AI Didn't Remove the Hard Part of Coding. It Moved It.

For decades, the bottleneck in software was producing the answer. That bottleneck is gone, and most developers are still training for it.

The skill we all optimized for

Everything about how developers learn was built around *know-how*: memorize the syntax, learn the patterns, practice until you can produce working code from a blank file. Interviews test it. Courses teach it. Careers were ranked by it.

Then LLMs made production nearly free. Describe what you want, and working-looking code appears in seconds. By Google's own account, more than a quarter of its new code is now AI-generated. The blank file is no longer the enemy.

It's tempting to conclude the job just got easier. The real question changed underneath us: if anyone can *generate* an answer, what separates a good developer from a bad one?

The scarce skill is no longer producing the answer. It is seeing, quickly and reliably, whether an answer is any good, and knowing how to critique the model into a better one. Call it *know-see*. Know-how ships the first draft. Know-see decides whether that draft belongs in production.

Why we keep training the wrong muscle

Because production was expensive for fifty years, our instincts equate typing with working and output with progress. Review was the low-status activity, the thing you did to other people's code, quickly, before getting back to "real work." Nobody built a career on being great at reading code. That habit is now exactly backwards.

What the evidence says

Generation is cheap. Verification is the new cost center. The METR randomized trial followed experienced open-source developers on their own mature codebases and found something uncomfortable: with AI tools they took 19% longer to finish tasks, while believing they had been 20% faster. Where did the time go? Not into typing. Developers accepted fewer than 44% of AI suggestions, and a majority reported major cleanup on the code they did accept. Reading, judging, and fixing, the know-see work, quietly absorbed the savings.

Your feeling of productivity is not a measurement. That same study exposed a roughly 40-point gap between perceived and actual speed. Google's DORA report, drawing on 39,000 professionals, found the same pattern at scale: as AI adoption rose 25%, delivery speed dipped and system stability dropped 7.2%, while three-quarters of developers *felt* more productive. Know-see starts with distrusting the feeling. Instrument reality: task completion times, defect rates, review depth. If you can't see your own performance clearly, you certainly can't see the model's.

The best output goes to the best critics. An LLM is like an infinitely fast junior colleague with unlimited confidence: the first draft arrives in seconds, polished-looking, occasionally wrong in ways designed to be missed. What improves the next draft isn't a vaguer "make it better", it's a precise critique. Name the flaw: "this ignores the timezone edge case," "this duplicates the retry logic in the client," "this test asserts nothing." The developers who get remarkable output are running a tight loop: define what *good* looks like before prompting, review the result like a hostile senior reviewing a junior's PR, and feed back specific, named defects. Being the main critic is not overhead on the AI workflow. It *is* the workflow.

Two fair objections. "*Other studies show real gains.*" True, a large study across Microsoft and Accenture found developers with Copilot completed 26% more tasks. Both findings can hold: gains concentrate where tasks are self-contained and verification is cheap; losses concentrate in complex, interconnected systems where seeing a subtle flaw is hard. The variable separating the two studies is precisely the cost of judgment. "*Models will get good enough that checking won't matter.*" The better models get, the more their failures shift from obvious to plausible, wrong in ways that read as right. Improvement raises the bar for the critic; it doesn't retire the critic.

A quick self-test

1. Can you tell within a minute whether a generated function is production-worthy, or do you find out in review, or worse, in production?
2. Do you read AI output with the same rigor you'd apply to a junior's pull request, or with the leniency you'd apply to your own code?
3. When output is mediocre, do you re-prompt with a named, specific critique, or accept it and patch by hand?

4. Do you measure your AI-assisted work, or trust how fast it feels?

Training know-see, deliberately

Read more code than you write. Reviewing, hostile, line-by-line, "what would break this?", is now the core practice, not the chore.

Write the definition of done before you prompt. If you can't state what good looks like, you'll accept whatever looks finished.

Build a critique vocabulary. Models respond to precision. "This is wrong" produces a reshuffle; "this leaks the connection on the error path" produces a fix.

AI didn't make developers replaceable. It made their judgment the entire product.

Sources: METR, "Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity" (RCT, 2025); Google DORA Report 2024; MIT/Princeton/UPenn Copilot field study; Alphabet Q3 2024 earnings call.

Revision #3
Created 8 July 2026 14:02:00 by EMB
Updated 10 August 2026 09:55:08 by EMB