Lessons from a Month of Vibe-Coding

Arun Rajappa / @appa / 2026.02.20

I vibe-coded 14 apps between January 6th and February 5th. You can see them at lab.appapappa.com (itself one of the vibe-coded apps).

Before getting to what I learned, a bit about the how and the why.

The How

All 14 apps were built using Codex; I used Claude Code on two of them as well. The stack: Django as the web framework, Tailwind + daisyUI for UI, with GSAP and Three.js for animations. A few use Postgres as the DB; one uses Redis for some caching. A couple of them integrate with OpenAI APIs, one of them with Resend APIs for mail integration. All the apps are deployed on Railway.

The highlight? I didn't write a single line of code. Everything was built through prompts alone.

I did handle a few technical tasks manually: Git check-ins, pushing to GitHub, setting up Railway for deployments (mostly, configuring environment variables), and making DNS record changes on GoDaddy.

The fastest app—Cuckoo—went from idea to deployed in just under two hours. The longest—Dumcee, which involved figuring out some WebRTC stuff—took about six hours. The most complete app, Bily, had 58 commits; most had fewer than 15.

During this same period, I separately experimented with building versions of my apps using Next.js/React for web; Flutter and Expo for mobile (plus one experiment with Ionic + Capacitor); and Lua/LÖVE2D for desktop/games (plus a couple of experiments with Flutter+Flame and Pygame).

The Why

Overall, I wanted to get a sense of what was the entry bar was for someone to build and deploy complete prototypes. I wanted to form an opinion on what I'd teach if I had to teach someone to vibe-code. And I wanted to get a visceral feel for the process—to understand the nitty gritty details of what works and what is painful.

As a developer, I wanted to uncover what minimal technical knowledge I needed—what aspects of building systems were encoded in my brain from years of work. I wanted to see how vibe-coding *felt*: was it empowering, disempowering, or something else entirely? I wanted to understand what stayed hard, what had become easy, and what the future looked like.

As a product manager, I wanted to push myself to create complete prototypes, and to scratch the itch of products I'd always wanted to build. I wanted to see what this did to my creative process, and to form my own opinion about what AI tools are doing to the work of building products.

Learnings & Reflections

It works. It is now possible to build and deploy complete, functional, beautiful prototypes using the state of vibe-coding in early 2026. This wasn't true a year ago.

The tools have matured. They maintain context, pick up from where they left off, and don't go off on tangents or down rabbit holes the way they used to. A year ago, I regularly had to throw away half-done apps because the tools got confused and made a big mess from which it was hard to recover. Now I can make consistent progress and have high confidence of finishing things that I start.

Vibe-coding is joyful. Many years ago, when I moved from C to Java and then to Ruby, I remember feeling a similar kind of joy. Progress in developer productivity from roughly 1990 to 2010 mostly trended upward—you could create faster, without attending to the minutiae of memory allocation; you could model human concepts faster using paradigms like object-oriented programming. Vibe-coding feels like that—you build at the speed of thought, using human language, human concepts.

However, that progress in developer productivity wasn't constant. The last 15 years of programming were a horrible morass. Web development was filled with competing frameworks, version mismatches, and browser compatibility issues. Mobile was somehow worse: multiple Android versions and device specs, different languages (Java, Kotlin, Objective-C, Swift), and a closed developer experience on iOS with and various lock-ins. Deployment was its own nightmare of Docker, AWS, and assorted yak-shaving.

What happened around 2010? My hypothesis is that it was a shift from a builder mindset to a startup/entrepreneur mindset. Everything being built suddenly needed a business plan, a path to millions of users, a plan to conquer the world. All the dev tooling shifted toward serving millions. The joy of building small was lost.

Building small is fun again. AI-coding tools break that pattern of being focused on scaled startups rather than small garage builders. Building small is fun again. You can build for yourself, without worrying about which framework, scaled growth, or a business plan. Have a problem? Build a solution. What fun, what joy!

The "compile time" parallel. Here's an apt comparison from back in the day: I would write some code, start the compile-build process, and walk away from my desk—usually to get coffee. During that time, I'd think about my code and plan what to build next. That downtime was my thinking-ahead time, and it kept me in flow. I found myself doing the

same thing now—prompting Codex to build something, walking away to think about the next feature, then returning to build it. I spent many hours at my desk this way, feature after feature. It brought back good memories.

What I'd teach. There is still some stuff you need to know. If I were designing a curriculum, I'd teach: how to prompt, how to test ("write evals," as they now say), how to check in code, and how to deploy. I'd cover web-architecture basics, OAuth and authentication, telemetry and logging, monitoring. I'd teach the basics of web design and typography. I'd teach product ideation and how to keep a product moving forward. And I'd spend a fair amount of time on problem-solving and debugging—because things will go wrong.

To summarize,

Vibe-coding is joyful, encourages a builder mindset, and keeps you in flow. Prompting and evals are a new type of abstraction, and this is in line with the broad secular trend in developer tooling. Nobody pines for the days when you had to understand assembly language to get anything done. Saying "people who vibe-code don't understand the internals" goes against history—not only in coding, but in every industry. People who drive cars know less about engines than ever, as do the people who build them.

Each time there's been a new abstraction—object-oriented programming, distributed systems—it changed the pedagogy of computer science. AI-coding will change it again, and it's best for us to embrace that and accelerate it.

The Valid Concerns

Can you go beyond prototypes? I've said you can build prototypes—but can you build, scale, and run full products? I think it's already happening. It feels possible, and the gaps are closing fast with newer models.

The JavaScript analogy. Vibe-coding isn't perfectly analogous to the jump from C to Java or Ruby, because one of the big benefits of moving from C to Java or Ruby was that outcomes were more predictable. Vibe-coding is more like someone who didn't know much about JavaScript starting to build websites—powerful, fragile, and potentially destructive in the wrong hands. The solution is better, more complete vibe-coding tools with better guardrails. As a simple example, I wish Codex would push back: *"Why don't we use GSAP instead of Three.js for these animations?"* or *"Why don't we separate this into APIs with an MVC architecture?"* Our AI tools today don't challenge our instructions, and that's a problem.

Is it a slot-machine or a gym? Being in flow while doing hard things is very different than being in flow because of being rewarded for doing easy things (slot machine). Are our AI-tools like being in flow while doing hard things, or are they like slot machines? The

onus to go deeper, understand more, and doing more difficult things now rests on the individual; there's a genuine concern that people will skip on understand and opt for just easy outcomes. The folks at fast.ai have a good essay on this.

A Few Closing Thoughts

What to build? When building is easy, the challenge shifts to choosing *what* to build. As a product manager, I loved spending time with customers and solving their problems. AI-coding tools push the focus toward a more customer-driven, user-insights-driven way of building. That's a win for everyone.

From prototype to product. The hard part for me in the last month was staying with projects—completing them, scaling them, thinking through GTM. Sticking with a project long enough to build something hard, defensible, and valuable requires a different kind of discipline than building the first version.

Monetization. A lot of the "SaaS is dead" discourse comes from the entrepreneur's perspective, not the builder's. How do you monetize when building is easy? What's your moat? These are valid business concerns. I don't have definitive answers, but I believe people who solve real problems for customers will always find a way to make money in B2B. I'm not overly worried.

Go-to-market. For consumer products where marketing is key, the problem shifts to GTM. Attention is scarce and getting scarcer. Creating network effects and building products people pay for is harder when every category is more crowded. The answer, again, is to go deeper into what users actually want. I also suspect we may see a shift from large, anonymous networks to smaller, close-knit communities of trust. That would be a welcome change.

What's Next

I learned a lot from this month. I felt joyful, creative, and excited as a builder. I stayed in flow longer and built many of the tools I'd wanted to build for myself, my family, and my friends.

I explored the tension between builder and entrepreneur. I thought through how to teach this emerging skill to people without technical backgrounds. I worried for a while that I wasn't going deep enough—and then I let go of that worry and found I was having fun.

In this month, I plan to whittle my project list down to a few ideas worth taking forward. For those, I'll go deeper into the details, understand the code more fully (mostly for my own satisfaction), and think through GTM, scale, and monetization. Stay tuned.

Vibe-coding democratizes building. It reduces the magic of casting spells and writing code. That can seem threatening to the magicians—but when everyone knows the spells, the world is a more magical place for all...