

# AI Coding will Prevent Expertise

The need for ongoing friction in long-term skill formation.



*"We see a future where intelligence is a utility like electricity or water and people buy it from us on a meter and use it for whatever they want to use it for" - Sam Altman of OpenAI*

In my previous article, [Agentic Coding is a Trap](#), I discussed the "skilled orchestrator paradox", where the skills required to manage AI agents for coding are the same ones that can be diminished through the continued use of said AI agents. Expertise was largely the differentiator; the more experienced a developer is, the *less likely* it is that they might experience skill atrophy, as the knowledge has had a chance to ossify after years of experience.

If you look around right now, you'll find the *vast majority* of those that are seeing the *most* benefits from these models are those that have had years, if not decades, of experience in the field (which predates AI tooling, of course). And any industry veteran will tell you the same: the bedrock of this knowledge comes from doing the work.

Developers who've entered the field around the time of LLMs are placed in a position where they don't have the benefit of longevity, but they are being guided (and sometimes mandated) to accelerate their efforts using coding assistants that require a history of expertise to wield effectively and *responsibly*.

It's an awkward place to be for that demographic, as it creates a scenario where a *novice* needs *expert-level* skills to leverage the tools and keep pace in the industry.

# The "Expert Novice"

We're currently sending very mixed signals to people across the industry. We're hammering in that if you're not using AI tools, you will be "left behind" by your peers who *are* using them. "AI won't replace you, someone using AI will" has been on repeat since 2023.

And in the same breath, it's also said that the way to get the best results from these models is to *apply higher-order thinking*; "vibe coding" is a dead end; you need to "move up the stack" and create robust specs, architect with good design patterns, and *always* review the outputs diligently so you *never* ship something you don't understand.

The skills to do so, however, are a function of someone who has experienced the friction and challenges over time that culminate in "good taste".

This leads to another situational paradox: If these tools *demand expertise*, yet the tools can *actively circumvent* the friction that *cultivates expertise*, then what is the path for one to *become an expert* so they can effectively use these tools?



## Confidence without Comprehension

One hope is that these models will end up *accelerating learning* as they are used for code generation. Junior developers can work with the same gravitas and confidence as industry veterans with their "personal AI tutor". Knowing syntax is increasingly less important, and any knowledge or ambiguity gaps are filled by the AI tool. The deeper mechanics of the code stay abstracted away, since the developer sits higher in the stack.

JetBrains, a major player in developer tools, recently cited a study titled "The Widening Gap: The Benefits and Harms of Generative AI for Novice Programmers", which painstakingly analyzed individual behaviors in live coding sessions, and tested their ability to learn coding with varying degrees of AI assistance. Their main takeaway was stark and counterintuitive:

*"Participants thought it was like having a personal tutor. From the data in our study ... we observed that they did not, in fact, use GenAI tools like a personal tutor. In fact, it was quite the opposite."*

The participants that leaned into heavier AI assistance:

- *"Often skipped crucial planning stages, finding that because they hadn't reasoned themselves into this position, Copilot had."*
- *"Finished with an 'illusion of competence' rather than true understanding."*

Counter to that, the participants that mitigated their usage of AI:

- *"Succeeded because they had developed 'negative expertise'—which is 'the ability to ignore incorrect or unhelpful GenAI suggestions'—allowing them to focus on writing their own solutions rather than being led astray."*
- *"Were able to use GenAI to accelerate, creating code they already intended to make."*

The novice developers who were the most unrestricted and confident in their AI usage *"had skipped crucial steps in the programming problem-solving process, and were now lost."*

Perhaps unsurprisingly, the novice developers who performed the best were the ones that *greatly mitigated* or *outright ignored* the AI coding assistance.



# Inverted Learning

Due to the self-directed nature of LLMs, *the more experience you have, the more benefit they provide* since you can accurately steer, audit, and verify the outputs. The less knowledge you have, the more they can *mislead you*. Interacting with LLMs for learning new skills takes the shape of an "inverted learning" model, a role reversal where the *student* is initially guiding the *mentor*, the mentor responds, and then the student, again, steers the mentor.

The process is precarious; LLMs are incredibly sensitive to the shape of the prompt. When you're exploring new domains, *you don't know what you don't know*, and the malleable and accommodating design of an LLM can lead you to believe you know more than you actually do.

If you're exploring territory that is even somewhat unfamiliar, you often don't even know the questions that you need to ask that could properly guide the model to providing the best answers. It begins to feel like a compass that always points north, wherever you *suggest* north *might* be.

From the same study that JetBrains highlights, even the most prepared students were derailed by the AI assistance due to this type of learning model: One participant demonstrated good fundamental planning and habits, but suddenly *"skipped crucial problem-solving planning stages, jumping directly to coding and was enticed by Copilot into quickly producing code"* and had to rely on the LLM to fix the error that the LLM introduced in the first place.

AI models lack judgment, empathy, and pedagogical intent, and the solutions provided are not rooted in *experience* but rather in patterns in the training data (*LLMs are, at their core, incredibly complex pattern interpolators*).

The infinite answer machine is tempting, and known to be addictive. It can unwind rather quickly, especially for inexperienced developers. Once you get deep enough into a generated solution, you are often beholden to the AI tool to also finish the job, circumventing the problem-solving friction that is required for the formation of a mental model (*and to be fair, senior developers are prone to this phenomenon, as well*).





# The Friction is a Feature

Expertise and mastery don't happen purely through observation and dialogue, but through experience, repetition, and trial and error; you have to fail to succeed. If I wanted to learn how to cook, I could watch a Master Chef work and make endless inquiries. After a month, I would be able to describe the perfectly medium-rare ribeye but never know what it's like to cook one, and I'd almost certainly overcook it on my first attempt.

Coding has endless moments of tracing obscure errors with no log file to help, experiencing the subtle performance differences of certain methods, or having to rewrite an approach when it's clear it won't going to scale.

This applied friction is directly what builds "developer intuition" (or "taste"). The Germans have a great word for this: Fingerspitzengefühl (fingertip feeling). It's the muscle memory that triggers when a developer looks at something and thinks, *"yeah...this is probably going to cause problems."* By avoiding the mechanics of the struggle, this intuition is never built.

In UPenn's large-scale 2025 study Generative AI without guardrails can harm learning, they followed 1,000 students using an LLM to learn mathematics and found students used AI as a crutch and ended up performing *17% worse than students with just a textbook* (and just as with the previous study, the students using the AI assistance *thought they were excelling*).

LLMs don't just have to generate code, though.

If leveraged as Socratic sparring partners instead of answer generators, studies have shown that "dialogic AI systems can meaningfully stimulate reflective, critical and independent thinking".

In that same UPenn study, they also tested a "Tutor" version by having students ask for help and then *independently solve the problem*. The GPT Tutor group performed an astonishing *127% better* in the AI-assisted practice session (although, interestingly, they scored about the same on the test as the textbook group).

This is effective because the model is *no longer being utilized as a means of production*, and it shifts the cognitive work back onto the individual. It's when the friction is still present that it creates a lasting imprint that leads to expertise.

Anthropic's 2026 study "*How AI assistance impacts the formation of coding skills*" came to similar conclusions:

*For novice workers in software engineering or any other industry, our study can be viewed as a small piece of evidence toward the value of intentional skill development with AI tools. Cognitive effort—and even getting painfully stuck—is likely important for fostering mastery.*

There's a certain sense of irony here: the most productive learning that can happen with an AI coding tool is when it isn't used to generate much of any code at all.



## Pipeline Collapse

If LLMs can write code *and debug* code, and agentic workflows can perform system design from the abundance of patterns in the training data, then what is the purpose of this knowledge in the first place? Programming will be done entirely in natural language, and we can dispense with the need to engage with the code because the models continue to improve and fill in any knowledge or ambiguity gaps. They will debug any issues that arise and manage any complexity that they introduce.

The trillion-dollar bet that is being made is: this knowledge won't matter, because LLMs will take up the slack and effectively become the new generation of "developers". It starts give off an aire of hubris that drove past no-code movements, and the fever dreams of CEOs, rather than the reality on the ground.

Coding/programming/software is a unique intersection of logic, math, problem-solving, critical thinking, planning, communication, and creativity. LLMs can detect patterns at a scale that no human ever could, but patterns only get you so far.

David Cramer, co-founder at Sentry (a performance and error tracking platform), put it succinctly in a recent interview:

*I think there's a type of person ... that inherently believes that LLM will get better enough that they will go back and fix this stuff, that it will be able to clean up all the junk that's been stacked up along the way. I don't think that's true. I think it's a science experiment.*

*You want to flex that you can generate all of your code and have hundreds of things going in parallel, I will flex and show you how broken the code is 100% of the time.*

## Will the pipeline collapse, or just change?

It really depends on whether we make the needed shift to a more pedagogical usage of these systems.

By continuing to focus on and promote AI coding workflows that prioritize code generation above deep understanding, *we are not cultivating the next generation of expertise* who will inherit the code that is being created today.



r/ExperiencedDevs • 7mo ago  
MoltenMirrors

...

### AI is a death trap for many junior devs. How do I mentor them out of it?

I'm noticing a pattern with many recent grads (yes, my company still hires them). Either they're excellent engineers who barely need any input from me, or they churn out broken AI slop that they don't understand well enough to even test.

In the latter case, I don't think they're lazy, necessarily (although some are). It's that they've forgotten how to learn new things. When AI is generating code for them they're not gaining experience with the capabilities of a framework nor how to architect something properly, so when the next feature comes along they don't even know how to properly craft the prompt. Then, when there are inevitably bugs, they rely on the AI to find them because they don't even know where to look or what to look for.

I use Claude and Gemini a lot, but there are only three use cases I've found where they actually save me time: looking up how to do something in an API or navigating an unfamiliar codebase, writing one-off scripts that pull data from multiple sources to do something useful, and generating unit tests when there are clear existing examples to replicate. Everything else, I end up churning too much on the prompt and it's faster to just write code myself.

There are a few tips I pass on to my juniors (always always have the AI tell you its plan before generating code; give it examples from our codebase to replicate so it follows our conventions), but I don't know how to help them gain the knowledge and experience they need to truly be effective.

Anyone have pointers to good resources for how to use AI to build your skills and become a better developer, not merely a faster one?



## My Approach: Friction First

Joel Spolsky presciently writes (*in 2002, no less*) in his *Law of Leaky Abstractions*:

*Code generation tools which pretend to abstract out something, like all abstractions, leak. And the only way to deal with the leaks competently is to learn about how the abstractions work ... the abstractions save us time working, but they don't save us time learning.*

If a developer wants to learn Java, they should probably not start with Spring Boot. If they want to learn JavaScript fundamentals, they should not start with React. If they want to become highly adept at CSS, they should not start with Tailwind. LLMs could be considered the ultimate leaky abstraction.

My advice here is very similar to my previous prescription.

If a developer wants to become an expert in programming, they should largely disregard the pure code generation capabilities of these models, and instead use them for *interactive documentation*, *dynamic tutorial generators*, and *Socratic exercises*.

It's not a panacea, of course: Using an AI tool as a tutor carries its own risks since it is susceptible to the same hallucinations as any other interactions, and it cannot be relied upon solely as a learning source. If you can't properly audit the accuracy of the generated code, then you can't audit the accuracy of the generated concept. If you use AI as a mentor, you must still verify its outputs against *official documentation*, *human peers*, and *actual trial and error*.

*"Coding's actually a great way to cement understanding. The more you program, the more you understand the domain that you're working in."*

— *Kent Beck, creator of Test-Driven Development*

Choosing this slower, more deliberate path is the best way to grow expertise, but I'm aware of how hard that is when the surrounding ecosystem is actively working against it. AI is being mandated (often recklessly) across companies, and baked into most software development tools and IDEs as they cater largely to senior engineers (*even with some tools like Cursor tucking away the code view unless the user specifically seeks it out*). Some companies are even forcing developers to only use AI for *all* coding tasks, regardless of experience level, and these companies will have to learn their own lessons.

However, for everyone else who is looking to strike a balance between deep learning (no pun) and productivity, there are qualifying questions you can ask to ensure your usage of these tools yields long-term benefits.

### My AI-assistance checklist:

- If I did not have access to an AI tool, *could I still accomplish this task?*
- Am I using the model to deepen my understanding, *or expedite the answer?*
- If I had to audit and verify the generated output, *could I adequately explain what was happening?*
- If I'm learning a new concept, *have I done proper research to know the right questions to ask?*
- Have I cross-referenced and verified the approach through other methods (*reading documentation, standard search tools, StackOverflow, Reddit*)?
- Is this a truly rote task that's been done 100 times before, or *a task that requires executive decision-making* somewhere in the process?

Even as a developer with decades of experience under my belt, I am still constantly referring to them throughout my daily work, especially when I am attempting to learn something new (which in this field, is neverending).

The key is to detect the difference between cognitive debt and cognitive offloading: Cognitive debt is *abdicating your judgment and decisions*, whereas cognitive offloading is *delegating the mechanical or tedious*.

As the Anthropic study mentioned, getting "painfully stuck" is a good thing. It takes discipline and effort to not drift back towards just generating answers, which might not even be accurate in the first place. LLMs

didn't suddenly rewrite the fundamentals of *how* we learn, but they did give us a new way to do so.

# Intelligence isn't a Commodity

The realignment I hope to see over the years is the understanding that skills don't develop without active participation. You must engage *directly* and *continuously* to experience the essential friction that culminates in expertise (even if it means moving more slowly).

If we stay fixated on lines of code and tokens burned while the expertise pipeline dries up over the years, Sam Altman's vision of selling intelligence back to us on a meter could become reality. Domain knowledge could become very hard to come by, and when one sits down to do any type of development work, there will be a pang of paralysis if that person does not have an active AI tool subscription at their side.

*LLMs are a static database of skills. They are interpolation engines. Software engineering, however, is an exercise in adaptation and novel problem-solving. You cannot interpolate your way through a completely unique system failure.*

— François Chollet, creator of ARC-AGI Benchmark

Subscribe for future articles:



Subscribe →





Learn more about me → 



© 2026 | All Rights Reserved