Cleaning Up a Vibe-Coded AI App and Shipping It to Production

When AI Writes Code, It Optimizes for “Works,” Not “Safe”

One of the first things I noticed was how easy it was to introduce security vulnerabilities when using tools like Supabase in combination with AI-generated code.

Company joins by code — and why that’s dangerous

To allow users to join a company, the app used a “company code” flow. Reasonable idea. The implementation, however, exposed a critical flaw.

The AI had generated logic that allowed *any authenticated user* to query companies by their join code in order to validate it. From the UI, this seemed harmless. But from the browser’s dev tools, it was trivial to call the underlying Supabase function directly and enumerate companies or retrieve sensitive metadata.

There was no meaningful boundary between:

- “the UI is allowed to do this”
- and “the backend must enforce this.”

The AI assumed the frontend was the gate. Attackers don’t respect that assumption.

Usage limits that existed only in the UI

The same pattern appeared with usage limits.

The UI respected limits. Buttons disabled correctly. Warnings showed up at the right time. But the Supabase functions themselves had **no enforcement**.

Anyone with basic knowledge of dev tools could bypass limits entirely by calling the functions directly. From the system’s perspective, nothing was wrong—the requests were valid.

This is a recurring theme with AI-generated systems: **policies exist conceptually, but not structurally.**

Velocity Without Foundations Is an Illusion

Another challenge wasn’t technical—it was organizational.

The product owner had built much of the app through “vibe coding”: iterating quickly, prioritizing visible progress, and reacting to stakeholder feedback in real time. That approach can be useful early on, but it creates blind spots.

While I was setting up:

- proper environment separation
- a branching strategy
- end-to-end tests

- and a staging pipeline that actually mirrored production

I kept hearing variations of:

What often goes unspoken is the assumption embedded in that question: that infrastructure work is optional, or secondary.

But stakeholders don't just care about UI polish. They care about:

- not breaking production
- predictable releases
- confidence that changes won't introduce silent regressions

A staging environment with real tests isn't overhead. It's the minimum cost of operating software responsibly.

Fix the Layout Before Adding More Features

Another lesson became clear early: the UI layout itself needed to change *before* new features were added.

When features are layered onto a layout that hasn't stabilized, every new addition increases cognitive and technical debt. Components become overloaded. State management becomes fragile. Small UI changes ripple unpredictably.

Cleaning this up early reduced the cost of every future change. It's the kind of decision that looks slow in the moment and fast in hindsight.

Re-thinking the Backend

Part of the cleanup involved reevaluating the backend entirely.

The original implementation leaned heavily on auto-generated logic and client-side orchestration. It worked—but it was difficult to reason about, test, and secure.

Moving critical paths to a Go backend gave us:

- explicit boundaries
- clearer ownership of business logic
- predictable performance
- and the ability to write tests that actually meant something

This wasn't about language preference. It was about control.

AI is excellent at scaffolding. Production systems still need *intentional architecture*.

Taking the Feature to Production

Once the foundations were in place, one of the previously "working" features could finally be shipped properly.

Not rewritten from scratch. Not replaced. Just corrected, hardened, and made real.

The result looked boring from the outside. That’s a compliment.

It behaved correctly under edge cases. It enforced rules at the backend. It survived being poked at from dev tools. It could be deployed with confidence.

That’s what production readiness actually feels like.

What This Experience Reinforced

AI has changed how quickly software can be assembled—but it hasn’t changed what makes software *good*.

The hard parts are still the same:

- defining boundaries
- enforcing invariants
- designing for failure
- and building systems that hold up when nobody is watching

AI can get you to “it works” faster than ever. Engineers are still needed to get you from “it works” to “we can ship this.”

That gap is where real engineering lives.