



The future of vibe-coding

Lately I've been spending a lot of time working in the vibe-coding space: not directly vibe-coding, but building the tooling and the resources to empower those who build using primarily AI.

This has brought me to think about one main thing: what is the future of vibe-coding? And, before that, is there even a future for vibe coding?

Introduction

There are two leading, opposite and polarized, positions when it comes to vibe coding:

- On one hand, someone says AI has extraordinary capacities, and that sooner or later it will replace software engineers. A company like Base44, which was reportedly the product of vibe-coding and exited with a million-dollars deal with Wix, is the perfect example for those who support this thesis.
- On the other hand, there are the skeptics, those who believe that AI will never truly be error-free, that it will be always flawed, limited, untrustworthy. Stories of security risks with Lovable, or Replit and Claude Code deleting prod databases, as well as Cursor getting rid of entire projects without backups, have been haunting Reddit and Twitter long enough for us to be well aware of the risks of vibe coding.

I wanna make it clear: I do not stand at neither of these polar opposites. I believe, as a matter of facts, that vibe coding has much more potential that many people reckon, but I am also firmly convinced that offloading the dev flow entirely and solely to AI (with the coder having no idea of what the code is doing) comes with high risks, not only in terms of security but also in terms of maintainability, CI/CD and future-proofing.

To me there is no question on whether vibe coding will have a future and survive - in a way or in the other, AI will continue to empower and shape this and the future

generations of builders: it's not a matter of *if*, it's a matter of *how*.

With this being said, let's dive right in the future perspectives of vibe coding.

Future forms of vibe-coding

Building Product Mock-Ups

Not much time ago someone from the product team at Google said that they were moving from a writing-first culture to a building-first one: with the advent of vibe-coding, indeed, it is now easier than ever to go from idea to minimum viable product in hours. Instead of an endless draft-review-rewrite process, you can now spend some hours designing, go to the dev team with a clear idea and a clear implementation track.

I think vibe-coding in this sense is extremely useful: no risk that you could interfere with prod code, no potential database cancellation, just feature-driven product development which can, in turn, help the devs ship faster.

Creating websites

Think about it: even if you run a bakery, a shoes shop or a book store, you need a website. Not necessarily to allow people to e-shop, place orders or take appointments on it, but at least to let the world know that you exist and what services do you offer. The problem is that, most of the times, a person that runs a non-IT business is unlikely to have advanced web development skills.

This is why products like Lovable and Bolt are perfect for these businesses: they can allow to build interactive, modern and dynamic websites without spending hundreds of dollars on professionals. Obviously, for more advanced capabilities (like booking or e-shopping) it would be still necessary to have some human supervision, but the time and the cost to get out in the web can be drastically reduced.

Drafting PoCs

Similarly to what we said above for product design, vibe coding can be of invaluable help for PoCs. Right now building proof of concepts often comes with downsides: sometimes you want to show features in the backend (like a new RAG

pipeline, an AI model or some innovative database logic), and you build minimal, not-good-looking frontend. Some other times, you want to show a new design, and you do not connect it with a backend. In both cases, your PoC loses value.

With vibe coding you can easily draft beautiful frontends and you can also build some backbone backend logic for your application, completing your PoC and adding some more value to it. Obviously, this might still need some effort from your side as a programmer, but the work will be lighter and faster, and also easier to iterate on.

These are the three main application I believe vibe-coding will be mostly used for on the long run, but I also think it is worth to take a quick look at the risks involved.

Future risks of vibe-coding

I think vibe-coding comes with three main risks categories:

- **Security:** security in the space of vibe-coding is a subject that is starting to emerge, and that many people have pointed out as one of the reasons that this programming practice won't survive. The point here is not only that LLMs often oversee security measures in the code they write, or apply them wrongly, but that also many of the people who use vibe-coding might not be familiar with security concepts and might take the AI's word (mostly hallucinated) for it, believing their applications to be secure when they are not. Also, there is another underexplored security issue that comes along with vibe-coding: LLMs can easily go off their guardrails, implementing malicious or scam-oriented application with the same confidence they would produce any other useful piece of software. In my idea of future vibe-coding, security shouldn't be a problem, but for those who are trying to build fully-functional applications right now, it could be something I would really look out for and ask for human help and expertise from security professionals.
- **Maintainability:** if you ever worked with an LLM, you know that oftentimes they offer extremely verbose and convoluted solutions. This is, most of the times, because they do not have deep knowledge of the documentation of every specific software, and they "take guesses" reproducing coding patterns that they learned during their training. These workarounds might actually end

up being functioning code, with some iterations, but most of the times this functioning code is made up by thousands of lines, huge monoliths, inefficient patterns, typing errors and verbose/irrelevant docstrings or comments. If the code then has to be passed on to a human engineer, it will be extremely hard for them to follow the logic, understand the workarounds and, in general, maintain code that they didn't write themselves, is badly documented and maybe doesn't even have tests. One solution to this would be trying to, as a vibe-coder, keep a "generation diary", documenting your journey, your intentions, the pieces of software built or refactored at every iteration - it doesn't have to be something you write yourself, you can also just copy-paste the explanations from the LLM, but at least you will be giving other people a guide on what was going on when the project was vibe-coded.

- **Future-proofing:** in a world where the software landscape changes with extreme rapidity, vibe-coding applications might very easily fall behind: due to the (necessary) cut-off date in the training data, LLMs might not have knowledge of the most cutting-edge technologies to develop software with, resulting oftentimes in products relying on deprecated dependencies, broken/discontinued packages or libraries with reported security or performance issues. Obviously you can mitigate this specific risk by providing up-to-date and detailed documentation about the libraries and services you want to build your application around (at LlamaIndex, we developed [vibe-llama](#) to serve this purpose), but it is also advisable that you yourself check for new solutions, vulnerability reports and other possible future-proofing threats to avoid that your application ceases to work in the near future.

These were my thoughts on the potential of vibe-coding for the future of software building: if you have takes to share about this, please feel free to do so on one of [my social platforms](#)!