How AI changed programming — and a practical guide to controlling it in your company

Francesco Barbato · Full Stack · Cybersecurity · AI

Published 6 March 2026 · 12 min read

ABSTRACT

AI didn't just speed up coding: it restructured the entire craft. What actually changed, what it means for your team, and a concrete framework to evaluate, control and get real value from AI tools without losing ownership of your codebase.

Keywords: AI engineering · Productivity · LLM · Software craft · Management · Code review

In 2021, a senior developer spent roughly 30% of their time writing boilerplate. By 2024, that number had collapsed toward zero for anyone using modern AI tooling. But the story of AI and programming is not a simple "productivity up" graph. It is a more complicated, more interesting, and in some ways more dangerous story, about what happens when a craft gets a powerful new instrument before it has developed the norms to use it wisely.

This article covers two things. First, an honest account of what AI actually changed in software development. Second, a practical framework for companies, whether you're managing an internal team or working with external vendors, to stay in control of work you increasingly can't read line by line.

1 What actually changed

The narrative often starts with GitHub Copilot in 2021. That's not wrong, but it undersells the depth of the shift. AI didn't just add an autocomplete layer. It changed the *nature* of several distinct activities in the development workflow.

1.1 1. Writing code vs. directing code

Before AI tooling, a developer's primary activity was **writing**: translating a mental model into syntax, one expression at a time. That activity required fluency, the ability to produce correct, idiomatic code from memory and pattern recall.

With capable AI assistants, the primary activity is shifting toward **directing**: specifying intent clearly enough that the model produces useful output, then critically evaluating what comes back. This is a genuinely different skill. It rewards clarity of thought, precision of specification and critical reading over syntactic fluency.

This is not a lesser skill. It is, arguably, a harder one, because you can no longer hide imprecise thinking behind the mechanical work of typing. If you can't explain clearly what you want, the model reveals that immediately.

1.2 2. The collapse of the blank page problem

One of the most consistent productivity gains is in **project initialisation**. Setting up a Next.js app with authentication, a PostgreSQL schema with sensible defaults, a Docker Compose file with correct networking: tasks that took hours of documentation-reading and trial and error can now be bootstrapped in minutes.

This is genuinely valuable, and genuinely risky. Fast bootstrapping hides the learning that happens when you struggle through setup manually. Developers who have only ever scaffolded projects with AI assistance sometimes lack the mental model to debug when the scaffold breaks in production.

1.3 3. Documentation, tests and the “boring” work

AI tools are remarkably good at tasks that developers historically avoided: writing docstrings, generating test cases, producing changelog entries, converting code comments into README sections. The friction cost of good documentation dropped by an order of magnitude.

In teams that have actually adopted this, the quality of repositories has improved significantly, not because developers became more disciplined, but because the cost of the discipline dropped.

1.4 4. Context-switching and cross-domain work

A full-stack developer in 2020 had strong opinions about which layers of the stack they were comfortable with. Asking a React developer to write a Kubernetes deployment file felt unreasonable. Today, with AI assistance, the blast radius of competence has expanded dramatically. A developer comfortable directing AI can produce reasonable work in adjacent domains: not expert-level, but functional and reviewable.

This has significant implications for team structure, hiring and how you scope work with external vendors.

1.5 5. What didn't change

It is equally important to name what AI has *not* changed, because the hype often elides these points:

- **System design.** AI can suggest patterns, but deciding which distributed architecture fits your scale, team and constraints is still a human judgment call that requires context no model has.
- **Debugging complex failures.** AI is good at fixing simple, well-defined bugs. It struggles with emergent failures in distributed systems where the problem is causal, not syntactic.

- **Business logic correctness.** A model can write code that compiles and passes tests but encodes the wrong business rule. Validating correctness against intent is still entirely a human responsibility.
- **Security judgment.** AI-generated code can introduce vulnerabilities that are syntactically plausible and logically subtle. No current model reliably reasons about threat models.

> **THE CORE TENSION**
>
> AI tools increase the speed at which code is produced. They do not proportionally increase the speed at which code is *understood, validated or owned*. This gap is where most AI-related development problems live.

2 The new risk landscape for companies

If you are a technical manager, CTO or business owner commissioning software today, AI tooling has introduced risks that didn't exist a few years ago, and most governance frameworks haven't caught up.

2.1 Opacity at scale

A developer using AI assistance can produce 3–5 times more lines of code per day. That means a vendor or employee can deliver a codebase that *looks* impressive by volume, compiles, passes basic tests, and is nonetheless brittle, unmaintainable, or riddled with copy-paste inconsistencies that the developer never understood well enough to catch.

The volume of output is no longer a reliable signal of effort or quality. Reviewing a 10,000-line pull request that was substantially AI-generated requires different skills and different tools than reviewing the same PR written manually.

2.2 Ownership and liability gaps

When a developer writes code manually, there is a reasonable presumption that they understand it. When they direct AI to write it and accept the output without deep review, that presumption breaks. If something goes wrong in production (a data breach, a billing error, a regulatory violation), the question of who understood the code becomes legally and practically important.

2.3 Intellectual property uncertainty

The legal landscape around AI-generated code and copyright is still developing. In most jurisdictions, AI output is not copyrightable by the user. For code that forms the core of a product, this may be relevant to your IP protection strategy.

2.4 Dependency and deskilling

Teams that rely heavily on AI for all code generation risk losing the baseline fluency needed to debug, extend or migrate their own codebase when AI tooling is unavailable, expensive, or produces incorrect output for a novel problem. This is a long-term organisational risk that doesn't show up on a quarterly productivity dashboard.

3 A practical control framework

The answer to these risks is not to ban AI tools, which is both futile and counterproductive. The answer is to build governance structures that capture the benefits while maintaining accountability and ownership. Here is a framework in five layers.

3.1 Layer 1: define what AI-assisted means in your contracts

If you work with external vendors or freelancers, your contracts almost certainly predate the AI tooling era. Update them. Specifically, define:

- Whether AI-assisted development is permitted (almost always: yes)
- That the vendor is fully responsible for reviewing, understanding and standing behind all AI-generated output as if they had written it manually
- That AI-generated code must pass the same review and testing standards as manually written code
- That ownership of the codebase transfers to you unencumbered, regardless of how it was generated

> **CONTRACT CLAUSE EXAMPLE**
>
> "The use of AI-assisted development tools is permitted. The Supplier warrants that all code delivered has been reviewed, tested and understood by a qualified developer, and accepts full responsibility for its correctness, security and maintainability regardless of the method of generation."

3.2 Layer 2: shift your review criteria

Traditional code review asks: *is this code correct?* In an AI-assisted world, you also need to ask: *does the author understand this code?*

Practical techniques for this:

- **Explanation requirements.** For any non-trivial function or module, require the developer to write a brief comment or PR description explaining *why* the approach was chosen, not just what it does. This is trivial for a developer who understands their code and very hard to fake convincingly.
- **Targeted questions in review.** Ask reviewers to pick 2–3 specific sections of each PR and ask the author to explain them in a call or written response. This is not a gotcha: it's a calibration tool.
- **Decision log requirement.** For architectural decisions, require a short decision record (ADR) explaining the alternatives considered and why the chosen approach fits the context.

3.3 Layer 3: automated quality gates

Set minimum automated standards that all code must pass before review, regardless of how it was written:

Table 1. Automated gates for every pull request

Gate	Tool examples	What it catches
Static analysis	ESLint, Pylint, SonarQube	Code smells, anti-patterns, complexity
Security scanning	Semgrep, Snyk, Trivy	Known vulnerability patterns, secrets in code
Test coverage	Jest, pytest-cov, Codecov	Untested paths, regression risk
Dependency audit	npm audit, pip-audit	Vulnerable or unlicensed dependencies
Type checking	TypeScript, mypy	Type errors that AI tools frequently introduce
Duplication detection	jscpd, SonarQube	Copy-paste patterns common in AI output

These gates should run on every pull request, automatically, with no manual bypass. They are not a substitute for human review: they are the floor below which no code reaches review at all.

3.4 Layer 4: the "explain and extend" test for vendors

When accepting a delivery from an external vendor, don't just run the code and check if it works. Run an **explain and extend** session:

1. Ask the vendor to walk you through the three most complex parts of the codebase, not reading comments, but explaining the logic in their own words.
2. Give them a small, well-defined extension task, something that requires understanding the existing architecture. It should take a competent developer 1–2 hours. Ask them to do it live or within a short fixed window.
3. Evaluate both the explanation quality and the extension quality. A developer who genuinely owns the code will handle both easily. A developer who generated without reviewing will struggle with one or both.

> **WHY THIS WORKS**
>
> AI tools make it easy to produce code. They don't make it easy to maintain a coherent mental model of a codebase you didn't build incrementally. The explain-and-extend test directly probes the thing AI cannot fake: genuine understanding of the specific system.

3.5 Layer 5: internal team — build AI fluency, not AI dependency

For your internal team, the goal is different from the vendor control goal. You want your developers to become *more capable* through AI, not more fragile. Some practices that work:

- **AI-assisted, human-explained.** Establish a team norm that using AI to write a function is fine, but the developer must be able to explain it in the next standup if asked. This keeps ownership without banning tooling.
- **Rotate the hard problems.** Deliberately assign debugging and architecture work that requires deep thinking and is unlikely to be solvable by prompting. This keeps skills sharp.
- **Prompt review as a skill.** Treat the quality of AI prompts as a reviewable artefact. Share effective prompts in a team knowledge base. Poor prompts that led to poor code are a learning opportunity, not a secret.

- **Time-box AI for learning contexts.** When a junior developer is learning a new area, consider having them attempt it manually first before using AI assistance. The struggle is the learning.

4 Evaluating AI tools for your organisation

If you're deciding which AI coding tools to adopt, here is a lightweight evaluation framework. Score each tool on these dimensions before committing:

Table 2. Six dimensions to score

Dimension	Questions to ask
Data privacy	Does the tool train on your code? Can you opt out? Is there an enterprise tier with data isolation?
IDE integration	Does it fit your team's existing toolchain without forcing workflow changes?
Language coverage	Is it strong in the languages and frameworks your team actually uses?
Context window	Can it reason about your whole codebase, or just the current file?
Auditability	Can you see what the model was given as context when it produced a suggestion?
Cost at scale	What is the per-seat cost at your team size? Are there usage limits that create friction?

4.1 The leading tools, as of early 2025

GitHub Copilot was the most widely deployed, with strong IDE integration and an enterprise tier that guarantees no training on your code. **Cursor** had become the preferred choice for developers who want deep codebase context: it can ingest a whole repository and reason across files, and its multi-file edits are powerful but need careful review. **Claude** (via API or claude.ai) was particularly strong for explanation, refactoring and architectural reasoning. **Codeium** was worth considering for cost-sensitive teams, with a generous free tier.

> **TOOLS CHANGE FASTER THAN CRITERIA**
>
> This list reflects early 2025, and products in this space change every few months. The six dimensions above are the part that lasts: re-run the evaluation every 6–12 months rather than trusting a ranking.

5 A note on AI agents and autonomous coding

The frontier is moving fast toward autonomous coding agents: systems that don't just suggest code but execute multi-step tasks, reading your codebase, writing tests, submitting pull requests and iterating on failures. Tools like Devin and SWE-agent were early examples.

For most companies, these are not ready for unsupervised use on critical codebases. But the direction is clear: a meaningful fraction of routine development tasks will be executed by agents with human approval at checkpoints, not at every line.

The governance frameworks you build now (code review standards, explanation requirements, automated quality gates) are exactly the checkpoints that will make agent-assisted development safe as it matures. Building them now is not just good practice for today's tools; it is infrastructure for the next wave.

> **KEY TAKEAWAY**
>
> AI did not replace programming. It replaced the parts of programming that didn't require thinking, and made the parts that do require thinking more exposed and more important. The companies that will navigate this best are those that invest in the human skills (critical reading, clear specification, architectural judgment) that AI cannot substitute, while building governance structures that keep them in ownership of what gets built.

6 Conclusion

The shift AI has brought to programming is real, substantial and still accelerating. Ignoring it is not a viable strategy. Neither is adopting it without structure. The middle path, deliberate adoption with clear ownership, quality gates and honest accountability, is harder than either extreme, but it is the only one that captures the upside without accumulating the technical and organisational debt that unstructured AI use creates.

If you take one thing from this article: the question is not whether your team or your vendors are using AI. They almost certainly are. The question is whether you have built the systems to stay in ownership of what those tools produce.

> **HOW TO CITE THIS ARTICLE**
>
> Francesco Barbato (2026). How AI changed programming — and a practical guide to controlling it in your company. localhost:3000.
>
> Online version: http://localhost:3000/en/blog/ai-changed-programming-practical-guide