CEU Universidad
San Pablo

Escuela Politécnica Superior

El futuro de la programación en la era de la IA

Constantino A. García
Director de Grado en Ingeniería de Sistemas de Información
Escuela Politécnica Superior
Universidad CEU San Pablo

Festividad de San José
Marzo de 2026

El futuro de la programación en la era de la IA

Constantino A. García
Director de Grado en Ingeniería de Sistemas de Información
Escuela Politécnica Superior
Universidad CEU San Pablo

Escuela Politécnica Superior
Universidad CEU San Pablo

El futuro de la programación en la era de la IA

Maquetación: Andrea Nieto Alonso (CEU *Ediciones*)

CEU *Ediciones*
Julián Romea 18, 28003 Madrid
www.ceuediciones.es

Depósito legal: M-8032-2026

Excelentísima y Magnífica Señora Rectora de la Universidad CEU San Pablo. Ilustrísimo Señor Director de la Escuela Politécnica Superior. Excelentísimas autoridades académicas, religiosas y civiles. Estimados compañeros, profesores, estudiantes, amigos todos:

Quiero comenzar agradeciendo a la Dirección de esta Escuela la confianza depositada al encomendarme esta lección magistral.

Introducción

Cuando el Director me lo propuso, no dudé sobre el tema. Porque en los últimos años hay una pregunta que me interesa especialmente, tanto como investigador en Inteligencia Artificial (IA) como profesor del Grado en Ingeniería de Sistemas de Información. ¿Qué va a ocurrir con la programación en la era de la IA? No como ejercicio especulativo, sino como pregunta que afecta directamente a los estudiantes que se sientan en estas aulas, a los profesores que les acompañamos, y a la forma en que concebimos nuestra disciplina.

Esta cuestión cobra aún más relevancia si observamos los avances recientes. Hoy, sistemas de IA no solo sugieren líneas de código, sino que son capaces de resolver problemas complejos en entornos reales sin intervención humana. Incluso, como ha señalado Jensen Huang, CEO de Nvidia, hay quienes afirman que «la IA puede convertir a cualquiera en programador». Pero, ¿qué significa esto realmente para quienes enseñamos a programar? ¿Y para quienes están aprendiendo a hacerlo?

Es innegable que estas herramientas representan un avance extraordinario. Sin embargo, este reconocimiento no puede ignorar un desafío fundamental: la IA está rodeada de un *hype* que, en ocasiones, nubla nuestra capacidad para distinguir entre el avance real y las expectativas infladas. Predicciones optimistas –a menudo provenientes de voces influyentes– pueden llevarnos a confundir el potencial con la realidad.

Por ello, me propuse partir de lo que la literatura científica nos permite afirmar con cierta solidez. Veremos de dónde vienen estas herramientas y cómo han evolucionado; y qué sabemos objetivamente sobre su impacto en la productividad y la calidad del código. Finalmente, y reconociendo que aquí entramos en terreno más especulativo, exploraré qué consecuencias podría tener todo esto para el perfil del programador del futuro y para nuestra manera de enseñar, porque son preguntas que, aunque aún no tengan respuestas definitivas, no podemos eludir.

Un poco de historia: de los LLMs a los agentes de código

Para situar el debate, conviene hacer un recorrido rápido por la historia de la programación mediante IA, sin pretender ser exhaustivo.

Todo parte de los modelos de lenguaje de gran escala (LLMs): sistemas entrenados sobre volúmenes masivos de texto –incluyendo enormes corpus de código fuente– que aprenden a predecir y generar secuencias de texto. La arquitectura *transformer*, propuesta en 2017, fue el salto técnico decisivo.

La primera generación de herramientas llegó solo cuatro años después, alrededor de 2021, con *GitHub Copilot* como paradigma. Llamémoslas *asistentes de completado*: el programador escribe, la herramienta sugiere la continuación. Es una autocompleción muy sofisticada –que puede sugerir funciones enteras a partir de un comentario– pero el programador conduce.

La segunda generación es cualitativamente distinta. Hablamos de *agentes autónomos de código*: sistemas que reciben una tarea en lenguaje natural y son capaces de planificar los pasos necesarios, explorar el repositorio, escribir código, ejecutarlo, interpretar los errores, corregirse y volver a intentarlo –todo ello sin intervención humana. Herramientas como *Claude Code*, *Codex*, *Devin* o *Cursor* representan este paradigma. El programador ya no conduce: supervisa, valida, orienta. El *benchmark* estándar del sector, *SWE-bench Verified* –que mide la capacidad de resolver *issues* reales de GitHub de forma autónoma– ha registrado un aumento significativo en las tasas de éxito: desde aproximadamente el 2% en 2023 hasta cerca del 80% en 2026, según datos recientes en configuraciones modernas con herramientas de orquestación. Todo en menos de tres años.

La aparición de agentes de código ha dado lugar a un fenómeno nuevo al que quiero prestar atención: el llamado *vibe coding*. El término lo acuñó Andrej Karpathy –uno de los fundadores de OpenAI– en febrero de 2025. La idea es sencilla: programar sin escribir código a mano, dejando que todo emerja de una conversación iterativa con la IA, hasta el punto de «olvidarse de que el código existe». Es una práctica que tiene sus virtudes –cualquiera puede construir un prototipo funcional– pero también sus riesgos. Volveremos a ella.

Tenemos ya el mapa de las herramientas. Pero antes de preguntarnos qué sabemos sobre su impacto, conviene recordar algo importante.

La adopción de herramientas de IA en el desarrollo de software ha dejado de ser una opción: se está convirtiendo en el estado por defecto de la industria. El informe DORA 2025 de Google, con casi 5,000 encuestados, documenta que aproximadamente el 90% de los profesionales tecnológicos utilizan IA en su trabajo.

¿Cómo afecta a estos programadores el uso de la IA?

Y aquí es donde el debate se complica.

Impacto empírico: copilots y agentes autónomos

Los estudios favorables

La evidencia más robusta a favor de los asistentes de completado proviene de experimentos controlados. El más ambicioso hasta la fecha, realizado por Microsoft y Accenture con casi 5.000 desarrolladores, confirma un incremento significativo del 26% en contribuciones de código integradas al proyecto (*pull requests*), con efectos más pronunciados en perfiles *junior*. Otros estudios muestran aceleraciones similares o superiores. Además, los beneficios parecen particularmente consistentes en algunos ámbitos concretos como la generación automática de tests, de documentación o en la exploración de *APIs* desconocidas, donde los modelos funcionan como una especie de documentación interactiva.

Los datos sobre agentes autónomos son más escasos, pero apuntan en una dirección parecida. Un estudio de Carnegie Mellon (CMU) encontró que la adopción de *Cursor* producía un aumento de la velocidad de desarrollo: las líneas de código aumentaban un 28.6%. Pero el efecto era transitorio: concentrado en los dos primeros meses tras la adopción, y disipado después. Volveremos a este estudio al hablar de calidad del código.

El estudio que nadie esperaba: METR 2025

Si uno se limitara a esta literatura, la conclusión sería clara: las herramientas de IA aceleran el desarrollo de software de forma sustancial.

Y sin embargo, en 2025 apareció un estudio que introdujo un matiz importante. Investigadores de METR –una organización independiente de evaluación de sistemas de IA– realizaron un experimento controlado con 16 desarrolladores expertos trabajando en repositorios que conocían en profundidad. Durante el experimento usaron *Cursor Pro* con *Claude Sonnet* de frontera en el momento del estudio.

El resultado fue el opuesto a lo esperado: cuando se permitía el uso de IA, los desarrolladores tardaban un 19% más que cuando no la usaban.

Este hallazgo no invalida los beneficios de la IA en otros contextos –como los propios autores subrayan–, pero demuestra que su impacto en la productividad depende profundamente del escenario: desde la experiencia del programador hasta la complejidad de la tarea o la familiaridad con el código. La aceleración, pues, no es universal ni automática.

Los autores identifican varios mecanismos plausibles que explican esta ralentización. Destaco tres. Primero, un optimismo sistemático sobre la utilidad de la IA: antes de comenzar el experimento, los desarrolladores esperaban ser un 24% más rápidos; después de completarlo seguían creyendo que habían sido un 20% más rápidos, a pesar de que la medición objetiva mostraba lo contrario: habían sido un 19% más lentos. La diferencia de percepción y realidad es de unos 40 puntos porcentuales.

Segundo, el coste de supervisión: los desarrolladores aceptaban menos del 44% de las generaciones y dedicaban tiempo considerable a revisar, limpiar o reescribir código generado.

Y finalmente, un límite más fundamental: los propios desarrolladores señalaban que los modelos no capturan el conocimiento tácito del proyecto, desde convenciones internas hasta decisiones arquitectónicas implícitas.

Calidad del código y deuda técnica

Si los hallazgos de METR nos obligan a matizar el impacto de la IA en la productividad, el estudio de CMU sobre *Cursor* reveló que, incluso cuando la velocidad aumenta, hay un coste oculto que persiste en el tiempo: la calidad del código.

El mismo estudio que documentó el incremento del 28.6% en líneas de código añadidas reveló que los repositorios acumularon un 30% más de avisos de análisis estático, un 41% más de complejidad de código y un 7% más de duplicación de líneas –y estos efectos, a diferencia de la ganancia de velocidad, eran persistentes. Los modelos de panel del estudio confirman que esa deuda técnica

acumulada reduce la velocidad de desarrollo en el futuro, creando un ciclo que se retroalimenta.

La empresa de análisis GitClear analizó millones de líneas de código de repositorios de Google, Microsoft y Meta entre 2020 y 2024. El código *churn* –escrito y revertido en menos de dos semanas, indicador clásico de deuda técnica– creció un 26% solo entre 2023 y 2024, y casi se duplicó respecto al nivel pre-IA de 2020. Los bloques de código duplicados de cinco o más líneas se multiplicaron por ocho solo en 2024. Y por primera vez en toda la serie histórica, las líneas copiadas y pegadas superaron a las líneas movidas –el principal indicador de actividad de refactorización y reorganización de código en el repositorio.

El informe DORA 2025 de Google apunta en la misma dirección: la adopción de IA mejora la velocidad de entrega, pero se asocia con mayor inestabilidad de los sistemas.

Seguridad: más código vulnerable, más confianza

El impacto en seguridad es el área de mayor consenso negativo en la literatura. En un estudio publicado en el *IEEE Symposium on Security and Privacy*, un equipo de investigadores evaluó el código generado por Copilot en escenarios con implicaciones de seguridad. Los resultados indicaron que, sin revisión humana posterior, aproximadamente el 40% del código contenía vulnerabilidades. Estudios posteriores con distintos modelos y dominios confirman sistemáticamente la misma tendencia.

Pero el hallazgo más importante no es el porcentaje en sí, sino su combinación con otro resultado que ya hemos visto. Un equipo de Stanford encontró que los participantes con acceso a un asistente de IA produjeron código significativamente menos seguro y expresaron más confianza en la seguridad de su código que los que no lo usaron. La herramienta no solo generaba vulnerabilidades: generaba una falsa sensación de seguridad que eliminaba el escrutinio crítico.

Síntesis: qué nos dice realmente la evidencia

El cuadro que emerge de toda esta evidencia es matizado, pero no menos interesante. Hay un progreso espectacular en *benchmarks* con beneficios de productividad reales y tangibles, aunque condicionales al contexto y probablemente menores de lo que se anuncia. Pero la evidencia también apunta a que la IA

genera deuda técnica y código menos seguro –y todavía no tenemos claro cuál será el coste de esto a largo plazo. La diferencia entre quien aprovecha estas herramientas y quien queda expuesto a sus riesgos depende, en gran medida, de la capacidad de supervisión del programador que las usa.

De todo ello se deriva también, con bastante claridad, un cambio de rol. El trabajo se desplaza de escribir código a evaluarlo, orquestarlo y refinarlo. Lo que importa ahora no es la velocidad de escritura, sino la capacidad de garantizar valor a través de la seguridad, la calidad y la confianza. El valor se traslada de la producción bruta a la supervisión inteligente.

Aceptemos entonces que el rol cambia. La pregunta inmediata es: ¿hacia dónde? ¿Qué competencias ganan valor cuando escribir código deja de ser lo central?

El futuro: programadores, ingenieros y educación

Las competencias que perduran

El consenso emergente en la literatura apunta a tres familias de competencias.

1. **Pensamiento computacional.** El pensamiento computacional –descomposición de problemas, abstracción, reconocimiento de patrones, diseño de algoritmos– no solo sigue siendo relevante en la era de la IA generativa, sino que se vuelve más necesario: es precisamente la capacidad que determina si un desarrollador puede evaluar críticamente el código que la IA produce. Un estudiante que entiende por qué un algoritmo es $O(n^2)$ y cuándo eso importa puede juzgar si la solución generada es adecuada para el problema en cuestión. Uno que solo sabe pedirle a la IA que «escriba una función que ordene esta lista» no tiene esa capacidad.
2. **Arquitectura y diseño de sistemas.** La creación de arquitecturas –decisiones sobre estructura de sistemas, interfaces, escalabilidad, mantenibilidad– se está convirtiendo la competencia diferencial del ingeniero. Pero la arquitectura no empieza en el diseño técnico: empieza en la comprensión de los requisitos, en la capacidad de navegar la ambigüedad de lo que un sistema debe hacer antes de decidir cómo hacerlo. Los agentes de IA pueden generar componentes; no pueden negociar con un cliente qué significa «que funcione bien», ni decidir de manera fiable cómo deben articularse esos componentes en un sistema que debe vivir, evolucionar y mantenerse durante años.

3. ***Code sense*: leer, evaluar y depurar**. Asentada sobre el pensamiento computacional, existe una competencia más aplicada: la capacidad de leer código que no has escrito tú –entender qué hace, por qué podría fallar, qué asume implícitamente. En la literatura de ingeniería de *software* esta habilidad se denomina *code sense*: una intuición entrenada sobre la calidad, los riesgos y las implicaciones del código. En un entorno donde gran parte del código lo genera una IA, esta es quizá la competencia central del programador del futuro: no producir código, sino juzgarlo.

Estas competencias son el ideal. La realidad, a veces, es otra.

El problema del *vibe coding* y la brecha de comprensión

Volvamos un momento al *vibe coding*. Cuando Karpathy acuñó el término, lo hizo con cierta admiración por la democratización que representa: cualquiera puede construir un prototipo funcional, sin necesidad de formación técnica. Y en este sentido, el fenómeno merece reconocimiento genuino. Equipos enteros de no-programadores pueden ahora construir sus propias herramientas sin depender de un departamento de ingeniería. Eso es una ampliación real del acceso a la tecnología.

Pero sus críticos señalaron rápidamente la otra cara. Y es que llevarse el *vibe coding* a desarrollos profesionales es probablemente una mala idea. *Fast Company* documentaba la «resaca del *vibe coding*»: desarrolladores describiendo «infiernos de desarrollo» al intentar mantener, escalar o depurar sistemas construidos sin comprensión. La práctica que funciona para un prototipo se convierte en una trampa en cuanto ese prototipo llega a producción.

Y aquí aparece una brecha. La distancia entre quienes tienen las competencias para evaluar lo que la IA produce y quienes no las tienen se está convirtiendo en la nueva fractura de la industria del software: entre los *vibe coders* puros –quienes usan la IA como una caja negra sin entender su *output*– y los programadores que, aún usando la IA, se sienten cómodos editando, corrigiendo y juzgando lo que reciben.

Simon Willison, un influyente desarrollador, lo resume con precisión: «Si un agente escribió cada línea de tu código, pero lo has revisado, testeado y comprendido, eso no es *vibe coding*: es usar un agente como asistente de escritura». En el desarrollo profesional, la línea divisoria no pasa por quién genera el código, sino por quién lo entiende –y, por tanto, por quién aprovecha la IA y quién sufre sus limitaciones.

Implicaciones para la educación

Si este diagnóstico es correcto –si la brecha entre quienes entienden la IA y quienes solo la consumen es real y creciente–, entonces la pregunta para nosotros, como escuela de ingeniería, es cómo evitar que nuestros estudiantes queden del lado equivocado de esa brecha. El debate ya no está en la adopción, que es inevitable, sino en la forma en que garantizamos que la IA sirva para potenciar –y no para erosionar– las competencias que definen a un ingeniero.

La literatura educativa apunta en una dirección coherente. El modelo que mejor funciona es uno de andamiaje progresivo: primero construir comprensión genuina sin asistencia, luego introducir la IA gradualmente con reflexión crítica explícita, y solo entonces permitir un uso más autónomo de las herramientas. Los estudiantes que acceden a la IA sin ese andamiaje previo no aprenden a programar: aprenden a gestionar *outputs* que no comprenden. Los que la usan para *aprender* –pidiendo explicaciones, generando variantes, entendiendo por qué funciona algo– obtienen resultados sistemáticamente mejores que los que la usan simplemente para *producir*.

Esto tiene consecuencias directas para los programas de ingeniería. Los cursos introductorios deben preservar el trabajo sin asistencia como base para desarrollar comprensión genuina. Los cursos avanzados deben desarrollar explícitamente la capacidad de leer, evaluar y depurar código generado por IA. Y los programas en su conjunto deben reforzar las competencias mencionadas que la evidencia identifica como no delegables.

La paradoja pedagógica final es esta: para formar ingenieros capaces de usar bien la IA, hay que formarlos primero sin ella. El estudiante que entiende por qué falla un algoritmo, cómo se diseña un sistema que debe escalar, y qué significa que un código sea correcto –ese estudiante puede confiarle trabajo a un agente. El que no entiende nada de eso no le está confiando trabajo a la IA: le está confiando su ignorancia.

Conclusiones

Empecé esta lección con una pregunta concreta: ¿qué va a ocurrir con la programación? La respuesta más honesta es que la programación ya ha cambiado y, sin duda, seguirá cambiando profundamente.

Y sin embargo, incluso si imaginamos un escenario muy optimista –en el que los sistemas de IA resuelven prácticamente todos los problemas de *benchmarks* como *SWE-bench Verified*– creo firmemente que los programadores seguirán siendo esenciales. O al menos, seguirán siendo esenciales quienes realicen tareas más allá de escribir código. Serán esenciales quienes diseñen sistemas que deben vivir, escalar y mantenerse durante años. Esenciales quienes naveguen la incertidumbre de los requisitos, la seguridad y el cumplimiento normativo en contextos que ningún *benchmark* contempla. Esenciales quienes asuman la responsabilidad –técnica, legal, humana– de convertir una idea en un producto que funciona en el mundo real.

El programador no va a desaparecer; va a cambiar. Y formar a ese programador es exactamente la responsabilidad que las escuelas de ingeniería no podemos delegar. El pensamiento computacional, la capacidad arquitectónica y de diseño, el *code sense*: competencias que aunque llevamos años cultivando en estas aulas, debemos potenciar y exigir con más convicción que nunca.

Termino reiterando mi admiración por los agentes de código. En el tiempo que hemos tardado en estar aquí reunidos, sistemas como los que hemos descrito habrán resuelto miles de *bugs* reales, escrito decenas de miles de líneas de código funcional, y ayudado a programadores de todo el mundo a hacer cosas que solos no habrían podido.

Pero una herramienta notable no es una herramienta sin riesgos. Y la diferencia entre usarla con criterio y sin él no la produce la tecnología: la produce la formación.

Y esa es, precisamente, la razón por la que seguimos aquí.

Muchas gracias.

Constantino A. García

Es Ingeniero de Telecomunicación (Premio Extraordinario y Tercer Premio Nacional) e Ingeniero en Informática (Premio Extraordinario), ambos títulos por la Universidad CEU San Pablo; Licenciado en Ciencias Matemáticas (especialidad en Estadística) y Licenciado en Ciencias Físicas (especialidad en Física General) por la Universidad Nacional de Educación a Distancia (UNED); y Doctor en Tecnologías de la Información por la Universidad de Santiago de Compostela (USC) (*Cum Laude*). Gran parte de su tesis doctoral se desarrolló gracias a una beca predoctoral de la Xunta de Galicia y una beca FPU del Ministerio de Educación, Cultura y Deporte.

Actualmente, desarrolla actividades docentes e investigadoras en la Universidad CEU San Pablo, donde se incorporó en el curso 2017-2018. Ha impartido docencia en los grados de Ingeniería de Sistemas de Información e Ingeniería Biomédica, en el máster de Ingeniería Biomédica y en cursos de doctorado. Desde septiembre de 2022, dirige el grado en Ingeniería de Sistemas de Información, y desde el curso 24/25, es profesor titular en el área de Ciencias de la Computación e Inteligencia Artificial.

Sus líneas de investigación principales se centran en la aplicación de modelos bayesianos y técnicas de aprendizaje automático para abordar problemas complejos, especialmente en los ámbitos de la electrocardiografía, el análisis de la variabilidad de la frecuencia cardíaca y la metabolómica. Cuenta con más de 30 publicaciones científicas en revistas y congresos internacionales, es autor de un libro científico con dos ediciones, y posee dos sexenios de investigación reconocidos. Además, ha participado en cinco proyectos financiados a nivel nacional o autonómico, ha colaborado en un contrato de transferencia con empresa, y es cotitular de tres registros de *software*.

Escuela Politécnica Superior
Universidad CEU San Pablo
Campus de Montepríncipe
28925 Alcorcón (Madrid)
Teléfono: 91 372 40 27, fax: 91 372 64 48
epssec@ceu.es, www.ceu.es/usp