Pair Programming con Inteligencia Artificial

Juan Diego Arias Durán
Sebastián Andrés Rodríguez
Pontificia Universidad Javeriana, Bogotá, Colombia

Resumen

Resumen— La práctica de *programación en pareja* (pair programming) ha sido tradicionalmente una técnica de desarrollo ágil donde dos desarrolladores trabajan de forma colaborativa. Con los recientes avances en *inteligencia artificial* (IA) generativa, ha surgido el concepto de *pair programming con IA*, en el cual un asistente de código colabora con un desarrollador humano. Este artículo ofrece una visión integral de esta práctica: desde su definición y evolución histórica con herramientas como GitHub Copilot, Cursor y Claude Code, hasta un análisis detallado de sus ventajas y desafíos. Adicionalmente, se examina la creciente demanda en el mercado laboral de habilidades relacionadas con la orquestación de IA, presentando datos sobre la valoración de estos perfiles. Se discute la rápida obsolescencia de los modelos, comparando el rendimiento de las herramientas líderes a fecha de noviembre de 2025, y se explora el fenómeno emergente del ”Vibe Coding”, analizando sus implicaciones para el futuro de la ingeniería de software.

Index Terms

Programación en parejas; Inteligencia Artificial; Colaboración Humano-IA; Desarrollo Ágil; Asistentes de código

I. Introducción

La incorporación de asistentes de programación basados en inteligencia artificial se ha vuelto una tendencia dominante en el desarrollo de software moderno. Para el año 2025, aproximadamente el 84 % de los desarrolladores emplean (o planean emplear) herramientas de IA en sus flujos de trabajo, integrándolas como parte habitual del proceso de codificación [1]. Esta adopción masiva indica que el *pair programming* con IA —es decir, la colaboración activa entre un programador humano y un “compañero” de IA— ha pasado de ser una curiosidad experimental a convertirse en una práctica extendida en la industria del software.

En esencia, la idea de un “programador en pareja” impulsado por IA se popularizó con la introducción de GitHub Copilot en 2021, anunciado explícitamente como *“tu programador en pareja de IA”* [2]. Herramientas como Copilot, basadas en modelos de lenguaje de gran escala, pueden integrarse directamente en el entorno de desarrollo (IDE) del programador y ofrecer sugerencias de código, completar funciones enteras y ayudar a resolver problemas en tiempo real [2]. Del mismo modo, otros asistentes (p. ej., Amazon CodeWhisperer, Tabnine, OpenAI ChatGPT) se han incorporado rápidamente al repertorio del desarrollador, ofreciendo distintas formas de colaboración inteligente.

Este artículo explora a profundidad la práctica de **programación en parejas con IA**. En las Secciones II y III se define el concepto y sus características. La Sección IV repasa la evolución histórica, incluyendo la llegada de editores inteligentes. Las Secciones V y VI discuten ventajas y desventajas. La Sección VII presenta casos de uso, y la Sección VIII ejemplos reales. La Sección IX analiza la demanda del mercado laboral para estas habilidades mediante una matriz de perfiles. La Sección X aborda la velocidad de avance de la IA generativa y compara el rendimiento de los modelos actuales. La Sección XI introduce el concepto de *vibe coding* y discute sus retos. La Sección XII presenta matrices de análisis que relacionan el pair programming con IA con principios de diseño, atributos de calidad, tácticas, patrones y perfiles de mercado. La Sección XIII distingue entre modelos fundacionales, herramientas de desarrollo asistidas por IA y modos de interacción (chat, completado, agentes, etc.). La Sección XIV realiza un análisis comparativo de costos entre el pair programming con IA y el vibe coding, considerando consumo de tokens, tiempo humano y deuda técnica. Finalmente, la Sección XV presenta las conclusiones del trabajo.

II. Definición de Pair Programming con IA

La **programación en pareja** (*pair programming*) se refiere a la técnica en la cual dos desarrolladores trabajan juntos en un mismo equipo, uno escribiendo código (*"conductor"*) y el otro revisando constantemente y orientando a nivel estratégico (*"navegante"*) [3]. Este método, formalizado dentro de las prácticas de la metodología ágil Extreme Programming a fines de los años 90, busca mejorar la calidad del software a través de la colaboración: el código es revisado y discutido en el momento mismo de su escritura, lo que redunda en menos errores y mejores soluciones de diseño [4]. Estudios clásicos reportaron que el desarrollo en parejas, aunque pudiese parecer menos "productivo" en términos de horas-hombre, produce código más robusto y un intercambio de conocimiento continuo entre los participantes [4]. De hecho, Kent Beck —creador de Extreme Programming— sostenía que *"los mejores programas y diseños se logran en parejas, porque ambos miembros pueden criticarse y encontrar errores mutuamente, aprovechando las mejores ideas de cada uno"* [3].

En el contexto actual, el concepto evoluciona hacia el **pair programming asistido por IA**. En vez de contar con dos humanos, uno de los miembros de la pareja es un agente de inteligencia artificial capaz de entender instrucciones en lenguaje natural y generar código en diversos lenguajes de programación [2]. Importante aclarar que la IA no reemplaza al desarrollador humano, sino que actúa como colaborador que *complementa* su trabajo [5]. Durante una sesión de programación en pareja con IA, el flujo típico involucra al humano escribiendo parte del código o describiendo la intención (por ejemplo, mediante comentarios o prompts), y la IA respondiendo con sugerencias de implementación, detección de posibles errores o mejoras en el código. Esta interacción iterativa configura una verdadera co-creación: la IA propone y el humano decide aceptar, editar o rechazar las propuestas, manteniendo siempre el control final sobre el código fuente [5].

En otras palabras, la programación en pareja con IA consiste en hacer partícipe a un sistema inteligente en el proceso de codificación de forma activa. El asistente automatizado *entiende el contexto* del código en desarrollo y genera aportes alineados a ese contexto [2], muy similar a como lo haría un colega humano que conoce el proyecto. Plataformas contemporáneas como GitHub Copilot describen su servicio precisamente como un "copiloto" de desarrollo que *"se integra en tu editor, tomando el contexto del código en el que trabajas y sugiriendo líneas completas o funciones enteras"* [2].

Cabe resaltar que este paradigma forma parte de una tendencia mayor denominada **IA colaborativa**, donde las herramientas de IA no son meros oráculos que dan respuestas aisladas, sino compañeros de trabajo que *participan activamente* en tareas creativas junto con el usuario [6]. En programación, esto se manifiesta mediante la asistencia continua en tiempo real: la IA no solo ejecuta órdenes, sino que también *propone, responde, sugiere, aprende y se adapta* durante el desarrollo [6]. Por ello, a esta modalidad a veces se le llama también *"programación asistida por IA"* o *"programación con copiloto"*, haciendo énfasis en la naturaleza cooperativa entre el desarrollador y el modelo de IA.

III. Características del enfoque

El pair programming con IA posee una serie de características distintivas que lo diferencian tanto de la programación individual tradicional como de la programación en parejas humana convencional. A continuación se resumen sus rasgos principales:

Sugerencias de código contextuales en tiempo real: El asistente de IA analiza el código fuente mientras se escribe y ofrece *sugerencias inmediatas* que completan líneas o bloques enteros de código, teniendo en cuenta el contexto actual del programa [2]. Por ejemplo, a medida que el desarrollador teclea el encabezado de una función o escribe un comentario describiendo cierta lógica, la IA puede proponer automáticamente la implementación más probable de esa función. Estas sugerencias suelen adaptarse al estilo y convenciones del proyecto, pues el modelo ha sido entrenado con enormes conjuntos de código fuente público [7].

Integración fluida en el IDE y rapidez de interacción: Las herramientas de pair programming con IA están diseñadas para integrarse sin fricciones en el entorno de trabajo del desarrollador [6]. Operan como extensiones o asistentes dentro de editores populares (Visual Studio Code, IntelliJ, etc.), mostrando las completaciones de forma semi-transparente en pantalla, casi como si otro desarrollador estuviese escribiendo en la misma ventana. Esto permite una interacción muy ágil: la IA responde en cuestión de milisegundos a las entradas del usuario, fomentando un ciclo de prueba y error sumamente rápido [6].

Adaptación al estilo del desarrollador y del código existente: Gracias a los modelos de lenguaje de gran escala y al análisis del contexto, la IA puede adaptar sus sugerencias para que coincidan con el *estilo de codificación existente* en el proyecto [6]. Por ejemplo, respetará convenciones de nomenclatura de variables, formato y estructuras que vea en el repositorio actual, logrando que el código sugerido se integre de forma natural con el escrito por humanos. Incluso, herramientas como Intellicode de Microsoft permitían entrenar el completador con el repositorio de la empresa para alinear las sugerencias a los estándares internos [8]. De igual modo, la IA "aprende" de las elecciones del usuario: si ciertas sugerencias son constantemente aceptadas o rechazadas, con el tiempo puede ajustar sus propuestas (hasta donde su modelo se lo permita).

Colaboración iterativa y feedback inmediato: En este enfoque, el flujo de trabajo consiste en un diálogo constante: el desarrollador escribe o edita código, la IA reacciona con una propuesta, el desarrollador la evalúa y posiblemente la incorpora (quizá con modificaciones), tras lo cual la IA vuelve a observar el nuevo estado del código y ofrece la siguiente sugerencia, y así sucesivamente [5]. Esta iteración continua significa que el programador recibe *feedback inmediato* sobre lo que va escribiendo. Por ejemplo, la IA puede señalar al instante un posible error lógico u omisión, o sugerir optimizaciones al código recién introducido. El resultado es un ciclo de desarrollo más dinámico y "conversacional" que el típico ciclo escribir-compilar-probar.

Soporte en múltiples tareas del ciclo de desarrollo: Aunque la principal función de estos asistentes es generar código, en la práctica moderna abarcan mucho más. Pueden explicar segmentos de código en lenguaje natural (útil para comprender bases de código heredado), sugerir casos de prueba o generar automáticamente pruebas unitarias, escribir comentarios o documentación básica para funciones, e incluso ayudar en tareas de *debugging* identificando la causa de ciertos errores [5] [5]. Por ejemplo, Copilot puede proponer casos de prueba para una función recién escrita, ahorrando tiempo en la creación de suites de testing. Chatbots como ChatGPT pueden revisar un bloque de código buscando vulnerabilidades de seguridad o puntos de mejora, actuando como un revisor de código automatizado. Todo esto amplía el alcance de la colaboración humano-IA más allá de únicamente "escribir líneas de código".

Requerimiento de indicaciones claras (*prompt engineering*): Para aprovechar al máximo un compañero de IA, el desarrollador debe aprender a comunicarse eficazmente con él. Esto implica desde escribir comentarios descriptivos de alta calidad que guíen al modelo (// Ordenar la lista de usuarios por edad de forma ascendente") hasta posiblemente ajustar configuraciones del asistente o proveer ejemplos del resultado esperado. A esta habilidad emergente se le conoce como *ingeniería de prompts*, y se ha vuelto parte integral de la colaboración con IA [5]. Un prompt bien formulado puede significar que la IA sugiera una solución casi completa y correcta, mientras que una instrucción ambigua puede resultar en código irrelevante. Por lo tanto, el desarrollador ahora asume también el rol de "entrenador en tiempo real" de la IA, afinando las instrucciones para guiarla hacia la solución deseada.

En conjunto, estas características hacen del pair programming con IA una experiencia diferente a la programación tradicional: más interactiva, rápida y con un flujo de ideas bidireccional entre humano y máquina. Sin embargo, también requieren que el programador desarrolle nuevas competencias (comunicación con la IA, verificación crítica de sus sugerencias, etc.) para utilizar la herramienta eficazmente, como discutiremos más adelante.

IV. Historia y evolución de la práctica

La práctica de programación en parejas tiene sus raíces bien establecidas en la industria de software desde finales del siglo XX. Si bien la idea de dos personas programando juntas surgió de manera informal en los primeros equipos de programación (por ejemplo, las programadoras pioneras del ENIAC ya colaboraban de esta forma en la década de 1940 [3]), fue con la metodología *Extreme Programming* (XP) de Kent Beck que el término "programación en pareja" cobró notoriedad y adopción amplia en 1999 [3]. XP promovió esta técnica como parte de sus prácticas centrales, enfatizando la mejora en calidad, la difusión del conocimiento y la simplicidad resultante de tener dos mentes resolviendo el mismo problema simultáneamente [4]. Durante las dos décadas siguientes, numerosas organizaciones implementaron pair programming humano en mayor o menor medida, reportando a menudo resultados positivos en términos de reducción de defectos e incremento en la comprensión colectiva del código, aunque también señalando retos como el aumento en esfuerzo humano y la necesidad de compatibilidad de personalidades entre los pares.

Paralelamente, desde hace muchos años existen herramientas de asistencia al programador, aunque inicialmente eran más básicas que "inteligentes". Por ejemplo, los entornos de desarrollo integrados (IDE) desde los 90 ofrecían **autocompletado de código** (como el clásico IntelliSense de Microsoft Visual Studio) que sugería nombres de variables o métodos según las letras que uno iba tipeando, basándose en reglas sintácticas o en el contexto estático del lenguaje. Estos primeros asistentes mejoraban la velocidad de tipeo y evitaban errores tipográficos, pero eran sistemas determinísticos, sin aprendizaje automático.

El verdadero punto de inflexión hacia asistentes de código "inteligentes" llegó a fines de la década de 2010. En 2018, surgió un producto llamado *TabNine* que fue pionero en aplicar modelos de aprendizaje profundo (específicamente basados en GPT-2) para completar código fuente de forma inteligente [8]. A diferencia de IntelliSense, TabNine no se limitaba a completar la siguiente palabra clave, sino que era capaz de generar varias líneas de código en distintos lenguajes, habiendo sido entrenado con millones de archivos de código de código abierto. Esto supuso el primer asistente de código impulsado por **redes neuronales** en entornos de desarrollo, demostrando que la IA podía aprender patrones de programación y aplicarlos a contextos nuevos de manera generalizada.

Poco después, en 2020-2021, OpenAI entrenó un modelo aún más potente, **OpenAI Codex**, derivado de GPT-3 pero especializado en generación de código fuente. Codex demostró una capacidad sorprendente para no solo completar líneas sino también sintetizar funciones completas a partir de descripciones en lenguaje natural [8]. GitHub, en colaboración con OpenAI, integró este modelo en un producto llamado **GitHub Copilot**, lanzado inicialmente como vista previa en junio de 2021 [2]. Copilot marcó un antes y un después, ya que se integró directamente en editores populares como VS Code, permitiendo a cualquier desarrollador tener un "par" de IA constantemente activo mientras codificaba. En su anuncio oficial se le denominó explícitamente "*AI pair programmer*" [2] y se destacaba que podía sugerir desde pequeñas líneas hasta bloques enteros de solución, adaptándose al código del usuario para ayudarle a completar su trabajo más rápido [2].

La evolución desde entonces ha sido vertiginosa. Para 2022, Copilot pasó a disponibilidad general y acumuló cientos de miles de usuarios, inspirando la aparición de múltiples competidores y herramientas complementarias. Amazon lanzó **CodeWhisperer** en 2022 como su servicio de sugerencia de código basado en IA, entrenado con código fuente diverso incluyendo repositorios públicos de Amazon. Startups como **TabNine** y **Codeium** mejoraron sus motores de completado e introdujeron innovaciones como la técnica *"Fill-in-the-Middle"* (rellenar huecos en el código aprovechando contexto bidireccional) para generar código con mayor coherencia en el medio de una función, no solo continuaciones lineales [8] [8].

A finales de 2022, irrumpió **ChatGPT** de OpenAI, una interfaz conversacional capaz de entender instrucciones complejas y generar respuestas detalladas, incluyendo código. Aunque ChatGPT no se integraba directamente en el IDE, muchos desarrolladores comenzaron a usarlo en paralelo para resolver dudas de programación. En respuesta,

las IDE modernas han incorporado chats integrados con modelos de IA. Sin embargo, la industria dio un paso más allá con la aparición de editores *"AI-native"* como **Cursor**. A diferencia de los plugins tradicionales, Cursor integra la IA en el núcleo del editor, permitiendo la indexación semántica de todo el código base y habilitando ediciones predictivas en múltiples líneas (Copilot++). Paralelamente, la familia de modelos **Claude** de Anthropic y herramientas como **Claude Code** han ganado prominencia por su capacidad de razonamiento superior y manejo de ventanas de contexto extensas, permitiendo a los desarrolladores abordar refactorizaciones complejas y análisis de arquitectura que superan las capacidades de los autocompletadores convencionales.

Hoy en día (2025), el ecosistema de pair programming con IA es rico y variado. Hay asistentes especializados en distintas etapas: algunos se enfocan en generar código a partir de diseños, otros en ayudar con la documentación, otros en refactorizar automáticamente código legado. Empresas grandes y pequeñas han adoptado estas herramientas. Se reporta que más del 50 % de los desarrolladores profesionales ya usan asistentes de IA en su trabajo diario [1], y casi todos al menos los han probado alguna vez [1]. En términos de impacto, estudios controlados han encontrado reducciones significativas en el tiempo de desarrollo: por ejemplo, un experimento de GitHub mostró que con Copilot los programadores completaban tareas un 55 % más rápido en promedio que un grupo de control sin Copilot [1]. De forma similar, Amazon ha informado aceleraciones del orden del 57 % en la finalización de tareas con su CodeWhisperer [1].

Otra tendencia reciente es la aparición de agentes de IA más autónomos que pueden encargarse de tareas completas. Por ejemplo, GitHub ha introducido modos "CLI" y de agente en Copilot (Copilot X) capaces de crear proyectos enteros a partir de una especificación básica o realizar múltiples pasos (crear un archivo, luego escribir su contenido, luego configurar algo) sin intervención humana directa [9]. Si bien aún están en etapas iniciales, vislumbran un futuro donde el rol de la IA en el desarrollo será aún más proactivo.

En suma, la práctica de programación en pareja ha pasado de ser un proceso exclusivamente humano a un modelo híbrido humano-máquina en un lapso relativamente corto. La historia reciente muestra una rápida coevolución: la IA haciéndose más capaz y ubicua en las herramientas de desarrollo, y los desarrolladores adaptando sus métodos de trabajo para aprovechar esta nueva clase de compañero digital. En la siguiente sección evaluaremos qué beneficios y riesgos concretos se han identificado al trabajar con este tipo de asistentes inteligentes.

V. Ventajas del pair programming con IA

Numerosos reportes técnicos, encuestas industriales y estudios iniciales han documentado las ventajas que ofrece la colaboración con una IA a la hora de programar. Entre las principales **ventajas** podemos destacar:

Incremento de velocidad y productividad: El beneficio más inmediato es la aceleración en la escritura de código. La IA puede generar automáticamente porciones considerables de código que de otro modo el desarrollador tendría que escribir manualmente, ahorrando tiempo especialmente en tareas repetitivas o boilerplate [7] [1]. Por ejemplo, un caso de estudio mostró que desarrolladores usando GitHub Copilot completaron nuevas funcionalidades un 34 % más rápido, y la escritura de pruebas unitarias hasta un 38 % más rápido, comparado con no usar la herramienta [10]. En general, más del 95 % de los programadores que han utilizado estos asistentes afirman que les ayudan a acelerar su trabajo cotidiano [10]. Este aumento de velocidad se traduce en ciclos de desarrollo más cortos y mayor capacidad de entrega de funcionalidades.

Reducción de errores y mejora en calidad de código: Actuando como un revisor constante, la IA puede ayudar a producir código más limpio y con menos defectos desde el inicio. Modelos entrenados en millones de ejemplos tienden a sugerir patrones considerados buenas prácticas (p. ej., estructuras eficientes, manejo adecuado de excepciones, corrección en sintaxis), lo que eleva la calidad del código escrito [7]. Además, al tener una visión "fresca" y amplia del contexto, la IA puede detectar descuidos que el programador humano pasa por alto, señalando errores lógicos potenciales o casos no cubiertos. En términos de calidad percibida, un estudio de GitHub reportó mejoras

en 8 dimensiones (legibilidad, mantenibilidad, ausencia de errores comunes, etc.) cuando los desarrolladores usaban Copilot, en comparación con código escrito sin esa asistencia [11]. Asimismo, la generación automática de pruebas unitarias por parte de la IA (o sugerir escenarios para las mismas) contribuye a identificar fallas antes de que el código llegue a producción, actuando como un mecanismo adicional de control de calidad.

Aprendizaje acelerado y transferencia de conocimiento: Un asistente de IA bien puede fungir como un mentor virtual. Para desarrolladores menos experimentados, ver las sugerencias que propone la IA —muchas veces incorporando patrones consolidados de la industria— es una forma directa de aprender nuevas técnicas y mejores prácticas en tiempo real [7]. Por ejemplo, un programador junior puede aprender cómo estructurar una consulta SQL segura o cómo implementar un algoritmo conocido observando la solución que propone la IA a partir de la descripción del problema. Incluso programadores seniors pueden descubrir atajos del lenguaje o bibliotecas que no conocían, gracias a la vastedad de conocimiento encapsulado en el modelo [7]. Varios testimonios resaltan que trabajar con IA les hizo "pensar diferente" y cuestionar supuestos, refinando sus habilidades de diseño [12]. En suma, la IA actúa como un repositorio viviente de conocimientos de programación que se pone a disposición durante el flujo de trabajo, facilitando una especie de capacitación continua *on-the-job*.

Enfoque en tareas de mayor nivel y alivio del trabajo tedioso: Delegar a la IA la escritura de código repetitivo o "mecánico" permite que el desarrollador humano concentre más su tiempo y energía en aspectos de alto nivel, creativos o críticos del sistema [7]. Por ejemplo, generar estructuras de datos boilerplate, código estándar de acceso a base de datos, bucles rutinarios, o documentación básica son tareas que el asistente puede resolver rápidamente. Así, el programador puede dedicar más esfuerzo a afinar la arquitectura general, las reglas de negocio o resolver problemas complejos que *sí* requieren juicio humano y creatividad. Esta redistribución del esfuerzo no solo aumenta la eficiencia, sino que también mejora la satisfacción del desarrollador, al liberarlo en parte de "picar código" monótono y permitirle enfocarse en desafíos más interesantes. Un desarrollador declaró que gracias a la IA ahora pasa menos tiempo frustrado con tareas repetitivas y más en la parte creativa del desarrollo [10].

Disponibilidad permanente y colaboración asíncrona: A diferencia de un colega humano, la IA está disponible las 24 horas, todos los días. Esto ofrece flexibilidad: un desarrollador puede "parearse" con la IA en cualquier momento que lo necesite, incluso fuera del horario típico o cuando otros miembros del equipo no están disponibles. Esto resulta especialmente útil en equipos distribuidos o cuando se trabaja en solitario. La IA tampoco se cansa ni pierde concentración, por lo que puede ser consultada tantas veces como se requiera sin desgaste (aunque el programador sí deba cuidarse de la fatiga). En escenarios de emergencia o trabajo contrarreloj, contar con un ayudante incansable puede marcar la diferencia para cumplir una fecha de entrega apretada [9] [9]. Un ejemplo real se dio en el Banco Galicia (Argentina) durante un fin de semana crítico: su equipo logró implementar contrarreloj una nueva funcionalidad regulatoria en la aplicación bancaria apoyándose fuertemente en Copilot, algo que consideraron hubiera sido "muy improbable" lograr a tiempo sin la asistencia de la IA [9] [9].

Potencial ahorro de costos a largo plazo: Si bien las herramientas de IA pueden implicar un costo de suscripción o infraestructura, muchas organizaciones vislumbran un retorno de inversión positivo gracias a los ahorros de tiempo y la reducción de errores que brindan. McKinsey estimó en 2024 que la adopción efectiva de asistentes de IA en desarrollo de software podría aumentar la productividad hasta en un 55 % y reducir costos en torno al 27 % [5]. Menos horas de programación para una misma entrega significa liberar capacidad del equipo para otros proyectos. Asimismo, código de mayor calidad implica menos fallos en producción y menos retrabajo, lo que disminuye costos asociados a mantenimiento y soporte. Si bien el impacto exacto varía según el contexto, empresas como la fintech Naranja X reportan ahorros de 2–3 horas diarias por desarrollador gracias al uso de Copilot, con una reducción del 50 % en el tiempo de resolución de ciertos problemas [9]. Esto claramente se traduce en optimización de recursos y podría justificar la inversión en estas tecnologías.

Colaboración y consistencia en equipos grandes: En equipos con muchos desarrolladores, un asistente de IA puede actuar como un agente homogeneizador, sugiriendo soluciones consistentes y estilo unificado, mitigando en parte

las divergencias de enfoque entre distintos miembros. Por ejemplo, si existe una forma óptima de implementar cierta función utilitaria, la IA tenderá a proponer esa misma forma a todos los miembros del equipo cuando se enfrenten a ese problema, lo que promueve consistencia en el código base. Además, puede funcionar como "miembro extra" que ayuda a un desarrollador novel a integrarse más rápido: en lugar de interrumpir a un colega para cada duda, puede consultarle primero a la IA cómo hacer algo del código base, obteniendo guía inmediata. Líderes técnicos señalan que estas herramientas les permiten a sus equipos lograr objetivos estratégicos más rápido y con mayor confiabilidad, mejorando a la vez la experiencia de los desarrolladores al darles un apoyo constante [9] [9].

En resumen, las ventajas del pair programming con IA abarcan mejoras tangibles en productividad, calidad y aprendizaje, así como beneficios operativos para las organizaciones. No es de extrañar que muchos lo consideren un cambio de paradigma en la forma de desarrollar software. No obstante, como toda nueva tecnología, también conlleva retos y posibles contras, que exploraremos en la siguiente sección.

VI. Desventajas y desafíos

Pese a sus atractivos beneficios, la integración de una IA como compañero de programación también presenta diversas **desventajas y desafíos** que es importante considerar:

Dependencia excesiva y posible erosión de habilidades: Existe el riesgo de que los desarrolladores se *acostumbren* a aceptar sugerencias de la IA sin un análisis crítico profundo, lo que podría derivar en una comprensión superficial del código que producen [7]. Si un programador confía ciegamente en las salidas del asistente, puede perder poco a poco destrezas fundamentales como depurar lógicamente un problema o implementar algoritmos desde cero. Esto es especialmente preocupante para perfiles junior: apoyarse en la IA puede acelerar su trabajo, pero si la utilizan como muleta constante podrían no afianzar los conceptos básicos por sí mismos [7]. En equipos donde la IA se convierte en el "primer recurso" para todo, podría generarse una dependencia tal que, ante la ausencia de la herramienta, el equipo tenga dificultades para rendir al mismo nivel. Además, a largo plazo esta externalización cognitiva podría repercutir en que menos desarrolladores dominen a fondo ciertos temas, ampliando la brecha entre quienes programan y quienes entienden realmente el código.

Necesidad de aprender a usar la IA (curva de aprendizaje): Irónicamente, aunque la IA simplifica muchas tareas, para utilizarla bien hay que adquirir nuevas habilidades. No todos los desarrolladores logran adaptarse de inmediato a este estilo de trabajo asistido. Aprender a redactar buenos *prompts*, interpretar las sugerencias (que a veces pueden ser enrevesadas) y saber cuándo confiar en la IA y cuándo ignorarla, es un arte que toma tiempo [7]. Durante el periodo de adaptación, algunos reportan que la IA incluso puede *ralentizar* el flujo (por ejemplo, sugiriendo cosas irrelevantes que distraen o teniendo que experimentar con distintas formas de pedir algo hasta obtener lo deseado) [7]. En ciertos casos, los desarrolladores pueden sentirse inicialmente frustrados hasta comprender cómo "pensar junto con la IA". Esta curva de aprendizaje implica un esfuerzo y tiempo adicional que los equipos deben invertir para sacarle verdadero provecho a la herramienta.

Sugerencias incorrectas o fuera de contexto: A pesar de los avances, las IAs actuales no son infalibles. Pueden generar código que *parece* válido pero contiene errores lógicos sutiles, o que simplemente no encaja con los requerimientos específicos del problema en cuestión [7]. Por ejemplo, se han observado casos donde el asistente propone soluciones genéricas que no aplican a ciertos dominios (p. ej., sugiere un algoritmo ineficiente para un conjunto de datos grande, porque en su entrenamiento vio ejemplos pequeños), o completa con código obsoleto/no óptimo. Si el desarrollador no detecta estas imperfecciones, podría introducir bugs o vulnerabilidades sin darse cuenta. Un ingeniero señaló que algunas recomendaciones de la IA, de ser aplicadas sin más, habrían roto el sistema en producción [12]. En entornos complejos con lógica de negocio intrincada, la IA suele carecer de entendimiento profundo del dominio, por lo que sus aportes deben tomarse con cautela. En síntesis, siempre se requiere la supervisión y validación humana de lo sugerido, lo cual demanda tiempo y atención.

Riesgos de privacidad y seguridad del código: Muchas herramientas de IA funcionan enviando fragmentos del código del usuario a servidores en la nube para ser procesados por el modelo. Esto puede presentar preocupaciones en torno a la confidencialidad del código fuente, especialmente en compañías que manejan código propietario sensible [7]. Existe temor a la fuga involuntaria de información: ¿podría la IA "aprender" de mi código privado y luego sugerir partes similares a otro usuario externo? Si bien proveedores como OpenAI y GitHub han implementado políticas para no retener datos de usuarios empresariales o filtrar secretos, algunas organizaciones con altos estándares de seguridad son reticentes a usar estos servicios en código crítico [7]. Adicionalmente, si la IA sugiere código que proviene de su entrenamiento, podría insertar inadvertidamente fragmentos que contengan vulnerabilidades conocidas o contravengan políticas de seguridad internas. También es posible que genere código aparentemente correcto pero inseguro (ej: consultas SQL sin sanitizar apropiadamente), por lo que el equipo debe estar alerta y quizá reforzar las revisiones de seguridad al usar estas herramientas.

Cuestiones de licencia, derechos de autor y ética: Un punto debatido ha sido que los modelos de IA de código son entrenados con repositorios públicos, algunos bajo licencias copyleft (como GPL). Ha habido casos donde Copilot sugirió bloques de código casi idénticos a implementaciones conocidas de proyectos open source, sin atribución [7]. Esto despierta dudas legales: si un desarrollador acepta esa sugerencia y la incluye en un producto cerrado, ¿podría estar infringiendo una licencia de software sin saberlo? La frontera no es clara aún, y están surgiendo discusiones éticas sobre la atribución debida y la propiedad intelectual del código generado por IA [7]. Si bien la mayoría de las salidas de la IA son un "remix" original de muchos datos, la posibilidad de filtraciones textuales existe. Por tanto, empresas están teniendo que desarrollar políticas sobre el uso aceptable de estos asistentes, definiendo cómo auditar el origen del código sugerido y prevenir posibles conflictos legales [5]. En el aspecto ético, también preocupa que los modelos puedan perpetuar sesgos: por ejemplo, generar ejemplos discriminatorios o con supuestos inapropiados (se citó un caso polémico donde Copilot completó código asignando sueldos menores a mujeres basado en datos sesgados de entrenamiento [7]). La verificación y corrección de estos sesgos es otro desafío a afrontar.

Impacto en la dinámica de equipos y roles: La introducción de IA en tareas de codificación suscita preguntas sobre el futuro de ciertos roles y la estructura de los equipos de desarrollo. Por un lado, desarrolladores menos experimentados podrían verse desplazados de algunas tareas rutinarias que ahora la IA realiza, lo que podría limitar sus oportunidades de aprender *haciendo* en proyectos reales [7]. Esto podría agrandar la brecha entre desarrolladores senior (que aprovechan la IA eficazmente gracias a su experiencia) y junior (que podrían ser relegados a un papel más pasivo de validación). Por otro lado, emergen roles nuevos, como "Ingeniero de Prompt" o "Especialista en Integración de IA", encargados de optimizar y monitorear el uso de estas herramientas en los equipos [5]. A nivel de organización, algunos temen que una alta dependencia en IA pueda reducir la interacción entre colegas humanos, afectando la transmisión orgánica de conocimientos y la mentoría clásica que ocurría en pair programming tradicional. También existe cierta resistencia al cambio: no todos los desarrolladores se sienten cómodos adoptando la IA, y obligarlos podría generar tensiones o división en el equipo [5]. Gestionar este cambio cultural es una consideración importante para líderes de ingeniería.

Falsos positivos y disminución temporal de confianza: En la práctica, se han observado situaciones donde la IA sugiere con mucha seguridad una solución que resulta ser incorrecta. Esta "confianza" aparente del asistente puede confundir al usuario. Si el desarrollador descubre varios errores provenientes de la IA, podría llegar a desconfiar y descartar incluso las sugerencias correctas, perdiendo así el beneficio. Alternativamente, si confía demasiado, puede pasar por alto errores sutiles incorporados por la IA. Encontrar el equilibrio adecuado de confianza en las recomendaciones de la IA es difícil. Un desarrollador describió que algunos días Copilot sugería soluciones brillantes, y otros días recomendaciones absurdas; este comportamiento inconsistente puede frustrar y mermar la confianza en la herramienta [12]. También, la constante necesidad de verificar a la IA puede suponer una carga cognitiva adicional: básicamente se realiza *code review* en tiempo real de cada snippet sugerido.

Limitaciones técnicas y de dominio: Aunque las IAs actuales tienen gran conocimiento general, pueden fallar en contextos muy específicos o con tecnologías muy nuevas. Si se está trabajando en un lenguaje de programación

desconocido o en un framework nuevo, tener a la IA de pareja puede acelerar la curva de aprendizaje. A medida que uno escribe, la IA sugiere sintaxis correcta, mejores prácticas de ese ecosistema y uso idiomático de las APIs. Esto es similar a programar junto a un colega experto en ese lenguaje. Por ejemplo, un desarrollador Python que deba escribir algo en Go puede guiarse con las completaciones que Copilot le brinda, imitando estructuras idiomáticas (como canales, goroutines, etc.) sin tener que buscar todo en la documentación. En la fintech Naranja X mencionada, un desarrollador de back-end logró resolver un problema de front-end fuera de su área de expertise apoyándose en Copilot, lo que demuestra cómo la IA puede cubrir brechas de conocimiento técnico en un equipo [9]. La efectividad del asistente depende en parte de que haya "visto" algo similar durante su entrenamiento. Además, estos modelos suelen tener un límite de contexto (cantidad de código que pueden considerar a la vez); en bases de código enormes puede que no tomen en cuenta partes relevantes que no entren en ese contexto, lo que limita su comprensión global del sistema. También hay que mencionar que a veces la IA simplemente no entiende lo que se le pide si la especificación es ambigua, devolviendo cualquier cosa que encaje estadísticamente, lo cual no siempre es lo correcto.

En conclusión, si bien la IA puede ser un poderoso aliado en el desarrollo de software, su uso eficaz requiere conciencia de estos riesgos y desafíos. Es crucial implementar buenas prácticas: siempre revisar y probar el código generado, continuar capacitando a los desarrolladores en fundamentos (para no volverlos dependientes ciegos), manejar con cuidado datos sensibles, y establecer lineamientos éticos y legales en torno a su utilización [5] [5]. De esta forma se pueden mitigar las desventajas y aprovechar los beneficios dentro de un marco responsable.

VII. Casos de uso recomendados

Dada la naturaleza de sus fortalezas y limitaciones, el pair programming con IA es más provechoso en ciertos escenarios y tipos de problemas. A continuación se enumeran varios **casos de uso** donde esta práctica puede aportar gran valor:

Proyectos con alta carga de código repetitivo o plantilla: Cuando el desarrollo involucra escribir muchos componentes similares o código *boilerplate* (por ejemplo, múltiples clases CRUD, getters/setters, mapeos de campos, código ceremonial de configuración), la IA destaca al generar rápidamente esas secciones a partir de uno o dos ejemplos. Por ejemplo, en una API con decenas de endpoints similares, Copilot puede sugerir la estructura de cada nuevo endpoint siguiendo el patrón de los existentes, ahorrando un tiempo considerable en mecanografía. Esto permite mantener consistencia en el estilo y reduce la fatiga del programador por tareas monótonas.

Escritura de pruebas unitarias y casos de prueba: Generar tests puede ser tan laborioso como escribir el código de la funcionalidad en sí. Los asistentes de IA pueden analizar una función o módulo y proponer automáticamente varios casos de prueba relevantes, incluyendo las aserciones esperadas [7]. Por ejemplo, dado un método esPrimo(n), la IA puede sugerir tests comprobando valores primos, no primos, casos borde (0, 1, negativos). Esto ayuda a mejorar la cobertura de pruebas de manera ágil. Startups con equipos pequeños han aprovechado esta capacidad para lograr altos niveles de testing sin distraer a los desarrolladores de la construcción de features principales [5].

Aprendizaje de nuevos lenguajes o frameworks sobre la marcha: Cuando un desarrollador incursiona en un lenguaje de programación desconocido o en un framework nuevo, tener a la IA de pareja puede acelerar la curva de aprendizaje. A medida que uno escribe, la IA sugiere sintaxis correcta, mejores prácticas de ese ecosistema y uso idiomático de las APIs. Esto es similar a programar junto a un colega humano. Por ejemplo, un desarrollador Python que deba escribir algo en Go puede guiarse con las completaciones que Copilot le brinda, imitando estructuras idiomáticas (tal vez aprovechando canales, goroutines, etc.) sin tener que buscar todo en la documentación. En la fintech Naranja X, un desarrollador de back-end logró resolver un problema de front-end fuera de su área de expertise apoyándose en Copilot, lo que demuestra cómo la IA puede cubrir brechas de conocimiento técnico en un equipo [9].

Refactorización y modernización de código legado (legacy): En proyectos antiguos o con *deuda técnica*, usar la IA como apoyo para refactorizar puede ser muy útil. Por ejemplo, al modernizar una aplicación escrita en un framework viejo hacia uno más nuevo, el desarrollador puede convertir uno de los módulos manualmente y luego dejar que la IA repita el patrón en los demás módulos. De hecho, se ha reportado un caso de migración de una aplicación de Angular a React donde Copilot permitió completar la tarea con aproximadamente 40 % menos de tiempo del estimado originalmente [10]. Asimismo, la IA puede sugerir automáticamente reemplazos de funciones o librerías obsoletas por equivalentes actuales, al haber "visto" muchas transformaciones similares en otros proyectos.

Asistencia en depuración de código (*debugging*): Cuando se enfrenta un bug difícil, la IA puede ayudar ofreciendo hipótesis o destacando partes del código potencialmente implicadas. Un flujo común es pegar el mensaje de error o describir el problema en un prompt (o comentario) y pedir a la IA orientación.

Por ejemplo: "`// BUG: esta función se cuelga con entradas grandes, sugerencias?`". La IA, con su entrenamiento, puede reconocer patrones de errores comunes y señalar posibles soluciones (quizá recomendar un algoritmo más eficiente o un manejo distinto de recursividad). Si bien no siempre dará en el clavo, puede ahorrar tiempo apuntando hacia pistas que el desarrollador puede probar. Esto es especialmente útil para errores de sintaxis o mal uso de APIs: Copilot/ChatGPT puede corregir un trozo de código mal escrito al identificar qué falla, actuando como un "patito de hule" que además responde con propuestas.

Exploración de soluciones alternativas y creatividad asistida: En problemas abiertos donde hay múltiples maneras de resolver algo, trabajar con la IA permite explorar enfoques que quizá el programador no habría considerado. Por ejemplo, ante una tarea de optimización, se le puede pedir a la IA "propón otra forma de hacer X" y ésta podría devolver un algoritmo diferente (tal vez uno más eficiente). Esto enriquece el proceso creativo, casi como hacer *brainstorming* con un colega. Algunos desarrolladores indican que la IA les ha sugerido patrones de diseño o funciones de librerías estándar desconocidas para ellos, ampliando su caja de herramientas [12]. En contextos de investigación o desarrollo de prototipos, este intercambio con la IA puede acelerar la obtención de resultados novedosos.

Documentación y explicación de código: Otra aplicación valiosa es usar la IA para generar descripciones de código para documentación o para comprender secciones complejas escritas por otros. Un desarrollador puede seleccionar una función complicada y pedirle a la IA "explica qué hace esto". La IA producirá un resumen en lenguaje natural, lo cual puede clarificar la intención. Esto ayuda al onboarding de nuevos miembros en un proyecto: en lugar de descifrar a mano cada componente, pueden apoyarse en la IA para obtener "resúmenes" rápidos. Igualmente, a la inversa, la IA puede sugerir comentarios y documentación en el momento en que el código es escrito, asegurando que se mantenga la práctica de documentar sin mucho esfuerzo adicional [5].

Proyectos con plazos muy ajustados: En situaciones donde el tiempo de desarrollo es crítico (p. ej., hackatones, demos rápidas para clientes, respuesta urgente a cambios de requerimientos), la IA acelera la producción de una versión funcional. Como se mencionó, el Banco Galicia logró implementar en un fin de semana un cambio regulatorio complejo apoyándose intensamente en Copilot [9]. En hackatones, desarrolladores solos pueden alcanzar a prototipar aplicaciones completas que normalmente requerirían equipos, gracias a asistentes que generan partes estándar del código. Obviamente, se debe tener cuidado con la calidad en estos casos, pero para entregar algo que funcione rápidamente, la IA es una gran aliada.

Equipos pequeños o desarrolladores en solitario: En startups o proyectos unipersonales, tener un "compañero" de IA suple en parte la falta de colegas para rebotar ideas o revisar código. Un equipo de 2 desarrolladores con IA puede lograr una productividad cercana a la de un equipo de mayor tamaño, al multiplicar su output con las sugerencias automáticas [5]. Esto democratiza el desarrollo: incluso programadores independientes pueden encarar proyectos más grandes o complejos contando con la asistencia constante de la IA para las partes repetitivas o menos especializadas.

En todos estos casos de uso, es importante recalcar que la IA brilla como ayudante pero no reemplaza el criterio del desarrollador. Las mejores situaciones para usarla son aquellas donde puede ahorrar tiempo y dar soporte, mientras el humano conserva la guía del diseño general y la validación final. Cuando se la emplea de forma estratégica, alineando sus fortalezas con las necesidades del proyecto, el pair programming con IA puede ser extremadamente efectivo y elevador de productividad.

VIII. Casos de aplicación reales en la industria

A medida que la adopción de la programación en pareja con IA se expande, surgen múltiples **casos reales** que ilustran su impacto en entornos profesionales. A continuación se presentan algunos ejemplos y experiencias reportadas en la industria del software:

Caso 1: Implementación crítica acelerada en banca (Banco Galicia, Argentina). En abril de 2025, los desarrolladores del Banco Galicia enfrentaron un desafío imprevisto: tras un cambio regulatorio anunciado un viernes por la tarde, necesitaban habilitar en su aplicación transacciones en dólares para el lunes siguiente, un plazo sumamente corto [9] [9]. Para lograrlo, el equipo técnico recurrió intensamente a GitHub Copilot durante ese fin de semana frenético. Según Germán Gross, jefe de arquitectura del banco, los asistentes de código les permitieron ganar la velocidad necesaria para hacer realidad la nueva funcionalidad a tiempo, *"con todos los controles de seguridad y calidad que eso merece"* [9]. Un desarrollador senior de Galicia afirmó que habría sido "muy improbable" cumplir con la meta sin Copilot, herramienta que ya usaba casi a diario [9]. Gracias a la IA, pudieron generar rápidamente las nuevas rutinas de negocio y pruebas, asegurando el cumplimiento regulatorio en cuestión de horas. Actualmente, alrededor de 500 desarrolladores en el banco utilizan Copilot activamente, y la dirección tecnológica señala que esta clase de asistentes se ha vuelto "obligatoria" para mantener la innovación y tiempos de respuesta en un sector tan competitivo [9]. En resumen, Copilot funcionó como un catalizador que multiplicó la capacidad del equipo en un momento crítico, sin comprometer seguridad ni fiabilidad.

Caso 2: Ahorro de tiempo y mejora de experiencia en fintech (Naranja X). Naranja X, una fintech argentina dedicada a servicios de crédito digital, integró GitHub Copilot tempranamente en su flujo de desarrollo. Según Sofía Baggini, líder de DevOps, sus desarrolladores ahorran en promedio de 2 a 3 horas de trabajo por día gracias a Copilot [9]. Tareas que antes tomaban una tarde entera, ahora se resuelven en cuestión de horas o minutos con la ayuda de la IA. En particular, Baggini menciona que el tiempo invertido en solucionar ciertos problemas técnicos se redujo en un 50 % [9]. Además, Copilot ha permitido que sus ingenieros gestionen múltiples proyectos simultáneamente con mayor calidad, encontrando soluciones creativas y cambiando de un lenguaje de programación a otro con rapidez [9]. Un desarrollador de pruebas automatizadas en Naranja X describe la IA como "un muy buen acelerador" que le ayuda a mantener un ritmo alto de entregas sin sacrificar la calidad [9]. Para la empresa, esto se traduce en sacar nuevas features al mercado más rápido y ofrecer mejor experiencia a sus usuarios finales. La adopción fue tan natural que consideran que tener estos copilotos de código "estaba en su ADN" innovador.

Caso 3: Estudio de productividad en empresa de desarrollo (Future Processing). Future Processing, una compañía de software europea, realizó un estudio interno al introducir GitHub Copilot en varios de sus proyectos comerciales [10]. Tras algunos meses de uso controlado, recogieron datos mediante encuestas y métricas de rendimiento. Encontraron que un 43 % de los desarrolladores comenzaron a usar Copilot a diario, y luego del piloto, el 80 % decidió que quería continuar utilizándolo [10]. En términos de productividad, midieron que al escribir código nuevo, los programadores con Copilot eran en promedio un 34 % más rápidos, y al escribir pruebas unitarias hasta un 38 % más rápidos [10]. De manera abrumadora (96 % de encuestados), coincidieron en que Copilot agilizó su trabajo diario [10]. Un ejemplo concreto fue la migración de un proyecto web de Angular a React, donde con apoyo de la IA se completó la migración aproximadamente 40 % más rápido de lo estimado inicialmente [10]. El estudio concluyó que Copilot permitió a sus equipos enfocarse más en la lógica de negocio, redujo frustraciones en tareas rutinarias y actuó como un recurso valioso de consulta rápida. También notaron que no todas las tareas se benefician

por igual: la mayor ganancia estuvo en codificación de nuevas funciones y en generación de tests, mientras que en tareas muy creativas o algoritmos complejos la ayuda fue más limitada, lo que tiene sentido dado el funcionamiento de la herramienta.

Caso 4: Mejora colaborativa en proyecto open source (experiencia individual). Un desarrollador contribuidor a un proyecto de código abierto relató su experiencia tras "parearse" con ChatGPT durante un mes de desarrollo activo [12]. Inicialmente escéptico, acabó sorprendido de cómo la IA amplificó su creatividad: le ayudó a articular mejor los problemas y a explorarlos paso a paso [12]. Por ejemplo, al enfrentarse a un bug complejo, le describió el problema a ChatGPT y éste le guió para dividirlo en sub-problemas más manejables [12]. Si bien la IA no dio la solución final, el proceso de conversarlo "con alguien" le hizo pensar más profundamente y llegar a una solución robusta. Asimismo, comenta que la IA le sugirió patrones de diseño y enfoques alternativos que terminaron mejorando la arquitectura del proyecto [12]. Notó también momentos cómicos: en algunos casos la IA devolvió sugerencias completamente fuera de lugar (p.ej., referencias a "blockchain cuántico" en un comentario) que le hicieron reír [12], recordándole que el control humano es indispensable. En balance, encontró que la IA no lo volvió ni más lento ni más rápido en términos netos, pero sí hizo el proceso "más reflexivo y menos solitario", aportando valor intangible a su flujo de trabajo.

Caso 5: Adopción a gran escala en empresas tecnológicas: Grandes corporaciones de software han empezado a integrar oficialmente estas herramientas en sus procesos. Por ejemplo, Microsoft (casa madre de GitHub) ofrece Copilot a sus desarrolladores internos y reportó que en ciertos equipos aumentó la velocidad de integración de código (time-to-merge) en alrededor de 50 % [11]. Empresas como Shopify y Duolingo participaron en estudios que mostraron mejoras en la felicidad y eficiencia de los desarrolladores con estas ayudas, alentando la adopción generalizada [11]. No obstante, también comparten lecciones aprendidas: la necesidad de seguir buenas prácticas de ingeniería (no confiar ciegamente, añadir revisiones automatizadas para codegen, etc.) y de entrenar al personal en el uso responsable [11]. La mayoría concluye que la pregunta ya no es si usar o no usar IA en desarrollo, sino cómo usarla mejor integrándola con las metas organizacionales y flujos DevOps existentes.

Estos casos evidencian en la práctica tanto los beneficios cuantitativos (horas ahorradas, velocidad incrementada, menos errores en producción) como los cualitativos (mejor experiencia de desarrollo, mayor creatividad, aprendizaje constante) de contar con un "par" de IA en el equipo. Desde sectores financieros altamente regulados hasta ágiles startups de tecnología, el consenso emergente es que, bien aplicada, esta colaboración humano-IA aporta ventajas competitivas reales. Vale la pena señalar que los éxitos vienen acompañados de la formulación de nuevas políticas y entrenamiento: las organizaciones que destacan en estos casos invirtieron en establecer normas de uso (por ejemplo, para evitar salida de datos sensibles, para revisión de licencias del código sugerido, etc.) y en preparar a sus desarrolladores para sacar el máximo provecho de la herramienta de forma segura y efectiva.

IX. Demanda del mercado laboral y perfiles asociados a la IA

La adopción de la inteligencia artificial en el desarrollo de software no solo transforma la manera de programar, sino también los criterios de contratación. A nivel global, informes recientes muestran que más del 50 % de los desarrolladores ya utilizan asistentes de código y que las organizaciones esperan que sus equipos sean capaces de integrarlos en los flujos de trabajo diarios [1], [5]. En este contexto, las habilidades de *pair programming con IA* y de orquestación de modelos comienzan a consolidarse como parte del **stack tecnológico esperado** de un desarrollador moderno, al mismo nivel que conocer lenguajes, frameworks y herramientas de nube.

En el caso colombiano, esta tendencia ya es visible. Según un estudio de IDC encargado por Deel y reseñado por el diario económico *Portafolio*, la inteligencia artificial se ha convertido en un criterio decisivo en los procesos de contratación:

- el 66 % de las empresas del país ya superó la etapa inicial de adopción de IA;
- 4 de cada 10 empresas contratan talento *junior* con habilidades específicas en IA;

- el 42 % está invirtiendo en programas de capacitación en IA por departamento, y un 31 % adicional planea hacerlo en los próximos 12 meses [13].

Estos datos sugieren que, para roles de entrada en tecnología, **no basta con conocer un lenguaje de programación**: se espera cierto dominio de herramientas de IA (copilotos, chatbots especializados, motores de búsqueda aumentados, etc.) y la capacidad de integrarlas al flujo de trabajo diario. La Figura 1 ilustra este punto a través del titular del artículo de *Portafolio*, donde se destaca explícitamente que las empresas colombianas "ya exigen habilidades en IA para contratar".

Figura 1. Titular del diario económico *Portafolio* (21 de noviembre de 2025) que resume los resultados del estudio de Deel: las empresas colombianas ya exigen habilidades en IA para nuevos procesos de contratación [13].

Además de esta demanda general de "saber IA", empiezan a aparecer roles especializados que hacen explícito el uso intensivo de enfoques de *vibe coding*. Un ejemplo es la vacante de *AI Vibe Coder* publicada por una empresa tecnológica para trabajar de forma remota desde Colombia (Figura 2). El anuncio enfatiza responsabilidades como convertir ideas de producto en prototipos funcionales, construir aplicaciones *full–stack* ligeras, trabajar en ciclos de retroalimentación muy cortos y mantener el código lo suficientemente organizado como para ser escalado por otros equipos. En la práctica, se trata de un rol centrado en **prototipar rápidamente con IA generativa**, alineado con las prácticas de *vibe coding* descritas en la Sección XI.

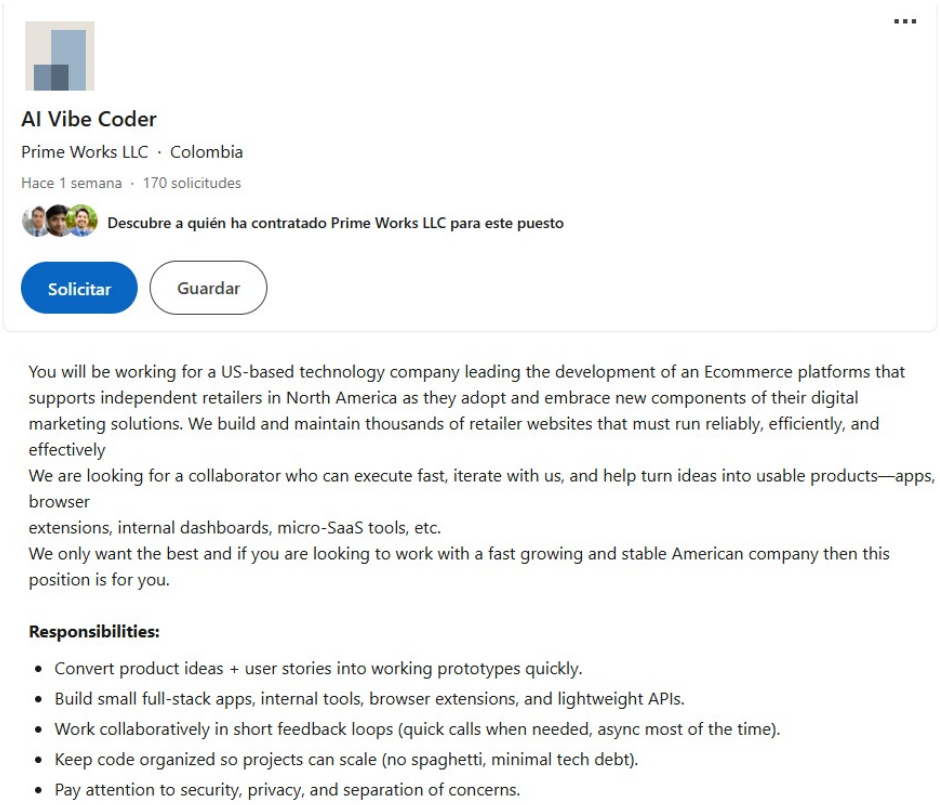

Figura 2. Ejemplo de oferta laboral para el rol de *AI Vibe Coder*, orientado a prototipar productos rápidamente utilizando IA generativa. El énfasis está en ciclos de feedback cortos, uso intensivo de IA y capacidad para convertir ideas en aplicaciones funcionales.

Para analizar esta demanda desde la perspectiva de perfiles en desarrollo de software y arquitectura, se propone la Matriz I. En ella se categoriza el nivel esperado de distintas competencias relacionadas con IA para cuatro perfiles típicos en equipos de ingeniería (incluyendo el uso de pair programming con IA como competencia transversal). Las categorías se expresan como: **B** (básica), **M** (media) y **A** (avanzada).

Cuadro I
MATRIZ DE PERFILES Y DEMANDA DE HABILIDADES RELACIONADAS CON IA EN EQUIPOS DE DESARROLLO

Perfil	**Alfabetización**	**Pair prog. con IA**	**Orquestación de IA**	**Gobernanza / Ética**
Desarrollador junior	B	M	B	B
Desarrollador mid–level	M	M	M	M
Desarrollador senior / TL	M	A	A	M
Arquitecto / Plataformas IA	M	A	A	A

En esta matriz, la *alfabetización en IA* incluye el dominio básico de conceptos (modelos, prompts, limitaciones); el *pair programming con IA* se refiere a la capacidad de integrar copilotos y asistentes conversacionales en el flujo de desarrollo diario; la *orquestación de IA* abarca el diseño de flujos donde intervienen varios modelos y herramientas (por ejemplo, combinar copilotos, búsquedas semánticas y herramientas internas); y la dimensión de *gobernanza/ética* reúne aspectos de seguridad, licenciamiento, protección de datos y uso responsable del código generado [5], [7].

Desde esta perspectiva, trabajar con IA pasa a entenderse como un **nuevo layer del stack tecnológico**. De manera simplificada, puede pensarse en un "stack ampliado" como el que se muestra en la Tabla II, donde las herramientas de IA se integran explícitamente como capa transversal:

Cuadro II
COMPARACIÓN ENTRE STACK TRADICIONAL Y STACK AMPLIADO CON IA

Capa	**Stack tradicional**	**Stack ampliado con IA**
Lenguajes y frameworks	Java, Python, JavaScript, Spring, React	Igual, más SDKs de IA (OpenAI, Anthropic, etc.)
Infraestructura	Nube, contenedores, CI/CD	Igual, más plataformas de inferencia y monitoreo de modelos
Colaboración	Git, code review, pair programming humano	Pair programming con IA, copilotos integrados al IDE
Capa IA	(no explícita)	Modelos fundacionales, agentes, herramientas de búsqueda aumentada
Gobierno	Normas de calidad, seguridad, licencias	Políticas de uso de IA, revisión de código generado, trazabilidad de prompts

Al enlazar los datos del mercado laboral colombiano, la aparición de vacantes específicas para *AI Vibe Coder* y esta visión de stack ampliado, se refuerza la idea de que dominar la IA —tanto en modalidades estructuradas de pair programming como en contextos de prototipado rápido con *vibe coding*— ya no es un lujo opcional, sino un componente esperado del perfil profesional en ingeniería de software. En consecuencia, formar a los estudiantes en estas prácticas no solo responde a una tendencia tecnológica, sino a una demanda concreta del mercado laboral nacional e internacional [5], [13].

X. VELOCIDAD DE AVANCE DE LA IA GENERATIVA Y COMPARACIÓN DE MODELOS

La práctica de pair programming con IA está íntimamente ligada a la evolución de los modelos subyacentes. Desde los primeros prototipos basados en GPT-2 y Codex hasta los modelos de 2024–2025 (familias GPT-4/4o, Claude 3.x, Gemini 1.5, modelos de razonamiento *o1*/*o3*, etc.), se ha observado una mejora rápida en tres dimensiones clave: **calidad de las sugerencias**, **tamaño de contexto** y **latencia de respuesta** [7], [8]. Esta mejora ha permitido que los asistentes pasen de completar líneas cortas a razonar sobre archivos enteros, refactorizar módulos grandes y mantener sesiones de pair programming de varias horas con contexto compartido.

Distintos estudios y reportes técnicos señalan que los modelos de 2024–2025 superan ampliamente a Codex y a los primeros modelos de Copilot tanto en benchmarks de razonamiento como en tareas prácticas de desarrollo [8]. Un ejemplo claro de esta aceleración es el **Epoch Capabilities Index (ECI)** de Epoch AI, que agrega los resultados de 179 modelos publicados entre 2023 y 2025 y les asigna un puntaje de capacidad global en múltiples tareas [14]. La Figura 3 muestra cómo, a medida que avanza el tiempo, los modelos recientes tienden a ocupar sistemáticamente posiciones con mayor puntaje en el eje vertical, evidenciando una mejora sostenida de capacidades.

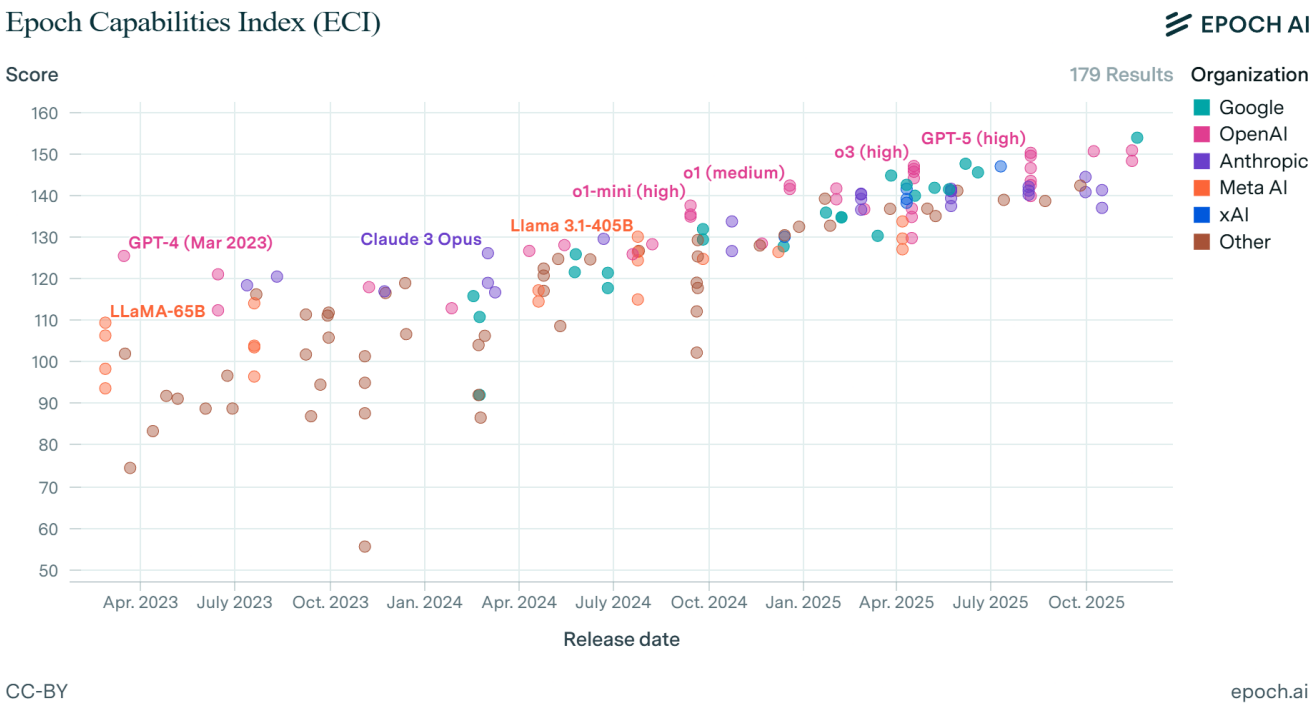

Figura 3. Evolución de las capacidades de modelos de IA generativa según el Epoch Capabilities Index (ECI). Cada punto representa un modelo evaluado; el eje horizontal indica la fecha de lanzamiento y el eje vertical su puntaje de capacidad global. Fuente: adaptación de datos de Epoch AI [14].

En el gráfico, los modelos lanzados alrededor de abril de 2023 (por ejemplo, GPT-4 y LLaMA-65B) se concentran aproximadamente en el rango de 100–130 puntos, mientras que en 2024 aparecen modelos como Claude 3 Opus y Llama 3.1–405B que se sitúan ligeramente por encima de esa franja. A partir de 2025, la "nube" de puntos se desplaza de forma clara hacia arriba: modelos de razonamiento como *o1* y *o3*, así como GPT-5 (*high*), alcanzan puntuaciones cercanas o superiores a 150, lo que indica un salto significativo en capacidad con respecto a la generación anterior [14]. Lo que mas resalta es que el modelo mas avanzado de la grafica es Gemini pro 3 de Google que salio apenas hace dos semanas.

Lo relevante para la programación en parejas con IA es que esta **tendencia monotónicamente creciente** no solo muestra que "los modelos nuevos son mejores", sino que la mejora es suficientemente rápida como para tener consecuencias prácticas en los flujos de trabajo:

- **Obsolescencia rápida de herramientas**: el intervalo entre generaciones que cambian de forma apreciable el nivel de capacidad se ha reducido a unos pocos trimestres. Un asistente de código basado en una familia de modelos de 2023 puede quedar claramente por detrás de uno de 2025 en tareas de razonamiento profundo, refactorización global o comprensión de repositorios grandes [8], [14].
- **Cambio en el rol del humano en la pareja**: conforme el modelo es capaz de proponer arquitecturas completas o modificar muchos archivos simultáneamente, el humano pasa de escribir líneas de código a diseñar constraints, revisar riesgos y decidir qué cambios aceptar [5], [7]. La capacidad extra del modelo se "invierte" en tareas de mayor nivel, no en seguir escribiendo más líneas por minuto.
- **Mayor variabilidad entre modelos**: el ECI muestra que diferentes organizaciones (OpenAI, Google, Anthropic, Meta, xAI, entre otras) ocupan distintos puntos en la frontera de capacidades [14]. Elegir "el compañero de IA adecuado" para un proyecto concreto se convierte así en una decisión técnica relevante: no es lo mismo optimizar por velocidad, coste, tamaño de contexto o capacidad de razonamiento.

Desde la perspectiva educativa, esta aceleración obliga a enseñar a los estudiantes no solo a usar una herramienta concreta (Copilot, Cursor, etc.), sino a:

1. entender las diferencias entre familias de modelos (razonamiento vs. velocidad vs. contexto),
2. diseñar experimentos básicos para comparar modelos en un mismo problema,
3. y documentar las decisiones de selección de modelo como parte de la arquitectura de software.

En otras palabras, el *pair programming con IA* en 2025 ya no es solo "usar Copilot", sino **configurar un ecosistema de modelos** que actúa como equipo extendido alrededor del desarrollador humano.

XI. Vibe Coding y sus retos

En paralelo al uso más "clásico" del pair programming con IA (donde el desarrollador revisa y entiende el código sugerido), ha surgido recientemente el concepto de **"Vibe Coding"**. De forma simplificada, el vibe coding describe una forma de programar donde el desarrollador:

- explica en lenguaje natural lo que quiere lograr,
- deja que la IA genere prácticamente todo el código,
- y se centra más en probar el comportamiento que en leer o comprender el código fuente subyacente [15].

La idea fue popularizada en 2025 por figuras como Andrej Karpathy y ha ganado visibilidad en medios generalistas y blogs técnicos, hasta el punto de ser reconocida como término del año por distintos medios y diccionarios [15].

Desde la perspectiva de ingeniería de software, el vibe coding puede verse como un *caso extremo* dentro del continuo de colaboración humano-IA:

- en el **pair programming con IA**, el humano mantiene un rol activo de diseño y revisión;
- en el **vibe coding**, el humano actúa más como "director de orquesta" que lanza prompts y valida resultados, pero rara vez inspecciona el código a fondo.

La Tabla III compara, de forma conceptual, ambos enfoques.

Cuadro III
Comparación conceptual entre pair programming con IA y vibe coding

Dimensión	Pair programming con IA	Vibe coding
Rol del desarrollador	Escribe parte del código, revisa y edita las sugerencias, toma decisiones de diseño explícitas.	Formula objetivos y prompts, ejecuta y prueba; interviene poco en la estructura interna del código.
Revisión del código	El código sugerido se lee y se entiende; se integra con estándares del equipo.	El código suele aceptarse como "caja negra", priorizando que "funcione" sobre cómo está implementado.
Riesgos principales	Dependencia de la IA, errores sutiles si no se revisa bien, sesgos en sugerencias [7].	Pérdida de entendimiento del sistema, deuda técnica elevada, posibles vulnerabilidades de seguridad y problemas de mantenimiento a largo plazo [15].
Contextos recomendados	Desarrollo profesional, proyectos de largo plazo, productos críticos, entornos regulados.	Prototipado rápido, proyectos personales, exploración creativa, pruebas de concepto de corta vida útil.

El auge del vibe coding plantea varios **retos** para la profesión:

- **Mantenibilidad y legibilidad**: código generado en sesiones de vibe coding tiende a ser difícil de mantener, ya que suele carecer de una arquitectura explícita y de documentación sistemática. Esto puede chocar frontalmente con los principios clásicos de diseño y con las necesidades de mantenimiento a mediano plazo [5], [7].
- **Erosión de habilidades fundamentales**: si los desarrolladores se acostumbran a no leer ni comprender el código, existe el riesgo de perder (o nunca desarrollar) competencias básicas en algoritmos, estructuras de datos y diseño de software. Esto es especialmente crítico en entornos educativos y para perfiles junior [7].

- **Seguridad y cumplimiento**: aceptar código "por vibra" sin revisión rigurosa puede introducir vulnerabilidades graves, uso incorrecto de librerías criptográficas, manejo inseguro de datos sensibles, etc. En sectores regulados, este enfoque es difícilmente compatible con auditorías y trazabilidad [5].
- **Gobernanza y responsabilidad**: si el equipo no entiende el código que ejecuta, se difumina la responsabilidad ante fallos o incidentes. En términos éticos y legales, sigue siendo el equipo humano quien responde por el comportamiento del sistema, no el modelo de IA.

A pesar de estos riesgos, el vibe coding también aporta aprendizajes útiles. Para estudiantes y personas no técnicas, puede servir como **puerta de entrada** para experimentar con ideas de software sin la barrera inicial del código, y como herramienta de prototipado rápido para explorar soluciones antes de invertir en una implementación "seria" [15]. En la medida en que se utilice de forma explícitamente acotada (prototipos, ideas, experimentos) y no como reemplazo de ingeniería rigurosa, puede complementarse con prácticas más disciplinadas de pair programming con IA.

Desde la perspectiva de Arquitectura de Software, una lectura razonable es:

> "El vibe coding puede ser una técnica aceptable para generar prototipos y descartar ideas rápidamente, pero la versión que llega a producción debe diseñarse y revisarse bajo principios de ingeniería y arquitectura explícitos".

En la práctica, esto sugiere un *pipeline* en el que:

1. se exploran ideas mediante vibe coding (rápido, de baja fricción),
2. se seleccionan las que tengan sentido de negocio,
3. y luego se reimplementan o consolidan mediante pair programming con IA, revisiones de código y prácticas de ingeniería tradicionales.

Integrar conscientemente ambos enfoques —vibe coding para explorar y pair programming con IA para construir y mantener— puede ofrecer lo mejor de ambos mundos, siempre y cuando exista una cultura fuerte de revisión técnica y responsabilidad sobre el código que finalmente se despliega en producción.

XII. Matrices de análisis

En esta sección se sintetiza el tema central del artículo —**pair programming con IA**— a la luz de distintos marcos clásicos de Arquitectura de Software y de la realidad del mercado laboral. Para ello se construyen cinco matrices: Principios SOLID vs. tema, Atributos de Calidad vs. tema, Tácticas vs. tema, Patrones vs. tema y Mercado Laboral vs. tema.

XII-A. Matriz de análisis de Principios SOLID vs. Tema

La Tabla IV analiza cómo el uso de asistentes de IA en programación en parejas se relaciona con los principios SOLID. El objetivo no es "forzar" el cumplimiento automático de estos principios, sino identificar dónde la IA puede apoyar y dónde puede amplificar malas decisiones de diseño si se usa sin criterio.

Cuadro IV
MATRIZ DE ANÁLISIS DE PRINCIPIOS SOLID VS. PAIR PROGRAMMING CON IA

Principio SOLID	**Impacto**	**Relación con pair programming con IA**
Responsabilidad Única (SRP)	Alto	La IA puede sugerir clases o funciones demasiado "grandes" si el prompt es ambiguo. El desarrollador debe guiarla para mantener responsabilidades claras y dividir el código en unidades coherentes.
Abierto/Cerrado (OCP)	Medio–Alto	Los copilotos facilitan extender comportamientos mediante nuevas clases o estrategias, pero también pueden inducir a modificar código existente de forma rápida. Se requiere disciplina para preferir extensiones sobre cambios destructivos.
Sustitución de Liskov (LSP)	Medio	La IA puede generar jerarquías de herencia sin respetar completamente las precondiciones o invariantes. El par humano debe revisar que las subclases mantengan contratos válidos y que los tests cubran estos casos.
Segregación de Interfaces (ISP)	Medio	Al generar interfaces a partir de descripciones en lenguaje natural, la IA tiende a agrupar muchas operaciones juntas. Un buen uso del pair programming con IA implica refinar esas interfaces hacia versiones más específicas y cohesivas.
Inversión de Dependencias (DIP)	Alto	Los asistentes favorecen patrones de inyección de dependencias y abstracciones reutilizables cuando se les guía correctamente, pero también pueden acoplar directamente a implementaciones concretas si el desarrollador no lo supervisa.

En conjunto, la matriz sugiere que el pair programming con IA puede ser un aliado para aplicar SOLID, siempre que el programador ejerza un rol activo de diseño y revisión. La IA acelera la escritura, pero la responsabilidad de mantener la estructura limpia sigue recayendo en el humano.

XII-B. Matriz de análisis de Atributos de Calidad vs. Tema

La Tabla V vincula el uso de IA en programación en parejas con algunos atributos de calidad típicos: mantenibilidad, rendimiento, seguridad, usabilidad del código y confiabilidad.

Cuadro V
MATRIZ DE ANÁLISIS DE ATRIBUTOS DE CALIDAD VS. PAIR PROGRAMMING CON IA

Atributo de calidad	**Impacto**	**Efecto del pair programming con IA**
Mantenibilidad	Alto	La IA puede homogenizar estilos y extraer funciones reutilizables, facilitando refactorizaciones frecuentes. Sin embargo, si se acepta código sin entenderlo, se incrementa la deuda técnica y se dificulta el mantenimiento futuro.
Rendimiento	Medio	Los modelos no siempre optimizan por eficiencia; pueden proponer soluciones correctas pero subóptimas. Es rol del humano identificar cuellos de botella y orientar a la IA a usar estructuras y algoritmos más eficientes.
Seguridad	Alto	Los asistentes pueden sugerir código inseguro (p. ej., consultas SQL sin sanitizar o validación insuficiente). En pair programming, el desarrollador debe revisar este aspecto con especial atención, complementando con análisis estático y pruebas de seguridad.
Usabilidad del código (legibilidad, claridad)	Alto	Cuando se les guía con buenos prompts, los asistentes tienden a generar nombres consistentes, comentarios y documentación básica, mejorando la legibilidad. Mal usados, pueden introducir código verboso o confuso.
Confiabilidad	Medio–Alto	La generación de pruebas unitarias y casos borde incrementa la probabilidad de detectar errores temprano. No obstante, la confiabilidad final depende de la integración de esas pruebas en el ciclo CI/CD y de la revisión humana del oráculo de cada test.

La matriz muestra que el impacto más claro se da en mantenibilidad, usabilidad del código y confiabilidad, mientras que rendimiento y seguridad exigen una vigilancia adicional para evitar que la velocidad que aporta la IA

se traduzca en errores costosos.

XII-C. Matriz de análisis de Tácticas vs. Tema

La Tabla VI relaciona algunas tácticas de diseño y arquitectura (orientadas a atributos de calidad) con el uso de pair programming con IA.

Cuadro VI
MATRIZ DE ANÁLISIS DE TÁCTICAS VS. PAIR PROGRAMMING CON IA

Táctica arquitectónica	**Impacto**	**Relación con el tema**
Encapsulación y separación de responsabilidades	Alto	El uso de IA para refactorizar facilita extraer componentes y servicios con límites más claros; el desarrollador puede pedir explícitamente "extraer este módulo" o "separar responsabilidades", acelerando la encapsulación.
Introducción de capas y puntos de extensión	Medio–Alto	La IA ayuda a generar adaptadores, interfaces y capas de servicio, reforzando tácticas de modifiabilidad. Sin embargo, requiere que el arquitecto exprese estos puntos de extensión en los prompts y en la estructura inicial.
Tácticas de testabilidad (inyección de dependencias, logs, aserciones)	Alto	Los asistentes pueden insertar pruebas, mocks y trazas en puntos clave del sistema, incrementando la observabilidad y la facilidad de prueba, sobre todo cuando se combinan con pipelines automatizados.
Tácticas de rendimiento (caching, uso eficiente de recursos)	Medio	La IA puede sugerir patrones de caching comunes, pero no tiene visibilidad directa del entorno de producción. Es necesario complementar con métricas y perfiles de rendimiento reales para decidir qué tácticas aplicar.
Tácticas de seguridad (validación, autenticación, manejo de errores)	Medio–Alto	Los modelos conocen bibliotecas y *best practices* de seguridad, pero pueden omitir controles en contextos específicos. El par humano debe validar que las tácticas propuestas sean consistentes con las políticas de seguridad de la organización.

En resumen, el pair programming con IA puede acelerar la aplicación de tácticas de modifiabilidad, testabilidad y, en menor medida, rendimiento y seguridad, siempre que el equipo mantenga una visión arquitectónica clara y traduzca esa intención en prompts y revisiones sistemáticas.

XII-D. Matriz de análisis de Patrones vs. Tema

La Tabla VII discute cómo la IA puede apoyar la aplicación de patrones de diseño y de arquitectura frecuentemente utilizados.

Cuadro VII
MATRIZ DE ANÁLISIS DE PATRONES VS. PAIR PROGRAMMING CON IA

Patrón	**Impacto**	**Relación con el tema**
Patrones de diseño (Strategy, Factory, Adapter, Observer, etc.)	Alto	La IA reconoce muchos de estos patrones y puede generar implementaciones estándar a partir de descripciones ("usar Strategy para seleccionar algoritmo de pago"). El riesgo es aplicarlos mecánicamente sin analizar si el patrón realmente es necesario.
Arquitectura en capas (Layered)	Medio–Alto	Los copilotos facilitan separar controladores, servicios y repositorios repitiendo el mismo esqueleto de capa. Esto refuerza la claridad de responsabilidades, siempre que el equipo mantenga la disciplina de no "saltar capas" en las integraciones.
Arquitecturas orientadas a servicios/microservicios	Medio	La IA puede generar esqueletos de servicios, APIs y contratos entre microservicios. Sin embargo, las decisiones de particionado, límites de contexto y manejo de datos distribuidos siguen siendo tareas críticas del arquitecto humano.
Pipes and Filters / Data pipeline	Medio	Los asistentes son útiles para encadenar transformaciones de datos (p. ej., pipelines de ETL o procesamiento de logs). Aun así, se requiere un diseño explícito del flujo y de los puntos de observabilidad.
Patrones de observabilidad (logging centralizado, métricas, tracing)	Alto	Con buenos prompts, la IA puede insertar puntos de log, métricas y trazas de forma sistemática en el código, facilitando la instrumentación necesaria para monitorear sistemas complejos.

La matriz evidencia que el valor de la IA está en acelerar la implementación de patrones, no en decidir cuál patrón es apropiado. Esa decisión sigue siendo responsabilidad del equipo de diseño y arquitectura.

XII-E. Matriz de análisis de Mercado Laboral vs. Tema

Finalmente, la Tabla VIII conecta el pair programming con IA con distintos perfiles profesionales y la demanda de habilidades de IA en el mercado laboral, en línea con los datos discutidos para Colombia y el contexto global [1], [13], [16].

Cuadro VIII
MATRIZ DE ANÁLISIS DE MERCADO LABORAL VS. PAIR PROGRAMMING CON IA

Perfil del mercado	**Demanda de IA**	**Implicaciones del pair programming con IA**
Desarrollador junior	Alta	Se espera que domine al menos un copiloto de código y pueda integrarlo en su flujo diario. El pair programming con IA se vuelve parte de su "stack mínimo", además de los lenguajes y frameworks básicos.
Desarrollador mid–level / senior	Muy alta	Debe combinar destreza técnica con capacidad de supervisar y corregir el código generado por la IA, guiando a perfiles junior y participando en decisiones de arquitectura donde la IA es un componente activo.
Arquitecto/a de software	Muy alta	Se espera que defina lineamientos de uso de IA, seleccione herramientas y modelos, y diseñe procesos de pair programming con IA que respeten los atributos de calidad del sistema y las políticas de la organización.
Orquestador de IA / Prompt engineer	Alta y creciente	Su rol gira explícitamente en torno a diseñar prompts, flujos de herramientas y evaluaciones de calidad para interacciones humano–IA. El pair programming con IA es su contexto natural de trabajo.
Especialista en plataformas de IA (MLOps/LLMOps)	Alta	Debe ofrecer entornos seguros y eficientes para que los equipos de desarrollo puedan usar modelos de IA como "pares" de programación, cuidando aspectos de coste, seguridad, latencia y cumplimiento normativo.

Esta última matriz refuerza la idea de que el pair programming con IA ya no es solo una curiosidad técnica, sino un componente central de los perfiles que el mercado valora. Dominar esta práctica aumenta la empleabilidad, especialmente en contextos donde las organizaciones ya exigen habilidades en IA como parte de sus procesos de contratación.

XIII. Modelos, herramientas y modos de interacción con la IA

En la discusión sobre programación en parejas con IA es frecuente que se mezclen términos como *modelo*, *herramienta*, *editor* o *agente*, como si fuesen sinónimos. Sin embargo, desde la perspectiva de Arquitectura de Software y de adopción profesional, es importante distinguir claramente estos conceptos. Esta sección aclara qué entendemos por **modelo fundacional** (por ejemplo, un modelo de la familia GPT), qué es una **herramienta** de desarrollo asistida por IA (como Cursor, Claude Code o Antigravity), la diferencia entre herramientas locales y remotas, y los principales **modos de interacción** (chat, completado, agente, etc.) que hoy se utilizan en el pair programming con IA.

XIII-A. ¿Qué es un modelo fundacional?

Un **modelo fundacional** (por ejemplo, un modelo de la familia GPT, Claude o Llama) es un sistema de aprendizaje profundo entrenado sobre grandes volúmenes de datos para aprender patrones generales de lenguaje y código [7], [8]. En este trabajo nos centramos en modelos de lenguaje de gran tamaño (*Large Language Models*, LLM), que reciben como entrada una secuencia de texto (prompt) y producen como salida otra secuencia de texto, que puede ser lenguaje natural, código fuente, instrucciones, etc.

Estos modelos:

- no "saben" de por sí nada sobre Git, IDEs ni pipelines de CI/CD; solo operan sobre texto;
- no tienen noción intrínseca de proyecto, archivo o repositorio, más allá de lo que se les provee en el prompt o en su contexto de entrada;
- son agnósticos del entorno de ejecución: la misma llamada al modelo puede provenir de un editor, de una API, de un chatbot o de una herramienta de línea de comandos.

En otras palabras, un modelo fundacional es el "motor estadístico" que genera y transforma texto. La experiencia de pair programming con IA emerge cuando este motor se integra en un entorno de desarrollo concreto, se le proporciona contexto del código y se definen dinámicas de interacción humano–máquina específicas [2], [5].

XIII-B. ¿Qué es una herramienta de desarrollo asistida por IA?

Por contraste, una **herramienta** de desarrollo asistida por IA es un producto de software que integra uno o varios modelos fundacionales dentro de un flujo de trabajo concreto. Ejemplos típicos son plugins de IDE como GitHub Copilot, editores "AI–native" como Cursor, entornos centrados en código como Claude Code, o extensiones como Antigravity. Estas herramientas:

- mantienen estado sobre el proyecto (archivos, árbol de directorios, historial de cambios);
- deciden qué fragmentos de código o contexto enviar al modelo y cómo presentar las respuestas;
- orquestan múltiples llamadas a modelos y servicios (búsquedas, herramientas internas, repositorios, etc.);
- ofrecen modos de uso específicos (completado en línea, chat contextual, refactorización de archivos, agentes que ejecutan acciones sobre el repositorio, entre otros).

La Tabla IX resume la distinción conceptual entre modelo y herramienta desde la perspectiva de la programación en parejas con IA.

Cuadro IX
DIFERENCIAS CONCEPTUALES ENTRE MODELO FUNDACIONAL Y HERRAMIENTA DE DESARROLLO ASISTIDA POR IA

Dimensión	**Modelo (GPT, Claude, Llama)**	**Herramienta (Cursor, Claude Code, Antigravity)**
Naturaleza	Motor de generación de texto/código	Producto de software que usa uno o varios modelos
Unidad de trabajo	Prompt de texto y respuesta	Proyecto, repositorio, archivos, tareas de desarrollo
Contexto	Solo lo que cabe en la ventana de contexto	Combina contexto del modelo con conocimiento del IDE y del sistema de archivos
Responsabilidad principal	Predecir la siguiente secuencia de tokens	Orquestar flujos de trabajo de desarrollo y experiencia de usuario
Relación con el dev	"Cerebro" estadístico	Interfaz de pair programming y automatización

Distinguir entre modelo y herramienta es clave para tomar decisiones arquitectónicas: por ejemplo, una organización puede usar el mismo modelo subyacente (p. ej. GPT-4) a través de distintas herramientas (Copilot, un bot interno, un editor personalizado) según el caso de uso.

XIII-C. Herramientas locales vs. herramientas remotas

Otra distinción relevante es entre **herramientas locales** y **herramientas remotas**. Una herramienta local es aquella cuyo componente principal de ejecución (incluyendo, en algunos casos, el modelo) corre en la propia máquina del desarrollador o dentro de la red interna de la organización. En cambio, una herramienta remota se apoya en servicios en la nube, típicamente a través de APIs expuestas por proveedores externos.

En el contexto de pair programming con IA, esto tiene implicaciones importantes:

- **Herramientas locales**: reducen preocupaciones de confidencialidad (el código no sale de la máquina o de la red corporativa), pueden ofrecer latencias más predecibles en entornos sin buena conectividad y permiten mayor control sobre la configuración del modelo. Suelen requerir más recursos de hardware y mantenimiento (GPU, actualizaciones, etc.).

- **Herramientas remotas**: delegan el cómputo pesado al proveedor de IA, permiten acceder rápidamente a los modelos más recientes y suelen integrarse fácilmente mediante plugins de IDE o APIs. Sin embargo, implican enviar fragmentos de código y contexto a servicios externos, lo que exige políticas claras de seguridad, privacidad y cumplimiento normativo [5], [7].

En la práctica, muchas organizaciones adoptan esquemas híbridos donde ciertas operaciones (por ejemplo, análisis estático y tests) se ejecutan localmente, mientras que las tareas de generación de código y explicación de cambios se apoyan en modelos remotos de última generación.

XIII-D. Modos de interacción: chat, completado y agentes

Finalmente, es útil diferenciar los principales **modos de interacción** que ofrecen las herramientas de programación asistida por IA, pues condicionan el tipo de pair programming que se puede realizar:

- **Modo completado (*inline completion*)**: el modelo sugiere una o varias líneas de código directamente en el editor, a medida que el desarrollador escribe. Es el modo clásico de herramientas como Copilot en su versión inicial [2]. Favorece un flujo muy rápido y "ligero" de pair programming, centrado en ahorrar tecleo y repetir patrones.

- **Modo chat**: la interacción ocurre en una ventana de conversación integrada al IDE o en una interfaz web; el desarrollador puede hacer preguntas en lenguaje natural sobre el código, pedir explicaciones, solicitar refactorizaciones y generar fragmentos que luego copia o aplica. Herramientas como Claude Code o ChatGPT en modo código explotan este paradigma [5]. Este modo se asemeja más al pair programming conversacional: se discuten decisiones, no solo se completa texto.

- **Modo agente**: la herramienta no solo genera código, sino que también *ejecuta acciones* sobre el proyecto (crear archivos, modificar múltiples ficheros, ejecutar tests, leer logs, etc.). Ejemplos son los "agentes de repositorio" que recorren el código, proponen cambios y los aplican automáticamente si se les da permiso. En este modo, la IA deja de ser únicamente un copiloto de sugerencias para convertirse en un colaborador que ejecuta pasos completos del flujo de desarrollo.

- **Otros modos especializados**: algunas herramientas ofrecen modos de *in-place edit* (seleccionar un bloque de código y pedir una transformación específica), *CLI asistida* (comandos de terminal guiados por IA) o *code review* automático. Todos ellos combinan los mismos elementos básicos: un modelo fundacional, una herramienta que orquesta el contexto y un canal de interacción adaptado a la tarea.

Comprender estas diferencias ayuda a diseñar mejores prácticas de pair programming con IA. No es lo mismo utilizar la IA únicamente como completer de líneas que integrarla en un flujo de trabajo donde el desarrollador conversa con el modelo, delega tareas a un agente y revisa los cambios antes de integrarlos al repositorio. En términos de Arquitectura de Software, elegir modelos, herramientas (locales o remotas) y modos de interacción apropiados forma parte de las decisiones de diseño que determinan cómo se construyen y gobiernan los sistemas asistidos por IA.

XIV. Análisis comparativo de costos: pair programming con IA vs. vibe coding

Al hablar de trabajo con modelos de IA en desarrollo de software, una pregunta práctica es: *¿qué enfoque resulta más costoso?* En esta sección se comparan, de forma cualitativa, los costos asociados al **pair programming con IA** frente al **vibe coding**, separando tres dimensiones principales:

- costo directo de uso del modelo (tokens y poder de cómputo);
- costo de tiempo humano durante el desarrollo;
- costo diferido asociado a calidad del código y deuda técnica.

No se pretende calcular valores monetarios exactos (pues dependen de cada proveedor y modelo), sino ofrecer un marco que permita razonar sobre qué enfoque tiende a ser más costoso en distintos escenarios.

XIV-A. Costos de tokens y poder de cómputo

Desde la perspectiva del modelo, el costo directo suele ser proporcional al número de **tokens procesados** (entrada + salida) y, en algunos casos, al tamaño del modelo utilizado. Si denotamos por C_{tok} el costo por token y por N_{tok} el número total de tokens consumidos en una sesión, el costo directo puede aproximarse como:

$$C_{IA} \approx C_{tok} \times N_{tok}.$$

En **pair programming con IA** estructurado:

- predominan llamadas de *completado en línea* y chats relativamente acotados al contexto inmediato (archivo o módulo en el que se trabaja);
- las interacciones son frecuentes pero pequeñas: muchas sugerencias cortas de código, refactorizaciones puntuales, explicaciones localizadas;
- el desarrollador tiende a editar, podar y reusar prompts, controlando mejor el tamaño de las entradas.

En **vibe coding**, por el contrario:

- es frecuente enviar prompts largos describiendo requisitos, copiando grandes fragmentos de código o incluso resúmenes de múltiples archivos;
- los modos de *agente* tienden a leer y escribir sobre varios ficheros en cada iteración, multiplicando las llamadas al modelo y el tamaño de contexto;
- se generan respuestas extensas (métodos completos, clases, incluso módulos enteros) que incrementan notablemente N_{tok}.

En términos generales, para una misma funcionalidad, **vibe coding suele consumir más tokens y más poder de cómputo** que un flujo disciplinado de pair programming con IA, porque privilegia menos la reutilización de contexto y más la generación de bloques grandes de código. Esta diferencia se amplifica cuando se usan modelos de razonamiento grandes en modo agente, que requieren múltiples rondas de planificación y verificación.

XIV-B. Costos de tiempo humano

Podemos esbozar un costo de tiempo humano como:

$$C_{humano} \approx T_{dev} \times C_{hora},$$

donde T_{dev} es el tiempo del desarrollador y C_{hora} su coste/hora.

En **pair programming con IA**:

- el desarrollador invierte más tiempo en *leer, revisar y refinar* las sugerencias;
- se realizan más decisiones conscientes de diseño, pruebas y ajustes por iteración;
- el tiempo de desarrollo por funcionalidad puede ser mayor que en un enfoque de vibe coding, especialmente en las primeras etapas del prototipo.

En **vibe coding**:

- se reduce el tiempo inicial para obtener algo que "funcione" (código ejecutable, prototipo navegable);
- se sacrifica parte del tiempo de revisión en profundidad: el foco está en probar el resultado, no en entender cada línea;
- para tareas de exploración o demostraciones, T_{dev} suele ser menor que con pair programming más riguroso.

Así, a corto plazo, **vibe coding tiende a minimizar el tiempo humano por funcionalidad**, a costa de delegar más decisiones al modelo y de asumir riesgos que se materializan después.

XIV-C. Costos diferidos: deuda técnica y mantenibilidad

Una tercera dimensión, menos visible pero crítica, es el costo diferido asociado a la **deuda técnica**. Podemos pensar en un costo total aproximado como:

$$C_{total} \approx C_{IA} + C_{humano} + C_{deuda},$$

donde C_{deuda} agrupa el esfuerzo futuro necesario para corregir errores, refactorizar código difícil de mantener y alinear la implementación con la arquitectura prevista.

En **pair programming con IA**:

- al revisar código en el momento, se reducen errores lógicos y decisiones arquitectónicas improvisadas;
- se tiende a mantener principios de diseño (SOLID, separación de capas, pruebas) desde el inicio;
- el valor de C_{deuda} suele ser moderado, especialmente en proyectos de largo plazo.

En **vibe coding**:

- es común aceptar soluciones "suficientemente buenas" sin una revisión detallada de estructura y calidad;
- el código generado puede ser difícil de entender o extender por otros miembros del equipo;
- aparecen más frecuentemente problemas de duplicación, acoplamiento excesivo y falta de pruebas, lo que aumenta el esfuerzo futuro de refactorización.

En proyectos que evolucionan hacia productos mantenibles, **el costo de deuda técnica asociado a vibe coding puede superar con creces cualquier ahorro inicial de tiempo o de tokens**. Por el contrario, un uso disciplinado

de pair programming con IA aumenta algo el esfuerzo inicial, pero reduce la probabilidad de reescrituras masivas o correcciones costosas más adelante.

XIV-D. Resumen comparativo

La Tabla X resume el análisis cualitativo de costos entre ambos enfoques para una misma funcionalidad desarrollada.

Cuadro X
COMPARACIÓN CUALITATIVA DE COSTOS ENTRE PAIR PROGRAMMING CON IA Y VIBE CODING

Dimensión de costo	**Pair programming con IA**	**Vibe coding**
Tokens y cómputo de IA	Consumo moderado y granular; prompts más pequeños, centrados en el archivo actual.	Consumo alto; prompts largos, respuestas masivas y posibles agentes que recorren todo el repositorio.
Tiempo humano inicial	Mayor tiempo de revisión, diseño y pruebas por funcionalidad.	Menor tiempo para llegar a "algo que funciona" (prototipo).
Deuda técnica y retrabajo futuro	Menor; el código tiende a ser más alineado con la arquitectura y principios de diseño.	Mayor; riesgo de reescrituras, bugs ocultos y refactorizaciones complejas.
Costo total en proyectos de largo plazo	Generalmente menor: más caro al inicio, pero más barato en mantenimiento.	Generalmente mayor: barato al iniciar, caro al sostener y escalar.
Adecuación al contexto	Ideal para productos críticos, regulados o de vida larga.	Adecuado para prototipos, experimentos y validación rápida de ideas.

En síntesis, **vibe coding suele ser más costoso en tokens y cómputo, y más barato en tiempo humano inicial**, mientras que **el pair programming con IA tiende a optimizar el costo total en proyectos de larga duración**, al reducir la deuda técnica y mejorar la calidad desde las primeras iteraciones. Desde la perspectiva de Arquitectura de Software, la elección entre ambos enfoques no debería ser dicotómica, sino contextual: vibe coding es útil para explorar y prototipar, mientras que el pair programming con IA proporciona la base para construir sistemas sostenibles y económicamente viables en el tiempo.

XV. CONCLUSIONES Y LECCIONES APRENDIDAS

El análisis realizado muestra que el *pair programming con IA* no es simplemente "usar una herramienta de autocompletado más avanzada", sino un cambio en la forma de pensar y organizar el trabajo de desarrollo. Al tratar a la IA como una pareja de programación—con responsabilidades claras, ciclos de feedback cortos y discusión explícita de decisiones de diseño—el equipo puede obtener beneficios significativos en términos de productividad, calidad del código y aprendizaje continuo. No obstante, estos beneficios solo se materializan cuando existe una intención clara de diseño, revisión y validación humana; la automatización sin criterio aumenta el riesgo de introducir errores sutiles, deuda técnica y soluciones que el equipo no comprende.

En contraste, el *vibe coding* ofrece una propuesta más radical de delegación, especialmente útil para prototipado rápido, generación de esqueleto de aplicaciones o exploración de alternativas de diseño. Sin embargo, el análisis de *Vibe Coding y sus retos*, junto con las matrices de *Atributos de Calidad*, *Tácticas* y *Patrones*, evidencia que este enfoque tiende a incrementar la probabilidad de deuda técnica y a debilitar la trazabilidad de las decisiones de arquitectura si no se acompaña de revisiones rigurosas y refactorización sistemática. En otras palabras, vibe coding puede ser una herramienta poderosa para acelerar el inicio de un proyecto, pero no reemplaza la disciplina de ingeniería necesaria para sostenerlo en el tiempo.

Las *Matrices de análisis* aportan una visión estructurada de estos hallazgos. La matriz de *Principios SOLID vs. Tema* sugiere que la IA puede ser tanto aliada como enemiga de un buen diseño: puede ayudar a aplicar

patrones comunes y a detectar olores de código, pero también incentiva atajos cuando las instrucciones del usuario priorizan velocidad sobre claridad. Las matrices de *Atributos de Calidad*, *Tácticas* y *Patrones* muestran que el pair programming con IA, bien encuadrado, tiende a favorecer mantenibilidad y claridad, mientras que un uso acrítico de vibe coding aumenta la complejidad accidental. La matriz de *Mercado Laboral vs. Tema* refuerza la idea de que los perfiles más demandados combinan competencias técnicas sólidas con habilidades para orquestar sistemas de IA, interpretar sus salidas críticamente y comunicarse efectivamente sobre decisiones asistidas por IA.

Desde la perspectiva de costos, el *Análisis comparativo de costos: pair programming con IA vs. vibe coding* confirma que el ahorro aparente en *Costos de tokens y poder de cómputo* o en *Costos de tiempo humano* puede ser engañoso si no se consideran los *Costos diferidos: deuda técnica y mantenibilidad*. En muchos casos, un enfoque ligeramente más lento pero más guiado (pair programming con IA) resulta, a medio plazo, más económico que generar grandes cantidades de código con vibe coding que luego requieren reescritura o extensas tareas de refactorización. El *Resumen comparativo* ayuda a identificar contextos en los que cada enfoque aporta mayor valor: prototipado, exploración y tareas de bajo riesgo para vibe coding; funcionalidades críticas, componentes nucleares y trabajo de diseño para pair programming con IA.

A partir de este recorrido, se pueden extraer varias lecciones aprendidas. Primero, la IA debe ser integrada en el flujo de trabajo como un colaborador al que se le asignan tareas y se le exigen justificaciones, no como un oráculo infalible. Segundo, la formación de las personas desarrolladoras debe incluir no solo competencias técnicas tradicionales, sino también habilidades para diseñar buenos *prompts*, evaluar la confiabilidad de las respuestas y articular criterios de aceptación que tomen en cuenta principios de diseño, atributos de calidad y contexto de negocio. Tercero, las organizaciones que adopten estas prácticas deberán ajustar sus procesos de revisión de código, gestión de conocimiento y gobernanza técnica para incorporar explícitamente el uso de IA.

En síntesis, el pair programming con IA y el vibe coding no son enfoques excluyentes, sino herramientas complementarias dentro de un mismo ecosistema de desarrollo asistido por IA. Utilizados con criterio, apoyados en principios de ingeniería sólidos y acompañados de una reflexión continua sobre sus impactos en el diseño, la calidad y el mercado laboral, pueden convertirse en catalizadores para construir software más rápido sin sacrificar su sostenibilidad. El reto para los equipos y profesionales no es solo aprender a *usar* estas herramientas, sino aprender a *trabajar con ellas* de manera responsable y estratégica.

REFERENCIAS

[1] M. Golovatenco, "AI Pair Programming Statistics: How Developers Use AI in 2025," Index.dev Blog, 2025, publicado 04 de noviembre de 2025. Disponible en: https://www.index.dev/blog/ai-pair-programming-statistics [Accedido 25/11/2025].

[2] N. Friedman, "Introducing GitHub Copilot: your AI pair programmer," GitHub Blog, 2021, publicado 29 de junio de 2021. Disponible en: https://github.blog/2021-06-29-introducing-github-copilot-ai-pair-programmer/ [Accedido 25/11/2025].

[3] B. B.ºckeler and N. Siessegger, "On Pair Programming," MartinFowler.com, 2020, publicado 15 de enero de 2020. Disponible en: https://martinfowler.com/articles/on-pair-programming.html [Accedido 25/11/2025].

[4] "Programaci'on extrema (secci'on Programaci'on en parejas)," Wikipedia, la enciclopedia libre, 2022, disponible en: https://es.wikipedia.org/wiki/Programaci%C3%B3n_extrema [Accedido 25/11/2025].

[5] Shakers, "IA generativa aplicada: Transformando flujos de trabajo en desarrollo de software," Blog de Shakers, 2023, disponible en: https://blog.shakersworks.com/ia-generativa-aplicada [Accedido 25/11/2025].

[6] ITDO, "Diseño colaborativo con IA: interfaces para cocrear con máquinas," Blog ITDO, 2023, disponible en: https://www.itdo.com/blog/diseno-colaborativo-con-ia-interfaces-para-cocrear-con-maquinas/ [Accedido 25/11/2025].

[7] V. Gopinath, "AI Pair Programming in 2025: The Good, Bad, and Ugly," Builder.io Blog, 2024, disponible en: https://www.builder.io/blog/ai-pair-programming [Accedido 25/11/2025].

[8] Sankalp, "The Evolution of AI-assisted coding features and developer interaction patterns," Blog personal, 2024, publicado 21 de diciembre de 2024. Disponible en: https://sankalp.bearblog.dev/evolution-of-ai-assisted-coding-features-and-developer-interaction-patterns/ [Accedido 25/11/2025].

[9] J. Montes, "En el sector financiero de Argentina, los desarrolladores utilizan la IA para impulsar la innovación," Microsoft News Center Latinoamérica, 2025, publicado el 19 de agosto de 2025. Disponible en: https://news.microsoft.com/latam/features/ia/galicia-naranja-x-github-copilot-es/ [Accedido 25/11/2025].

[10] E. Magrel, "GitHub Copilot speeding up developers work by 30

[11] GitHub, “Measuring the impact of GitHub Copilot,” GitHub Resources, 2023, disponible en: https://resources.github.com/learn/pathways/copilot/essentials/measuring-the-impact-of-github-copilot/ [Accedido 25/11/2025].
[12] A. Gupta, “I Spent 30 Days Pair Programming with AI—Here’s What It Taught Me,” DEV Community Blog, 2024, publicado 18 de enero de 2024. Disponible en: https://dev.to/arpitstack/i-spent-30-days-pair-programming-with-ai-heres-what-it-taught-me-4dal [Accedido 25/11/2025].
[13] C. Gómez Guasca, “Empresas colombianas ya exigen habilidades en IA para contratar, según estudio de Deel,” Portafolio, 2025, publicado el 21 de noviembre de 2025. Disponible en: https://www.portafolio.co/economia/empleo/empresas-colombianas-ya-exigen-habilidades-en-ia-para-contratar-segun-estudio-de-deel-483476 [Accedido 25/11/2025].
[14] Epoch AI, “Epoch Capabilities Index (ECI),” Epoch AI, 2025, disponible en: https://epoch.ai [Accedido 25/11/2025].
[15] Wikipedia, “Vibe coding,” Wikipedia, the free encyclopedia, 2025, disponible en: https://en.wikipedia.org/wiki/Vibe_coding [Accedido 25/11/2025].
[16] S. Teki, “Impact of AI on the 2025 Software Engineering Job Market,” Blog personal, 2025, disponible en: https://www.sundeepteki.org/advice/impact-of-ai-on-the-2025-software-engineering-job-market [Accedido 25/11/2025].