

"Do You Still Read the Code?"

14 SEPTEMBER 2026

There is considerable difference between that question and a more innocuous “do you read the code?” The addition of *still* smuggles with it a theory of progress: reading code is something on its way out, like memorising phone numbers or unfolding paper maps, and the asker seems to want to know if you by chance aren’t one of the obscurantists still clinging to the old ways.

I use AI extensively, and I read what it produces. This is a deliberate choice about how I want to develop software, at least at my job, where I’m expected to be reasonably responsible for the code that I commit and deploy. Other people make a different choice, sometimes with considerable care. But we have begun sharing codebases without necessarily agreeing on what either path asks of our colleagues.

It is very hard to tell which choice will prevail. The *still* presumes that the matter has been settled, and indeed producing a working application these days has become quite a bit easier. But finding out what it costs to maintain it through changes of requirements, developers, and tooling takes considerably longer. We are making commitments with respect to how teams work now, and their consequences will only later be felt and understood. The confidence with which either mindset announces its victory seems somewhat premature.

Accelerators and vibecoders

As far as I can tell, there are two prevalent approaches to using AI these days.

Accelerators use AI to help translate their understanding into code. They intend to retain enough understanding of the implementation to explain the reasoning behind the translation from intent to code, anticipate the consequences of changes, and maintain the resulting model and its implementation. Reading the generated code is part of that commitment.

For accelerators, language models and harnesses exist in roughly the same category as text editors and their plugins: they can now code faster. They invest in their continued ability to explain and change the implementation. Large changes are slow to review, a generated diff that is determined to incorrectly implement a model is rewritten, cognitive debt¹ piles up whenever the team's reading falls behind the generating, and the whole thing requires discipline that is very hard to keep.

Vibecoders aim to delegate implementation and its continued revision to AI. Their attention moves towards specifying the desired behaviour, supplying context and domain knowledge, and establishing ways to determine whether the result is satisfactory. Understanding every implementation detail is no longer an intended product of their work.

Vibecoders expect language models to abstract away implementation, putting them in the same category as compilers and frameworks: there should no longer be a need to understand technical details. They invest in their continued ability to specify, regenerate, and evaluate code. Intent debt² may accumulate as requirements are rewritten or forgotten, context drifts between sessions, engineering time goes into curating what the agents get to see to keep them away from the “dumb zone,”³ and the whole thing stands or falls with the quality of a model that is controlled by Anthropic or OpenAI.

The distinction concerns the developer's relationship to the output, rather than how much of it the model writes.⁴ An accelerator might generate almost every line of a feature and still understand what was built, taking ownership of the reasoning behind it.

A vibecoder might spend considerable time refining a specification and its acceptance criteria, while deliberately treating the resulting implementation as disposable. Being an accelerator doesn't require knowing the answer before asking the model—you can use generated code to explore a problem you don't yet understand, provided that understanding is something you intend to acquire before the code ends up on the master branch.⁵

Naur is still undefeated

My own preference comes from what I think programming is for. I wrote before about how programming is essentially pure applied philosophy. I hadn't read, at that time, Peter Naur's essay,⁶ since brought to renewed prominence by the advent of LLMS, making a similar point thirty-five years earlier.

It is essentially the same argument, made better, by someone much smarter: the code is not the real product of programming. If anything, we mistake the code for an asset, while more often than not it's indeed a liability. The real product, the asset, is the model, or the theory, behind it. The theory is the kind of knowledge that lets a person not merely do something, but explain it, answer questions about it, use it to project the future, and adapt it if circumstances change. The programmer who possesses it is capable of three things:

“The programmer having the theory of the program can explain how the solution relates to the affairs of the world that it helps to handle. (...) The programmer having the theory of the program can explain *why* each part of the program is what it is. (...) The programmer having the theory of the program is able to respond constructively to any demand for a modification of the program so as to support the affairs of the world in a new manner.”

In other words, source code is only one kind of product of the activity of programming, but because it's more visible, it's treated as more valuable than the understanding acquired while producing it. It's as if we treated the steam coming out of a coal power plant cooling tower as the main output, rather than the electricity, merely because electricity is invisible.

A part of the job of a software engineer is to name and structure the parts of the business which have until now been understood only implicitly. This has benefits outside of the code: it can help the people whose work we are modelling understand what they do and see aspects of their work in a new light.

Programming is one way to sort the grains of sand of reality.⁷ We distinguish one thing from another, give distinctions names, and build a system whose behaviour lets us discover where our sorting was useful, and where it was mistaken.

Naming things in terms of their immediate appearance locks us into a certain mode of understanding of the product which may be accidental; naming them after their essential attributes is more flexible, but requires much more effort to discover and structure. And running software can only test the consequences of the sorting. Judging whether the model fits the domain still requires contact with the people and processes being modelled. A green test suite only proves that the program does what you said, not that what you said corresponds to reality.

How a programmer understands the domain changes as he works through the implementation. You might discuss a requirement or even whiteboard it, but your conceptual analysis might not survive first contact with the code. When you must account for every case, you might completely change your understanding of the problem and be forced back to square one.

The implementation, then, is not merely a translation of the specification into runnable code, but one of the places where it is revised. Vibecoders seem to bet that the theory can be built by specifying and running software, without dealing with its implementation. I am not so sure about that.

Choice and drift

Choosing the accelerator or vibecoder approach involves different commitments, even though the same person might choose differently for different projects, or even submodules within the same project.

For example, I might lean into vibecoding in throwaway or exploratory projects, where I don't actually care about the underlying implementation and just want to see something on the screen. The difficulty is that one can cease to maintain an understanding of the implementation without ever consciously thinking about how to correctly delegate it. You start skimming the diff instead of reading it, you need a lot of time to recall or even come up with a justification for a specific implementation choice, and eventually the only practical way to find out what it is that you've actually built is to ask the model, with no way to verify if what the model tells you is correct.

This is why I don't think of the two approaches as some kind of spectrum. You can be an accelerator in one module and a vibecoder in another, but not halfway in either. A drifting accelerator doesn't end up somewhere between the two approaches; he ends up a vibecoder by default, without the harness around specifications and evaluations that a deliberate vibecoder would have built to make up for the missing theory.

For my own commercial work, reading every line of the generated code is a good way to make sure I don't drift. It doesn't guarantee my understanding, but it provides repeated opportunities to find out where the understanding and the implementation diverge.

And they diverge because agents still make dumb choices, and often don't respect some human dimensions, like time, that aren't expressible as textual output. An agent that added tests just looks at the terminal output; it doesn't really matter to that agent that the tests now take three times longer to run. But speed matters to a human. Likewise, agents with polluted context windows drifting into the "dumb zone" will start making obvious local mistakes like declaring React components inside other components, or violating the rules of hooks.

A marriage of inconvenience

I don't want this to sound like an accelerator manifesto, because I am entirely open to the possibility that the vibecoders are right and AI will make looking at code obsolete—I'm just not seeing enough compelling evidence for that conclusion yet.

What I am certain of, though, is that there is one obviously harmful practice: putting people who prefer different approaches on the same team without establishing expectations and boundaries beforehand, and then keeping to them.

An accelerator may inherit the work of reconstructing understanding that the author never intended to retain or never had in the first place. A vibecoder may be asked to explain incidental implementation choices, even though he spent a considerable amount of time refining his development process to intentionally make those choices disposable. Either can make the other's work harder by imposing an unstated maintenance expectation.

In ordinary correspondence, sending someone unfiltered AI output is impolite. But when it starts to concern code, intent, and responsibility for issues in production, courtesy becomes an engineering concern. Before merging changes, colleagues need to know how those changes are meant to be maintained: through a developer's understanding of the implementation, through specification and an established process of code generation and rigorous testing, or through some combination of the two, as long as it's clear which parts are which.

Whether reading is enough

None of this makes the accelerator's position comfortable.

Skills deteriorate when they are not used, and it remains to be seen if merely reviewing AI-generated code, rather than typing it out, is enough to take over and switch back to manual coding if the situation should demand it. Some circumstances that were often speculated about, such as the major AI labs going bankrupt or hiking prices, seem to be no longer a concern (we have access to many providers of open-weights models of high enough quality to trundle along if it comes to it), but there might be some dangers to automation that we do not foresee.⁸ And perhaps whoever inherits a vibecoded, or even accelerated, codebase might find it prohibitively difficult to maintain.

Declaring that a human remains responsible is easy. Arranging the work so that the human remains capable of exercising that responsibility is considerably more difficult

and requires very multidimensional decision-making. It might turn out that the accelerator approach is just a fast track to burnout.

Reviewing everything involves constraints on how much unfamiliar work you can do at once. When coding with AI, you still need to make sure to model your requirements and ensure that the implementation doesn't drift; otherwise the job of self-reviewing the output is the worst kind of work—"very boring but very responsible, yet there is no opportunity to acquire or maintain the qualities required to handle the responsibilities."⁹

It's a conscious choice of a demanding practice whose success requires more than good intentions. If software development is ever fully automated, having programmers perform manual coding tasks just to maintain their skill might become a necessary cost for software companies in order to have someone around who can take over in a moment of need.¹⁰

Intent to code

What the AI revolution seems to have revealed about code review is that we never cared about the quality of the code in itself, but about the understanding of the product *expressed by* the quality of the implementation. Code quality was a useful proxy for that understanding in the time before Claude Code, but a language model can now feign that understanding convincingly. The presence of generated code in the codebase makes it more important to examine the understanding and the implementation separately.

A reviewer needs to distinguish requirements, deliberate implementation decisions, inherited conventions, and choices for which no rationale was recorded. Those distinctions should remain connected to the code as it changes.

The point is not whether a specific piece of code was generated or typed on a keyboard. Humans make incidental choices that are hard to explain, too, but it's considerably more difficult to make a great many of them and still end up with working software. Agents do that all the time, but the fact that they do so is not a reason to dismiss them outright: it's still entirely possible for them to implement explicit decisions faithfully.

Consider the following story: the business requirement is that a user may request password reset links and that they expire. A developer chooses a particular expiry period. An agent chooses how to represent and check it. The resulting code implements those decisions in several places, but the intent is lost—when a reader comes later and sees a value of `expiryTime = 6h`, he can see what the software does, but the six hours could have come from several places: an explicit business requirement, an existing convention, a considered trade-off, or a guess that no one challenged. To decide whether the value should change, the reader needs to know what justified it and whether those circumstances still hold. The decision’s result is in the code, but on its own it doesn’t preserve enough of its history to reconsider it fully. This is intent debt, and reading every line of the code doesn’t pay it off.

One promising way agents can help here is by leveraging their ability to summarise, letting a reviewer trace those relationships and see where a choice lacks an explanation. If you review the code yourself, you can use a tool like Crit to make sure the implementation didn’t stray from what you intended. Or you can use CodeRabbit’s Change Stack, or my own intent-stack skill, to gather context and generate an aid for the reviewer.

As for automation, greater delegation makes durable goals, constraints, validation criteria, and relevant context increasingly difficult to preserve as the generated codebase grows. Simultaneously, that preservation becomes increasingly important with the growth of the number of lines of code, because the agents doing the implementation might make contradictory decisions simply because they didn’t chance upon the relevant piece of context, or they ascribed authority to information that was meant to be disposable. Those records must remain usable when sessions end, agents change, and the implementation is regenerated.

The various experiments in building “software factories” and “graph engineering” seem to me to be attempts at making that delegation deliberate by cataloguing intent in ever more fractal structures of summary, so that trees or graphs or swarms of agents can communicate at different levels of generality without polluting each other’s context windows with things that aren’t relevant to them. I’m not very in tune with this approach, but a reasonable starting point might be reviewing StrongDM’s account of their software factory, which runs on two rules: no writing code by humans, and no

reviewing code by humans. Instead, they validate agents' work against scenarios kept outside the codebase.¹¹

Two paths

There are, then, at least two kinds of progress in AI-assisted programming: one helps developers understand and interact with the implementation, and one aims to remove the need for that interaction altogether. While the latter seems to be getting more of the limelight, these are different paths, they require different tools and different approaches. Both may improve, but it does not follow that the goal of the accelerator is to become a vibecoder. Ceasing to read code is not, in itself, progress.

But before asking whether your colleague *still* reads code, perhaps consider whether you're *still* expecting him to maintain yours.

WRITTEN BY

Marek Chotoborski

Product engineer. I work with domain experts to turn their knowledge into software. If you have a problem you cannot yet put into words, my [office hours](#) are open, or you can [write to me](#) directly.

ZANLIB.DEV © 2026 • EST. 2017 • PRIVACY

