

- [learnelm.dev](#)
- [Jesus!?](#)
- [About](#)
- [Recommended Books](#)
- [Podcast \(🇩🇰\)](#)
- [Blog](#)

[There Is Still No Silver Bullet](#)

An Elm Primer for React Developers is available now:

[Paperback](#) / [eBook](#).

August 13, 2026 11-minute read

[Programming](#) • [Software-Engineering](#) • [Theory](#) • [Llm](#) • [Ai](#) • [Craft](#) • [Brooks](#) • [Programming-as-Theory-Building](#)

44 people reading this right now

Of all the monsters that fill the nightmares of our folklore, none terrify more than werewolves, because they transform unexpectedly from the familiar into horrors. For these, one seeks bullets of silver that can magically lay them to rest.

That’s how the most famous essay in software engineering opens. Fred Brooks, [“No Silver Bullet: Essence and Accidents of Software Engineering”](#), presented at the IFIP conference in 1986. It turns forty this year, which feels like the right moment to point out that it has never once been wrong.

The essay’s famous claim is a prediction: “There is no single development, in either technology or management technique, which by itself promises even one order-of-magnitude improvement within a decade in productivity, in reliability, in simplicity.” People mostly remember the title (if even that) and skip the argument. Which is a shame, because the argument is the part that explains the last three years of our industry better than anything written in the last three years of our industry 🙄.

The essence and the accident [Link to heading](#)

Brooks splits the work of software into two piles.

The *essence* is the conceptual construct: the interlocking concepts, the data and their relationships, the decisions about what the system should mean and do and refuse to do. “I believe the hard part of building software to be the specification, design, and testing of this conceptual construct,” he writes, “not the labor of representing it and testing the fidelity of the representation.”

The *accident* is that labor of representing. Syntax. Build tooling. Boilerplate. The twenty-something nearly identical factories (gotta love Java, right?). The typing.

If you’ve read my [earlier posts in this series](#), that first pile might sound familiar. Peter Naur, one year before Brooks, called it the *theory* of the program: the shared mental model that the source code only partially represents. Two men within the same decade, arriving at the same place from opposite ends; Naur from watching a compiler team hand their program over to fresh programmers — source and documentation included — and seeing the newcomers’ patches fight the original design anyway, Brooks from watching projects miss every estimate he’d ever seen. Both concluded that the hard part of programming is building a model that can be communicated to others.

(Neither of them said code is easy, mind you, though half the internet is currently misquoting them on that.)

And from that split, Brooks derives his prediction with arithmetic “simple enough to do on a napkin”: a tool can only compress the accident. So unless the accident is nine tenths of your job, no tool, however magical, can make you ten times faster. The essence sets the pace.

He even reviewed AI In 1986 [Link to heading](#)

Brooks didn’t just make an abstract prediction and leave the werewolf-hunting to us. He went through the silver bullets of his day one by one: Ada, object-oriented programming, program verification, graphical programming — and(!) yes, artificial intelligence and expert systems, each with its own section. His verdict was the same each time: useful, some genuinely so, but all of it aimed at the accident.

The AI of 1986 was not the AI of 2026, obviously. But his reasoning never depended on capability in the first place. What matters is *which pile* the help lands in. “The hardest single part of building a software system is deciding precisely what to build,” he writes. No tool that receives a specification can help with the fact that producing the specification is the actual work.

Forty years of silver bullets later (CASE tools, 4GLs, UML, no-code, low-code), the essay is undefeated. Now it’s up against the strongest candidate yet, at least if we measure by hype. LLMs are probably, and I mean this sincerely, the most effective accident-compressor ever built. They write the boilerplate and scaffold the tests, and they’ll produce the twentieth nearly identical factory without a word of complaint. [I use them for exactly this](#), in a box, with supervision.

The mirror [Link to heading](#)

If Brooks is right that tools compress only the accident, then every AI productivity claim is accidentally a confession:

Your AI speedup is a measurement of how much of your job was accident.

If Claude Code really did make you ten times faster, then, by Brooks’ napkin math, nine tenths of what you were doing was the labor of *representation*, not the conceptual work (or in my own terms: the engineering and craft). My intention is not to insult, btw (we’ll get to the actual insult shortly). Accident is real work, somebody has to do it, and our industry has spent two decades manufacturing mountains of it (I say this as someone who has configured webpack, and enabled ephemeral storage on Azure VMs 😊). For some roles the ratio really is that lopsided, and for those roles the compression is a mercy.

Turn the mirror around, though: the senior engineers reporting modest gains aren’t slow adopters or Luddites in denial. They’re telling you, with a number, what their week is made of: domain conversations, design decisions, naming, review, the slow work of figuring out what the system should refuse to represent. All essence, and none of it gets faster just because the typing does.

A small example from my own desk. When I built the little Elm testimonials widget for this site, [I let an AI turn a pile of messy DOM nodes into clean JSON](#) — pure accident, and it spared me a genuinely boring half hour. Then I wrote the widget itself by hand, and it took roughly what it would have taken me in 2019. Not because I refused the help on principle (though, to be honest I would have — I did it for *fun* after all!), but because the hours went into deciding which states should be impossible and what the messages should mean. The all-to-the-agent crowd tends to miss where those decisions got made: half of them happened *while* w

“Code was never the hard part” [Link to heading](#)

Which brings me to a phrase making the rounds, one that sounds like it agrees with everything above: “code was never the hard part.” You’ve heard it before, I’m quite sure. The real work was always the requirements and the stakeholder meetings, so let the machine handle the trivial coding bit and free the humans for Higher Things.

Senko Rašić recently wrote [a response to that phrase](#) calling it an insult to all programmers, and he’s right. If code was never the hard part, why did this industry spend two decades hunting “10x ninja rockstar” coders and grilling candidates on whiteboards? Why do [Clean Code](#) and [SICP](#) exist, and why did generations of us wear them out? You don’t build a fifty-year literature of craft around the easy part of a job. The people saying this now, mostly to sell you something, are rewriting the history of a profession.

But that is not what Brooks said, however often he gets conscripted into saying it. The line between essence and accident does not run between coding and everything else, with meetings on the important side and keyboards on the trivial side. It runs *through the middle of coding itself*. When you write real code — choose a name or reject a type, feel a design resist you and change course because of it; you are doing essence work with a compiler watching. The accident is the residue inside that activity: the transcription, the ceremony/boilerplate, factory number twenty. And Naur’s whole point, the one [this series pounds repeatedly on](#), is that the theory doesn’t get built in meetings and then merely “implemented”. It largely gets built at the keyboard, [in the old gym](#). The typing was never the point, and further: the writing was never just typing.

So the sales pitch (“let AI write the code so you can focus on what matters”) lands exactly backwards for the people it’s aimed at; it is, quite precicely, wrong. For a programmer worth the title, writing the code is where a good share of the what-matters gets done. (And again, don’t underestimate this part either: It’s also fun.)



The essence gap [Link to heading](#)

In 2025, METR ran [an actual randomized controlled trial](#): sixteen experienced open-source developers, working in their own mature codebases, tasks randomly assigned with and without AI assistance. With AI they were 19% slower. Their own estimate, afterwards, was that the AI had sped them up by about 20%.¹

An ish forty-percentage-point gap between felt speed and measured speed! And Brooks would have called both the sign of that gap and the reason for it. The acceleration is real and it is *felt* — the boilerplate appears instantly, the diff grows, the tool is visibly doing things. But these were experts in codebases they knew deeply, meaning their work was almost all essence: deciding, judging, verifying against a theory of the system they carried in their heads. Compress the accident of a job that’s mostly essence and you get exactly what the study found: developers who felt faster and measured slower.

The 19% has a second explanation too, and it’s the one from the section above: for engineers like that, writing the code is how the theory stays current. Hand the writing to a tool and what’s left is auditing someone else’s guesses against a mental model that has quietly stopped being fed. That’s a different job, a worse one, and the stopwatch noticed. Meanwhile the dashboards go up and to the right while the outcomes stand still. (The churn and duplication data pointing the same direction is a story I’ve [already told](#), so I’ll spare you the recap.)

I can hear the objection forming: “but productivity has improved by an order of magnitude since 1986 — compilers, garbage collection, open source, the cloud!” True, and Brooks never said otherwise. His bet was about a *single* technology, within a *decade*, and it has held every time someone claimed to have shot werewolf. The cumulative gains come from decades of chipping away at accident, which is his model.



Grow designers, not prompters [Link to heading](#)

People forget that “No Silver Bullet” doesn’t end in despair. After dismantling every bullet on the market, Brooks tells you where he thinks the real leverage is, and it isn’t a technology at all. “The central question in how to improve the software art centers, as it always has, on people.” His concrete proposal is to find and deliberately grow great designers (the people who can do the essence work) with the same seriousness companies apply to growing managers.

Forty years later, that’s still the whole game, and I’d argue the stakes have gone up. The essence work is what this whole series has been about: [the theory](#), [the knowledge that can’t be written down](#), and [the institutions where both get passed on](#). None of that compresses, and all of it is currently being starved in the name of a speedup that Brooks’ arithmetic says cannot arrive.

And no, this isn’t a sorting of people into two camps. Everyone’s week contains both piles. A junior converting DOM sludge on Monday might be doing the team’s deepest domain modeling on Tuesday. The question is which pile your calendar — and your employer’s incentives — actually protects.

The habits that follow from taking Brooks seriously are not complicated:

- Sort your own week honestly. Look back at last week: how many hours went to deciding what the system should mean, and how many to expressing decisions already made? That second number is your ceiling on any AI speedup, and it’s worth knowing before your manager reads a vendor benchmark.
- Point the AI at the accident, on purpose, with constraints. Set the theory yourself and let the tool do the typing inside it — [I’ve written about what that looks like](#), and it works.
- Protect the essence hours from the velocity dashboard. Design conversations and domain modeling will always look slow in Jira, because Jira only knows how to count the accident.
- If you lead people: grow designers. Brooks said it in 1986 and the transmission mechanism hasn’t changed — [apprenticeship](#), [code review](#), [working at the shoulder of someone who knows](#). No prompt produces a designer.

Brooks ended his essay with *people*, so I’ll end mine with him. The industry is currently spending hundreds of billions on the shiniest bullet ever cast, aimed — as every bullet before it — at the part of the monster that was never the problem. When the “tenfold gains” keep failing to appear on the balance sheets, and the napkin says they will keep failing, companies will go looking for the people who can move the essence: the ones who kept building theories while everyone else was compressing accident.

Code remains what it has always been: a craft, not a solved problem. (And killing werewolves is fun.)

1. Yes, I know about [the follow-up](#) METR published in February 2026, where the sign flips to a (statistically inconclusive) speedup. Perhaps contrary to its intention, it mostly confirms my suspicion: METR spends a whole section explaining that wider adoption has made the thing nearly impossible to measure — developers now refuse to join studies where they can’t use AI, and 30–50% admitted to holding back exactly the tasks they figured AI would win. The well is poisoned, and the people poisoning it are the ones being measured. Even at face value the new estimates land around “modest” anyway. IMHO the napkin math still stands. 

Search?

Email

Subscribe

© 2021 - 2026 Christian Ekrem

805
Shares





Search?

805
Shares

