

sean goedecke June 22, 2025 | [software design](#), [tech companies](#), [large codebases](#)

Pure and impure software engineering

Why do solo game developers tend to get into fights with big tech engineers? Why do high-profile external hires to large companies often fizzle out? Why is AI-assisted development amazing for some engineers and completely useless for others?

I think it's because some engineers are doing very different kinds of work to other engineers. Those two types of engineers often assume their counterparts are simply incompetent, but they're really just working in different fields.

Pure and impure engineering

There are two very different kinds of programming work. The first kind - pure engineering - is interested in solving a technical problem as perfectly as possible. Open-source work is often like this: some engineer wants to write the best HTTP requests library, or their ideal game engine. The second kind - impure engineering - is interested in solving a real-world problem as efficiently as possible. Paid tech company work is often like this: engineers are asked to deliver some project or feature as well as they can do it by the deadline.

In pure software engineering, what you're doing is close to art or research. It's close to art because the engineer is driven by an aesthetic sense (e.g. of what makes a good library or game engine). It's close to research because it's open-ended: once the engineer arrives at a solution, they can continue testing and tinkering forever, trying (and usually discarding) new approaches.

Impure software engineering is more like plumbing or construction. The engineer's aesthetic sense is subordinated to someone else's (usually their employer's) needs. They're building a solution to someone else's problem. And

since it's someone else's problem, it has to actually be finished to schedule, which means compromising.

If you've spent any time reading my posts, you'll know what kind of work I spend most of my time on: the impure kind of work. I am pragmatic to a fault. But I do have a lot of respect for pure work. I come from a background in academic philosophy, which (along with mathematics) is pretty similar to pure engineering.

Pure engineering is less important in the 2020s

This pure/impure distinction has been obscured by the fact that **there used to be much more scope for pure engineering at large tech companies**. In the 2010s, [times were different](#). Companies were driven almost entirely by hype, and they were hiring more engineers than they knew what to do with. Funding pure engineering projects solved both of those problems: it produced impressive open-source artifacts that made the company look good to prospective engineering hires, and it provided a bottomless pit of useful-looking work for those engineers to do.

Even impure engineering got colonized by pure engineering. Companies burned hundreds of thousands of engineer-hours migrating from monoliths to microservices, or from HTTP service calls to event-sourced architecture, or from event-sourced architecture to full CQRS, and so on. A lot of very skilled engineers found their niche in navigating these hard, technical projects.

But like I said, those times are gone. Tech companies now have to make money. Hiring has slowed down dramatically, and companies are tightening their belts. A lot of pure engineers have had a rough time navigating this transition. From their perspective, work has all of a sudden become much more political. But what's really happened is that their previous role - which was effectively a covert developer marketing position - isn't being funded in the current market.

What tech companies need

Tech companies need both kinds of work, but not in equal quantities. They rely on pure engineering to build components that solve tightly-scoped technical

problems. For reasons that will become clear, tech companies prefer to get these components from open-source (tools like Kafka, Redis, or even the programming languages themselves). But large tech companies always build some things internally, because their requirements are so specific. For example, GitHub has its own highly-performant HTML parsing code (instead of using something like Nokgiri) because it relies so much on rendering Markdown everywhere.

I don't think I have to explain why tech companies need impure engineering. Almost everything a tech company does is predicated on the need to ship some new feature or capability as soon as possible. On top of that, almost all tech company decisions are a compromise between tens or hundreds of people. The kind of engineer who can fulfil those needs is the kind of engineer who's happy to do impure work.

Of course, any engineer can do both pure and impure work. But in my experience, engineers who are mainly interested in pure work do impure work badly: they struggle to compromise, they panic under deadlines, they aren't as good as holding unwieldy codebases in their head, and so on. [As I keep saying](#), it takes a lot of skill to ship in a large tech company. Likewise, engineers who are mainly interested in impure work (like me!) struggle with pure work: they're too eager to settle for a working hacky solution, and they often don't have the technical expertise to see the right path forward.

Why impure engineering is valuable

There are probably people reading this right now and thinking that this is just a distinction between competent and incompetent engineers. Since pure work is more fundamental and technically harder, isn't the pure/impure distinction just a distinction between engineers who are smart enough to build fundamental components and engineers who aren't? Isn't it like the distinction between the electrical engineers who design microchips and the hobbyists who plug them together out of a kit to build their PCs?

No, not really. This view is kind of like saying that engineers are just people who weren't smart enough to do physics, or physicists aren't smart enough to do pure mathematics. They're different fields that require different skills.

Clashes between pure and impure engineers

I'm going to reference some four-year-old drama here. In June 2021, Casey Muratori [got into a fight](#) with the Windows Terminal developers over a point of [performance](#). I think it's clear that Casey was right about the technical matter - as a competent game engine programmer, he understood the performance characteristics of the problem well enough to point out that Windows Terminal was doing it inefficiently.

To their credit, the Windows Terminal team went and implemented the feature anyway, after taking a bit of time to cool down. It became available in settings in [Feb 2022](#). I think that's not actually an unreasonable timeline for a team that had its own work to do and had to actually implement the feature (the GitHub comments have some fun details about supporting multi-glyph code points).

Does this mean that Casey is a better engineer than the entire Windows Terminal team? I don't think so. It means that if you're a pure engineer with a particular area of expertise, you can almost always outperform impure engineers who are working inside a tech company to build a product.

Many pure engineers underrate just how hard it is to do impure engineering well. When you're doing pure engineering, it's just you against the problem. By comparison, impure engineering is a brawl: you're fighting decades of previous technical decisions, competing political views about how the product ought to work, consensus among your colleagues or the company at large, and in general *much more incidental complexity* driven by the accumulation of [wicked features](#) that provide enormous business value at the cost of adding drag to every single piece of feature work. There's a reason that impure engineering is so highly paid.

As another case study, George Hotz is another very competent pure engineer who famously joined Twitter in an attempt to "fix search" and [failed](#). As it turns out, doing impure engineering at large tech companies is really hard! Now he's working on [tinygrad](#), which is a perfect example of a pure engineering project: a maximally-simple, highly-performant deep learning framework.

I don't think this section would be complete without mentioning Jonathan Blow

(another game dev who has been working on his own programming language [Jai](#) for over ten years). There are [so many](#) examples of Jonathan saying that slow performance of software is due to engineers who don't know how to do their jobs and ought to be fired. Like Casey Muratori, he can see the silly mistakes being made.

Impure engineering is sometimes worth trading away performance

I'm sure it's frustrating to use slow software and know that you could have built it faster. I've been in that position myself. But tech companies are (mostly) rational economic actors who (mostly) do things because it's profitable to do so. They're not exclusively hiring elite performance engineers because those engineers *do not produce the most business value for the company*¹. Tech companies would like faster software, all things being equal. But they're willing to trade off performance against a number of other things.

As a user, I agree with the tech companies! I use Visual Studio Code to do almost all my engineering work. I used to use Neovim and Alacritty, which was considerably snappier. But I switched away from that because VS Code has better support for some features I use a lot (e.g. GitHub Codespaces). As it turns out, I am also willing to trade away performance to get other things I value.

I want to be really clear here that **I know some of these performance problems are not *technical* tradeoffs**. With the right engineers, you could make almost all big-tech-company software much faster in the same amount of development time. But I just don't think the kind of engineers who can do this performance work will be able to generally function as well as the kind of engineers big tech companies already hire². As with [fraud](#), the optimal rate of performance blunders is non-zero.

I'm not saying that working on tinygrad or game engines is easier than doing plumbing work at large companies. I'm saying they're both really hard, and that engineers who do pure work often underrate just how hard it is to do the impure work well.

AI is most helpful for impure engineering

This distinction between pure and impure engineering also drives very different attitudes towards the role of LLMs in software development. Pure engineers - like the Twitter game devs in the previous section - are typically [dismissive](#), saying that LLMs produce trash code and just aren't useful for real work. On the other hand, LLMs are a big part of my own developer workflow (maybe a 30% speedup, on par with type systems or debuggers). What's going on here?

Think about what pure engineering is like. You're working on a problem you understand very well (because you've been working on it for a long time) that is not well-understood by the developer community at large (or it wouldn't be interesting enough for you to work on). You're constantly operating at the limits of your technical expertise. And you're able to spend as much time as you want to make the right decision. It's understandable that LLMs wouldn't add much to that. For every decision you make, you're going to be much smarter than the LLM³.

Impure engineering is different. You're typically working on a problem you only have a loose working understanding of (because the company needs it in order to deliver some project). That problem is usually not novel, it's just novel to you. It's rare that you get to work on a problem that you have a thorough technical understanding of, and you're usually working to a tight deadline. For some of the decisions you make, the LLM will thus be as smart or smarter than you, and you can get a lot of value from asking it for advice or review.

With this in mind, I can see why pure engineers are baffled by the hype around AI. It must seem so strange: every time they try to use LLMs, they're completely unhelpful. But I think this reflects a certain narrowness of vision.

Final thoughts

I think it's commonly (and correctly) understood that pure engineering is difficult, valuable work. Almost everything we do as engineers depends on it: from the programming languages and libraries we use to the open-source services and databases we build our systems from.

However, I want to defend impure engineering as also difficult and valuable. Large tech companies already know this and hire based on it, so it's already very lucrative to develop your impure engineering skills. But when engineers talk about engineering, they sometimes pretend that pure engineering is the only kind of engineering there is.

edit: this got some comments on [Hacker News](#). Many commenters enjoyed the post but wish I'd written instead about some other distinction that they find more compelling. I want to clarify that I do not think that [pure engineering is going away](#). There will always be a lot of pure engineering work that needs doing.

1. Do Leetcode-based hiring practices serve as a counter-example to this? Leetcode problems are definitely pure engineering. I don't think so. For one, Leetcode hiring dates back before the bottom fell out of the industry. For two, I think it's mostly a convenience thing - if there was a Leetcode-style test that could measure impure engineering skills, tech companies would be all over it. (Also, big tech companies do need some amount of pure engineering, even today, and they do actively try and hire skilled pure engineers for that work.)



2. Of course there are exceptions. Just as some people are both Olympic athletes and mathematical geniuses, some engineers are amazing at both pure and impure engineering. These people can usually write their own ticket.



3. About a month after I wrote this, the now-famous [METR study](#) was released, which I take to experimentally demonstrate the exact point I'm making here: engineers who are very familiar with their pure codebases do not get much speed-up from AI tooling.



If you liked this post, consider [subscribing](#) to email updates about my new posts, or [sharing it on Hacker News](#).

Here's a preview of a related post that shares tags with this one.

Wicked features

Why is working at large tech companies so hard?

It's because a small subset of “wicked features” dominate everything else. If you're building a todo app, adding the ability to attach images to todo items might be a large feature, but it's not a wicked feature. However, offering your todo app as a webapp and a standalone executable is a wicked feature. What's the difference? Wicked features are features that must be considered *every time you build any other feature*.

[Continue reading...](#)

[subscribe](#) | [about](#) | [podcasts](#) | [popular](#) | [tags](#) | [rss](#)

Search