# Come l'IA ha cambiato la programmazione — e una guida pratica per controllarla in azienda

**Francesco Barbato** · Full Stack · Cybersecurity · AI

Pubblicato il 6 marzo 2026 · 13 min di lettura

**SOMMARIO**

L'IA non ha solo accelerato la scrittura del codice: ha ristrutturato l'intero mestiere. Cosa è cambiato davvero, cosa significa per il tuo team e un metodo concreto per valutare, controllare e ottenere valore reale dagli strumenti di IA senza perdere il controllo del codice.

**Parole chiave:** Ingegneria dell'IA · Produttività · LLM · Mestiere del software · Gestione · Revisione del codice

Nel 2021 uno sviluppatore senior passava circa il 30% del tempo a scrivere codice ripetitivo. Nel 2024 quella quota era crollata verso zero per chiunque usasse i moderni strumenti di IA. Ma la storia dell'IA e della programmazione non è un semplice grafico di “produttività in crescita”. È una storia più complicata, più interessante e per certi versi più pericolosa: cosa succede quando un mestiere riceve un nuovo strumento potente prima di aver sviluppato le regole per usarlo con giudizio.

Questo articolo affronta due cose. Primo, un resoconto onesto di cosa l'IA ha davvero cambiato nello sviluppo software. Secondo, un metodo pratico per le aziende, che gestiscano un team interno o lavorino con fornitori esterni, per restare in controllo di un lavoro che sempre meno si riesce a leggere riga per riga.

## 1 Cosa è cambiato davvero

Il racconto parte spesso da GitHub Copilot nel 2021. Non è sbagliato, ma sottovaluta la profondità del cambiamento. L'IA non ha aggiunto soltanto uno strato di completamento automatico. Ha cambiato la *natura* di diverse attività distinte del processo di sviluppo.

### 1.1 1. Scrivere il codice o dirigere il codice

Prima degli strumenti di IA, l'attività principale di uno sviluppatore era **scrivere**: tradurre un modello mentale in sintassi, un'espressione alla volta. Serviva fluidità, la capacità di produrre codice corretto e idiomatico a memoria.

Con assistenti di IA capaci, l'attività principale si sposta verso il **dirigere**: esprimere l'intenzione con abbastanza chiarezza da ottenere un risultato utile dal modello, e poi valutare con spirito critico quello che torna indietro. È una competenza davvero diversa. Premia la chiarezza di pensiero, la precisione nelle specifiche e la lettura critica più della scioltezza sintattica.

Non è una competenza minore. È, probabilmente, più difficile, perché non si può più nascondere un pensiero impreciso dietro il lavoro meccanico della digitazione. Se non sai spiegare con chiarezza cosa vuoi, il modello lo rivela subito.

### 1.2 2. La fine del problema della pagina bianca

Uno dei guadagni di produttività più costanti riguarda l'**avvio dei progetti**. Impostare un'app Next.js con autenticazione, uno schema PostgreSQL con valori predefiniti sensati, un file Docker Compose con la rete configurata correttamente: attività che richiedevano ore di lettura della documentazione e tentativi ora si avviano in pochi minuti.

È un vantaggio reale, e un rischio reale. Partire in fretta nasconde l'apprendimento che avviene quando si fatica con la configurazione a mano. Chi ha sempre creato i progetti con l'aiuto dell'IA a volte non ha il modello mentale per fare debug quando la struttura iniziale si rompe in produzione.

### 1.3 3. Documentazione, test e il lavoro “noioso”

Gli strumenti di IA sono sorprendentemente bravi nei compiti che gli sviluppatori hanno sempre evitato: scrivere docstring, generare casi di test, redigere le voci del changelog, trasformare i commenti del codice in sezioni del README. Il costo di attrito di una buona documentazione è calato di un ordine di grandezza.

Nei team che l'hanno davvero adottato, la qualità dei repository è migliorata in modo significativo, non perché gli sviluppatori siano diventati più disciplinati, ma perché il costo della disciplina è sceso.

### 1.4 4. Cambio di contesto e lavoro tra ambiti diversi

Uno sviluppatore full stack nel 2020 aveva idee ben precise su quali livelli dello stack padroneggiava. Chiedere a uno sviluppatore React di scrivere un file di deployment per Kubernetes sembrava irragionevole. Oggi, con l'aiuto dell'IA, il raggio di competenza si è allargato enormemente. Chi sa dirigere l'IA può produrre un lavoro ragionevole in ambiti vicini: non da esperto, ma funzionante e rivedibile.

Questo ha implicazioni importanti per la struttura dei team, le assunzioni e il modo in cui si definisce il lavoro con i fornitori esterni.

### 1.5 5. Cosa non è cambiato

È altrettanto importante dire cosa l'IA *non* ha cambiato, perché l'entusiasmo tende a trascurarlo:

- **La progettazione dei sistemi.** L'IA può suggerire schemi, ma decidere quale architettura distribuita si adatta alla tua scala, al tuo team e ai tuoi vincoli resta un giudizio umano che richiede un contesto che nessun modello possiede.

- **Il debug dei guasti complessi.** L'IA è brava a correggere bug semplici e ben definiti. Fatica con i guasti emergenti nei sistemi distribuiti, dove il problema è causale e non sintattico.
- **La correttezza della logica di business.** Un modello può scrivere codice che compila e supera i test ma codifica la regola sbagliata. Verificare la correttezza rispetto all'intenzione resta interamente una responsabilità umana.
- **Il giudizio sulla sicurezza.** Il codice generato dall'IA può introdurre vulnerabilità plausibili dal punto di vista sintattico e sottili dal punto di vista logico. Nessun modello attuale ragiona in modo affidabile sui modelli di minaccia.

> **LA TENSIONE CENTRALE**
>
> Gli strumenti di IA aumentano la velocità con cui il codice viene prodotto. Non aumentano nella stessa misura la velocità con cui viene *capito, validato o posseduto*. In questo divario nascono la maggior parte dei problemi di sviluppo legati all'IA.

## 2 Il nuovo panorama dei rischi per le aziende

Se sei un responsabile tecnico, un CTO o un imprenditore che commissiona software oggi, gli strumenti di IA hanno introdotto rischi che pochi anni fa non esistevano, e la maggior parte dei modelli di governance non li ha ancora recepiti.

### 2.1 Opacità su larga scala

Uno sviluppatore che usa l'IA può produrre da 3 a 5 volte più righe di codice al giorno. Questo significa che un fornitore o un dipendente può consegnare una base di codice che *sembra* impressionante per volume, compila, supera i test di base, ed è comunque fragile, difficile da mantenere o piena di incoerenze da copia-incolla che lo sviluppatore non ha mai capito abbastanza da individuare.

Il volume prodotto non è più un segnale affidabile di impegno o di qualità. Rivedere una pull request da 10.000 righe in gran parte generata dall'IA richiede competenze e strumenti diversi rispetto alla stessa PR scritta a mano.

### 2.2 Vuoti di responsabilità e di proprietà

Quando uno sviluppatore scrive il codice a mano, si può ragionevolmente presumere che lo capisca. Quando lo fa scrivere all'IA e accetta il risultato senza una revisione approfondita, quella presunzione cade. Se qualcosa va storto in produzione (una violazione di dati, un errore di fatturazione, una violazione normativa), la domanda su chi capisse il codice diventa importante, sul piano legale e pratico.

### 2.3 Incertezza sulla proprietà intellettuale

Il quadro giuridico su codice generato dall'IA e diritto d'autore è ancora in evoluzione. Nella maggior parte degli ordinamenti, il risultato prodotto dall'IA non è tutelabile dall'utente come opera protetta. Per il codice che costituisce il cuore di un prodotto, può contare nella strategia di protezione della proprietà intellettuale.

### 2.4 Dipendenza e perdita di competenze

I team che si affidano molto all'IA per tutta la generazione di codice rischiano di perdere la fluidità di base necessaria per fare debug, estendere o migrare il proprio codice quando gli strumenti di IA non sono disponibili, costano troppo o producono risultati sbagliati su un problema nuovo. È un rischio organizzativo di lungo periodo che non compare nel cruscotto trimestrale della produttività.

## 3 Un metodo pratico per il controllo

La risposta a questi rischi non è vietare gli strumenti di IA, cosa vana e controproducente. La risposta è costruire strutture di governo che colgano i vantaggi mantenendo responsabilità e proprietà. Ecco un metodo in cinque livelli.

### 3.1 Livello 1: definisci nei contratti cosa significa "con l'aiuto dell'IA"

Se lavori con fornitori esterni o freelance, i tuoi contratti quasi certamente sono precedenti all'era degli strumenti di IA. Aggiornali. In particolare, definisci:

- Se lo sviluppo assistito dall'IA è consentito (quasi sempre: sì)
- Che il fornitore è pienamente responsabile di rivedere, capire e garantire tutto ciò che l'IA produce, come se lo avesse scritto a mano
- Che il codice generato dall'IA deve superare gli stessi standard di revisione e di test del codice scritto a mano
- Che la proprietà del codice passa a te senza vincoli, indipendentemente da come è stato generato

> **ESEMPIO DI CLAUSOLA CONTRATTUALE**
>
> "L'uso di strumenti di sviluppo assistito dall'intelligenza artificiale è consentito. Il Fornitore garantisce che tutto il codice consegnato è stato rivisto, testato e compreso da uno sviluppatore qualificato, e si assume la piena responsabilità della sua correttezza, sicurezza e manutenibilità, a prescindere dal metodo di generazione."

### 3.2 Livello 2: sposta i criteri di revisione

La revisione tradizionale del codice si chiede: *questo codice è corretto?* In un mondo assistito dall'IA bisogna chiedersi anche: *chi l'ha scritto lo capisce?*

Alcune tecniche pratiche:

- **Spiegazione obbligatoria.** Per ogni funzione o modulo non banale, chiedi allo sviluppatore di scrivere un breve commento o una descrizione della PR che spieghi *perché* è stato scelto quell'approccio, non solo cosa fa. È banale per chi capisce il proprio codice e molto difficile da simulare in modo convincente.
- **Domande mirate in revisione.** Chiedi ai revisori di scegliere 2 o 3 punti specifici di ogni PR e di far spiegare all'autore, in una chiamata o per iscritto. Non è un tranello: è uno strumento di calibrazione.

- **Registro delle decisioni.** Per le decisioni architetturali, chiedi un breve documento (ADR) che spieghi le alternative considerate e perché quella scelta si adatta al contesto.

### 3.3 Livello 3: controlli automatici di qualità

Fissa standard automatici minimi che tutto il codice deve superare prima della revisione, a prescindere da come è stato scritto:

**Tabella 1**. Controlli automatici per ogni pull request

| Controllo | Esempi di strumenti | Cosa individua |
|---|---|---|
| Analisi statica | ESLint, Pylint, SonarQube | Segnali di cattivo codice, anti-pattern, complessità |
| Scansione di sicurezza | Semgrep, Snyk, Trivy | Schemi di vulnerabilità noti, segreti nel codice |
| Copertura dei test | Jest, pytest-cov, Codecov | Percorsi non testati, rischio di regressioni |
| Audit delle dipendenze | npm audit, pip-audit | Dipendenze vulnerabili o senza licenza |
| Controllo dei tipi | TypeScript, mypy | Errori di tipo che gli strumenti di IA introducono spesso |
| Rilevamento dei duplicati | jscpd, SonarQube | Schemi da copia-incolla tipici dell'output dell'IA |

Questi controlli devono girare su ogni pull request, in automatico, senza scorciatoie manuali. Non sostituiscono la revisione umana: sono il livello minimo sotto il quale nessun codice arriva alla revisione.

### 3.4 Livello 4: il test “spiega ed estendi” per i fornitori

Quando accetti una consegna da un fornitore esterno, non limitarti a eseguire il codice e verificare che funzioni. Fai una sessione di **spiega ed estendi**:

1. Chiedi al fornitore di guidarti attraverso le tre parti più complesse del codice, non leggendo i commenti ma spiegando la logica con parole sue.
2. Dagli un piccolo compito di estensione ben definito, che richieda di capire l'architettura esistente. Deve richiedere a uno sviluppatore competente 1 o 2 ore. Chiedigli di farlo dal vivo o in una breve finestra di tempo fissa.
3. Valuta sia la qualità della spiegazione sia quella dell'estensione. Chi possiede davvero il codice se la cava facilmente in entrambe. Chi ha generato senza rivedere farà fatica in una o in entrambe.

> **PERCHÉ FUNZIONA**
>
> Gli strumenti di IA rendono facile produrre codice. Non rendono facile mantenere un modello mentale coerente di una base di codice che non si è costruita passo dopo passo. Il test spiega ed estendi mette alla prova direttamente ciò che l'IA non può simulare: la comprensione autentica di quel sistema specifico.

### 3.5 Livello 5: il team interno — costruire competenza con l'IA, non dipendenza

Per il team interno l'obiettivo è diverso da quello del controllo dei fornitori. Vuoi che i tuoi sviluppatori diventino *più capaci* grazie all'IA, non più fragili. Alcune pratiche che funzionano:

- **Assistito dall'IA, spiegato da una persona.** Stabilisci come regola del team che usare l'IA per scrivere una funzione va bene, ma lo sviluppatore deve saperla spiegare al prossimo standup, se richiesto. Così si mantiene la responsabilità senza vietare gli strumenti.
- **Ruota i problemi difficili.** Assegna di proposito attività di debug e architettura che richiedono un pensiero profondo e difficilmente si risolvono con una richiesta. Così le competenze restano allenate.
- **La revisione delle richieste come competenza.** Tratta la qualità delle richieste all'IA come un artefatto da rivedere. Condividi le richieste efficaci in una base di conoscenza del team. Le richieste scadenti che hanno prodotto codice scadente sono un'occasione di apprendimento, non un segreto.
- **Limita l'IA nei contesti di apprendimento.** Quando uno sviluppatore junior impara un ambito nuovo, valuta di fargli provare prima a mano, prima di usare l'IA. La fatica è parte dell'apprendimento.

## 4 Valutare gli strumenti di IA per la tua organizzazione

Se stai decidendo quali strumenti di IA per il codice adottare, ecco un metodo di valutazione leggero. Dai un punteggio a ogni strumento su queste dimensioni prima di impegnarti:

**Tabella 2**. Sei dimensioni da valutare

| Dimensione | Domande da porsi |
|---|---|
| **Privacy dei dati** | Lo strumento si addestra sul tuo codice? Si può disattivare? Esiste un livello aziendale con isolamento dei dati? |
| **Integrazione con l'IDE** | Si adatta agli strumenti già in uso dal team senza imporre cambi di flusso di lavoro? |
| **Copertura dei linguaggi** | È forte nei linguaggi e nei framework che il tuo team usa davvero? |
| **Finestra di contesto** | Può ragionare su tutta la tua base di codice o solo sul file corrente? |
| **Verificabilità** | Puoi vedere quale contesto ha ricevuto il modello quando ha prodotto un suggerimento? |
| **Costo su larga scala** | Quanto costa per postazione con la dimensione del tuo team? Ci sono limiti di utilizzo che creano attrito? |

### 4.1 Gli strumenti principali, a inizio 2025

**GitHub Copilot** era il più diffuso, con una forte integrazione con gli IDE e un livello aziendale che garantisce di non addestrarsi sul tuo codice. **Cursor** era diventato la scelta preferita da chi vuole un contesto profondo sul codice: può leggere un intero repository e ragionare tra più file, e le sue modifiche su più file sono potenti ma vanno riviste con attenzione. **Claude** (via API o claude.ai) era particolarmente forte per la spiegazione, il refactoring e il ragionamento architetturale. **Codeium** era da considerare per i team attenti ai costi, con un livello gratuito generoso.

> **GLI STRUMENTI CAMBIANO PIÙ IN FRETTA DEI CRITERI**
>
> Questo elenco riflette l'inizio del 2025, e i prodotti di questo settore cambiano ogni pochi mesi. Le sei dimensioni qui sopra sono la parte che dura: ripeti la valutazione ogni 6-12 mesi, invece di fidarti di una classifica.

## 5 Una nota su agenti di IA e programmazione autonoma

La frontiera si sta muovendo in fretta verso gli agenti di programmazione autonomi: sistemi che non si limitano a suggerire codice ma eseguono compiti in più passaggi, leggendo la base di codice, scrivendo test, inviando pull request e correggendo gli errori. Strumenti come Devin e SWE-agent ne sono stati i primi esempi.

Per la maggior parte delle aziende non sono pronti per un uso senza supervisione su basi di codice critiche. Ma la direzione è chiara: una quota rilevante dei compiti di sviluppo di routine sarà eseguita da agenti con l'approvazione umana ai punti di controllo, non a ogni riga.

I modelli di governo che costruisci ora (standard di revisione, spiegazioni obbligatorie, controlli automatici di qualità) sono esattamente i punti di controllo che renderanno sicuro lo sviluppo con gli agenti quando maturerà. Costruirli adesso non è solo buona pratica per gli strumenti di oggi: è l'infrastruttura per la prossima ondata.

> **IN SINTESI**
>
> L'IA non ha sostituito la programmazione. Ha sostituito le parti della programmazione che non richiedevano di pensare, e ha reso più esposte e più importanti quelle che lo richiedono. Le aziende che se la caveranno meglio sono quelle che investono nelle competenze umane (lettura critica, specifiche chiare, giudizio architetturale) che l'IA non può sostituire, costruendo al tempo stesso strutture di governo che le tengano in possesso di ciò che viene costruito.

## 6 Conclusione

Il cambiamento portato dall'IA nella programmazione è reale, sostanziale e ancora in accelerazione. Ignorarlo non è una strategia praticabile. Adottarlo senza struttura nemmeno. La via di mezzo, un'adozione deliberata con proprietà chiara, controlli di qualità e responsabilità onesta, è più difficile di entrambi gli estremi, ma è l'unica che coglie i vantaggi senza accumulare il debito tecnico e organizzativo che un uso non strutturato dell'IA produce.

Se porti via una sola cosa da questo articolo: la domanda non è se il tuo team o i tuoi fornitori usano l'IA. Quasi certamente la usano. La domanda è se hai costruito i sistemi per restare in possesso di ciò che quegli strumenti producono.

**COME CITARE QUESTO ARTICOLO**

Francesco Barbato (2026). Come l'IA ha cambiato la programmazione — e una guida pratica per controllarla in azienda. localhost:3000.

Versione online: http://localhost:3000/it/blog/come-ia-ha-cambiato-programmazione-guida-pratica