)	docs.docker.com/compose
GitHub Actions	CI/CD automation built into GitHub with thousands of marketplace integrations	github.com/features/actions
Jenkins	Open-source automation server with extensive plugin ecosystem for enterprise CI/CD	jenkins.io
CircleCI	Cloud-native CI/CD with fast builds, parallelism, and Docker layer caching	circleci.com

A.4 Cloud Providers and Hosting

Provider	Best For	Website
AWS	Maximum flexibility, broadest service catalog, enterprise scale	aws.amazon.com
Azure	Microsoft ecosystem integration, .NET workloads, enterprise compliance	azure.microsoft.com
GCP	AI/ML workloads, Kubernetes, BigQuery analytics	cloud.google.com
Vercel	Zero-config Next.js and frontend deployments, edge functions	vercel.com
Railway	Simple full-stack hosting, one-click databases, great for MVPs	railway.app
Render	Affordable managed hosting with auto-deploys from Git	render.com
Cloudflare Workers	Edge computing, global CDN, serverless at scale	workers.cloudflare.com
DigitalOcean	Straightforward VPS and managed services, predictable pricing	digitalocean.com
Fly.io	Global app deployment with built-in multi-region support	fly.io

A.5 Security and Compliance Tools

Tool	Description	Website
Snyk	Dependency vulnerability scanning, container security, and code analysis	snyk.io
Dependabot	Automated dependency updates and security alerts (built into GitHub)	github.com/dependabot
SonarQube	Static code analysis for bugs, vulnerabilities, and code smells	sonarqube.org
CodeQL	Semantic code analysis engine for finding zero-day vulnerabilities	codeql.github.com
Trivy	Open-source scanner for containers, filesystems, and infrastructure-as-code	trivy.dev
OWASP ZAP	Open-source dynamic application security testing (DAST)	zapproxy.org
HashiCorp Vault	Centralized secrets management, encryption, and access control	vaultproject.io
AWS Secrets Manager	Cloud-native secrets storage and rotation for AWS workloads	aws.amazon.com/secrets-manager
Checkov	Infrastructure-as-code security scanning for Terraform, CloudFormation, and Kubernetes	checkov.io

A.6 Infrastructure as Code

Tool	Description	Website
Terraform	Cloud-agnostic infrastructure provisioning with declarative HCL configuration	terraform.io
Pulumi	Infrastructure as code using real programming languages (TypeScript, Python, Go)	pulumi.com
AWS CloudFormation	Native AWS infrastructure provisioning with JSON/YAML templates	aws.amazon.com/cloudformation
AWS CDK	Define AWS infrastructure using TypeScript, Python, or Java with high-level constructs	aws.amazon.com/cdk

A.7 Learning Resources

Books, courses, and references to deepen your understanding of the concepts covered in each chapter.

- **Prompt engineering:** Anthropic's official prompt engineering documentation (docs.anthropic.com). The most practical, up-to-date guide to getting the best results from AI agents.
- **System design:** "Designing Data-Intensive Applications" by Martin Kleppmann. The definitive resource for understanding distributed systems, databases, and data pipelines.
- **Architecture patterns:** "Building Microservices" by Sam Newman and "Clean Architecture" by Robert C. Martin. Both provide frameworks for making sound architectural decisions.
- **Docker and containers:** Docker's official Getting Started guide (docs.docker.com/get-started) and "Docker Deep Dive" by Nigel Poulton.
- **Cloud infrastructure:** Each major provider's free training: AWS Skill Builder, Microsoft Learn, Google Cloud Skills Boost. All offer hands-on labs at no cost.
- **Git and GitHub:** GitHub's Skills platform (skills.github.com) offers interactive courses. "Pro Git" by Scott Chacon is available free at git-scm.com/book.
- **Security:** OWASP's Top 10 (owasp.org/Top10) and their Web Security Testing Guide. The SANS Institute's free resources for foundational security knowledge.
- **The 12-Factor App:** 12factor.net — the original methodology for building cloud-native applications. Short, practical, and still highly relevant.

Appendix B: Prompt Templates Library

Ready-to-use prompt templates organized by development phase. Copy, customize the bracketed sections, and use with your preferred AI agent. These templates follow the CONTEXT → CONSTRAINTS → OUTPUT framework from Chapter 3. For detailed walkthroughs of how these templates work in practice—including the bakery website, CPA CRM, and code refactoring examples—see Chapter 3, sections 3.5 through 3.7.

B.1 Project Scaffolding

Use when starting a new project from scratch. This template produces a complete project structure with configuration files, folder hierarchy, and boilerplate.

TEMPLATE — New Project Scaffold

Create a new [type of application: web app / API / mobile app / CLI tool] called [project name]. Purpose: [One to two sentences describing what the application does and who it serves.] Tech stack: [Framework] with [language]. Use [database] for data storage. Use [CSS framework or UI library] for styling. Project structure: Set up a clean folder structure with separation between [routes/controllers], [business logic/services], and [data access/models]. Include configuration for [linting tool], [formatter], and [test framework]. Include: A README with setup instructions, a .gitignore appropriate for this stack, environment variable configuration with a .env.example file, and a basic health check endpoint or landing page to confirm the app runs. Do not include: [Any specific things you want excluded, e.g., authentication yet, payment processing, etc.]

B.2 Code Review

Use when you want AI to review existing code for quality, security, or performance issues.

TEMPLATE — Code Review

Review the following [language] code for [security vulnerabilities / performance issues / maintainability / all of the above]. Context: This code is part of a [type of application] that [brief description of what the system does]. It handles [describe the data or user interactions this

code touches]. [Paste your code here] For each issue found, provide: 1. The specific line or section with the problem 2. Why it's a problem (what could go wrong) 3. A corrected version of the code 4. The severity level (critical / warning / suggestion) Also note any positive patterns worth keeping.

B.3 Debugging

Use when you're stuck on a bug and need systematic help identifying the root cause.

TEMPLATE — Debugging Assistance

I'm encountering a bug in my [language/framework] application. Here's the situation: Expected behavior: [What should happen] Actual behavior: [What is happening instead] Error message (if any): [Paste the full error output] Relevant code: [Paste the code section where the issue occurs] What I've already tried: - [Attempt 1 and what happened] - [Attempt 2 and what happened] Environment: [OS, language version, framework version, relevant dependencies] Please help me identify the root cause and provide a fix. Explain why the bug is occurring so I can avoid it in the future.

B.4 Architecture Design

Use when planning a new system or major feature that requires architectural decisions.

TEMPLATE — Architecture Design

I'm designing the architecture for [system/feature name]. Help me think through the design. Business requirements: - [Requirement 1] - [Requirement 2] - [Requirement 3] Constraints: - Expected users/load: [number and pattern] - Budget: [cloud budget or team size constraints] - Timeline: [when this needs to be operational] - Compliance: [any regulatory requirements: HIPAA, SOC 2, GDPR, etc.] - Existing systems: [what this needs to integrate with] Please provide: 1. Two to three architectural options with diagrams (Mermaid format) 2. Trade-offs for each option (cost, complexity, scalability, maintainability) 3. Your recommended approach and why 4. Key risks and mitigation strategies 5. A

suggested implementation roadmap

B.5 Docker Configuration

Use when containerizing an application for the first time or optimizing an existing Docker setup.

TEMPLATE — Docker Setup

Create a production-ready Docker configuration for my [language/framework] application. Application details: - Runtime: [Node.js 20 / Python 3.12 / Go 1.22 / etc.] - Build process: [npm run build / pip install / go build / etc.] - Entry point: [npm start / python app.py / ./main / etc.] - Port: [port number] - Dependencies: [database, cache, message queue, etc.] Please provide: 1. A multi-stage Dockerfile optimized for small image size and fast builds 2. A docker-compose.yml with the app and all dependencies configured 3. A .dockerignore file 4. Environment variable handling with sensible defaults 5. Health check configuration 6. Comments explaining each significant decision Security requirements: Run as non-root user, no unnecessary packages in final image, pin base image versions.

B.6 CI/CD Pipeline

Use when setting up automated testing and deployment workflows.

TEMPLATE — CI/CD Pipeline (GitHub Actions)

Create a GitHub Actions CI/CD pipeline for my [language/framework] project. Pipeline requirements: On pull request: Run linting, type checking, unit tests, and security scanning. Block merge if any step fails. On merge to main: Run full test suite, build Docker image, push to [container registry], and deploy to [staging environment]. On release tag: Deploy to production with [deployment strategy: rolling / blue-green / canary]. Project specifics: - Package manager: [npm / yarn / pnpm / pip / etc.] - Test command: [npm test / pytest / go test / etc.] - Build command: [npm run build / etc.] - Deploy target: [AWS ECS / Vercel / Railway / Kubernetes / etc.] Include: Caching for dependencies, parallel job execution where possible, Slack notifications on failure, and

environment-specific secrets handling.

B.7 Security Audit

Use when you want a thorough security review of your application or infrastructure.

TEMPLATE — Security Audit

Perform a security audit of my [type of application]. I want to identify vulnerabilities before deployment. Application overview: - Stack: [frontend framework, backend framework, database, hosting] - Authentication: [how users log in] - Data handled: [types of sensitive data: PII, financial, health, etc.] - External integrations: [third-party APIs, payment processors, etc.] [Paste relevant code, configuration files, or describe the architecture] Please evaluate against the OWASP Top 10 and check for: 1. Injection vulnerabilities (SQL, XSS, command injection) 2. Authentication and session management weaknesses 3. Sensitive data exposure (logging, error messages, API responses) 4. Security misconfigurations (default credentials, open ports, verbose errors) 5. Dependency vulnerabilities 6. API security (rate limiting, input validation, authorization) 7. Infrastructure security (Docker, cloud configuration, secrets management) For each finding, provide severity, impact, and a specific remediation with code examples.

B.8 Database Schema Design

Use when designing a new database or restructuring an existing one.

TEMPLATE — Database Schema

Design a database schema for [application/feature name]. Business entities and their relationships: - [Entity 1]: [describe fields and purpose] - [Entity 2]: [describe fields and purpose] - [Relationship]: [Entity 1] has [one/many] [Entity 2] Requirements: - Database: [PostgreSQL / MySQL / MongoDB / etc.] - ORM: [Prisma / TypeORM / Drizzle / SQLAlchemy / none] - Must support: [specific queries or access patterns that matter] - Performance: [expected row counts, read/write ratio] Please provide: 1. Complete schema with tables, columns, types,

and constraints 2. Indexes for common query patterns 3. Migration files or ORM schema definition 4. Seed data script with realistic sample records 5. Notes on any normalization decisions or intentional denormalization

Appendix C: Pre-Deployment Security Checklist

Use this checklist before every deployment. Not every item applies to every project, but reviewing each category ensures nothing critical is missed. Items marked with an asterisk (*) are non-negotiable for any application handling user data.

C.1 Authentication and Access Control

- *** Passwords hashed with bcrypt, scrypt, or Argon2** — never stored in plaintext or with weak algorithms (MD5, SHA-1).
- *** Multi-factor authentication available** — enforced for admin accounts and recommended for all users.
- *** Session tokens are cryptographically random** — with appropriate expiration and secure cookie flags (HttpOnly, Secure, SameSite).
- *** Role-based access control (RBAC) implemented** — users can only access resources appropriate to their role.
- **Rate limiting on login endpoints** — with account lockout or exponential backoff after failed attempts.
- **OAuth/SSO tokens validated server-side** — never trust client-side token claims without verification.

C.2 Input Validation and Injection Prevention

- *** All user inputs validated and sanitized server-side** — client-side validation is for UX only, never for security.
- *** Parameterized queries or ORM used for all database access** — no raw SQL string concatenation.
- *** Output encoding applied for all user-generated content** — prevents XSS in HTML, JavaScript, URL, and CSS contexts.
- **Content Security Policy (CSP) headers configured** — restricts script sources and mitigates XSS impact.
- **File upload validation** — check file type, size limits, and scan for malware. Never serve uploads from the same domain.
- **API input validation with schema enforcement** — reject malformed requests at the edge before they reach business logic.

C.3 Data Protection

- *** HTTPS enforced on all endpoints** — with TLS 1.2 or higher. HTTP requests redirect to HTTPS.
- *** Sensitive data encrypted at rest** — database encryption, encrypted file storage, and encrypted backups.
- *** No secrets in source code or Git history** — use environment variables and a secrets manager (Vault, AWS Secrets Manager).
- *** PII and sensitive fields masked in logs** — never log passwords, tokens, SSNs, credit card numbers, or API keys.
- **Database backups encrypted and tested** — verify restore procedures regularly, not just backup creation.
- **Data retention policy defined** — delete data that is no longer needed. Comply with GDPR, CCPA, or relevant regulations.

C.4 Dependency and Supply Chain Security

- *** Dependency vulnerability scanning enabled** — Dependabot, Snyk, or equivalent running on every build.
- *** Lock files committed to Git** — package-lock.json, yarn.lock, or equivalent ensures reproducible builds.
- **No unnecessary dependencies** — audit and remove packages that are unused or duplicated.
- **Base Docker images pinned to specific versions** — never use “latest” tags in production Dockerfiles.
- **Container images scanned with Trivy or Grype** — before pushing to any registry.
- **Software Bill of Materials (SBOM) generated** — for regulatory compliance and incident response traceability.

C.5 Infrastructure and Deployment

- *** Production environment variables separate from development** — no shared secrets between environments.
- *** Containers run as non-root users** — with minimal filesystem permissions and read-only root filesystems where possible.
- *** Firewall rules restrict network access** — databases and internal services not exposed to the public internet.
- **Health check endpoints configured** — for load balancers and orchestrators to detect and replace unhealthy instances.

- **Logging and monitoring active** — centralized log aggregation with alerts for error rate spikes, unusual traffic patterns, and failed authentication attempts.
- **Rollback procedure documented and tested** — you can revert to the previous deployment within minutes if a release introduces issues.

C.6 API Security

- *** Authentication required on all non-public endpoints** — with proper token validation on every request.
- *** Rate limiting enforced on all API endpoints** — with stricter limits on authentication and data-mutation routes.
- **CORS configured restrictively** — only allow origins that need access. Never use wildcard (*) in production.
- **API versioning strategy in place** — so breaking changes don't silently break client applications.
- **Error responses do not leak internal details** — no stack traces, database errors, or server paths in production error messages.
- **Request size limits configured** — prevent denial-of-service through oversized payloads.

C.7 Compliance and Documentation

- **Privacy policy published and accurate** — reflects actual data collection, storage, and sharing practices.
- **Cookie consent implemented** — if serving users in GDPR or similar jurisdictions.
- **Incident response plan documented** — who to contact, how to contain a breach, and notification timelines.
- **Architecture decision records maintained** — documenting security-relevant design choices and their rationale.
- **Access audit trail in place** — log who accessed what data and when, especially for sensitive records.
- **Regulatory requirements mapped** — HIPAA, SOC 2, PCI DSS, GDPR, or CCPA requirements identified and addressed for your specific application.