Tasty Software

Why taste is the one thing the machines can't hand you

By Hamza Hafeez @hmza.cc

If you had asked a developer a few years ago what makes someone great at their job, you'd have heard the usual stuff. Knows data structures. Types fast. Understands the framework. Can debug at 2 a.m. without crying.

Nobody would have said "taste."

Taste sounded like a word for fashion people, coffee snobs, and that one friend who has strong opinions about fonts. It wasn't a thing you put on a CV. It definitely wasn't something you got hired for.

Well, that's about to flip completely. In this essay, I want to walk through how we got here, why so many people my age are scared, and why I think the answer to the fear is a word we used to ignore.

How software used to get made

A couple of years ago, engineers and teams had a very particular way of building and maintaining software. And nobody questioned it.

Companies looked at what the market wanted, built that, shipped it, read the reviews, and iterated. The process was basically treated as part of company's culture: this is how *we* do things here.

And code? Code was "good" if it ticked the boxes. Clean. Maintainable. Scalable. Type safe. If it passed the rules and the benchmarks, it got merged. Done. Next ticket.

There was also a clear hierarchy. Code review was the job of senior engineers, the people with deep experience in the domain who could stare at a diff and *feel* that something was off. Meanwhile juniors were allowed to do what juniors do: practice their skills, write code, make mistakes, and slowly turn into seniors.

It was a pipeline. Slow, but it worked. Writing code was the expensive part, so humans did it, and other humans checked it.

Then writing code became a token problem

AI coding agents showed up. Cursor, Claude Code, Devin, and a growing list of others.

Suddenly writing code was only a *token* problem. You feed in a prompt, tokens come out, and code appears. And those tokens cost less than a team of junior engineers.

Think about what that does to the old pipeline. The bottom rung of the ladder, the part where juniors learn by writing, just got cheaper to automate than to hire.

So it moved. Writing code was no longer the hard part. **Reviewing it was.** Now the senior's problem was a mountain of generated code that somebody still had to read.

But it didn't stop there. Multi-agent harnesses showed up, along with a ton of startup tools built around them, and code review landed in the hands of agents too. One agent writes, another agent reviews, a third agent runs the tests.

Therefore you got news that Lauren Tan ships 4,000 PRs a month, working through Slack (She’s a very famous engineer at SpaceXSI, on the team of Grok Bot). Whether or not you buy that number, the point is the same: one person plus agents can now move at a speed that used to need a whole department.

So the obvious question is...

What happened to the engineers?

When AI tried to code, and then humans started using AI to code, over time, a lot of people slowly forgot how to write code.. It’s natural. If you've ever stopped memorizing phone numbers because your phone does it, you know the feeling.

But many of them (apparently) could look at a piece of code, understand what was actually going on, and then tell their agents *amazing* prompts. They wrote agent.md and rules.md files, little instruction manuals that shaped how the agent behaves, and those files produced brilliant outputs.

Some went more further. They started building their own systems for rules, linting, evals, testing, and review. They were the ones like Lauren who ships unbelievable code to production in the world’s biggest organization.

Same tools. Same models. Completely different results.

Hold onto that, because it's the whole essay.

The fear era

But before we get to the answer, I want to be honest about what this period felt like, because it was rough.

"Software is dead."

The irony is that the rise of AI had pulled a lot of people *into* computer science in the first place. AI was exciting, so everyone wanted to build it or go with the wave. And then the hype turned around and said: why bother, the machine will do it.

That created a wave of frustration, anxiety, and a loss of motivation toward programming and writing software. Young students stopped being excited to learn how to code. And their paths got confusing too, because honestly, nobody knew what they should do. Teachers didn't know. Seniors didn't know. The internet definitely didn't know.

And for people who already had careers, it wasn't peaceful either. Massive layoffs hit, even at big tech. If *they* weren't safe, who was?

That's the part that hit young developers hardest.. the fear of losing a career before it even started. I'm a CS student. I felt this one personally.

So is software dead?

I think we were asking the wrong question.

You're the chef now, the plot twisted

Imagine this, every kitchen in the world gets the same equipment. Same stove, same knives, same ingredients delivered to the door. Everyone has access to everything.

Who wins?

The one who just cooks anything? Naah, not a chance.

It’s the chef with taste.

A great chef doesn't win cuz of skills and tools. They win cuz they know what the dish is *supposed* to taste like. They know when it needs to be presented. They know when it's too much. They can taste a sauce and say "something's missing" before they can even explain what. Give them all the tools and they use them *efficiently* because they have a sense of taste, and they know how to handle every spice on the shelf.

Now swap the kitchen for a codebase. The agents are the stove. The models are the ingredients. Cursor, Claude Code, whatever comes next, those are the brushes, and everyone has the same set.

In a world where everyone can generate code, it’s obvious that folks have to develop new tastes and their own recipes to stand out.

The same goes for languages and frameworks. It used to matter a lot whether you knew Rust or Go or the hot new library. In an era where the tools and languages are one prompt away, that matters less and more less. What matters is what you decide to build, what you decide to cut, and whether you can tell good from almost-good.

Okay, but what actually *is* taste?

Let me make it concrete. In software, taste is things like:

- **Taste is the “NO” thing that triggers.. Naah man, that’s not ready.**
- **Knowing how it should feel.** Does the app respond instantly? Does the error message sound like a human wrote it? Does the flow make sense without a tutorial?
- **Knowing how it should be built.** Is this abstraction earning its place, or is it complexity in a nice outfit?
- **Noticing what's off.** Looking at agent-generated code that technically works and thinking "nope, this will hurt us in six months."

The old rules (clean, scalable, type safe, passes the benchmark) were checklists. Checklists are exactly what machines are great at. A checklist can be automated, and that's the whole point. And.. the checklist stays exactly where it was.

Taste can't be a checklist. It's the thing you use when the checklist is already green and something still feels wrong.

There's a nice side effect too. The old culture said "this is how our team does it." Taste says "this is not how it *should* be done". I'm encoding mine into hmstack, open source, coming soon.. fr ^^

How do you build taste?

That’s the big one. If taste is the new superpower, can you train it? Obviously, not in a mystical way. A few things I believe in:

1. Consume great stuff on purpose. Use products that feel amazing and ask *why* they feel amazing. Read open source code from projects people respect. Look at how the best teams write their docs, their commit messages, their APIs.

2. Build things and ship them. Taste comes from the gap between what you imagined and what actually came out. You only learn that gap by finishing things, putting them in front of real people, and watching what confuses them.

I mostly emphasize on “markdown driven development”, but that’s another day’s read,

3. Review code, a lot. Even if agents write it. Especially if agents write it. Reading code is how you develop the eye that catches bad stuff.

4. Don't completely forget how to write code. This is my bias, but I think it's real. You don't need to hand-write everything, but if you can't write it, you can't judge it. The chefs who win knows what happens when you burn an onion.

5. Have opinions, then argue with them. Write down why you like something. Then try to prove yourself wrong. Taste isn't about being snobby, it's about being *specific*.

6. Build your own harness. Write your rules files. Build your own lints and evals. Every time you encode a preference into a system, you're teaching the machine *your* taste, and you're also forcing yourself to figure out what your taste actually is.

So, is software dead?

The act of *typing* code stopped. That's true, and pretending otherwise would be cope. If your whole value was translating tickets into syntax, that part is getting automated, and it's okay, better maybe.

But software itself? Read this closely.. SOFTWARE IS MORE ALIVE THAN EVER. There's just more of it, made faster, by more people. And when there's an ocean of software, tasty software is what stands out.

To my fellow students and young developers who've been staring at the "software is dead" headlines and feeling their motivation drain: I get it. But I'd gently flip the framing. You're being promoted to chef. The job is now to decide what's worth making, to notice when it's not right yet, and to care enough to fix it.

Everyone gets the same brushes now. The only thing left to compete on is what you paint.

So go build things. Look closely at stuff you love. Notice what's off in stuff you don't. And keep tasting.

The kitchen is open, and nobody has told you what to cook ^^.