SAAS Vibecoding - Here are my thoughts & lessons

The Stack

Let’s jump into the stack I used:

[Lovable](https://lovable.dev/) is one of the more popular vibe coding tools. I chose it for its simplicity and because it uses Supabase for DB and backend, and I wanted to check out Supabase for quite some time now. I also did not want to lose touch with the data and code. While some other vibe coding tools do everything for you (i.e., base44), I wanted to have access to the data and backend code.

[Supabase](https://supabase.com/) is a managed Postgres service with a nice dashboard and a simple-to-use SDK. You can build authorization super easily, build APIs, and manage your DB. It is connected to Lovable, and if you want to have migrations in your code, you should do everything via the Lovable chat and not do anything manually in the Supabase admin.

[Heygen](https://www.heygen.com/) is a SaaS allowing you to integrate real-time avatars into your website and customize their knowledge. It is based on OpenAI, of course, but the avatar technology is amazing. I’ve used their API to create the interview sessions.

[Cursor](https://www.cursor.com/). I guess no need to introduce the VS Code fork that is changing the engineering world. I used Cursor on hard problems that Lovable could not implement, mainly by choosing a super model manually that I guess Lovable just doesn’t use on a regular basis. I mainly used Cursor for the integration of the Heygen SDK, which was not trivial.

[Cloudflare Pages](https://pages.cloudflare.com/). While Lovable has its own deployment, for some reason it did not work for me. I just bought a domain at GoDaddy and then deployed it to Cloudflare Pages. The integration from Lovable to Cloudflare via GitHub is super simple and makes life easy. I did have some challenges with the build issue, but I threw the errors into Cursor and it came up with a solution to fix the build.

[Lemon Squeezy](https://www.lemonsqueezy.com/). No Stripe in Israel, so I used Lemon Squeezy for payments. If it catches on, I will just use Stripe Atlas to open an entity in Delaware, but for now it’s not worth the trouble.

Why you still need engineers in the process?

This is a bit more technical, but if you build your own SAAS and thinking about vibecoding, you should consider the following:

Architecture. When I just started, I wrote a brief description of the project and wanted to see what I’d get. The overall UI and website flow were good, but once you want real functionality — you must be super detailed in your prompt and bring all your years of experience to life.

Security. As I mentioned above, Supabase uses a frontend SDK to connect the project to the DB. If you don’t use Row Level Security (RLS) correctly, you will end up with users able to change their credits or any other data. Unfortunately, the AI is not there yet. It made some very horrible security designs and I had to fix them. True to the challenge, I changed them with prompts, but you need to work in small tasks and fix issues one by one. I don’t see how it could be done in one long prompt or how someone non-technical could actually think about it if the AI is not doing it automatically. Too many of the core logics ended up in the frontend and not in the backend until I manually prompted them out.

DB Management. Surprisingly, this part went quite well, but I still had to manually prompt indexes and proper data types, as the AI had some weird choices.

Integrating Heygen SDK. This was the hardest part of the project. I tried using MCP via [gitmcp.io](http://gitmcp.io/) and putting the SDK docs URL into Cursor. It partially worked. I also ended up copying & pasting code from the SDK demo repository into the Cursor chat window and prompting it to “use this code to implement the SDK.” There were a lot of logic issues in the implementation causing some race conditions, slowness, and other bugs that I ended up reading the code and debugging manually (but still fixed everything using prompts). The integration took most of the time in the project.

React rendering and multiple calls to the server. The basic setup of the project in Lovable caused a lot of re-rendering of components, which in turn caused multiple requests to the server on every page load. I had to prompt my way out of it and only caught it via the Network or Console tabs in the browser’s DevTools. A regular person would not have noticed and I guess would end up with a slow project or a huge server bill.

Deploying the project, integrating SMTP, Google Auth, payment and other overheads also benefited from my technical experience. For some, you could prompt, but you still need to know what you’re looking for (DNS management, signing up for Google Auth or SMTP servers). Even if you provide clear documentation, I don’t see how non-technical users could actually do it.

Refactoring. Towards the end of the project, I decided to clean the code. I imported it to Cursor and used Gemini 2.5 Pro to go over all the code (big context window), removing uncalled functions, removing old components, and basically cleaning and refactoring. It worked pretty well with a couple of bugs (of course) that were fixed very quickly. I asked a very experienced friend to give feedback to the react code (overall, it is not a huge project) and he thinks it is good enough and meets the industry standards.