SDLC VIBE CODING HANDBOOK

The Complete Guide to Building Software with AI

From Requirements to Maintenance — A Practitioner's Framework

Jekardah Tech Review Desk • 2026 Edition • jekardah.com

SDLC
VIBE CODING
HANDBOOK

The Complete Guide to Building Software with AI
From Requirements to Maintenance — A Practitioner's Framework

Jekardah Tech Review Desk
rominur@gmail.com
2026 Edition
jekardah.com

SDLC Vibe Coding Handbook

The Complete Guide to Building Software with AI

Published by Jekardah Tech Review Desk, March 2026.

Author & Editor: rominur@gmail.com

Website: jekardah.com | Telegram: t.me/Jekardah_AI

This handbook synthesizes research from IBM, NIST, OWASP, METR, Anthropic, Google Cloud, AWS, GitHub, and the global developer community. All code examples use the TaskFlow case study — a project management tool for small teams.

Table of Contents

PART I

FOUNDATIONS

Origins, Frameworks, and the Evidence Behind Productivity Claims

Chapter 1: The Origin of Vibe Coding

1.1 The Tweet That Launched a Methodology

On February 2, 2025, Andrej Karpathy—computer scientist, OpenAI co-founder, and former Tesla AI director—posted a 185-word message on platform X that would fundamentally change how the world discusses software development. The tweet introduced the term “vibe coding” and garnered over 4.5 million views within weeks.

Karpathy described a workflow where the developer fully surrenders the coding process to AI. He was using Cursor Composer with Claude Sonnet, speaking through SuperWhisper (voice-to-text), and accepting all code changes without reading diffs. When encountering errors, he would simply copy-paste error messages without any additional commentary. The code, he admitted, grew beyond his usual comprehension.

Critically, Karpathy’s closing sentence read: “It’s not too bad for throwaway weekend projects, but still quite amusing.” He explicitly limited this approach to small, disposable projects. This disclaimer, however, was frequently omitted in the narratives that followed, as the term was adopted far beyond its original scope.

The tweet was particularly impactful because of Karpathy’s stature in the AI community. As someone who had led Tesla’s Autopilot team and co-founded OpenAI, his endorsement—even of a casual workflow—carried enormous weight. His earlier January 2023 claim that “The hottest new programming language is English” had already primed the community for this kind of paradigm shift.

> **Karpathy’s Original Definition**
>
> Vibe coding is an approach where developers “fully give in to the vibes, embrace exponentials, and forget that the code even exists.” The developer accepts all AI code without review, does not read diffs, and only copy-pastes error messages. This was explicitly intended for weekend throwaway projects, not production software.

1.2 From Tweet to Global Phenomenon

The term “vibe coding” spread with extraordinary velocity. Merriam-Webster listed it as slang by March 2025. Collins English Dictionary later selected it as their Word of the Year for 2025. Y Combinator’s Jared Friedman revealed that 25% of startups in YC’s Winter 2025 batch had codebases that were 95% AI-generated. An academic paper on arXiv compared the tweet’s cultural influence to Knuth’s literate programming manifestos.

The adoption wave progressed through three distinct phases. The first phase (February–April 2025) was pure euphoria. Developers from diverse backgrounds experimented with AI coding tools and shared impressive results on social media. Productivity claims of 5x, 10x, even 100x were common. The second phase (May–August 2025) brought the reality check. Reports of security vulnerabilities, unmaintainable code, and catastrophic production failures began to accumulate. The third phase (September 2025–present) represents maturation, as the developer community began developing structured frameworks that preserve vibe coding's speed advantages while eliminating its weaknesses.

Phase	Period	Sentiment	Key Events
1. Euphoria	Feb–Apr 2025	Excited, experimental	YC 25% AI codebases, viral demos
2. Reality Check	May–Aug 2025	Skeptical, cautious	Security breaches, METR study
3. Maturation	Sep 2025–now	Structured, pragmatic	SDD, AGENTS.md, frameworks

1.3 Karpathy's One-Year Retrospective (February 2026)

Exactly one year after his original tweet, Karpathy posted a retrospective that provided important perspective. He acknowledged that the tweet "minted a fitting name at the right moment for something a lot of people were feeling at the same time." He also proposed "agentic engineering" as a more mature successor term for structured AI-assisted development.

Notably, Karpathy's latest project—Nanochat—was written manually without AI assistance, because AI agents "weren't helpful enough" for that particular project. This demonstrated that even the coiner of vibe coding understood its limitations and recognized that some problems still require human craftsmanship.

1.4 The Critics: Key Voices

Not everyone welcomed vibe coding enthusiastically. Andrew Ng, one of the most influential figures in AI, called the term "unfortunate" and "misleading" at the LangChain Interrupt conference in May 2025. He argued that AI-assisted coding is "a deeply intellectual exercise," not mere vibes. Simon Willison, Django co-creator, drew a critical definitional line: "If an LLM wrote every line of your code, but you've reviewed, tested, and understood it all, that's not vibe coding."

Addy Osmani, well-known Google Chrome engineer and author of Beyond Vibe Coding (O'Reilly, 2025), took a more nuanced position. He defined an "AI-Assisted Development Spectrum" ranging from pure vibe coding (no review, no understanding) to fully structured AI-assisted engineering (specification-driven,

reviewed, tested). He also identified “The 70% Problem”: AI gets you 70% of the way quickly, but the remaining 30% requires deep engineering expertise.

Critic	Position	Core Argument
Andrej Karpathy	Creator, evolved to “agentic engineering”	Useful for throwaway projects; right name at right time
Andrew Ng	Critical of name and concept	AI-assisted coding is an intellectual exercise, not vibes
Simon Willison	Strict definitional boundary	Reviewed AI code is not vibe coding
Addy Osmani	Structured evolution	Defined the AI-Assisted Development Spectrum
Casey West	Governance-focused	Proposed Agentic Delivery Lifecycle (ADLC)

Chapter 2: SDLC Frameworks and Vibe Coding

2.1 How Vibe Coding Maps to Traditional SDLC

Vibe coding does not replace the Software Development Life Cycle—it compresses and reshapes its phases. The development phase that once took weeks can collapse to minutes with AI assistance. However, the upstream phases (requirements, design) and downstream phases (testing, security, maintenance) become more critical than ever before.

Every major published vibe coding guide maps the approach against traditional frameworks. Understanding these relationships is essential for building an effective methodology that captures AI's speed without sacrificing quality.

Waterfall Model

Waterfall provides the historical baseline from which vibe coding departs. Its sequential phases (requirements → design → development → testing → deployment → maintenance) still appear in nearly all vibe coding SDLC guides. The key difference: the development phase undergoes dramatic compression. Interestingly, counter-frameworks like Spec-Driven Development actually reinstate Waterfall-like sequential discipline (spec → design → tasks → implementation) as a corrective to vibe coding's chaos.

Agile/Scrum

Agile is the most directly referenced traditional framework in the vibe coding ecosystem. IBM explicitly states that vibe coding aligns with principles of fast-prototyping, iterative development, and cyclical feedback loops. The BMAD Method (Breakthrough Method for Agile AI-Driven Development) maps 21 AI agents to Scrum roles—Product Owner, Scrum Master, Developer, and QA—creating a fully simulated Agile team. AWS's AI-Driven Development Lifecycle reimagines Agile terminology entirely: "sprints" become "bolts" (measured in hours, not weeks), and "epics" become "Units of Work."

DevOps/CI/CD

DevOps is heavily integrated into modern vibe coding workflows. Google Cloud coined "vibe deploying" for single-prompt deployment to Cloud Run. Multiple frameworks envision AI agents managing entire CI/CD pipelines autonomously, from test execution to production deployment to rollback decisions.

V-Model

The V-Model's verification/validation pairing appears implicitly in the Vibe Coding Framework's core principle of "Verification Before Trust" and in Amazon Kiro's spec-to-verification pipeline. Every specification generates corresponding verification criteria, mirroring the V-Model's fundamental structure.

Evolution: Traditional SDLC → Vibe Coding → Structured AI-Assisted

```
┌──────────────────────────────────────────────────────────────┐
│  TRADITIONAL          VIBE CODING (2025)    STRUCTURED (2026)  │
├──────────────────────────────────────────────────────────────┤
│  Requirements: Weeks   Requirements: SKIP    Requirements: Hours │
│  Design: Weeks         Design: SKIP          Design: Hours (AI)  │
│  Development: Months   Development: Hours    Development: Hours  │
│  Testing: Weeks        Testing: Minimal      Testing: AI-gen'd   │
│  Deploy: Days          Deploy: Minutes       Deploy: CI/CD auto  │
│  Maintain: Ongoing     Maintain: CRISIS      Maintain: Monitored │
└──────────────────────────────────────────────────────────────┘
```

2.2 Emerging AI-Native Frameworks (2024–2026)

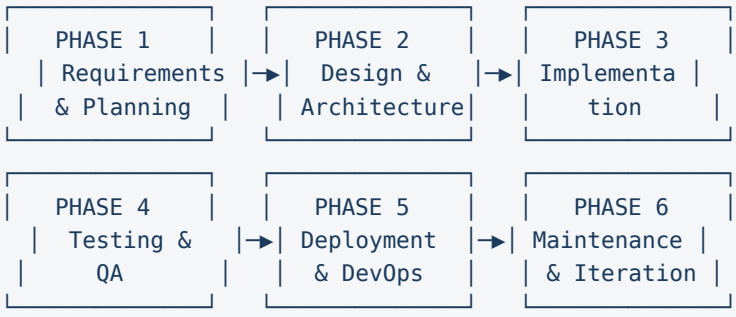
Over the past two years, several entirely new frameworks have emerged, purpose-built for AI-assisted development. Each offers a distinct approach to the same fundamental challenge: how to harness AI's power without sacrificing quality, security, and maintainability.

Framework	Proponent	Year	Core Innovation
AI-DLC	AWS	2025	"Bolts" replace sprints; time measured in hours, not weeks
Agentic SDLC	PwC, EPAM	2025–2026	Autonomous agent teams across all SDLC phases
BMAD Method	Open Source	2025	21 specialized AI agents simulating full Scrum team
Spec-Driven Dev	GitHub, Amazon Kiro	2024–2025	Specification as executable artifact; structured pipeline
Structured Vibe	Microsoft	2025	GitHub Copilot agents assigned Issues like team members
ADLC	Casey West	2025	Five-phase governance model for autonomous AI agents

2.3 The Vibe Coding SDLC: 6 Phases with AI

This handbook uses a Vibe Coding SDLC framework consisting of six phases, with AI acting as co-pilot at every step. The framework combines Waterfall discipline (sequential phases), Agile speed (rapid iteration), and DevOps automation (CI/CD). The key structural insight consistent across all guides: vibe coding compresses the development phase while making upstream and downstream phases more critical than ever. As Mohamed A. Hassan wrote, "The SDLC bottleneck was never coding."

Vibe Coding SDLC — 6 Phases



AI = Co-pilot at every phase. Human = Decision maker.

Phase	Input	Output	AI Role	Est. Time
1. Requirements	Raw idea	PRD, CLAUDE.md, User Stories	Brainstorming, validation	~3 hrs
2. Design	PRD	ERD, API Design, Wireframes	Schema generation, review	~2.5 hrs
3. Implementation	Design docs	Working features (vertical slices)	Code generation, pair prog.	~15–25 hrs
4. Testing	Working code	Test suites, security report	Test generation (70–80%)	~4–6 hrs
5. Deployment	Tested code	Live production app	CI/CD setup, config	~2–3 hrs
6. Maintenance	Live app	Bug fixes, new features	Debugging, refactoring	Ongoing

Chapter 3: Evidence — Productivity Claims vs Reality

3.1 The “100x More Expensive” Myth

The claim that fixing bugs in production costs 100x more than in the requirements phase is one of the most frequently cited “facts” in software engineering. This claim is typically attributed to an “IBM Systems Sciences Institute study” and NIST data. However, a forensic investigation by Laurent Bossavit in his book The Leprechauns of Software Engineering reveals that this claim lacks solid empirical foundation.

The IBM Systems Sciences Institute was not a research body but an internal corporate training program in Los Angeles, operating from approximately 1967 to 1982. The cited source ([IBM81]) consists of course notes, not a published study. The specific ratios (1:6.5:15:60–100) trace through Roger Pressman’s 1987 textbook back to Barry Boehm’s 1976 paper, where Bossavit found the underlying data was relabeled from studies measuring computer runtime costs (not SDLC phase costs) and partially derived from student projects. A 2016 paper (arXiv:1609.04886) attempting to verify the claim found weak or no statistical support.

> **Warning About the IBM/NIST Claim**
>
> The principle that bugs cost more when found later is directionally plausible. However, the specific 100x figure is effectively an urban legend without strong empirical basis. The real NIST study (2002) does exist and found software bugs cost the US economy $59.5 billion annually, but its phase-cost data explicitly references Boehm’s earlier work—making it derivative rather than independently validated.

3.2 AI Coding Productivity: What the Research Actually Shows

The “3–5x productivity improvement” claim from AI coding tools is one of vibe coding’s primary selling points. However, no single rigorous study supports this figure. The actual evidence landscape paints a far more nuanced picture, with some studies showing improvements and others showing AI tools actually slowing developers down.

Study	Year	Method	Finding	Caveat
GitHub/ Microsoft	2023	RCT	55.8% faster (~1.5x)	Single bounded task only (HTTP server in JS), funded by GitHub
Google Internal	2024	RCT, 96 engineers	21% faster	Specific task (logging), internal

Study	Year	Method	Finding	Caveat
McKinsey	2023	Mixed methods	Up to 2x faster	Complex tasks showed small or no gains
METR	Jul 2025	RCT, 16 devs, 246 tasks	19% SLOWER with AI	Most rigorous independent study to date
Google DORA	2024	Survey + metrics	-1.5% speed per 25% AI adoption	System stability dropped 7.2% additionally

Critical Finding: The METR Study

The METR study is the most rigorous randomized controlled trial on AI coding productivity conducted to date. 16 experienced open-source developers worked on 246 real-world tasks. Result: developers were 19% SLOWER with AI tools. Even more striking: developers predicted AI would speed them up by 24% and believed afterward that AI had helped by 20%, creating a 39-percentage-point perception-reality gap. This suggests developers systematically overestimate AI's contribution.

3.3 Security Vulnerabilities: Alarming Data

Security represents the area where AI-generated code shows the most serious weaknesses. The data from multiple independent sources converges on a deeply concerning picture that any responsible practitioner must take into account.

Metric	Figure	Source
AI code containing vulnerabilities	45%	Veracode 2025 State of Software Security
Java AI-generated failure rate	70%	Veracode 2025
Cross-site scripting failure rate	86%	Veracode 2025
Lovable apps with critical flaws	10.3% (170 of 1,645)	CVE-2025-48757
Vulnerabilities in vibe-coded apps	2,000+	Escape.tech scan of 5,600 public apps
Exposed secrets in vibe-coded apps	400+	Escape.tech
Images leaked via open Firebase	72,000 (incl. 13K gov IDs)	Tea App breach, July 2025

3.4 Code Quality Degradation

GitClear's analysis of 211 million lines of code revealed troubling trends since the mass adoption of AI coding tools. Code refactoring dropped from 25% of changed lines (2021) to under 10% (2024). Code duplication increased approximately 4x. Code churn (code written then immediately deleted or replaced) nearly doubled. CodeRabbit's December 2025 study of 470 GitHub PRs found that AI co-authored code contained 1.7x more major issues and 2.74x higher security vulnerabilities than human-written code.

Fast Company's September 2025 investigation found that 16 of 18 CTOs surveyed had experienced production disasters from AI-generated code. Analysts project $1.5 trillion in technical debt by 2027 driven by vibe coding approaches, with an estimated 8,000+ startups needing complete rebuilds.

3.5 Implications for This Handbook

The data above does not mean AI coding tools are useless—far from it. These tools are extraordinarily powerful when used correctly. The implications for this handbook are:

1. AI productivity gains are strongest for repetitive, boilerplate tasks—not for architecture design or complex debugging. Use AI to accelerate execution, not to replace thinking.

2. Every line of AI-generated code MUST be reviewed by a human. Karpathy's original "Accept All" pattern must never be used for production code.

3. Security scanning is not optional—it is mandatory. With 45% of AI code containing vulnerabilities, automated scanning in the CI/CD pipeline is the bare minimum.

4. Testing effort should INCREASE, not decrease, when using AI. AI can generate 70–80% of test boilerplate, but edge cases and business logic remain human responsibility.

PART II

REQUIREMENTS & PLANNING

The Foundation That Determines Everything

Chapter 4: From Brain Dump to Structured Vision

4.1 Why Planning Remains the Most Important Phase

Although the specific “100x more expensive” claim is empirically questionable (see Chapter 3), the principle that fixing problems earlier is far cheaper and more efficient remains valid both logically and practically. In the context of vibe coding, planning becomes even MORE important: AI coding assistants will happily write thousands of lines of wrong code if given bad requirements. AI accelerates execution but does not correct direction. Garbage requirements in, garbage code out—only faster.

In Vibe Coding SDLC, planning does not mean creating 100-page documents. It means four things: having a clear vision of what you are building, writing a CLAUDE.md that gives AI complete context, defining testable user stories, and choosing the right tech stack. Total time: 2–4 hours for a medium project. The result: a coding phase that is 3–5x faster because AI has the correct context from the start.

4.2 The Three-Step Brain Dump Process

1. Write Raw Ideas (5 minutes) — Brain dump everything in your head. No structure needed. Random bullet points are fine. The goal is to get everything from brain to text.

2. AI-Assisted Structuring (15 minutes) — Paste your brain dump to AI with a structured prompt. AI will validate, organize, and identify gaps. This becomes the material for your PRD.

3. Define MVP Scope (10 minutes) — From all features, select 3–5 core features that MUST exist in version one. The rest goes to the backlog. Principle: if everything is important, nothing is important.

4.3 Prompt Pattern: Brain Dump to Vision

```
## Prompt for AI-Assisted Brain Dump Structuring

I want to build [brief description].
Features I have in mind: [paste raw bullet points]

Please help me:
1. Validate if this idea makes sense (market, competitors)
2. Identify core features vs nice-to-have
3. Define target user persona and anti-persona
4. Write a one-paragraph vision statement
```

```
5. List 3 key differentiators from competitors
6. Suggest MVP scope (minimum viable product)
```

4.4 Case Study: TaskFlow Vision

> **Vision Statement — TaskFlow**
>
> TaskFlow is a project management tool for small teams (5–15 people) that combines a task board, time tracking, and automated reporting in one simple interface—without Jira's complexity or Trello's limitations. MVP Scope: Task Board + Time Tracking + Basic Reports. Phase 2: Slack integration, advanced analytics.

Chapter 5: Product Requirements Document (PRD)

5.1 What a PRD Is and Why It Matters

A PRD is the “contract” between you (product owner) and the developer (in this case, AI + you). It defines what is being built, for whom, and how to measure success. In Vibe Coding, the PRD also serves as the primary context given to AI during coding—the clearer the PRD, the better the code AI produces. Without a PRD, AI lacks context about project scope and will make incorrect assumptions about target users, over-engineer architecture, or add unrequested features.

5.2 PRD Structure for Vibe Coding

Section	Content	Length
1. Vision & Goals	One paragraph explaining what and why	1 paragraph
2. Target Users	Primary persona, secondary, and anti-persona	3–5 bullets
3. Core Features (MVP)	Features that MUST exist in version one	3–5 features with descriptions
4. Non-functional Reqs	Performance, scalability, security, mobile	Specification table
5. Success Metrics	Measurable KPIs for evaluating success	3–5 KPIs
6. Out of Scope	Explicitly NOT included in this version	Phase 2+ feature list

5.3 Complete PRD Template: TaskFlow

```
# PRD: TaskFlow — Project Management for Small Teams

## 1. Vision & Goals
TaskFlow is a simple PM tool for small teams (5-15 people)
combining task board, time tracking, and automated reporting
in a single interface.

## 2. Target Users
Primary: Startup teams (5-15 people), tech & non-tech
Secondary: Freelancers managing multiple clients
Anti-persona: Enterprise teams (100+ people) — NOT our target

## 3. Core Features (MVP)
F1: Kanban Board — Drag-drop task cards, customizable columns
F2: Time Tracking — Start/stop timer per task, daily log
F3: Weekly Report — Auto-generated, emailed to team lead
F4: User Auth — Email/password + Google OAuth
F5: Team Management — Invite, roles (admin/member)
```

```
## 4. Non-functional Requirements
Performance: Page load < 2s, API response < 200ms
Scalability: Support 1,000 concurrent users
Security: OWASP Top 10 compliance, data encryption at rest
Mobile: Responsive web (not native app for MVP)

## 5. Success Metrics
KPI 1: 100 teams sign up in first month
KPI 2: 60% weekly active rate
KPI 3: NPS > 40

## 6. Out of Scope (Phase 2+)
- Slack/Discord integration
- Advanced analytics dashboard
- Native mobile app
- AI task suggestions
- Gantt chart view
```

Chapter 6: CLAUDE.md — Your Project's DNA

6.1 The Most Important File in Vibe Coding

CLAUDE.md (or its variants AGENTS.md, .cursorrules) is a file that AI coding assistants read at the start of every session. It contains your tech stack, coding conventions, project structure, database schema, API patterns, and rules that MUST be followed. Without this file, AI will guess—and frequently guess wrong. With this file, AI becomes a developer who deeply understands your project's context, producing consistent, mergeable code.

6.2 History and Standards

CLAUDE.md was introduced with Claude Code's launch in February 2025 (GA May 2025). It supports hierarchical nesting where subdirectory files override parent instructions. Anthropic engineer Boris Cherny emphasized keeping instructions minimal—too many instructions degrade instruction-following quality uniformly.

AGENTS.md emerged as an open, cross-platform specification from a consortium including OpenAI, Sourcegraph, Google, and Cursor. Released August 2025 and donated to the Linux Foundation's Agentic AI Foundation in December 2025, it has been adopted by over 60,000 open-source projects and works across Codex, Cursor, Copilot, Gemini CLI, and other tools.

File	Tool	Format	Hierarchy	Adoption
CLAUDE.md	Claude Code	Free-form Markdown	Yes (subdirectory override)	Anthropic ecosystem
AGENTS.md	Multi-tool	Structured markdown	Yes	60,000+ OSS projects
.cursorrules (legacy)	Cursor IDE	Plain text	No	Deprecated Feb 2025
.cursor/rules/*.mdc	Cursor v0.46+	MDC format	Yes (4 rule types)	Cursor ecosystem

6.3 Complete CLAUDE.md Template

```
# CLAUDE.md — TaskFlow Project

## Project Overview
TaskFlow: PM tool for small teams. Kanban + Time Tracking + Reports.
Target: MVP launch, responsive web app.
```

```
## Tech Stack
Frontend: Next.js 15 (App Router), TypeScript, Tailwind CSS v4
Backend: Next.js API Routes + Server Actions
Database: PostgreSQL 16 via Prisma ORM
Auth: NextAuth.js v5 (Google OAuth + credentials)
Hosting: Vercel (frontend) + Neon (PostgreSQL serverless)
Testing: Vitest + Playwright (E2E)

## Coding Rules (NON-NEGOTIABLE)
1. ALWAYS use TypeScript strict mode
2. ALWAYS handle errors — no unhandled promises
3. ALWAYS validate input with Zod schemas
4. ALWAYS use parameterized queries (no raw SQL)
5. NEVER hardcode secrets — use env vars
6. NEVER commit to main directly — always PR
7. Every function MUST have JSDoc comment
8. Max 50 lines per function, max 200 lines per file

## File Structure
src/app/          # Next.js App Router pages
src/components/   # Reusable UI components
src/lib/          # Utilities, helpers, configs
src/server/       # Server-side logic, API handlers
prisma/           # Database schema & migrations
tests/            # Unit & integration tests

## API Patterns
- RESTful naming: /api/tasks, /api/tasks/[id]
- Always return { data, error, meta } format
- Use Server Actions for mutations
- Rate limit: 100 req/min per user

## Security Rules
- Zod validation on EVERY endpoint
- CSRF protection via NextAuth
- Helmet headers in next.config.js
- SQL injection prevention via Prisma
- XSS prevention via React escaping
```

Anti-pattern: Write Once, Never Update

CLAUDE.md is not a static document. Every time there is an architecture change, new dependency, or new convention, CLAUDE.md must be updated. An outdated file is worse than no file at all because it gives AI incorrect information, producing code that conflicts with the actual project state.

Chapter 7: User Stories & Acceptance Criteria

7.1 Origin of the User Story Format

The user story format “As a [user], I want [action] so that [benefit]” was created in 2001 by the team at Connextra, a UK company—attributed primarily to Rachel Davies. Kent Beck had introduced the broader concept at the Chrysler C3 project in 1997, and Ron Jeffries named the three aspects (Card, Conversation, Confirmation) in 2001. Mike Cohn’s 2004 book User Stories Applied became the canonical reference, cited by Martin Fowler as the industry standard.

7.2 Why User Stories Are Critical for AI

In vibe coding, user stories serve a dual purpose. First, they define “done”—without clear acceptance criteria, neither you nor AI knows when to stop. Second, they function as highly effective prompts: you can feed a user story plus its acceptance criteria directly to AI and request implementation. Structured input yields structured output.

7.3 TaskFlow User Stories

ID	Story	Key Acceptance Criteria
US-001	As a team member, I want to create a new task so I can track my work	Click “+” to add; form with title/description/assignee/due date; appears in correct column; real-time sync; title validation 3–100 chars
US-002	As a team member, I want to drag a task between columns to update status	Desktop + mobile drag-drop; column highlights on hover; immediate DB update; real-time sync; 5-second undo
US-003	As a team member, I want to start/stop a timer to track time per task	Play/pause per task; HH:MM:SS display; only one timer per user; log saved on stop; daily total in sidebar
US-004	As a team lead, I want auto-generated weekly reports to track progress	Auto-generated Mondays; tasks completed + hours/member + blockers; emailed to lead; PDF export
US-005	As a user, I want to sign in with Google so I don’t need another password	Google OAuth button; auto-create on first login; profile from Google; protected routes redirect

Chapter 8: Tech Stack Selection & Scaffolding

8.1 The AI Compatibility Dimension

In vibe coding, tech stack selection has an additional dimension compared to traditional development: how well AI coding assistants support the stack. AI tools perform significantly better with popular frameworks that have extensive training data. Next.js, React, Python/FastAPI, and PostgreSQL receive excellent support from all major AI tools. Niche or very new frameworks may produce less accurate code.

8.2 Recommended Stack Matrix

Category	Recommendation	Alternatives	AI Support	Rationale
Frontend	Next.js 15 + TypeScript	Nuxt, SvelteKit, Remix	Excellent	SSR, SEO, largest ecosystem
Styling	Tailwind CSS v4	CSS Modules, styled-comp.	Excellent	AI generates Tailwind extremely well
Backend	Next.js API Routes/Actions	FastAPI, Express, Hono	Excellent	Fullstack in one repo, simple deploy
Database	PostgreSQL + Prisma ORM	MySQL, MongoDB, Drizzle	Excellent	Type-safe, migrations, AI-friendly
Auth	NextAuth.js v5	Clerk, Supabase Auth	Good	Free, flexible, well-documented
Hosting	Vercel + Neon	Railway, Fly.io, AWS	N/A	Zero-config deploy, serverless, free tier
Testing	Vitest + Playwright	Jest, Cypress	Good	Fast, modern, ESM-native

8.3 Project Scaffolding

```
# 1. Create Next.js project
$ npx create-next-app@latest taskflow \
    --typescript --tailwind --app --eslint
$ cd taskflow

# 2. Install dependencies
$ npm install prisma @prisma/client next-auth@beta zod
$ npm install -D vitest @playwright/test

# 3. Initialize Prisma
$ npx prisma init --datasource-provider postgresql
```

```
# 4. Create CLAUDE.md (copy template from Chapter 6)
$ touch CLAUDE.md

# 5. Create folder structure
$ mkdir -p src/{components,lib,server,tests}

# 6. Initialize Git + first commit
$ git init && git add -A
$ git commit -m "feat: initial project scaffolding"

# 7. Push to GitHub
$ gh repo create taskflow --private --push
```

8.4 Sprint Estimation with AI

User Story	Subtasks	Vibe Est.	Traditional Est.	Savings
US-001: Create Task	UI form + API + DB + validation	2–3 hrs	8–12 hrs	~4x
US-002: Drag-Drop	DnD library + state + API + real-time	3–5 hrs	16–24 hrs	~4x
US-003: Time Tracking	Timer UI + start/stop API + log	2–4 hrs	10–16 hrs	~4x
US-004: Weekly Report	Query + PDF generation + email	3–5 hrs	12–20 hrs	~4x
US-005: Auth	NextAuth config + Google + routes	1–2 hrs	6–10 hrs	~5x
TOTAL MVP		11–19 hrs	52–82 hrs	~4x faster

Caveat on 4x Estimation

The 3–5x speed estimates assume the developer is already familiar with the stack, CLAUDE.md is comprehensive, and requirements are clear. The METR study showed experienced developers were actually 19% slower with AI on familiar codebases. For beginners, these estimates may be overly optimistic. Use them as guidelines, not promises.

8.5 Requirements & Planning Checklist

#	Item	Deliverable	Time
1	Clear vision statement	Section in PRD	15 min
2	Target user defined	Persona + anti-persona	10 min
3	MVP scope defined	Feature list + out-of-scope	20 min

#	Item	Deliverable	Time
4	PRD complete	PRD.md file	30 min
5	CLAUDE.md written	CLAUDE.md at repo root	30 min
6	User stories with AC	user-stories.md	45 min
7	Tech stack decided	Documented in CLAUDE.md	15 min
8	Project scaffolded	GitHub repo + initial commit	10 min
9	Sprint plan ready	Task list with estimates	15 min

PART III

DESIGN & ARCHITECTURE

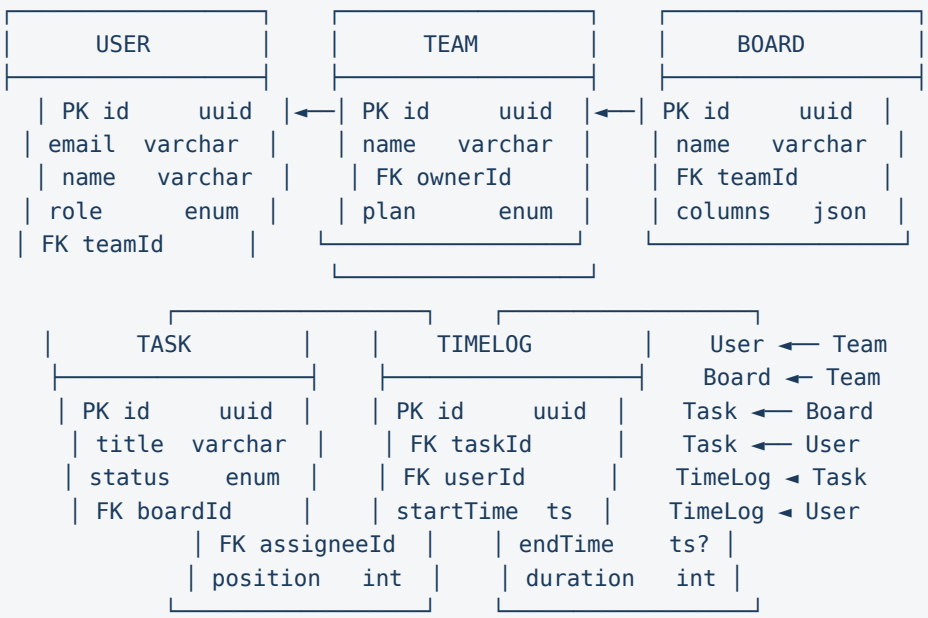
The Blueprint Before Building

Chapter 9: Database Schema Design (ERD)

9.1 Entity Relationship Diagram

The database schema is the data layer foundation of any application. In vibe coding, a well-designed schema enables AI to generate consistent queries and APIs. Defining the schema upfront in the PRD and CLAUDE.md ensures that every AI-generated database operation is correct from the start. Below is the complete ERD for TaskFlow:

Entity Relationship Diagram — TaskFlow



9.2 Prisma Schema

```
// prisma/schema.prisma
generator client { provider = "prisma-client-js" }
datasource db { provider = "postgresql"; url = env("DATABASE_URL") }

enum Role { ADMIN  MEMBER }
enum TaskStatus { TODO  IN_PROGRESS  REVIEW  DONE }

model User {
  id        String    @id @default(uuid())
  email     String    @unique
  name      String
  avatarUrl String?
  role      Role      @default(MEMBER)
  teamId    String?
  team      Team?     @relation(fields: [teamId], references: [id])
```

```
  tasks     Task[]    @relation("assignee")
  timeLogs  TimeLog[]
  createdAt DateTime  @default(now())
  updatedAt DateTime  @updatedAt
}

model Team {
  id       String  @id @default(uuid())
  name     String
  ownerId  String
  members  User[]
  boards   Board[]
  createdAt DateTime @default(now())
}

model Board {
  id      String @id @default(uuid())
  name    String
  teamId  String
  team    Team   @relation(fields: [teamId], references: [id])
  columns Json   @default("[\"Todo\",\"In Progress\",\"Done\"]")
  tasks   Task[]
}

model Task {
  id          String     @id @default(uuid())
  title       String
  description String?
  status      TaskStatus @default(TODO)
  position    Int        @default(0)
  assigneeId  String?
  assignee    User?      @relation("assignee", fields: [assigneeId],
references: [id])
  boardId     String
  board       Board      @relation(fields: [boardId], references: [id])
  timeLogs    TimeLog[]
  dueDate     DateTime?
  createdAt   DateTime   @default(now())
  updatedAt   DateTime   @updatedAt
  @@index([boardId, status])
  @@index([assigneeId])
}

model TimeLog {
  id        String   @id @default(uuid())
  taskId    String
  task      Task     @relation(fields: [taskId], references: [id])
  userId    String
```

```
  user      User     @relation(fields: [userId], references: [id])
  startTime DateTime @default(now())
  endTime   DateTime?
  duration  Int?     // seconds, computed on stop
  @@index([taskId])
  @@index([userId, startTime])
}
```

Chapter 10: API Design, UI Wireframes & Security

10.1 RESTful API Endpoint Design

Method	Endpoint	Description	Auth	Response
GET	/api/boards	List boards in team	Required	{data: Board[]}
GET	/api/boards/[id]	Get board with tasks	Required	{data: Board + Task[]}
POST	/api/tasks	Create new task	Required	{data: Task}
PATCH	/api/tasks/[id]	Update task	Required	{data: Task}
DELETE	/api/tasks/[id]	Delete task	Admin only	{success: true}
POST	/api/time/start	Start timer	Required	{data: TimeLog}
POST	/api/time/stop	Stop running timer	Required	{data: TimeLog}
GET	/api/reports/weekly	Weekly summary	Admin only	{data: WeeklyReport}

10.2 Standard API Response Format

```
// src/lib/api-response.ts
type ApiResponse<T> = {
  data: T | null;
  error: string | null;
  meta?: { page?: number; total?: number; timestamp: string; };
};
```

10.3 Component Architecture: Atomic Design

Atomic Design, introduced by Brad Frost in June 2013 and expanded into a book in January 2016, provides a five-level hierarchy (atoms → molecules → organisms → templates → pages) that maps naturally to how AI agents generate and organize UI components. This shared vocabulary enables coherent multi-file component generation.

Level	TaskFlow Examples	Reusability
Atoms	Button, Input, Avatar, Badge, Spinner	100% generic, used everywhere
Molecules	TaskCard, TimeTracker, StatCard, UserPill	Domain-specific but reusable

Level	TaskFlow Examples	Reusability
Organisms	KanbanColumn, Sidebar, ReportTable, TaskDetailPanel	Page sections, composed from molecules
Templates	DashboardLayout, AuthLayout, BoardLayout	Page structures without data
Pages	/dashboard, /boards/[id], /reports, /settings	Unique, data-connected

10.4 Security Architecture

Layer	Implementation	Location
Authentication	NextAuth.js v5: Google OAuth + email/password	Middleware + API routes
Authorization	Role-based: ADMIN can delete, MEMBER edit own only	API route handlers
Input Validation	Zod schemas on EVERY endpoint and form	Server Actions + API routes
SQL Injection	Prisma ORM (parameterized queries by default)	All database queries
XSS Prevention	React default escaping + DOMPurify for rich text	All rendered content
CSRF	NextAuth built-in CSRF tokens	All mutations
Rate Limiting	100 req/min per user, 10 req/min for auth	Middleware
Security Headers	Helmet: X-Frame-Options, CSP, HSTS	next.config.js
Secrets Management	Environment variables only; .env.local gitignored	Vercel env config

10.5 Architecture Decision Records (ADR)

Architecture Decision Records were formalized by Michael Nygard in his blog post “Documenting Architecture Decisions” on November 15, 2011. ThoughtWorks placed ADRs in its “Adopt” radar category in 2018. Each ADR documents WHY a decision was made, not just WHAT was decided—critical for future developers (human or AI) who need to understand context.

```
# ADR-001: Next.js 15 over Remix
Status: Accepted | Date: 2026-03-15

Context: Need React-based fullstack framework.
Options: Next.js 15, Remix, SvelteKit.
```

```
Decision: Next.js 15 with App Router.
Reasons:
  1. Largest ecosystem and community
  2. Best AI coding assistant support (most training data)
  3. Vercel deployment is zero-config
  4. Server Actions reduce API boilerplate
  5. Team has prior Next.js experience

Consequences:
  - Locked to React ecosystem (acceptable)
  - Vercel hosting recommended (acceptable for MVP)
```

Chapter 10B: State Management Strategy

10B.1 Server State vs Client State

In Next.js App Router applications, state management requires a clear distinction between server state (data from the database, fetched via Server Components and Server Actions) and client state (UI interactions like open modals, drag positions, form inputs). Confusing these two categories is one of the most common architectural mistakes in modern React applications, and AI coding tools frequently make this error when not given explicit guidance in CLAUDE.md.

Server state should be the single source of truth for all persistent data. In TaskFlow, this means task titles, statuses, time logs, and team membership are always fetched from the database via Server Components or Server Actions. Changes to server state always go through the server (via Server Actions or API routes), never through client-side state management alone.

Client state handles ephemeral UI concerns: whether a modal is open, which task card is being dragged, what text has been typed into a form field but not yet submitted, which column is highlighted during a drag-over event. This state lives in React's useState or useReducer hooks and disappears when the component unmounts.

10B.2 The State Decision Matrix

Data Type	Example in TaskFlow	Where It Lives	Update Mechanism
Persistent entity data	Task title, status, assignee	Server (PostgreSQL via Prisma)	Server Action → revalidatePath
User session/auth	Current user, role, team	Server (NextAuth session)	getServerSession()
UI interaction state	Modal open, drag position	Client (useState)	Event handlers
Form input state	Title field, description field	Client (useState or useForm)	onChange handlers
Optimistic updates	Task moved to new column	Client temporarily, then server	useState + Server Action
Derived/ computed	Task count per column	Computed from server data	No separate state needed
Real-time sync	Other users' changes	Server → Client (WebSocket/SSE)	Subscription + state update

10B.3 Optimistic Updates Pattern

Optimistic updates provide instant UI feedback while server operations complete in the background. This pattern is critical for drag-and-drop interactions in the Kanban board, where users expect immediate visual response. The implementation follows a three-step sequence: update local state immediately, send the mutation to the server, and revert local state if the server operation fails.

```
// Optimistic update pattern for task status change
"use client";
import { useState, useOptimistic } from "react";
import { updateTaskStatus } from "@/server/tasks";

function KanbanBoard({ tasks: initialTasks }) {
  const [optimisticTasks, setOptimistic] = useOptimistic(
    initialTasks,
    (state, { taskId, newStatus }) =>
      state.map(t => t.id === taskId
        ? { ...t, status: newStatus }
        : t
      )
  );

  async function handleDragEnd(taskId, newStatus) {
    // Step 1: Optimistic UI update (instant)
    setOptimistic({ taskId, newStatus });

    // Step 2: Server mutation (background)
    const result = await updateTaskStatus(taskId, newStatus);

    // Step 3: If failed, revalidation reverts to server state
    if (result.error) {
      toast.error("Failed to move task. Reverted.");
    }
  }

  return (/* render optimisticTasks */);
}
```

10B.4 What NOT to Use

For a project of TaskFlow's scale, avoid Redux, Zustand, Jotai, or other client-side state management libraries. Next.js App Router with Server Components already handles server state efficiently, and React's built-in useState/useOptimistic handles client state. Adding a state management library introduces unnecessary complexity, increases bundle size, and confuses AI coding assistants who may generate conflicting state update patterns. Reserve third-party state

management for applications with genuinely complex client-side state requirements (real-time collaborative editors, complex form workflows, offline-first applications).

Chapter 10C: UI Wireframes with AI

10C.1 Text-Based Wireframes: Why They Work

In Vibe Coding, you do not need pixel-perfect Figma mockups. Structured text descriptions of each page are sufficient because AI coding assistants can translate well-written layout descriptions directly into React components with Tailwind CSS. The key is being specific about layout structure, component hierarchy, and interaction patterns rather than visual details like exact colors or font sizes (which CLAUDE.md already defines).

Text-based wireframes have several advantages over visual mockups in an AI-assisted workflow. They are version-controlled alongside code, directly pasteable into AI prompts, easier to update than Figma files, and do not require design tool expertise. The trade-off is that they cannot communicate visual nuances—but for MVP development, structural clarity matters more than visual polish.

10C.2 Dashboard Wireframe

```
## Page: Dashboard (/dashboard)
Layout: Sidebar (left, 240px fixed) + Main Content (right, fluid)

Sidebar:
  - Logo + app name (top, h-16, border-bottom)
  - Navigation links: Dashboard, Boards, Reports, Settings
    (active state: bg-blue-50, text-blue-600, left border)
  - Team members list (avatars, 32px, with online indicator dot)
  - "Invite Member" button (bottom, secondary style)

Main Content:
  - Header bar: Page title (left) + User avatar dropdown (right)
  - Stats cards row (4 cards, equal width, gap-4):
    | Total Tasks | In Progress | Completed | Hours This Week |
    Each: number (2xl, bold) + label (sm, gray) + trend arrow
  - Recent Activity feed (card, max-h-96, scrollable):
    Each item: avatar + "[name] [action] [task]" + timestamp
  - FAB (floating action button): bottom-right, "+" icon
    Opens: quick-add task modal
```

10C.3 Board View Wireframe

```
## Page: Board View (/boards/[id])
Layout: Full-width, no sidebar

Header:
  - Back arrow (to dashboard) + Board name (editable inline)
```

```
  - Filter dropdown (by assignee, status, due date)
  - "Add Column" button (ghost/outline style)

Kanban Columns (horizontal scroll, gap-4, min-w-[280px] each):
  Column Header:
    - Column name + task count badge + "+" add task button
    - Drag handle for column reordering (Phase 2)
  Task Cards (draggable, gap-2):
    - Title (font-medium, truncate at 2 lines)
    - Assignee avatar (bottom-left, 24px)
    - Due date (bottom-right, text-xs, red if overdue)
    - Timer button (play icon, top-right corner)
  Drop Zone: dashed border when dragging over

Task Detail (slide-out panel, right, w-[480px]):
  - Title (editable, text-xl)
  - Description (rich text editor, optional)
  - Status dropdown, Assignee dropdown, Due date picker
  - Time logs history (table: date, duration, user)
  - Comments section (Phase 2)
```

10C.4 Prompt Pattern: Wireframe to Component

```
## Prompt to convert wireframe to React component

Convert this wireframe to a React component:
[paste wireframe text]

Requirements:
- Use Next.js 14+ App Router conventions
- Tailwind CSS for all styling (no custom CSS)
- TypeScript with proper types for all props
- Follow component structure from CLAUDE.md
- Responsive: works on mobile (stack columns vertically)
- Include loading skeleton states
- Use shadcn/ui components where applicable
```

PART IV

IMPLEMENTATION

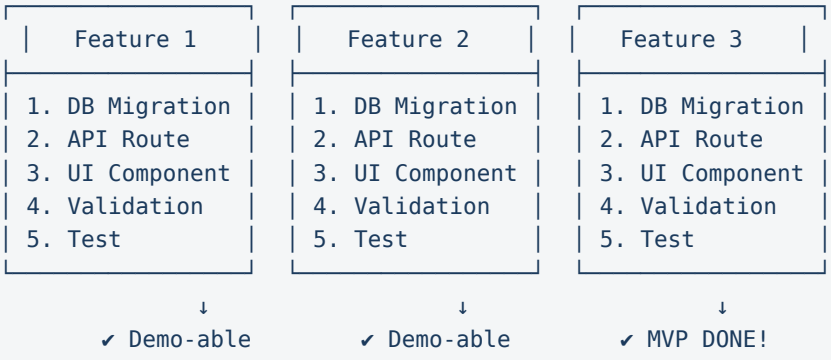
Vibe Coding in Action

Chapter 11: Vibe Coding Workflow — Vertical Slicing

11.1 Feature-by-Feature, Not File-by-File

A common mistake: writing all API routes first, then all components, then connecting them. This produces code that is difficult to test and full of bugs at integration time. The Vibe Coding workflow uses vertical slicing: complete one feature end-to-end (database + API + UI + test), verify it works, then move to the next feature. The concept of vertical slicing originated from XP/Agile communities, with Bill Wake popularizing the cake analogy in 2003 through his INVEST criteria. This approach is particularly relevant to vibe coding because AI agents can implement a complete feature across the full stack in a single session.

Vertical Slice Workflow



Chapter 12: Prompt Patterns for Production Code

The quality of your prompts directly determines the quality of AI-generated code. Below are seven proven patterns that consistently produce production-quality output:

#	Pattern	When to Use	Example Prompt
1	Context-First	Starting any new task	Given CLAUDE.md context, implement the Task creation API...
2	Spec-Driven	API endpoints, DB schemas	Implement exactly per the API design doc: POST /api/tasks...
3	Test-First (TDD)	Critical business logic	Write tests first for task status transitions, then implement to pass
4	Refactor	Improving existing code	Refactor this to extract reusable hooks. Keep exact same behavior.
5	Fix-and-Explain	Debugging errors	This error occurs at line 42. Fix it and explain why it happened.

#	Pattern	When to Use	Example Prompt
6	Complete-the-Pattern	Repetitive work	Here's the Task API. Create the Board API following the same pattern.
7	Review-and-Improve	Pre-merge code review	Review for security vulns, performance issues, and best practices.

Chapter 13: Live Coding — Task CRUD Implementation

13.1 The 5-Step Server Action Pattern

Every server action in the project follows a mandatory 5-step pattern: (1) Authentication check, (2) Input validation via Zod, (3) Authorization check, (4) Database operation wrapped in try/catch, (5) Cache revalidation. This pattern MUST remain consistent across all endpoints—and CLAUDE.md ensures AI follows it.

```
"use server";
import { prisma } from "@/lib/db";
import { z } from "zod";
import { getServerSession } from "next-auth";
import { revalidatePath } from "next/cache";

const CreateTaskSchema = z.object({
  title: z.string().min(3).max(100),
  description: z.string().max(2000).optional(),
  boardId: z.string().uuid(),
  assigneeId: z.string().uuid().optional(),
  dueDate: z.string().datetime().optional(),
});

export async function createTask(formData: FormData) {
  // Step 1: Auth check
  const session = await getServerSession();
  if (!session?.user) return { error: "Unauthorized" };

  // Step 2: Input validation (Zod)
  const raw = Object.fromEntries(formData);
  const parsed = CreateTaskSchema.safeParse(raw);
  if (!parsed.success) return { error: parsed.error.flatten() };

  // Step 3: Authorization — user must be in board's team
  const board = await prisma.board.findUnique({
    where: { id: parsed.data.boardId },
    include: { team: { include: { members: true } } },
```

```
  });
  if (!board?.team.members.some(
    m => m.email === session.user.email
  )) return { error: "Not a member of this team" };

  // Step 4: Database operation (try/catch)
  try {
    const task = await prisma.task.create({
      data: {
        title: parsed.data.title,
        description: parsed.data.description,
        boardId: parsed.data.boardId,
        assigneeId: parsed.data.assigneeId,
        dueDate: parsed.data.dueDate
          ? new Date(parsed.data.dueDate) : null,
      },
    });

    // Step 5: Cache revalidation
    revalidatePath(`/boards/${parsed.data.boardId}`);
    return { data: task };
  } catch (err) {
    console.error("Failed to create task:", err);
    return { error: "Failed to create task. Please try again." };
  }
}
```

13.2 Git Workflow

Branch	Naming Convention	Purpose	Lifetime
main	main	Production-ready code only	Permanent
develop	develop	Integration branch (optional for solo)	Permanent
feature	feat/US-001-create-task	One branch per user story	Until merged
fix	fix/timer-not-stopping	Bug fixes	Until merged

```
# Feature branch workflow
$ git checkout -b feat/US-001-create-task

# Small, atomic commits
$ git add -A && git commit -m "feat(task): add Prisma schema"
$ git add -A && git commit -m "feat(task): add createTask server action"
$ git add -A && git commit -m "feat(task): add TaskCard component"
$ git add -A && git commit -m "test(task): add unit tests"
```

```
# Push and create PR
$ git push -u origin feat/US-001-create-task
$ gh pr create --title "feat: US-001 Create Task"

# After AI review + tests pass: merge
$ gh pr merge --squash
```

Chapter 12B: Prompt Engineering Deep Dive

12B.1 Context-First Pattern: Full Example

The Context-First pattern is the most important prompt pattern in vibe coding. It ensures AI has all necessary context before generating code. Without context, AI produces generic code that may conflict with your project's architecture, coding standards, or security requirements. With context, AI produces code that fits seamlessly into your existing codebase. The difference between a novice vibe coder and an expert is primarily the quality of context they provide.

Below is a complete example showing the difference between a poor prompt and a context-rich prompt for the same task. The poor prompt produces code that works in isolation but violates multiple CLAUDE.md rules. The context-rich prompt produces production-ready code on the first attempt.

Poor Prompt (No Context)

```
## BAD: Vague, no context
"Create a function to update a task status"

## AI produces:
function updateTask(id, status) {  // No TypeScript!
  return fetch(`/api/tasks/${id}`, {
    method: 'PUT',              // Should be PATCH per API design
    body: JSON.stringify({ status })
  });                           // No error handling!
}                                // No auth! No validation!
```

Good Prompt (Context-First)

```
## GOOD: Full context from CLAUDE.md + API design

Given our CLAUDE.md rules:
- TypeScript strict mode, Zod validation on all inputs
- Server Actions for mutations (not client-side fetch)
- 5-step pattern: auth, validate, authorize, operate, revalidate
- Return { data, error, meta } format

And our API design:
- PATCH /api/tasks/[id] for status updates
- TaskStatus enum: TODO | IN_PROGRESS | REVIEW | DONE

Implement updateTaskStatus as a Server Action in src/server/tasks.ts
that changes a task's status. Include authorization check that
only team members can update tasks on their board.
```

The context-rich prompt references specific CLAUDE.md rules, the API design document, and the exact file location. AI now has enough information to produce code that follows all project conventions, uses the correct patterns, and handles authorization properly. This prompt takes 30 seconds longer to write but saves 15–30 minutes of post-generation cleanup.

12B.2 Spec-Driven Pattern: From User Story to Code

The Spec-Driven pattern takes a user story with acceptance criteria and produces a complete implementation. This pattern works exceptionally well because user stories provide both the business context (what the user needs) and the technical requirements (acceptance criteria that map directly to test cases).

```
## Spec-Driven Prompt

Implement US-002 (Drag-Drop Task Between Columns):

User Story: As a team member, I want to drag a task between
columns so that I can update task status visually.

Acceptance Criteria:
1. Drag-drop works on desktop (mouse) and mobile (touch)
2. Target column highlights when task is dragged over it
3. Task status updates in database immediately after drop
4. Other team members see the move in real-time
5. Undo available for 5 seconds after move (toast with undo btn)

Technical Context:
- Use @dnd-kit library (already installed)
- Server Action: updateTaskStatus in src/server/tasks.ts
- Optimistic update pattern (useOptimistic hook)
- Follow 5-step server action pattern from CLAUDE.md

Generate:
1. The KanbanBoard component (src/components/kanban-board.tsx)
2. The KanbanColumn component (src/components/kanban-column.tsx)
3. Unit tests for the updateTaskStatus server action
4. E2E test for the drag-drop flow
```

12B.3 Test-First (TDD) Pattern

The Test-First pattern inverts the typical vibe coding workflow. Instead of asking AI to write code and then tests, you ask AI to write tests first based on the acceptance criteria, then implement the code to make those tests pass. This produces more robust code because the tests define the behavior boundary

before implementation begins. It is particularly valuable for critical business logic where edge cases matter.

```
## TDD Prompt

For the time tracking feature (US-003), write TESTS FIRST:

Test cases for startTimer server action:
1. Returns TimeLog when starting timer on valid task
2. Auto-stops any running timer before starting new one
3. Rejects if user is not authenticated (returns error)
4. Rejects if taskId is not a valid UUID
5. Rejects if task doesn't exist (Prisma P2003 error)
6. Rejects if user is not a member of the task's team
7. Handles concurrent start requests gracefully

Use Vitest + Prisma mock. AAA pattern (Arrange-Act-Assert).
After tests are written, implement startTimer to pass all tests.
```

12B.4 Complete-the-Pattern: Scaling Efficiently

The Complete-the-Pattern prompt is the highest-leverage pattern in vibe coding. Once you have one well-implemented feature (with proper auth, validation, error handling, and tests), you can ask AI to replicate the same pattern for similar features. This produces remarkably consistent code across the entire project because AI extracts and applies the conventions from the reference implementation. It is especially effective for CRUD operations, API endpoints, and form components that share structural similarity.

```
## Complete-the-Pattern Prompt

Here is the complete implementation of the Task CRUD feature:
- Server Actions: src/server/tasks.ts
- UI Components: src/components/task-card.tsx
- Tests: tests/unit/task-validation.test.ts

Now create the Board CRUD feature following the EXACT same patterns:
- Same 5-step server action structure
- Same Zod validation approach
- Same error handling and response format
- Same test structure and coverage

Board-specific requirements:
- Only ADMIN role can create/delete boards
- Board columns field is JSON (default: ["Todo","In Progress","Done"])
- Deleting a board should cascade-delete all tasks (confirm dialog)
```

Chapter 13B: Live Coding — Kanban Board UI

13B.1 The Kanban Board Component

The Kanban board is TaskFlow's primary interface and the most complex UI component in the MVP. It requires drag-and-drop between columns, real-time updates when other team members make changes, optimistic UI updates for instant feedback, and responsive design that works on both desktop and mobile. We use @dnd-kit as the drag-and-drop library because it has excellent React 18 support, accessible by default, and strong AI training data coverage.

```
// src/components/kanban-board.tsx
"use client";
import { DndContext, closestCorners, DragEndEvent } from "@dnd-kit/core";
import { SortableContext, verticalListSortingStrategy } from
"@dnd-kit/sortable";
import { useOptimistic, startTransition } from "react";
import { updateTaskStatus } from "@/server/tasks";
import { TaskCard } from "./task-card";
import { KanbanColumn } from "./kanban-column";
import type { BoardWithTasks, TaskStatus } from "@/lib/types";

const COLUMNS: TaskStatus[] = ["TODO", "IN_PROGRESS", "REVIEW", "DONE"];

type Props = { board: BoardWithTasks };

export function KanbanBoard({ board }: Props) {
  const [optimisticTasks, setOptimistic] = useOptimistic(
    board.tasks,
    (state, update: { id: string; status: TaskStatus }) =>
      state.map(t => t.id === update.id
        ? { ...t, status: update.status } : t),
  );

  async function handleDragEnd(event: DragEndEvent) {
    const { active, over } = event;
    if (!over || active.id === over.id) return;

    const taskId = active.id as string;
    const newStatus = over.id as TaskStatus;

    // Optimistic: instant UI update
    startTransition(() => {
      setOptimistic({ id: taskId, status: newStatus });
    });

    // Server: persist change
```

```
    await updateTaskStatus(taskId, newStatus);
  }

  return (
    <DndContext
      collisionDetection={closestCorners}
      onDragEnd={handleDragEnd}
    >
      <div className="flex gap-4 overflow-x-auto p-4 min-h-screen">
        {COLUMNS.map(status => {
          const tasks = optimisticTasks.filter(
            t => t.status === status
          );
          return (
            <KanbanColumn key={status} status={status} count={tasks.length}>
              <SortableContext items={tasks}
strategy={verticalListSortingStrategy}>
                {tasks.map(task => (
                  <TaskCard key={task.id} task={task} />
                ))}
              </SortableContext>
            </KanbanColumn>
          );
        })}
      </div>
    </DndContext>
  );
}
```

13B.2 The Task Card Component

```
// src/components/task-card.tsx
"use client";
import { useSortable } from "@dnd-kit/sortable";
import { CSS } from "@dnd-kit/utilities";
import { Avatar } from "./ui/avatar";
import { TimerButton } from "./timer-button";
import type { Task } from "@/lib/types";

export function TaskCard({ task }: { task: Task }) {
  const { attributes, listeners, setNodeRef, transform, transition,
    isDragging } = useSortable({ id: task.id });

  const style = {
    transform: CSS.Transform.toString(transform),
    transition,
    opacity: isDragging ? 0.5 : 1,
```

```
  };

  return (
    <div ref={setNodeRef} style={style} {...attributes} {...listeners}
      className="bg-white rounded-lg border p-3 shadow-sm
        hover:shadow-md transition-shadow cursor-grab active:cursor-grabbing"
    >
      <p className="font-medium text-sm line-clamp-2">{task.title}</p>
      <div className="flex items-center justify-between mt-2">
        {task.assignee && <Avatar user={task.assignee} size="sm" />}
        <TimerButton taskId={task.id} />
        {task.dueDate && (
          <span className={`text-xs ${isOverdue(task.dueDate)
            ? "text-red-500" : "text-gray-400"}`}>
            {formatDate(task.dueDate)}
          </span>
        )}
      </div>
    </div>
  );
}
```

Chapter 13C: Live Coding — Time Tracker

13C.1 Timer Server Actions

The time tracking feature allows team members to start and stop a timer on any task, recording the duration for reporting purposes. A critical constraint: only one timer can be active per user at any time. Starting a new timer automatically stops the previous one. This business rule must be enforced at the server level, not just in the UI, because multiple browser tabs or devices could be involved.

```
// src/server/time.ts
"use server";
import { prisma } from "@/lib/db";
import { z } from "zod";
import { getServerSession } from "next-auth";
import { revalidatePath } from "next/cache";

const StartTimerSchema = z.object({
  taskId: z.string().uuid(),
});

/** Start a timer. Auto-stops any running timer for this user. */
export async function startTimer(formData: FormData) {
  const session = await getServerSession();
  if (!session?.user) return { error: "Unauthorized" };

  const parsed = StartTimerSchema.safeParse(
    Object.fromEntries(formData)
  );
  if (!parsed.success) return { error: parsed.error.flatten() };

  const userId = session.user.id;

  // Auto-stop any currently running timer
  const running = await prisma.timeLog.findFirst({
    where: { userId, endTime: null },
  });

  if (running) {
    const duration = Math.floor(
      (Date.now() - running.startTime.getTime()) / 1000
    );
    await prisma.timeLog.update({
      where: { id: running.id },
      data: { endTime: new Date(), duration },
    });
  }
```

```
  // Start new timer
  try {
    const log = await prisma.timeLog.create({
      data: { taskId: parsed.data.taskId, userId },
    });
    revalidatePath("/");
    return { data: log };
  } catch (err) {
    return { error: "Failed to start timer" };
  }
}

/** Stop the currently running timer for this user. */
export async function stopTimer() {
  const session = await getServerSession();
  if (!session?.user) return { error: "Unauthorized" };

  const running = await prisma.timeLog.findFirst({
    where: { userId: session.user.id, endTime: null },
  });

  if (!running) return { error: "No timer running" };

  const duration = Math.floor(
    (Date.now() - running.startTime.getTime()) / 1000
  );

  try {
    const log = await prisma.timeLog.update({
      where: { id: running.id },
      data: { endTime: new Date(), duration },
    });
    revalidatePath("/");
    return { data: log };
  } catch (err) {
    return { error: "Failed to stop timer" };
  }
}
```

13C.2 Timer UI Component

```
// src/components/timer-button.tsx
"use client";
import { useState, useEffect } from "react";
import { startTimer, stopTimer } from "@/server/time";
import { Play, Pause } from "lucide-react";
```

```
export function TimerButton({ taskId, activeLog }) {
  const [elapsed, setElapsed] = useState(0);
  const isRunning = !!activeLog;

  useEffect(() => {
    if (!isRunning) { setElapsed(0); return; }
    const start = new Date(activeLog.startTime).getTime();
    const tick = () => setElapsed(
      Math.floor((Date.now() - start) / 1000)
    );
    tick();
    const id = setInterval(tick, 1000);
    return () => clearInterval(id);
  }, [isRunning, activeLog]);

  const hh = String(Math.floor(elapsed / 3600)).padStart(2, "0");
  const mm = String(Math.floor((elapsed % 3600) / 60)).padStart(2, "0");
  const ss = String(elapsed % 60).padStart(2, "0");

  return (
    <button onClick={async () => {
      const fd = new FormData();
      fd.set("taskId", taskId);
      isRunning ? await stopTimer() : await startTimer(fd);
    }}>
      {isRunning ? <Pause size={14} /> : <Play size={14} />}
      {isRunning && <span className="ml-1 text-xs font-mono">
        {hh}:{mm}:{ss}</span>}
    </button>
  );
}
```

Chapter 13D: Live Coding — Weekly Report Generation

13D.1 Report Data Aggregation

The weekly report is one of TaskFlow's most valuable features for team leads. It automatically aggregates data from the past week: tasks completed per team member, total hours tracked, tasks in progress, and potential blockers (tasks that have been in the same status for more than 3 days). The report is generated via a Server Action that queries the database with date-range filters and produces structured data for both the UI display and email delivery.

```
// src/server/reports.ts
"use server";
import { prisma } from "@/lib/db";
import { getServerSession } from "next-auth";
import { startOfWeek, endOfWeek, subWeeks } from "date-fns";

type WeeklyReport = {
  period: { start: Date; end: Date };
  summary: {
    tasksCompleted: number;
    tasksInProgress: number;
    totalHoursTracked: number;
    blockers: number;
  };
  memberStats: Array<{
    name: string;
    email: string;
    tasksCompleted: number;
    hoursTracked: number;
  }>;
  blockedTasks: Array<{
    title: string;
    status: string;
    assignee: string;
    daysSinceLastUpdate: number;
  }>;
};

export async function generateWeeklyReport(
  teamId: string
): Promise<{ data: WeeklyReport | null; error: string | null }> {
  // Step 1: Auth
  const session = await getServerSession();
  if (!session?.user) return { data: null, error: "Unauthorized" };
```

```
// Step 2: Check admin role
const user = await prisma.user.findUnique({
  where: { email: session.user.email! },
});
if (user?.role !== "ADMIN") {
  return { data: null, error: "Admin access required" };
}

// Step 3: Calculate date range
const now = new Date();
const weekStart = startOfWeek(subWeeks(now, 1), {
  weekStartsOn: 1, // Monday
});
const weekEnd = endOfWeek(subWeeks(now, 1), {
  weekStartsOn: 1,
});

try {
  // Step 4: Aggregate data
  const completedTasks = await prisma.task.findMany({
    where: {
      board: { teamId },
      status: "DONE",
      updatedAt: { gte: weekStart, lte: weekEnd },
    },
    include: { assignee: true },
  });

  const timeLogs = await prisma.timeLog.findMany({
    where: {
      user: { teamId },
      startTime: { gte: weekStart, lte: weekEnd },
      endTime: { not: null },
    },
    include: { user: true },
  });

  // Blocked: same status for 3+ days
  const threeDaysAgo = new Date(
    Date.now() - 3 * 24 * 60 * 60 * 1000
  );
  const blocked = await prisma.task.findMany({
    where: {
      board: { teamId },
      status: { in: ["TODO", "IN_PROGRESS", "REVIEW"] },
      updatedAt: { lt: threeDaysAgo },
    },
    include: { assignee: true },
```

```
    });

    // Step 5: Build report
    const totalSeconds = timeLogs.reduce(
      (sum, log) => sum + (log.duration ?? 0), 0
    );

    // Group by member
    const memberMap = new Map<string, {
      name: string; email: string;
      tasksCompleted: number; seconds: number;
    }>();
    // ... aggregation logic ...

    return {
      data: {
        period: { start: weekStart, end: weekEnd },
        summary: {
          tasksCompleted: completedTasks.length,
          tasksInProgress: 0, // calculated separately
          totalHoursTracked: +(totalSeconds / 3600).toFixed(1),
          blockers: blocked.length,
        },
        memberStats: Array.from(memberMap.values()).map(m => ({
          ...m,
          hoursTracked: +(m.seconds / 3600).toFixed(1),
        })),
        blockedTasks: blocked.map(t => ({
          title: t.title,
          status: t.status,
          assignee: t.assignee?.name ?? "Unassigned",
          daysSinceLastUpdate: Math.floor(
            (Date.now() - t.updatedAt.getTime()) / 86400000
          ),
        })),
      },
      error: null,
    };
  } catch (err) {
    console.error("Report generation failed:", err);
    return { data: null, error: "Failed to generate report" };
  }
}
```

13D.2 Report UI Component

The weekly report renders as a dashboard-style page with four key sections: a summary stats row at the top, a member performance table, a blocked tasks warning list, and a chart showing daily task completion over the week. The component uses React Server Components for initial data fetching and client components only for interactive elements like the PDF export button.

```
// src/app/reports/page.tsx
import { generateWeeklyReport } from "@/server/reports";
import { getTeamId } from "@/lib/auth";
import { StatCard } from "@/components/stat-card";
import { MemberTable } from "@/components/member-table";
import { BlockersList } from "@/components/blockers-list";
import { ExportButton } from "@/components/export-button";

export default async function ReportsPage() {
  const teamId = await getTeamId();
  const { data: report, error } = await generateWeeklyReport(teamId);

  if (error) return <div className="p-8">Error: {error}</div>;
  if (!report) return <div className="p-8">No data available.</div>;

  return (
    <div className="p-6 max-w-6xl mx-auto">
      <div className="flex justify-between items-center mb-6">
        <h1 className="text-2xl font-bold">Weekly Report</h1>
        <ExportButton report={report} />
      </div>

      {/* Summary Stats */}
      <div className="grid grid-cols-4 gap-4 mb-8">
        <StatCard label="Completed" value={report.summary.tasksCompleted} />
        <StatCard label="In Progress" value={report.summary.tasksInProgress} />
        <StatCard label="Hours Tracked"
value={report.summary.totalHoursTracked} />
        <StatCard label="Blockers" value={report.summary.blockers}
          variant={report.summary.blockers > 0 ? "warning" : "default"} />
      </div>

      {/* Member Performance */}
      <MemberTable members={report.memberStats} />

      {/* Blocked Tasks */}
      {report.blockedTasks.length > 0 && (
        <BlockersList blockers={report.blockedTasks} />
      )}
    </div>
```

```
  );
}
```

PART V

TESTING & QA

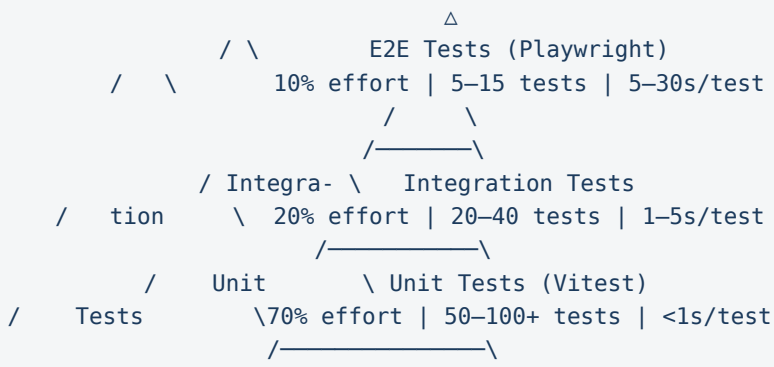
Making Sure the Code Actually Works

Chapter 14: The Testing Pyramid

14.1 Origin and Principles

The Testing Pyramid was created by Mike Cohn, first sketched in conversation with Lisa Crispin around 2003–2004 and formally published in his 2009 book Succeeding with Agile. Its three layers (unit tests at the base, service/integration tests in the middle, UI/E2E tests at the top) remain the industry standard. The principle is simple: write many fast, cheap unit tests (70% of effort), some integration tests (20%), and few slow, expensive E2E tests (10%).

Testing Pyramid



14.2 Unit Tests with Vitest

```
// tests/unit/task-validation.test.ts
import { describe, it, expect } from "vitest";
import { CreateTaskSchema } from "@/lib/schemas";

describe("CreateTaskSchema", () => {
  it("accepts valid task data", () => {
    const result = CreateTaskSchema.safeParse({
      title: "Fix login bug",
      boardId: "550e8400-e29b-41d4-a716-446655440000",
    });
    expect(result.success).toBe(true);
  });

  it("rejects title shorter than 3 characters", () => {
    const result = CreateTaskSchema.safeParse({
      title: "ab",
      boardId: "550e8400-e29b-41d4-a716-446655440000",
    });
    expect(result.success).toBe(false);
  });
```

```
  it("rejects missing boardId", () => {
    const result = CreateTaskSchema.safeParse({ title: "Valid" });
    expect(result.success).toBe(false);
  });

  it("handles SQL injection attempt safely", () => {
    const result = CreateTaskSchema.safeParse({
      title: "'; DROP TABLE tasks; --",
      boardId: "550e8400-e29b-41d4-a716-446655440000",
    });
    // Zod passes (valid string >3 chars)
    // Prisma ORM parameterizes = NO injection possible
    expect(result.success).toBe(true);
  });
});
```

14.3 E2E Tests with Playwright

```
// tests/e2e/kanban.spec.ts
import { test, expect } from "@playwright/test";

test("user can create and move a task", async ({ page }) => {
  // Login
  await page.goto("/login");
  await page.fill("[name=email]", "test@example.com");
  await page.fill("[name=password]", "testpass123");
  await page.click("button[type=submit]");
  await expect(page).toHaveURL("/dashboard");

  // Navigate to board
  await page.click("text=My Board");
  await expect(page.getByText("TODO")).toBeVisible();

  // Create task
  await page.click("[data-testid=add-task-btn]");
  await page.fill("[name=title]", "E2E Test Task");
  await page.click("[data-testid=submit-task]");

  // Verify
  await expect(page.getByText("E2E Test Task")).toBeVisible();
});
```

14.4 Security Scanning Tools

Tool	What It Checks	Command	Frequency
npm audit	Known vulnerabilities in npm packages	npm audit --production	Every PR
Snyk	Deep dependency scan + fix suggestions	snyk test	Daily
gitleaks	Secrets accidentally committed	gitleaks detect	Every commit (pre-commit hook)
ESLint security	Common code security patterns	eslint-plugin-security	Every save
Lighthouse	Best practices, SEO, accessibility	npx lighthouse URL	Pre-release

14.5 Coverage Goals

Metric	Target	Tool
Line coverage	>80%	vitest --coverage
Branch coverage	>70%	vitest --coverage
E2E critical paths	100% (all user stories)	Playwright
Security vulnerabilities	0 high/critical	npm audit + Snyk
Secrets in repository	0	gitleaks
Lighthouse score	>90 (all categories)	Lighthouse CI

Chapter 14D: Integration Tests

14D.1 What Integration Tests Cover

Integration tests verify that multiple components work together correctly. In the testing pyramid, they sit between unit tests (which test individual functions in isolation) and E2E tests (which test complete user flows in a real browser). For TaskFlow, integration tests focus on three areas: Server Action + Database interactions (does createTask actually persist to the database?), API route request/response cycles (does the endpoint return the correct status codes and response format?), and authentication/authorization flows (does an unauthorized request get properly rejected?).

The gap between unit tests and E2E tests is where most production bugs hide. Unit tests mock the database, so they cannot catch Prisma query errors, foreign key violations, or data type mismatches. E2E tests are slow and expensive, so you cannot write enough of them to cover all edge cases. Integration tests fill this critical middle ground by testing real database interactions without the overhead of a full browser environment.

14D.2 Setting Up Prisma Test Environment

```
// tests/setup.ts — Test database configuration
import { PrismaClient } from "@prisma/client";
import { execSync } from "child_process";

const TEST_DB = process.env.TEST_DATABASE_URL;
const prisma = new PrismaClient({
  datasources: { db: { url: TEST_DB } },
});

// Before all tests: push schema to test DB
beforeAll(async () => {
  execSync(`DATABASE_URL=${TEST_DB} npx prisma db push --force-reset`,
    { stdio: "inherit" }
  );
});

// After each test: clean all tables
afterEach(async () => {
  await prisma.timeLog.deleteMany();
  await prisma.task.deleteMany();
  await prisma.board.deleteMany();
  await prisma.user.deleteMany();
  await prisma.team.deleteMany();
});
```

```
afterAll(async () => { await prisma.$disconnect(); });

export { prisma };
```

14D.3 Integration Test Examples

```
// tests/integration/tasks.test.ts
import { describe, it, expect } from "vitest";
import { prisma } from "../setup";

describe("Task CRUD Integration", () => {
  async function seedTeamAndBoard() {
    const team = await prisma.team.create({
      data: { name: "Test Team", ownerId: "owner-1" },
    });
    const user = await prisma.user.create({
      data: {
        email: "test@example.com", name: "Test User",
        teamId: team.id,
      },
    });
    const board = await prisma.board.create({
      data: { name: "Test Board", teamId: team.id },
    });
    return { team, user, board };
  }

  it("creates a task and persists to database", async () => {
    const { user, board } = await seedTeamAndBoard();

    const task = await prisma.task.create({
      data: {
        title: "Integration Test Task",
        boardId: board.id,
        assigneeId: user.id,
      },
    });

    expect(task.id).toBeDefined();
    expect(task.title).toBe("Integration Test Task");
    expect(task.status).toBe("TODO"); // Default
    expect(task.boardId).toBe(board.id);
  });

  it("enforces foreign key: rejects invalid boardId", async () => {
    await expect(
      prisma.task.create({
```

```
        data: {
          title: "Orphan Task",
          boardId: "nonexistent-uuid",
        },
      })
    ).rejects.toThrow(); // Prisma P2003
  });

  it("cascades: composite index query works", async () => {
    const { board } = await seedTeamAndBoard();
    await prisma.task.createMany({
      data: [
        { title: "A", boardId: board.id, status: "TODO" },
        { title: "B", boardId: board.id, status: "DONE" },
        { title: "C", boardId: board.id, status: "TODO" },
      ],
    });

    const todos = await prisma.task.findMany({
      where: { boardId: board.id, status: "TODO" },
    });

    expect(todos).toHaveLength(2);
    expect(todos.every(t => t.status === "TODO")).toBe(true);
  });
});
```

14D.4 Integration Test Coverage Targets

Area	What to Test	Min. Test Count
Task CRUD	Create, read, update status, delete + auth checks	8–10 tests
Board CRUD	Create, list, delete cascade + admin-only checks	6–8 tests
Time Tracking	Start, stop, auto-stop, concurrent handling	6–8 tests
Auth Flow	Login, session creation, protected route access	4–5 tests
Weekly Report	Aggregation query correctness, date filtering	3–4 tests

Chapter 14B: Error Handling Patterns

14B.1 The Four Layers of Error Handling

Production applications require error handling at four distinct layers, each serving a different purpose. The validation layer catches malformed input before it reaches business logic. The business logic layer handles domain-specific errors like insufficient permissions or invalid state transitions. The infrastructure layer handles database connection failures, network timeouts, and third-party API errors. The presentation layer translates technical errors into user-friendly messages. AI-generated code frequently handles only the first two layers and ignores infrastructure and presentation errors, leading to cryptic error messages or silent failures in production.

Layer	Purpose	Example	Implementation
1. Validation	Catch malformed input early	Title too short, invalid UUID	Zod schemas at entry points
2. Business Logic	Domain-specific rules	User not team member, timer already running	Explicit checks in server actions
3. Infrastructure	System-level failures	DB connection lost, Prisma timeout	try/catch + retry + circuit breaker
4. Presentation	User-friendly messages	"Something went wrong" vs stack trace	Error boundaries + toast notifications

14B.2 Server-Side Error Handling Pattern

```
// src/lib/errors.ts
export class AppError extends Error {
  constructor(
    message: string,
    public code: string,
    public statusCode: number = 400,
    public isOperational: boolean = true
  ) {
    super(message);
    this.name = "AppError";
  }
}

// Usage in server actions:
export async function moveTask(taskId: string, newStatus: TaskStatus) {
  try {
    // ... validation + auth + business logic ...
    const task = await prisma.task.update({
```

```
      where: { id: taskId },
      data: { status: newStatus },
    });
    return { data: task, error: null };
  } catch (err) {
    // Prisma-specific errors
    if (err.code === "P2025") {
      return { data: null, error: "Task not found" };
    }
    if (err.code === "P2002") {
      return { data: null, error: "Duplicate entry" };
    }
    // Log unexpected errors for investigation
    console.error("Unexpected error in moveTask:", err);
    Sentry.captureException(err);
    return { data: null, error: "An unexpected error occurred" };
  }
}
```

14B.3 Client-Side Error Boundary

```
// src/components/error-boundary.tsx
"use client";
import { Component, ReactNode } from "react";

interface Props { children: ReactNode; fallback?: ReactNode; }
interface State { hasError: boolean; error?: Error; }

export class ErrorBoundary extends Component<Props, State> {
  state: State = { hasError: false };

  static getDerivedStateFromError(error: Error): State {
    return { hasError: true, error };
  }

  componentDidCatch(error: Error, info: React.ErrorInfo) {
    console.error("Error boundary caught:", error, info);
    // Report to Sentry in production
  }

  render() {
    if (this.state.hasError) {
      return this.props.fallback || (
        <div className="p-8 text-center">
          <h2>Something went wrong</h2>
          <button onClick={() => this.setState({ hasError: false })}>
            Try Again
```

```
          </button>
        </div>
      );
    }
    return this.props.children;
  }
}
```

14B.4 Common Error Scenarios and Solutions

Scenario	Symptom	Root Cause	Solution
Drag-drop fails silently	Task snaps back to original column	Server action throws, no error handling in UI	Add toast notification on error; revert optimistic state
Timer shows negative time	Display shows -00:01:23	Clock skew between client and server	Use server timestamp for startTime; Math.max(0, elapsed)
Concurrent timer starts	Two timers running for same user	Race condition in auto-stop logic	Add database transaction; use SELECT FOR UPDATE
Stale board data	New tasks from other users not visible	Cache not invalidated after peer actions	Add polling or WebSocket for real-time sync
File upload timeout	Spinner spins forever	No timeout on fetch call	Add AbortController with 30s timeout
Rate limit hit	Rapid clicks cause 429 errors	No client-side throttling	Debounce mutation calls; disable button during request

Chapter 14C: Code Review with AI

14C.1 The AI Code Review Workflow

Before merging any feature branch, the code should undergo AI-assisted review focusing on security, performance, and compliance with CLAUDE.md standards. This is not a replacement for human review—it is an additional safety net that catches common issues quickly. The workflow consists of three steps: prompt AI with the code and review criteria, analyze AI's findings for validity (AI sometimes flags false positives), and address genuine issues before merge.

14C.2 Review Prompt Template

```
## AI Code Review Prompt

Review the following file for:

1. SECURITY: Injection vulnerabilities, auth bypass,
   data leaks, missing input validation, hardcoded secrets

2. PERFORMANCE: N+1 queries, missing database indexes,
   unnecessary re-renders, large bundle imports

3. ERROR HANDLING: Unhandled promises, missing try/catch,
   generic error messages that leak implementation details

4. TYPESCRIPT: Any `any` types, missing return types,
   unsafe type assertions, non-strict comparisons

5. CLAUDE.md COMPLIANCE: Violations of project coding rules
   (max 50 lines/function, JSDoc comments, Zod validation)

Format your response as:
- CRITICAL: Must fix before merge (security, data loss)
- WARNING: Should fix (performance, best practices)
- INFO: Nice-to-have improvements (style, readability)
```

14C.3 Example AI Review Output

Below is an example of what a thorough AI code review looks like for the createTask server action. Understanding this output format helps you quickly triage AI findings and focus on genuine issues.

Example AI Review: createTask Server Action

CRITICAL: None found. Auth, validation, and authorization checks are properly

implemented. WARNING #1: The board query includes team.members which loads ALL member data. Use select() to fetch only email fields needed for the authorization check. This is an N+1-adjacent issue that will worsen as teams grow. WARNING #2: Error message in catch block is generic ("Failed to create task"). Consider differentiating between Prisma P2003 (foreign key violation = invalid boardId) and P2002 (unique constraint = duplicate) for better debugging. INFO #1: Consider extracting the authorization check ("is user a member of this board's team") into a reusable middleware function since it will be needed in updateTask, deleteTask, and all time-tracking endpoints.

14C.4 Per-Feature Review Checklist

#	Check	How to Verify
1	Zod validation on all inputs	Search for raw FormData/JSON access without .safeParse()
2	Auth + authorization on all endpoints	Test: unauthenticated request returns 401
3	Error handling with try/catch	No unhandled promise rejections in server actions
4	TypeScript passes strict mode	Run: npx tsc --noEmit
5	ESLint passes with no warnings	Run: npx eslint . --max-warnings=0
6	All tests pass	Run: npx vitest run
7	AI code review completed	Claude Code or Cursor review pass
8	Feature works end-to-end manually	Manual browser testing on localhost

PART VI

DEPLOYMENT & DEVOPS

From Localhost to Production

Chapter 15: CI/CD Pipeline

15.1 GitHub Actions: Auto-Test, Migrate, Deploy

```
# .github/workflows/ci.yml
name: CI/CD Pipeline
on:
  push: { branches: [main] }
  pull_request: { branches: [main] }

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v4
        with: { node-version: 20, cache: npm }
      - run: npm ci
      - run: npx tsc --noEmit            # Type check
      - run: npx eslint .                 # Lint
      - run: npx vitest run               # Unit + integration
      - run: npx playwright install --with-deps
      - run: npx playwright test           # E2E
      - run: npm audit --production       # Security audit

  migrate:
    needs: test
    if: github.ref == 'refs/heads/main'
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v4
        with: { node-version: 20, cache: npm }
      - run: npm ci
      - run: npx prisma migrate deploy   # Apply DB migrations
        env:
          DATABASE_URL: ${{ secrets.DATABASE_URL }}

  deploy:
    needs: [test, migrate]
    if: github.ref == 'refs/heads/main'
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: amondnet/vercel-action@v25
        with:
          vercel-token: ${{ secrets.VERCEL_TOKEN }}
```

```
          vercel-args: '--prod'
```

15.2 Environment Management

Variable	Dev (.env.local)	Production	Notes
DATABASE_URL	localhost:5432/dev_db	Neon production URL	NEVER share DB between dev & prod
NEXTAUTH_SECRET	dev-secret-123	64-char random string	Generate: openssl rand -base64 32
NEXTAUTH_URL	http://localhost:3000	https://taskflow.app	Must match actual domain
SENTRY_DSN	(optional)	https://sentry.io/prod-dsn	Error tracking in production

15.3 Database Migration Strategy

Command	Environment	Behavior	Data Safety
prisma db push	Development ONLY	Direct schema sync, may reset data	Data may be lost!
prisma migrate dev	Development	Generates SQL migration file	Safe
prisma migrate deploy	Production / CI	Applies migrations sequentially	Safe

CRITICAL: Always Backup Before Migrating Production

Before running any migration in production: (1) Create a full database backup, (2) Test the migration in staging first, (3) Never use destructive migrations (DROP COLUMN, RENAME TABLE) without a data migration plan, (4) Database rollbacks are NOT automatic—prepare reverse migrations. Prisma migrate deploy applies migrations forward only.

Chapter 16: Monitoring & Launch Checklist

16.1 Monitoring Setup

What	Tool (Free Tier)	Setup Time	Why It Matters
Error Tracking	Sentry	5 min	Full stack traces with user context + alerting
Performance	Vercel Analytics	0 min	Core Web Vitals and API response times

What	Tool (Free Tier)	Setup Time	Why It Matters
		(built-in)	
Uptime	BetterUptime	2 min	Instant alerts when your site goes down
Logs	Vercel Logs	0 min (built-in)	Server-side request and error logs
Usage Analytics	PostHog / Plausible	10 min	Privacy-friendly product usage data

16.2 Launch Checklist

#	Item	Verification Method
1	CI/CD pipeline running; all tests passing	Check GitHub Actions green
2	Production environment variables set correctly	Verify in Vercel dashboard
3	Database migrations applied to production	Run prisma migrate status
4	Custom domain configured with SSL active	Visit https://yourdomain.com
5	Sentry error tracking connected and verified	Trigger test error, check Sentry
6	Uptime monitoring active	Confirm BetterUptime dashboard
7	Database backup schedule configured	Verify Neon auto-backup settings
8	Rate limiting enabled on all endpoints	Test with rapid requests
9	Security headers configured (CSP, HSTS, etc.)	Run securityheaders.com scan
10	Basic load test passed	Artillery or k6 smoke test

Chapter 15B: Environment Setup & .env Configuration

15B.1 The .env.example File

Every project must include a .env.example file that documents all required environment variables without containing actual secret values. This file serves three purposes: it tells new developers what variables they need to configure, it gives AI coding assistants the correct variable names to reference, and it acts as documentation for deployment configuration. Without this file, developers (and AI) frequently hardcode secrets or use incorrect variable names.

```
# .env.example — Copy to .env.local and fill in values

# Database (Neon PostgreSQL)
DATABASE_URL="postgresql://user:password@host/dbname?sslmode=require"

# NextAuth.js
NEXTAUTH_SECRET="generate-with-openssl-rand-base64-32"
NEXTAUTH_URL="http://localhost:3000"

# Google OAuth (console.cloud.google.com)
GOOGLE_CLIENT_ID="your-client-id.apps.googleusercontent.com"
GOOGLE_CLIENT_SECRET="your-client-secret"

# Error Tracking (optional in dev)
SENTRY_DSN=""

# Testing
TEST_DATABASE_URL="postgresql://user:password@localhost/taskflow_test"
```

15B.2 Environment-Specific Configuration

Setting	Development	Staging	Production
DATABASE_URL	localhost:5432/taskflow_dev	Neon staging branch	Neon production branch
NEXTAUTH_URL	http://localhost:3000	https://staging.taskflow.app	https://taskflow.app
SENTRY_DSN	Empty (no tracking)	Staging DSN	Production DSN
LOG_LEVEL	debug	info	warn
RATE_LIMIT	Disabled	100 req/min	100 req/min
GOOGLE_OAUTH	Test credentials	Same as prod	Production credentials

15B.3 Vercel Environment Variable Setup

Vercel provides three environment scopes: Production, Preview, and Development. Each scope can have different values for the same variable name, enabling safe separation between environments. Production variables are used only for the main branch deployment. Preview variables apply to pull request deployments. Development variables are available when running vercel dev locally.

```
# Set production environment variables via CLI
$ vercel env add DATABASE_URL production
# Paste your Neon production connection string

$ vercel env add NEXTAUTH_SECRET production
# Paste output of: openssl rand -base64 32

$ vercel env add NEXTAUTH_URL production
# Enter: https://taskflow.app

$ vercel env add GOOGLE_CLIENT_ID production
$ vercel env add GOOGLE_CLIENT_SECRET production

# Verify all variables are set
$ vercel env ls

# Alternative: use Vercel Dashboard
# Settings > Environment Variables > Add for each scope
```

15B.4 DNS and SSL Configuration

For custom domains on Vercel, the setup is straightforward but frequently missed in deployment guides. Add your domain in Vercel's project settings, then configure DNS records at your domain registrar. Vercel automatically provisions and renews SSL certificates via Let's Encrypt once DNS propagation is complete. The entire process typically takes 5–15 minutes.

Record Type	Name	Value	Purpose
A	@	76.76.21.21	Root domain (taskflow.app)
CNAME	www	cname.vercel-dns.com	www subdomain redirect

Tip: Verify SSL Configuration

After DNS propagation (usually 5–30 minutes), verify your SSL setup by visiting https://yourdomain.com and checking the lock icon. You can also run a

comprehensive check at ssllabs.com/ssltest. Vercel’s automatic SSL typically scores A+ on SSL Labs without any additional configuration. If SSL is not working after 30 minutes, verify your DNS records are correct and there are no conflicting records.

Chapter 16B: Cost Optimization

16B.1 When Free Tiers Run Out

Understanding when free tiers expire is critical for budget planning. Many vibe coding guides gloss over costs, but production applications quickly exceed free tier limits as user bases grow. Below is a realistic cost projection for TaskFlow at various scales.

Resource	Free Tier Limit	TaskFlow at 100 Users	TaskFlow at 1,000 Users	TaskFlow at 10,000 Users
Vercel (hosting)	100GB bandwidth/mo	~5GB (~$0)	~50GB (~$0)	~500GB (~$150/mo)
Neon (PostgreSQL)	0.5GB storage, 190hr compute	~100MB (~$0)	~2GB (~$19/mo)	~20GB (~$69/mo)
Sentry (errors)	5K events/mo	~1K (~$0)	~10K (~$26/mo)	~100K (~$80/mo)
Vercel Analytics	2,500 events/mo	~1K (~$0)	~10K (~$14/mo)	Included in Pro
TOTAL		~$0/mo	~$59/mo	~$299/mo

16B.2 Optimization Strategies

Area	Strategy	Estimated Savings	Implementation Effort
Images	Next.js Image optimization + WebP format	40–60% bandwidth reduction	Low (built-in)
API Caching	Cache GET responses with stale-while-revalidate	50–70% fewer DB queries	Medium
Database	Connection pooling via Prisma + select() for partial reads	30–50% compute reduction	Low
Bundle Size	Dynamic imports + tree-shaking analysis	20–40% smaller JS	Medium
Static Pages	ISR (Incremental Static Regeneration) for reports	80–90% fewer function invocations	Low
Error Noise	Filter repetitive errors + set Sentry sample rate to 0.1	90% fewer Sentry events	Low

PART VII

MAINTENANCE & ITERATION

Software Is Never Finished

Chapter 17: Bug Tracking & Performance Monitoring

17.1 Bug Priority System (P0–P4)

Priority	Definition	Response Time	Examples
P0 Critical	System down, data loss, security breach	Fix within hours	Production crash, auth bypass, data corruption
P1 High	Major feature broken, many users affected	Fix within 1 day	Payment fails, tasks not saving, login broken
P2 Medium	Feature partially broken, workaround exists	Fix within 1 week	Drag-drop glitchy on mobile, report formatting
P3 Low	Minor visual issue, rare edge case	Fix in next sprint	Typo, color mismatch, tooltip position
P4 Cosmetic	Nice-to-have improvement	Backlog	Animation smoothness, icon alignment

17.2 AI-Assisted Bug Fixing Workflow

The AI-assisted debugging workflow follows five steps: (1) Paste the error log, stack trace, and surrounding context to AI, (2) AI diagnoses the root cause and suggests a fix, (3) You verify the fix makes logical sense and doesn't introduce new issues, (4) AI generates a regression test to prevent recurrence, (5) You review, test, and merge. Typical resolution time drops from 2 hours (manual) to 15–30 minutes with AI assistance. For P0/P1 issues, an experienced developer should still make the final call.

17.3 Performance Monitoring Targets

Metric	Target	How to Measure	Action if Failing
LCP (Largest Contentful Paint)	< 2.5s	Vercel Analytics	Optimize images, reduce JS bundle, add caching
FID (First Input Delay)	< 100ms	Web Vitals API	Code-split large bundles, reduce main thread work
CLS (Cumulative Layout Shift)	< 0.1	Lighthouse	Set explicit image dimensions, avoid layout shifts
API Response p95	< 500ms	Server logs / Sentry	Add DB indexes, optimize queries, implement caching
Error Rate	< 0.1%	Sentry dashboard	Fix top errors, add React error boundaries
Database Query	<	Prisma query	Add indexes, fix N+1 queries, use select()

Metric	Target	How to Measure	Action if Failing
Time	100ms	logs	

Chapter 18: Feature Iteration & Dependency Management

18.1 Mini-SDLC for Every New Feature

After MVP launch, every new feature follows a mini-SDLC cycle that mirrors the full six-phase framework: (1) Requirement: write a user story with acceptance criteria, (2) Design: update schema, API design, and wireframe as needed, (3) Implement: build as a vertical slice, (4) Test: write unit and E2E tests, (5) Deploy: merge to main, auto-deploy via CI/CD, (6) Monitor: watch error rates and performance metrics. Target timeline: 1–3 days for small features, 1–2 weeks for large features.

18.2 Dependency Update Strategy

```
# Check for outdated dependencies
$ npm outdated

# Safe update (patch + minor versions)
$ npm update
$ npx vitest run  # ALWAYS test after updating!

# Setup automated dependency updates
# Create .github/dependabot.yml:
version: 2
updates:
  - package-ecosystem: npm
    directory: /
    schedule:
      interval: weekly
    open-pull-requests-limit: 10
```

18.3 CLAUDE.md Update Protocol

CLAUDE.md must be treated as a living document. It should be updated whenever any of the following occurs: a new dependency is added, a coding convention changes, a new database table or column is created, an API endpoint is added or modified, a security rule is established, or an architecture decision is made. Outdated CLAUDE.md files are worse than no file at all because they feed AI incorrect context.

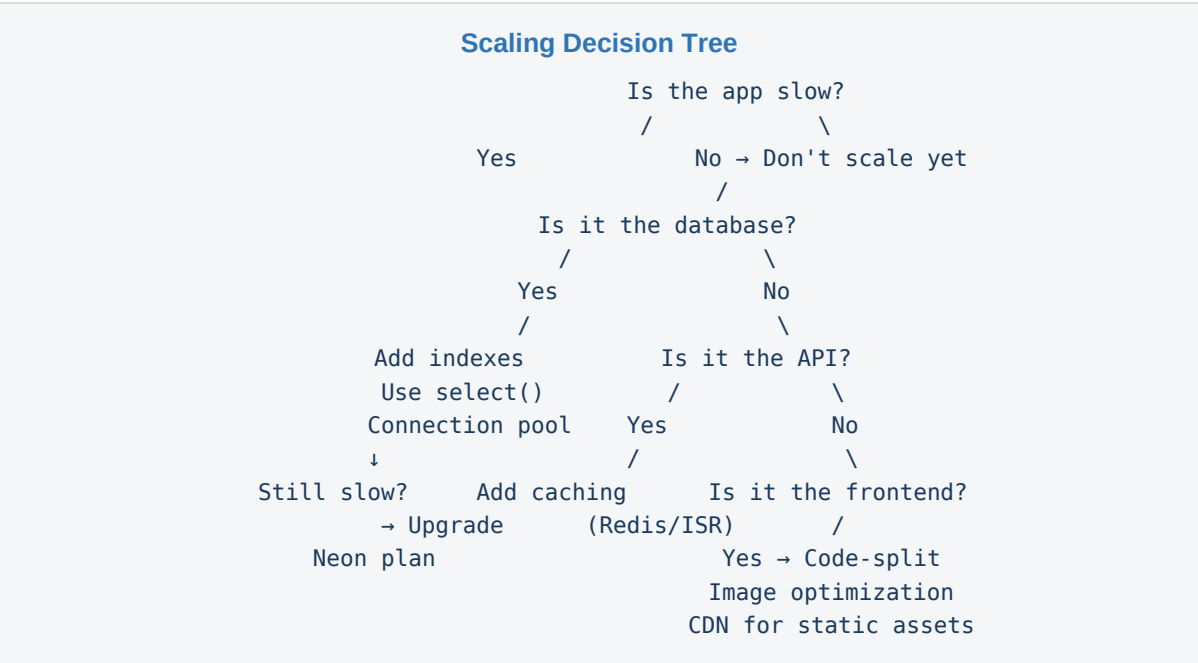
Chapter 18B: Scaling Playbook

18B.1 When to Scale

Scaling decisions should be driven by metrics, not assumptions. TaskFlow's PRD targets 1,000 concurrent users, but the MVP architecture (Vercel serverless + Neon PostgreSQL) can comfortably handle this without modification. The question is: when does the architecture need to evolve? The answer is found in monitoring data, not gut feeling.

Signal	Threshold	Action Required
API response p95	> 1 second consistently	Add database indexes; implement response caching
Database connections	> 80% of pool limit	Enable Prisma connection pooling; consider PgBouncer
Vercel function duration	> 10 seconds regularly	Optimize slow queries; add background job processing
Database storage	> 80% of plan limit	Upgrade Neon plan; archive old data
Monthly bandwidth	> 80% of Vercel limit	Add CDN for static assets; optimize image delivery
Error rate	> 1% sustained	Performance audit; fix top error sources
User complaints	Increasing trend	User experience audit; load testing

18B.2 Scaling Decision Tree



18B.3 Architecture Evolution Path

TaskFlow's architecture should evolve gradually based on actual needs, not anticipated ones. Premature optimization is a common anti-pattern in vibe coding because AI makes it easy to add complexity. Below is the recommended evolution path, where each stage is triggered by specific metrics rather than user count assumptions.

Stage	Architecture	Trigger	Key Changes
1. MVP	Vercel + Neon Free	Launch	Serverless functions, managed Postgres, auto-scale
2. Growth	Vercel Pro + Neon Scale	p95 > 500ms or 5K users	Connection pooling, Redis cache, ISR for reports
3. Scale	Vercel Enterprise + dedicated DB	p95 > 1s or 50K users	Background jobs (Inngest), WebSocket server, CDN
4. Platform	Custom infra (K8s/ECS)	Complex requirements	Microservices, event-driven, multi-region

18B.4 The YAGNI Principle in Scaling

YAGNI (You Aren't Gonna Need It) is especially important in the vibe coding era because AI makes it trivially easy to add microservices, message queues, and distributed caches before they are needed. Every piece of infrastructure you add is a piece you must maintain, monitor, and pay for. The correct approach is: start simple, measure everything, and add complexity only when metrics demand it. Most applications never reach Stage 3. Almost no MVPs need Stage 4. If you find yourself designing a Kubernetes deployment for a product that has zero users, you have succumbed to architecture astronautics, and no amount of AI assistance can save you from that.

PART VIII

ADVANCED TOPICS

Criticism, the Future, and the Evolution of Methodology

Chapter 19: Criticism, Failures & The Vibe Coding Hangover

19.1 Documented Security Incidents

The “Vibe Coding Hangover” describes the wave of problems that surfaced after the initial euphoria. Fast Company reported that 16 of 18 CTOs surveyed had experienced production disasters from AI-generated code. The Lovable vulnerability (CVE-2025-48757) exposed critical Row Level Security misconfigurations in 170 of 1,645 applications—10.3% had critical flaws, with one app leaking data of 18,000+ users including K-12 students. Escape.tech’s analysis of 5,600+ publicly accessible vibe-coded apps found 2,000+ vulnerabilities and 400+ exposed secrets. The Tea App breach (July 2025) exposed 72,000 images including 13,000 government ID photos through completely open Firebase storage.

19.2 Developer Skill Atrophy

Stack Overflow’s 2025 survey found that 20% of developers felt less confident in their own problem-solving abilities after intensive AI tool usage. Overall positive sentiment toward AI coding tools dropped from 77% (2023) to 60% (2025). A Stanford study found that employment among software developers aged 22–25 fell nearly 20% between 2022 and 2025, raising serious questions about the pipeline of engineering talent being developed.

19.3 Technical Debt Projections

Analysts project $1.5 trillion in accumulated technical debt by 2027 driven by vibe coding approaches, with an estimated 8,000+ startups requiring complete codebase rebuilds. GitClear’s analysis of 211 million lines showed code refactoring declining from 25% to under 10%, code duplication rising 4x, and code churn nearly doubling—all indicators of mounting technical debt.

19.4 The Indonesian Developer Perspective

Indonesia has developed a remarkably active vibe coding ecosystem with platforms like vibengoding.id (7 free courses, 70+ lessons), VibeDev ID (community forums and weekly events), and BelajarVibeCoding.com. Universities including BINUS, Telkom University, and Universitas Siber Muhammadiyah have engaged formally. However, Indonesian developer sentiment leans skeptical—BINUS reported 68% of users experienced debugging difficulties, and developers are described as a “willing but reluctant” generation using AI due to FOMO but not trusting results.

Chapter 20: Spec-Driven Development — The Future

20.1 From Vibes to Specifications

The most significant evolution in AI-assisted development is the emergence of Spec-Driven Development (SDD) as vibe coding's disciplined successor. GitHub's Spec Kit (released September 2024) provides a structured CLI workflow: /specify → /plan → /tasks → /implement. Amazon's Kiro IDE enforces a pipeline of spec → design → tasks → implementation. The open-source BMAD Method simulates a full Agile team with 21 specialized AI agents.

Addy Osmani's book Beyond Vibe Coding (O'Reilly, 2025) defines an "AI-Assisted Development Spectrum" from pure vibe coding to AI-assisted engineering and identifies "The 70% Problem"—AI gets you 70% of the way quickly, but the last 30% requires deep engineering expertise that cannot be vibed through.

20.2 The Five Principles of the Vibe Coding Framework

The community-driven Vibe Coding Framework (docs.vibe-coding-framework.com) codifies five core principles for responsible AI-assisted development:

#	Principle	Description
1	Augmentation Not Replacement	AI amplifies human capabilities; it does not substitute for them
2	Verification Before Trust	Every AI output must be verified before acceptance
3	Maintainability First	Code must be maintainable by humans, not just functional
4	Security by Design	Security is built in from the start, not bolted on after
5	Knowledge Preservation	Documentation and understanding must be actively maintained

20.3 The Academic Frontier

Research on vibe coding is maturing rapidly. The first VibeX International Workshop on Vibe Coding and Vibe Researching is scheduled for the EASE 2026 conference in Glasgow (June 2026). Multiple systematic studies have been published on arXiv, and the field is developing formal taxonomies, evaluation frameworks, and best-practice guidelines grounded in empirical evidence rather than anecdote.

Chapter 19B: OWASP Top 10 for Vibe-Coded Applications

19B.1 Why OWASP Matters More for AI-Generated Code

The OWASP Top 10 is a standard awareness document for web application security risks, now in its 8th edition (2025 release candidate, announced November 6, 2025). For vibe-coded applications, OWASP compliance is not optional—it is existential. Veracode's 2025 data shows 45% of AI-generated code contains OWASP vulnerabilities. The 2025 edition introduces a new category, "Software Supply Chain Failures," directly relevant to vibe coding where practitioners accept AI-suggested dependencies without verification.

This section maps each OWASP Top 10 risk to specific vibe coding scenarios and provides concrete mitigation strategies using the TaskFlow tech stack. Every mitigation has been implemented or referenced in earlier chapters of this handbook.

19B.2 OWASP Top 10 (2025) Mapped to Vibe Coding

#	OWASP Risk	Vibe Coding Scenario	TaskFlow Mitigation
A01	Broken Access Control	AI forgets authorization checks; generates endpoints without role verification	5-step pattern enforces auth + authz on every Server Action (Ch. 13)
A02	Cryptographic Failures	AI stores passwords in plaintext; uses weak hashing; hardcodes secrets	NextAuth handles hashing; all secrets in env vars (Ch. 6 CLAUDE.md rules)
A03	Injection	AI generates raw SQL queries or uses string interpolation in queries	Prisma ORM parameterizes all queries by default (Ch. 9); Zod validates all inputs (Ch. 13)
A04	Insecure Design	AI generates code without threat modeling; no defense-in-depth	Security architecture designed upfront (Ch. 10); ADR documents security decisions
A05	Security Misconfiguration	AI leaves debug mode on; default credentials; permissive CORS	Helmet headers in next.config.js (Ch. 10); rate limiting in middleware
A06	Vulnerable Components	AI suggests outdated packages with known CVEs	npm audit in CI/CD (Ch. 15); Dependabot weekly updates (Ch. 18)
A07	Auth Failures	AI implements custom auth with flaws; no brute-force protection	NextAuth.js v5 handles auth; rate limit 10 req/min on auth endpoints (Ch. 10)
A08	Data Integrity Failures	AI uses untrusted CDN links; no subresource integrity	Dependencies pinned in package-lock.json; npm audit on every PR

#	OWASP Risk	Vibe Coding Scenario	TaskFlow Mitigation
A09	Logging Failures	AI generates no logging; errors silently swallowed	Sentry integration (Ch. 16); console.error in catch blocks; structured logging
A10	SSRF	AI generates fetch calls to user-supplied URLs without validation	Validate all URLs server-side; whitelist allowed external domains

19B.3 The Lovable Incident: A Case Study in A01

The Lovable vulnerability (CVE-2025-48757) is the most significant documented security incident in the vibe coding era and serves as a textbook example of OWASP A01 (Broken Access Control). Lovable is an AI-powered application builder that generates full-stack applications from natural language descriptions. The platform uses Supabase as its database layer, which provides Row Level Security (RLS) as a built-in access control mechanism.

The problem: AI-generated applications consistently failed to configure RLS policies correctly. Of 1,645 applications scanned, 170 (10.3%) had critical Row Level Security misconfigurations that allowed any authenticated user—or in some cases, any anonymous visitor—to read, modify, or delete data belonging to other users. One particularly severe case exposed data of over 18,000 users, including K-12 students, through a completely open database.

The root cause was systemic rather than incidental. The AI code generator understood how to create database tables and write CRUD operations but did not consistently generate the corresponding security policies. This is precisely the kind of error that the 5-step Server Action pattern in this handbook (Chapter 13) is designed to prevent: by making authorization a mandatory step in every data operation, the risk of forgotten access controls is dramatically reduced.

Key Lesson from the Lovable Incident

AI code generators are optimized to produce code that works (functional correctness), not code that is secure (security correctness). Access control, input validation, and authorization must be enforced through systematic patterns and automated checks, never left to AI's discretion. The CLAUDE.md rule "ALWAYS check authorization" exists specifically to prevent this class of vulnerability.

19B.4 The Tea App Breach: A Case Study in A05

The Tea App breach (July 2025) exposed 72,000 images including 13,000 government identification documents through completely unsecured Firebase storage. The application was built using vibe coding techniques with a Firebase backend. The AI-generated code created Firebase storage buckets with default

permissions that allowed public read access to all uploaded files—including highly sensitive identity documents.

This incident illustrates OWASP A05 (Security Misconfiguration) in its purest form. The AI generated functional code that successfully uploaded and retrieved files, but the storage bucket security rules were left in their default permissive state. No human reviewed the Firebase security rules before deployment. No security scanning tool was included in the deployment pipeline. The breach was only discovered when a security researcher found the exposed bucket during routine scanning.

Prevention in the TaskFlow context: the Launch Checklist (Chapter 16) includes explicit security header verification and access control auditing. The CI/CD pipeline (Chapter 15) includes npm audit for dependency scanning. For applications using Firebase or Supabase, an additional step should be added to verify storage and database security rules programmatically before deployment.

Chapter 20C: Real-World Case Studies

20C.1 Success: YC Winter 2025 Batch

Y Combinator's Winter 2025 batch provides the most visible success story for AI-assisted development. Jared Friedman revealed that 25% of startups in the batch had codebases that were 95% AI-generated. These companies were able to build MVPs in weeks rather than months, iterate based on user feedback rapidly, and focus founder time on product-market fit rather than coding. The critical success factor, according to multiple YC founders, was not the AI tools themselves but the clarity of specifications and requirements provided to those tools.

However, the YC data also contains a cautionary element. Several companies from the batch have subsequently reported significant challenges scaling their AI-generated codebases beyond MVP stage. The code that was fast to generate proved difficult to maintain, refactor, and extend. This aligns with Addy Osmani's "70% Problem": AI excels at getting you to a working prototype quickly, but the engineering effort required for production hardening, performance optimization, and long-term maintenance is substantial and cannot be vibed through.

20C.2 Success: Neuronworks Indonesia

PT Neuronworks Indonesia in Bandung provides a counter-example to the prevailing Indonesian skepticism about vibe coding. A three-person team successfully used AI-assisted development to build production AI analytics platforms for enterprise clients. Their approach mirrors this handbook's methodology: detailed specifications written upfront, CLAUDE.md-style context files for every project, mandatory code review of all AI output, and comprehensive testing. Their success demonstrates that vibe coding can work for production software when practiced with discipline—the key word being discipline.

20C.3 Failure: Fast Company CTO Survey

Fast Company's September 2025 investigation surveyed 18 CTOs about their experience with AI-generated code in production. The results were sobering: 16 of 18 (89%) had experienced at least one production disaster directly attributable to AI-generated code. Common failure modes included security vulnerabilities that passed code review (because reviewers trusted the AI), performance degradation from inefficient queries that worked fine with small datasets but failed at scale, and subtle logic bugs in edge cases that AI-generated tests did not cover.

The most frequently cited root cause was not the AI tools themselves but organizational practices: pressure to ship faster led teams to reduce code review thoroughness, skip manual testing, and deploy without adequate monitoring. In

every case, the failure could have been prevented by following standard SDLC practices—the same practices this handbook systematizes.

20C.4 Lessons Across All Case Studies

Three patterns emerge consistently across success and failure case studies:

1. Specification quality is the primary determinant of outcome. Teams that invest 2–4 hours in requirements, PRD, and CLAUDE.md consistently outperform teams that jump straight to prompting AI. The specification is the multiplier; AI is the multiplicand. A 10x multiplier on a zero-quality specification still produces zero.

2. Code review discipline cannot be relaxed because AI wrote the code. If anything, review must be more rigorous for AI-generated code because AI produces plausible-looking code that may contain subtle errors—errors that human-written code is less likely to contain because humans have domain context that AI lacks.

3. Testing and monitoring are non-negotiable regardless of how the code was produced. The YC successes included comprehensive testing; the Fast Company failures did not. The difference is not whether AI tools were used, but whether engineering discipline was maintained around those tools.

Chapter 20B: AI Coding Tools Comparison

20B.1 Major AI Coding Tools (2026)

The AI coding tool landscape has evolved rapidly since Karpathy's original tweet. Understanding the strengths, limitations, and optimal use cases for each tool is essential for choosing the right workflow. Below is a comprehensive comparison of the major tools available as of March 2026, evaluated across dimensions that matter most for structured vibe coding.

Tool	Type	Best For	Context Window	Price (Individual)
Claude Code	CLI agent	Agentic coding, complex refactoring, multi-file changes	200K tokens	$20/mo (Pro) or API usage
Cursor	IDE (VS Code fork)	In-editor AI, tab completion, inline chat, composer mode	Variable by model	$20/mo (Pro)
GitHub Copilot	IDE extension	Tab completion, inline suggestions, Copilot Workspace	Variable by model	$10–$39/mo
Windsurf (Codeium)	IDE	Fast completions, Cascade multi-file agent	128K tokens	Free tier generous
Amazon Q Developer	IDE extension + CLI	AWS-integrated development, security scanning	Variable	Free tier + Pro $19/mo
Gemini Code Assist	IDE extension + CLI	Google Cloud integration, large context	1M+ tokens	Free tier + Standard

20B.2 Choosing the Right Tool for Your Workflow

The choice of AI coding tool depends on your workflow preferences, project complexity, and team size. For solo developers working on MVP projects (like TaskFlow), Claude Code or Cursor are the recommended choices because they offer the strongest agentic capabilities—the ability to make multi-file changes, run commands, and understand project-wide context. For larger teams with existing GitHub workflows, GitHub Copilot integrates seamlessly with pull requests and code review processes. For teams heavily invested in AWS or Google Cloud, the respective vendor tools offer deeper infrastructure integration.

Importantly, these tools are not mutually exclusive. Many developers use a combination: Cursor for day-to-day in-editor coding, Claude Code for complex refactoring tasks and architectural changes, and GitHub Copilot for quick inline completions during code review. The CLAUDE.md and AGENTS.md standards

ensure consistent behavior regardless of which tool is being used at any given moment.

20B.3 Tool Configuration for Structured Vibe Coding

Each tool reads project context from a different configuration file, though the industry is converging on AGENTS.md as a cross-platform standard. Ensuring your project has the correct configuration files for your tool stack is a critical setup step that many developers skip, leading to inconsistent AI output.

Tool	Config File	Location	Key Contents
Claude Code	CLAUDE.md	Project root + subdirectories	Rules, tech stack, patterns, conventions
Cursor	.cursor/rules/*.mdc	.cursor/ directory	4 rule types: Always, Auto, Agent, Manual
GitHub Copilot	.github/copilot-instructions.md	.github/ directory	Project conventions, style preferences
All tools	AGENTS.md	Project root	Cross-platform agent instructions (Linux Foundation standard)

APPENDICES

Appendix A: Anti-Patterns Reference

Anti-pattern	What Happens	Solution
No CLAUDE.md	AI generates inconsistent code, wrong framework choices	ALWAYS create CLAUDE.md before writing any code
Scope Creep	"Just one more feature" endlessly; project never ships	Define MVP scope; stick to it. Phase 2 for everything else.
No User Stories	Coding without knowing what “done” looks like	Every feature MUST have acceptance criteria BEFORE coding
Wrong Tech Stack	Niche framework, poor AI support, many bugs	Choose popular stacks (Next.js, React, PostgreSQL)
Skip Scaffolding	Start coding in empty folder, structure becomes chaotic	5 minutes of setup saves 5 hours of refactoring
Over-planning	2 weeks of planning, 0 lines of code. Analysis paralysis.	Planning = 2–4 hours for medium projects. Then START.
Accept All Diffs	AI code enters production without review	ALWAYS review every change. Never blind-accept.
No Testing	"Works on my machine" = production crash	Minimum: unit tests + E2E for all critical paths
Hardcoded Secrets	API keys in source code, leaked to GitHub	ALWAYS use environment variables; .env.local gitignored
No Monitoring	No visibility when production is broken	Sentry + uptime monitoring is the bare minimum

Appendix B: Phase Checklists

Phase 1: Requirements & Planning (~3 hours)

#	Deliverable	Output File	Time
1	Clear vision statement	Section in PRD.md	15 min
2	Target user persona + anti-persona	Section in PRD.md	10 min
3	MVP scope (3–5 core features)	Feature list in PRD.md	20 min
4	Complete PRD	PRD.md	30 min
5	CLAUDE.md written	CLAUDE.md at repo root	30 min
6	User stories with acceptance criteria	user-stories.md	45 min
7	Tech stack decided and documented	In CLAUDE.md	15 min
8	Project scaffolded	GitHub repo + initial commit	10 min
9	Sprint plan with estimates	Task list / board	15 min

Phase 2: Design & Architecture (~2.5 hours)

#	Deliverable	Output File	Time
1	Entity Relationship Diagram	docs/erd.md or SVG	20 min
2	Complete Prisma schema	prisma/schema.prisma	30 min
3	API endpoint design table	docs/api-design.md	20 min
4	Standard response format	src/lib/api-response.ts	10 min
5	UI wireframes (text descriptions)	docs/wireframes.md	30 min
6	Component architecture tree	docs/components.md	15 min
7	Security architecture documented	CLAUDE.md + docs/security.md	15 min
8	ADRs for major decisions	docs/adr/*.md	20 min

Appendix C: Glossary

Term	Definition
Vibe Coding	AI-first approach where developers surrender coding to AI. Coined by Andrej Karpathy, February 2, 2025.
CLAUDE.md	Project configuration file read by Claude Code at session start; contains rules, stack, and context.
AGENTS.md	Open cross-platform standard for AI coding agent instructions. Linux Foundation, August 2025. 60K+ projects.
Vertical Slicing	Building features end-to-end (DB + API + UI + Test) before moving to the next feature.
PRD	Product Requirements Document. Defines WHAT is built, FOR WHOM, and HOW success is measured.
Atomic Design	UI methodology by Brad Frost (2013): atoms → molecules → organisms → templates → pages.
ADR	Architecture Decision Record. Documents WHY an architecture decision was made, not just WHAT.
Testing Pyramid	Principle by Mike Cohn (2009): many unit tests (70%), some integration (20%), few E2E (10%).
SDD	Spec-Driven Development. Structured evolution: specify → plan → tasks → implement.
METR Study	Most rigorous RCT: 16 developers, 246 tasks. AI tools made developers 19% SLOWER.
Agentic Engineering	Karpathy’s 2026 successor term for “vibe coding”—more mature and structured.
OWASP Top 10	Top 10 web application security risks. 2025 edition adds Software Supply Chain Failures.
The 70% Problem	Osmani’s observation: AI gets you 70% done fast; the last 30% requires deep engineering.
Vibe Coding Hangover	The wave of security breaches, tech debt, and failures following the vibe coding euphoria.

Appendix D: References

Primary Sources:

Karpathy, A. (2025). “There’s a new kind of coding I call vibe coding.” X/Twitter, February 2, 2025.

Karpathy, A. (2026). One-year retrospective on vibe coding. X/Twitter, February 2026.

METR. (2025). Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity. arXiv:2507.09089.

Bossavit, L. (2015). The Leprechauns of Software Engineering. Leanpub.

Osmani, A. (2025). Beyond Vibe Coding: A Guide to AI-Assisted Development. O’Reilly Media.

Cohn, M. (2004). User Stories Applied: For Agile Software Development. Addison-Wesley.

Cohn, M. (2009). Succeeding with Agile: Software Development Using Scrum. Addison-Wesley.

Frost, B. (2016). Atomic Design. Self-published (atomicdesign.bradfrost.com).

Nygard, M. (2011). Documenting Architecture Decisions. Blog post, November 15, 2011.

Technical Documentation:

Anthropic. Claude Code & CLAUDE.md documentation. docs.anthropic.com.

OpenAI, Sourcegraph, Google, Cursor, et al. (2025). AGENTS.md specification.

Vercel. Next.js 15 documentation. nextjs.org/docs.

Prisma. Prisma ORM documentation. prisma.io/docs.

OWASP Foundation. OWASP Top 10 — 2025. owasp.org/Top10.

Security & Quality Data:

Veracode. (2025). State of Software Security report. 45% of AI code contains vulnerabilities.

Escape.tech. (2025). 2,000+ vulnerabilities found in 5,600+ publicly accessible vibe-coded apps.

CVE-2025-48757. Lovable Row Level Security misconfiguration affecting 170 applications.

GitClear. (2025). Analysis of 211 million lines of code: duplication up 4x, refactoring down to <10%.

CodeRabbit. (2025). 470 GitHub PRs: AI co-authored code has 1.7x more major issues, 2.74x more security vulns.

Productivity Studies:

Peng, S. et al. (2023). The Impact of AI on Developer Productivity. arXiv:2302.06590.

Google. (2024). Internal RCT with 96 engineers: 21% faster on specific logging task.

McKinsey & Company. (2023). Unleashing Developer Productivity with Generative AI.

Google DORA. (2024). Every 25% increase in AI adoption = 1.5% speed dip + 7.2% stability drop.

Frameworks & Standards:

AWS. (2025). AI-Driven Development Life Cycle (AI-DLC). aws.amazon.com/blogs/devops.

Microsoft. (2025). An AI-led SDLC with Azure and GitHub. techcommunity.microsoft.com.

Casey West. (2025). The Agentic Manifesto: Engineering in the Era of Autonomy. caseywest.com.

Vibe Coding Framework community. (2025–2026). docs.vibe-coding-framework.com.

SDLC VIBE CODING HANDBOOK

AI is a co-pilot, not an autopilot.

What separates an ordinary developer from a great one
is not coding speed —
it is the quality of thinking before coding.

Jekardah Tech Review Desk
rominur@gmail.com
jekardah.com • t.me/Jekardah_AI
2026 Edition

Appendix E: Complete Prompt Library

E.1 Planning Phase Prompts

Task	Prompt Template
Brain dump to vision	I want to build [description]. Features in mind: [list]. Help me: validate idea, identify core vs nice-to-have, define persona, write vision statement, suggest MVP scope.
Generate PRD	Based on this vision: [paste vision]. Generate a PRD with sections: Vision & Goals, Target Users (include anti-persona), Core Features (MVP only), Non-functional Requirements, Success Metrics, Out of Scope.
Write user stories	For feature [name]: Write user stories in "As a [user], I want [action] so that [benefit]" format. Include 4–6 specific, testable acceptance criteria per story. Use IDs like US-001.

Task	Prompt Template
Select tech stack	For a [type] application targeting [audience] with these features: [list]. Recommend a tech stack optimized for: (1) AI coding assistant support, (2) developer productivity, (3) deployment simplicity. Justify each choice.

E.2 Design Phase Prompts

Task	Prompt Template
Generate ERD	Based on this PRD: [paste]. Design an Entity Relationship Diagram with: table names, column names and types, primary/foreign keys, cardinality, indexes for common queries.
Write Prisma schema	Convert this ERD: [paste] into a complete Prisma schema with: enum types, relations, default values, composite indexes, timestamps (createdAt/updatedAt).
Design API endpoints	Based on these user stories: [paste]. Design RESTful API endpoints in table format: Method, Path, Description, Auth Level, Request Body, Response Format. Use {data, error, meta} standard.
Create wireframe	Design a text-based wireframe for [page name]. Include: layout structure (sidebar/main/header), component list with approximate sizing, interaction descriptions, responsive behavior notes.

E.3 Implementation Phase Prompts

Task	Prompt Template
Implement server action	Given CLAUDE.md context and API design, implement [action name] as a Server Action following the 5-step pattern: auth check, Zod validation, authorization, try/catch DB operation, cache revalidation. Return {data, error} format.
Build UI component	Convert this wireframe: [paste] to a React component. Use: TypeScript, Tailwind CSS, proper props typing, loading/error states. Follow Atomic Design—this is a [atom/molecule/organism].
Complete the pattern	Here is the complete [Feature A] implementation: [paste files]. Create [Feature B] following the EXACT same patterns for: server actions, UI components, validation schemas, and tests. [Feature B]-specific: [requirements].
Debug error	Error: [paste full error + stack trace]. Context: [what I was doing]. File: [filename]. This error occurs when [trigger]. Fix it and explain the root cause so I can avoid it in the future.

E.4 Testing Phase Prompts

Task	Prompt Template
Generate unit tests	Read [file path] and generate comprehensive unit tests: (1) Test every Zod validation rule, (2) Test happy path and error paths, (3) Edge cases: empty string, max length, SQL injection, (4) Use Vitest, AAA pattern. Generate at least 12 test cases.
Generate E2E test	Write a Playwright E2E test for user story: [paste US with acceptance criteria]. Include: login flow, navigation, action, verification of each AC. Use data-testid attributes. Add meaningful wait conditions.
Security review	Review [file] for: (1) Injection vulnerabilities, (2) Auth bypass, (3) Data leaks, (4) Missing input validation, (5) Hardcoded secrets. Format: CRITICAL / WARNING / INFO with specific fix instructions.
AI code review	Review [file] for: security, performance (N+1 queries, missing indexes), error handling completeness, TypeScript strictness, edge cases, and CLAUDE.md rule compliance. Output: CRITICAL / WARNING / INFO.

E.5 Deployment & Maintenance Prompts

Task	Prompt Template
Setup CI/CD	Generate a GitHub Actions workflow for a Next.js + Prisma project that: type-checks, lints, runs unit tests, runs E2E tests (Playwright), runs security audit, applies DB migrations, and deploys to Vercel. Include proper secret handling.
Diagnose performance	Our API endpoint [path] has p95 latency of [X]ms (target: <500ms). Prisma query log shows: [paste]. Database: PostgreSQL. Suggest: specific indexes to add, query optimizations, caching strategies.
Plan migration	I need to add [column/table] to the production database. Current schema: [paste relevant model]. Write: (1) Prisma migration, (2) Data migration script if needed, (3) Rollback plan, (4) Verification queries to run after migration.
Refactor code	Refactor [file]: (1) Extract reusable hooks/utils, (2) Split into files <200 lines, (3) Add TypeScript types (remove any), (4) Add JSDoc. CRITICAL: Write tests for current behavior FIRST, then refactor, then verify tests pass.

Appendix F: Quick Reference Cards

F.1 Server Action 5-Step Pattern

```
// EVERY Server Action follows this pattern:
export async function actionName(formData: FormData) {
  // 1. AUTH — verify user is logged in
  const session = await getServerSession();
  if (!session?.user) return { error: "Unauthorized" };

  // 2. VALIDATE — parse input with Zod
  const parsed = Schema.safeParse(Object.fromEntries(formData));
  if (!parsed.success) return { error: parsed.error.flatten() };

  // 3. AUTHORIZE — check permissions
  // (user must be team member, admin for delete, etc.)

  // 4. OPERATE — database operation in try/catch
  try {
    const result = await prisma.model.create({ data: {...} });
    // 5. REVALIDATE — clear cache for affected pages
    revalidatePath("/affected-page");
    return { data: result };
  } catch (err) {
    console.error(err);
    return { error: "Operation failed" };
  }
}
```

F.2 Git Commit Message Convention

Prefix	When to Use	Example
feat:	New feature or functionality	feat(task): add drag-drop between columns
fix:	Bug fix	fix(timer): prevent negative elapsed time
test:	Adding or updating tests	test(task): add integration tests for CRUD
docs:	Documentation changes	docs: update CLAUDE.md with new API endpoints
refactor:	Code restructuring (no behavior change)	refactor(board): extract useBoard custom hook
chore:	Build, config, dependency updates	chore: upgrade Prisma to 5.x
style:	Formatting, whitespace (no logic change)	style: fix ESLint warnings in components/
perf:	Performance improvement	perf(db): add composite index on [boardId,

Prefix	When to Use	Example
		status]

F.3 Prisma Common Commands

Command	Environment	What It Does
npx prisma generate	Dev + CI	Regenerate TypeScript client from schema
npx prisma db push	Dev ONLY	Sync schema to DB (may lose data)
npx prisma migrate dev	Dev	Create migration file + apply to dev DB
npx prisma migrate deploy	CI/Production	Apply pending migrations (safe, sequential)
npx prisma migrate status	Any	Show pending migrations
npx prisma studio	Dev	Open visual DB browser at localhost:5555
npx prisma db seed	Dev	Run seed script for test data

F.4 Essential Terminal Commands

Task	Command
Start dev server	npm run dev
Type check (no emit)	npx tsc --noEmit
Run all linting	npx eslint . --max-warnings=0
Run unit tests	npx vitest run
Run unit tests (watch)	npx vitest
Run E2E tests (headless)	npx playwright test
Run E2E tests (visible browser)	npx playwright test --headed
Run security audit	npm audit --production
Check for outdated packages	npm outdated
Generate test coverage report	npx vitest run --coverage
Build production bundle	npm run build
Analyze bundle size	npx @next/bundle-analyzer