Code completion is the easy part

There's a tempting story going around: soon AI will write the code, and humans will "just" do architecture. It's a pleasant fantasy because it turns software engineering into a clean split between thinking and typing.

The MIT CSAIL paper summarized by MIT News pushes back on that split. The thesis is simple and slightly brutal: code completion is the easy part; the hard part is everything else.

Software engineering is not LeetCode

One reason the narrative goes wrong is that it compresses software engineering into "the undergrad programming part": implement a function from a neat spec, or solve an interview puzzle. Real work is broader and messier: refactors that slowly improve design, migrations that reshape a business, and an ongoing fight against bugs, regressions, and security issues.

MIT's summary lists the kinds of tasks that dominate practice: large-scale migrations (e.g., moving millions of lines from COBOL to Java), continuous testing and analysis (fuzzing, property-based testing), and the maintenance grind (documentation, summarizing history for new teammates, reviewing pull requests for correctness, performance, security, and style).

This is why "AI writes code" is not the end of the story. The question is whether it can write code that belongs to a living codebase: one that already has conventions, constraints, unwritten assumptions, and a deployment pipeline that is merciless to small mistakes.

Evaluation is the bottleneck

Engineering organizations run on feedback loops. If AI is going to do real engineering work, we need strong ways to measure whether it improved the codebase or quietly made it worse.

The MIT News piece calls out a gap between what we evaluate and what we need. Many metrics and benchmarks are still biased toward short, self-contained tasks, while real high-stakes engineering includes performance-critical rewrites, long refactors, and multi-step changes that must survive integration and future edits.

It also notes that popular yardsticks like SWE-Bench—patching a GitHub issue—can be useful but remain close to the "small spec" paradigm, often touching only hundreds of lines. That leaves out scenarios like AI-assisted refactors, pair-programming workflows, and changes that span huge codebases or require careful performance validation.

The thin control channel problem

Even when code generation "works," the developer experience can be fragile. MIT's summary describes today's interaction as a thin line: you ask for a change and get back a large, unstructured blob of code (sometimes with unit tests), but you still don't have fine-grained control over the model's decisions.

A key missing feature is an explicit uncertainty interface. In the article, Alex Gu points out that without a channel for the AI to surface confidence ("this part is solid, this part you should double-check"), teams risk trusting hallucinated logic that compiles and looks clean, but fails under production conditions.

Another missing piece is disciplined deferral: the model noticing when the request is underspecified and asking for clarification instead of guessing. In mature engineering, "ask before you guess" is not politeness; it's reliability.

Scale breaks the illusion

Small demos hide the hardest part: context. The MIT News summary emphasizes that models struggle with large codebases (millions of lines) and with proprietary repositories where conventions, internal helpers, and architectural patterns are out-of-distribution relative to public GitHub training data.

The failure mode is painfully familiar: code that looks plausible but calls non-existent functions, violates internal style rules, or fails CI. The output is not "random"; it is specifically dangerous because it is coherent enough to pass a quick glance.

The same problem shows up in retrieval: the article notes that models can retrieve code that looks similar by name or syntax rather than by functionality. That is, the system finds something that matches the surface form, not the underlying semantics, and then builds on a false foundation.

What changes for engineers

The natural conclusion is not "AI can't do it." It's that the profession's leverage shifts from typing to constraint design. If implementation becomes cheap, the scarce skill becomes specifying intent precisely and building systems that reject incorrect work quickly.

Concretely, this means doubling down on tools and interfaces that make software behavior testable and inspectable: stronger regression suites, property-based tests where they pay off, better static analysis, better observability in production, and better benchmarks that measure things like refactor quality, bug-fix longevity, and migration correctness.

The framing that stuck with me is that the core engineering problem becomes: how to make a model a collaborator you can audit. Not "can it write code?", but "can it expose its assumptions, surface uncertainty, and operate inside feedback loops that punish wrongness?"

Contributor: Alessandro Linzi