El Efecto Bullet

Deuda Cognitiva y la Ilusión de la Velocidad en la Era de la IA

Autor: Ariel Fabian **Área:** Ingeniería de Software y Psicología Cognitiva **Fecha:** Octubre de 2026

Palabras Clave: *Ingeniería de software con IA, deuda cognitiva, peaje cognitivo, vibe-coding, psicología del ajedrez, agrupamiento (chunking), apuro de tiempo, síndrome de Kotov, pensamiento profiláctico, teoría del programa.*

Keywords: *AI-assisted software engineering, cognitive debt, proof-of-comprehension gate, vibe-coding, chess psychology, chunking, time trouble, Kotov syndrome, prophylactic thinking, theory building.*

RESUMEN

Los agentes de Inteligencia Artificial generan código en segundos. Pero escribir código rápido no es lo mismo que entenderlo. Cuando un equipo acepta código generado sin tomarse el tiempo de leerlo y analizarlo, acumula **deuda cognitiva**: sistemas que funcionan, pero que nadie en el equipo sabe modificar con seguridad. Así, las fallas pueden llegar directo a producción. Sostenemos que el problema no es la herramienta sino el ritmo: una persona necesita tiempo para entender lógica de software y no puede hacerlo a la velocidad con la que un modelo genera texto. Para analizar este problema usamos la psicología del ajedrez, un campo con ochenta años de investigación sobre memoria, decisiones bajo presión de tiempo e interacción humano-máquina. Planteamos que, cuando el ritmo supera la capacidad de lectura, el desarrollador pasa a una revisión superficial y empieza a aprobar cambios que no entiende. Como respuesta, proponemos cinco reglas de trabajo adaptadas del ajedrez. Las dos centrales son la **regla del 10 + 10**, que establece una reserva mínima obligatoria para leer y asimilar el código proporcional al esfuerzo previo de desarrollo, y un **peaje cognitivo antes del commit**, que exige al programador justificar el cambio por sí mismo sin explicaciones predigeridas de la IA. El objetivo es que el equipo entienda lo que construye sin renunciar a la velocidad. Por último, proponemos un diseño experimental para poner a prueba estas reglas en equipos reales.

ABSTRACT

Artificial Intelligence coding agents generate code in seconds. Yet writing code quickly is not the same as understanding it. When a team accepts generated code without taking the time to read and analyze it, it accumulates **cognitive debt**: systems that work today, but that no one on the team knows how to safely modify. As a result, latent flaws can reach production directly. We argue that the problem is not the tool itself, but the pace: a person needs time to assimilate software logic and cannot do so at the speed at which a model generates text. To analyze this problem, we draw on chess psychology—a field with eighty years of experimental research on memory, decision-making under time pressure, and human-machine interaction. We contend that when the pace of generation exceeds reading capacity, developers shift to superficial scanning and begin approving changes they do not understand. In response, we propose five working rules adapted from chess. The two core practices are the **10 + 10 rule**, which establishes a mandatory minimum time reservation to read and assimilate the code proportional to previous development effort, and a **pre-commit cognitive toll**, which requires the developer to justify the change themselves without pre-chewed AI explanations. The goal is for teams to understand what they build without giving up speed. Finally, we outline an experimental design to test these rules in real-world engineering teams.

Introducción

En la ingeniería de software, la escritura de código representa solo una fracción inicial del ciclo de vida de un sistema; el desafío central —y donde radica el costo real del proyecto— reside en su estabilización: la verificación de invariantes, la prueba bajo condiciones de carga, la depuración de casos de borde y la capacidad de mantener el comportamiento predecible a lo largo del tiempo.

En el paradigma tradicional, el ritmo deliberado del desarrollo manual cumplía una función estabilizadora implícita: al requerir tiempo para planificar y estructurar cada componente, el programador construía paralelamente en su mente un modelo operativo del sistema. Como formalizó Peter Naur (1985) en *Programming as Theory Building*, la programación consiste en desarrollar y sostener una teoría viva sobre cómo el software mapea el dominio real; sin esa teoría, el sistema no puede modificarse de forma segura.

La irrupción de la IA generativa rompió este equilibrio: acelera drásticamente la escritura de código, pero compromete la comprensión necesaria para estabilizarlo. La herramienta permite generar código a una velocidad sin precedentes, creando una apariencia inmediata de progreso. Sin embargo, esta rapidez temprana prescinde del trabajo analítico necesario para estabilizar el sistema:

- **Revisiones más superficiales:** Al verse inundados por grandes volúmenes de código artificial, los equipos reducen la rigurosidad de sus revisiones. Investigadores de CMU y Stanford (He & Vasilescu, 2026) registraron en 196.000 Pull Requests que la tasa de revisión humana profunda cayó del 48% al 21%.
- **Dificultad para entender y depurar:** Sin un modelo mental consolidado, el equipo pierde la facultad de intervenir sobre el código ante comportamientos anómalos. Un ensayo experimental controlado con desarrolladores (Shen & Tamkin, 2025) registra una caída de 17 puntos porcentuales en la comprensión conceptual al aprender una librería nueva, concentrándose la mayor brecha precisamente en las tareas de lectura de código y depuración (*debugging*).
- **Desplazamiento del costo a producción:** Al omitirse la estabilización durante el desarrollo, las fallas se transfieren directamente a la fase operativa. Reportes de telemetría de industria (Faros AI, 2026) apuntan a esta misma tendencia: en organizaciones con adopción masiva y desregulada de IA, los cambios aprobados sin revisión previa subieron un 76,3% y los incidentes críticos en producción aumentaron un 125,4% (patrón observacional que coincide con la caída en la revisión profunda documentada por He & Vasilescu).

El resultado es que el tiempo que se ahorra al escribir el código se pierde después en fallas y en **deuda cognitiva** (Osmani, 2026) y en **deuda de intención** (Storey, 2026): la pérdida progresiva del propósito arquitectónico original, donde el equipo mantiene sistemas construidos en pocas horas pero cuya lógica interna nadie comprende con precisión.

Para entender las causas de este problema y buscar una solución, usamos la psicología del ajedrez como analogía. El ajedrez cuenta con más de ochenta años de investigación experimental que estudió cinco fenómenos muy parecidos a los que hoy vemos en el desarrollo de software:

- Cómo almacena información la memoria de trabajo y el reconocimiento de estructuras mediante agrupamiento perceptivo (*chunking*) (De Groot, 1946; Chase & Simon, 1973; Hermans, 2021).
- Qué le ocurre al cálculo mental cuando se reduce drásticamente el tiempo disponible en el reloj (Kotov, 1971; Rowson, 2000).
- La diferencia entre solucionar un problema puntual y mantener un sistema coherente a largo plazo (la tríada de Kaspárov, 2007: material, tiempo y calidad).
- Por qué la reducción del tiempo analítico amplifica la asimetría entre principiantes y expertos (Dreyfus, 1972/1992; Dreyfus & Dreyfus, 1986).
- Cómo reacciona un ser humano cuando se apoya en sistemas que calculan mejor que él (Poulidis et al., 2025; Bilalić et al., 2026).

La analogía se apoya en los mecanismos cognitivos del profesional: la memoria de trabajo retiene de forma activa apenas 4 ± 1 estructuras funcionales (*chunks*, Cowan, 2001; Chase & Simon, 1973), el apuro de tiempo inhibe el pensamiento analítico deliberado (el Sistema 2 deliberativo conceptualizado por Kahneman, 2011) y la delegación continua en la IA atrofia el esfuerzo analítico necesario para aprender.

Al mismo tiempo, delimitamos el alcance de la comparación: el ajedrez es un sistema cerrado, formal y de información perfecta, mientras que la ingeniería de software es un dominio abierto, ambiguo y sociotécnico (Brooks, 1987). Esta diferencia delimita lo que la analogía puede probar: el ajedrez permite aislar mecanismos cognitivos, pero no predice por sí solo la magnitud cuantitativa de sus efectos en un dominio abierto. Sugiere, no obstante, que los riesgos de operar sin modelos mentales compartidos (Naur, 1985) son sustancialmente mayores en software, donde no existe un tablero discreto ni una función de evaluación matemática inmediata en cada cambio. Esa conjetura, por tanto, requiere contrastación empírica propia.

A lo largo de este trabajo articulamos esta correspondencia a través de cinco ejes fundamentales:

1. **La lectura de código:** El código de IA compila y parece correcto, pero es código ajeno. Como el revisor no participó en las decisiones de diseño ni descartó alternativas, enfrentarse a cientos de líneas juntas satura la memoria de trabajo y le impide entender cómo funciona realmente el sistema (Naur, 1985; Hermans, 2021).
2. **Táctica vs. estrategia:** Priorizar el cierre rápido de tickets a costa de descuidar la arquitectura equivale a ganar material a corto plazo mientras se arruina la posición a largo plazo (la tríada de Kaspárov: material, tiempo y calidad).
3. **El Efecto Bullet:** La velocidad de la IA no iguala a principiantes y expertos. Al igual que en el ajedrez a un minuto (ritmo *bullet*), cuando no hay tiempo para analizar en detalle, cada uno depende exclusivamente de lo que ya sabe: el desarrollador experimentado detecta riesgos de inmediato gracias a sus modelos mentales y a su experiencia previa, mientras que el principiante acepta soluciones que no comprende y queda desarmado cuando el sistema falla en producción.
4. **El apuro de tiempo:** La presión por entregar rápido acorta los tiempos de revisión. Frente a cientos de líneas de código que no escribió, el desarrollador experimenta una dinámica análoga al apuro de tiempo ajedrecístico (Rowson, 2000; Kotov, 1971): ante la falta de tiempo, deja de lado el análisis crítico y aprueba sin revisar (*silent approval*). En los ensayos de Shaw & Nave (2026), bajo presión de tiempo las chances relativas de aceptar sin cuestionar respuestas erróneas de la IA se multiplicaron por más de tres (Odds Ratio = 3,19).
5. **Los límites del *vibe-coding*:** Programar delegando todo en la IA sin reflexionar perjudica el aprendizaje y la autonomía técnica. En un ensayo experimental, Poulidis et al. (2025) observaron que el acceso libre a la ayuda de la IA reduce el aprendizaje a largo plazo porque elimina el esfuerzo necesario para consolidar conceptos. En una escala mucho mayor, el análisis de más de 11,6 millones de decisiones tomadas por ajedrecistas de élite (Bilalić et al., 2026) apunta a una conclusión parecida: la potencia de cálculo externa no transfiere comprensión al ser humano de forma automática. Entender un sistema sigue demandando tiempo y análisis propio.

Para abordar estos problemas, proponemos un marco de trabajo basado en cinco principios adaptados del ajedrez:

- **Control de ritmo:** desarrollo reflexivo y pausado en componentes críticos.
- **Incremento de reloj:** reserva obligatoria de tiempo para entender el código (la regla del 10 + 10).
- **Evaluación socrática:** validación activa de la comprensión antes del commit (el peaje cognitivo).
- **Pensamiento profiláctico:** auditoría preventiva de fallas y contingencias antes del despliegue.
- **El modelo centauro:** primacía del método de trabajo por encima de la potencia de la herramienta.

Organización del documento

Las Secciones 1 a 5 desarrollan el diagnóstico cognitivo sobre cada uno de los ejes analizados. La Sección 6 formaliza las cinco reglas de trabajo, detalla el funcionamiento de la herramienta y describe el protocolo experimental para su validación futura. Finalmente, cerramos con nuestras conclusiones.

---

1. La Lectura de Código y la Saturación Mental

1.1. El cansancio al revisar código de IA

Cuando un desarrollador experimentado revisa un Pull Request generado por IA, se cansa mucho más rápido que cuando revisa código escrito por un compañero de equipo. Aunque el código compila y pasa pruebas unitarias básicas, entenderlo requiere un esfuerzo mental agotador.

1.2. La causa cognitiva: el agrupamiento

En un experimento fundacional, el psicólogo y ajedrecista Adrian de Groot (1946/1965) mostró posiciones de partidas reales durante cinco segundos a grandes maestros y a jugadores aficionados. Los maestros reconstruyeron con precisión más del 90% del tablero; los jugadores menos experimentados, en cambio, cayeron a una fracción reducida de esa cifra (entre el 25% y el 50% según su nivel de juego). De Groot comprobó que los maestros no calculaban más rápido ni analizaban muchas más jugadas hacia adelante, sino que entendían la posición de un solo golpe de vista.

Casi tres décadas después, William Chase y Herbert Simon (1973) completaron el diseño experimental con una variante decisiva: ubicaron las piezas sobre el tablero de forma totalmente aleatoria. En esta condición artificial, la ventaja de los grandes maestros desapareció: recordaron prácticamente lo mismo que los principiantes (entre 3 y 4 piezas aisladas, Chase & Simon, 1973).

Chase y Simon demostraron así el mecanismo del **agrupamiento (*chunking*)**: la memoria de trabajo —el espacio mental transitorio donde retenemos y procesamos información activa (Baddeley, 1992)— tiene una capacidad de apenas 4 ± 1 elementos en simultáneo (Cowan, 2001). Por eso, no puede procesar grandes volúmenes de información nueva a la vez. Los maestros no memorizan piezas aisladas, sino estructuras con sentido (*chunks*) consolidadas en su memoria de largo plazo tras miles de horas de práctica ("un enroque debilitado", "una cadena de peones central"). Cada una de estas estructuras ocupa un único lugar en su memoria de trabajo. Cuando la posición carece de sentido lógico —como en el tablero con piezas ubicadas al azar—, la experiencia previa no sirve y la memoria de trabajo se satura de inmediato.

1.3. El código ajeno y la ilusión de fluidez

En *The Programmer's Brain*, Felienne Hermans explica que la lectura de software se apoya en el mismo principio de agrupamiento (*chunking*): un programador experimentado no lee el código instrucción por instrucción, sino que reconoce patrones conocidos de inmediato (por ejemplo, una búsqueda o una validación de permisos). Esto libera espacio en su memoria de trabajo para concentrarse en la lógica del negocio.

No obstante, el código generado por IA altera este proceso por tres razones:

- **La apariencia de corrección:** A diferencia del experimento de Chase y Simon con piezas ubicadas al azar en el tablero, el código de la IA compila, tiene sangría prolija y parece correcto a primera vista. En términos del modelo clásico de comprensión de programas de Nancy Pennington (1987), la IA entrega un **modelo de programa** (*Program Model*) superficialmente impecable —sintaxis limpia y flujo de control legible—, pero priva al programador de construir el **modelo de situación** (*Situation Model*): la comprensión de cómo interactúa ese código con las reglas del dominio real. Quien lo revisa se encuentra

ante **código ajeno**: no tomó las decisiones de diseño, no descartó alternativas y no construyó la *Teoría del Programa* (Naur, 1985).

- **El esfuerzo de revisión:** La presentación ordenada induce la llamada **ilusión de fluidez** (Bjork, 1994): como el código se lee fácil, parece bien resuelto, pero exige un esfuerzo mental deliberado para verificar si realmente cumple con las reglas del negocio, la concurrencia o los casos límite.
- **La asimetría de tiempos:** Escribir 500 líneas de código le demandaba a un desarrollador varios días de diseño mental. La IA genera ese mismo volumen en un par de segundos. El revisor recibe de golpe una gran cantidad de código que no concibió; en pocos minutos su memoria de trabajo se satura por sobrecarga cognitiva (Sweller, 1988; Cowan, 2001) y el cambio se aprueba sin una revisión profunda.

---

2. Táctica vs. Estrategia

2.1. El error de optimizar solo lo inmediato

Un agente de IA es excelente resolviendo problemas puntuales: si se le pide un endpoint o una consulta SQL para un formulario, lo genera en segundos. El ticket se cierra rápido y el equipo siente que avanza a gran velocidad.

A pesar de ello, el agente no ve el panorama completo:

- No sabe que esa consulta bloqueará una tabla crítica en horas de alta concurrencia.
- No sabe que está duplicando lógica que ya existía en otro módulo.
- No contempla cómo evolucionará el sistema cuando cambie una normativa impositiva dentro de seis meses.

2.2. La lección del ajedrez

En ajedrez, la diferencia entre táctica y estrategia es clara:

- **Táctica:** Resolver una situación inmediata mediante el cálculo a corto plazo (responder a una exigencia del momento o lograr un objetivo puntual).
- **Estrategia:** Planificar a largo plazo la solidez y la coordinación general de la posición.

Garry Kaspárov (2007) cita una frase histórica de Savielly Tartakower:

> *"La táctica consiste en saber qué hacer **cuando hay algo que hacer**; la estrategia consiste en saber qué hacer **cuando no hay nada que hacer**."*

Los modelos generativos tienden a optimizar lo que el prompt o el contexto inmediato especifica, careciendo de acceso autónomo a las reglas no escritas del negocio, a las dependencias ocultas y a la evolución histórica de la arquitectura.

Kaspárov define además que todo proyecto depende de tres elementos inseparables:

1. **Material:** En software, las líneas de código generadas y los tickets cerrados (lo fácil de medir).
2. **Tiempo:** La velocidad de entrega.
3. **Calidad:** La solidez de la arquitectura y la comprensión del equipo.

Con la incorporación de los agentes de IA, surge la tentación natural de priorizar más material en menos tiempo sin advertir el impacto silencioso sobre la calidad: acumular líneas de código y tickets cerrados pierde valor si el sistema se vuelve frágil y el equipo deja de comprender cómo funciona.

---

3. El Efecto Bullet

3.1. La promesa del junior acelerado

En la industria suele argumentarse que la IA democratiza la pericia técnica: un ensayo controlado sobre desarrollo de software con GitHub Copilot (Peng et al., 2023) sugiere que los programadores novatos o de menor experiencia relativa son quienes registran los mayores incrementos porcentuales en velocidad de escritura y resolución inicial de tareas (un patrón documentado también en tareas cognitivas generales; Brynjolfsson et al., 2023). A la inversa, en tareas complejas sobre repositorios maduros con ingenieros experimentados, un ensayo controlado preliminar con herramientas de inicios de 2025 (Becker et al., 2025) observó que, en ese contexto específico sobre repositorios maduros, el uso de asistentes no aceleró a los expertos e incluso incrementó el tiempo de resolución en un 19%, documentando una posible ilusión de velocidad: los desarrolladores estimaban haber sido un 20% más rápidos cuando en la práctica fueron más lentos. A partir de estos datos, suele asumirse apresuradamente que la rapidez para generar código compensa la falta de trayectoria o de conocimientos profundos.

Sin embargo, esta aparente equiparación confunde la **velocidad de emisión sintáctica** con la **comprensión conceptual del sistema**. La herramienta acelera la escritura superficial, pero priva al principiante de la fricción analítica con el código —la fase formativa donde se cometen errores, se depuran fallas y se construyen los esquemas mentales descritos por Dreyfus y Dreyfus (1986)—, amenazando la transmisión de pericia y la formación de los futuros ingenieros senior.

3.2. Lo que enseña el ritmo bullet

Es como creer que un aficionado al ajedrez, con una computadora veloz y un mouse rápido, puede ganarle a un Gran Maestro jugando a un minuto por partida (ritmo *bullet*).

El modelo de Dreyfus y Dreyfus (1986) sobre cómo adquirimos destrezas explica por qué esto no ocurre: cuando el tiempo de reloj se reduce a unos pocos segundos, el experto mantiene una ventaja enorme porque reconoce situaciones críticas de inmediato gracias a su experiencia previa. El principiante, en cambio, necesita tiempo para recordar y aplicar reglas paso a paso. Sin tiempo, no puede razonar. Dreyfus (1972/1992) distinguió entre dos formas de conocimiento:

- **El principiante** opera mediante reglas conscientes (*knowing-that*): necesita tiempo para pensar cada paso. A un segundo por jugada entra en pánico y comete errores graves.
- **El experto** actúa por reconocimiento tácito de patrones (*knowing-how*): identifica de golpe qué pasa en el tablero tras años de práctica, sin necesidad de calcular cada regla desde cero.

En el desarrollo de software asistido por IA ocurre algo parecido, con riesgos claros para ambos perfiles:

A. El riesgo para el principiante:
Cuando la IA genera cientos de líneas en un instante, el desarrollador experimentado detecta rápido si hay problemas de arquitectura o de concurrencia gracias a su experiencia. El principiante no tiene esos esquemas mentales: ve que el código compila, pasa dos pruebas básicas y asume que está bien. El resultado es la aceptación pasiva (rendición cognitiva): se incorporan al sistema módulos que nadie en el equipo entiende a fondo y que serán imposibles de depurar cuando fallen en producción.

B. El riesgo para el programador experimentado:
El problema de la velocidad no afecta únicamente al novato. En ajedrez, los ritmos rápidos ponen a prueba el reflejo táctico inmediato, pero la comprensión profunda del juego se apoya en el ajedrez clásico, donde hay tiempo para calcular variantes. Como señalan los estudios sobre práctica deliberada (Ericsson et al., 1993), depender exclusivamente de respuestas automáticas termina por consolidar hábitos superficiales y debilita la capacidad de análisis crítico.

Si sometemos a un programador senior a la presión constante de revisar y aprobar código de IA en segundos, el cerebro se acostumbra a validar sintaxis al vuelo en lugar de pensar el diseño estructural, los contratos entre módulos y la tolerancia a fallos.

4. El Apuro de Tiempo

4.1. Aprobar sin revisar

El estudio de He & Vasilescu (2026) constató que cuando el volumen de código generado por IA desborda al equipo, la tasa de revisión profunda cae a menos de la mitad (del 48% al 21%). Se generalizan así las aprobaciones sin revisión real (*silent approval*), que en el día a día suelen reducirse a mirar el código por arriba durante un par de minutos.

4.2. El síndrome de Kotov y el sesgo de automatización

Este comportamiento refleja una dinámica análoga a la descrita en el ajedrez bajo **apuro de tiempo** y el **síndrome de Kotov (1971)**:

- Alexander Kotov describió al jugador que analiza una variante, duda; pasa a otra, vuelve a dudar; consume casi todo su tiempo en el reloj y, cuando le quedan apenas segundos para que se le caiga la aguja (en los relojes mecánicos de la época, la caída física de la aguja marcaba que el tiempo se había terminado y la partida estaba perdida), entra en pánico y juega lo primero que se le ocurre, cometiendo un error fatal.
- Jonathan Rowson (2000), en su análisis filosófico del error ajedrecístico, argumenta que el apuro de tiempo no constituye un mero problema cronométrico, sino una falla psicológica en la gestión de la duda: el jugador consume su tiempo en vacilaciones improductivas y, al límite del reloj, termina ejecutando jugadas impulsivas sin cálculo deliberado.

Si bien en ajedrez el síndrome de Kotov ilustra cómo el apuro de tiempo induce errores fatales por vacilación y pánico, en el desarrollo de software opera como una **analogía ilustrativa del colapso bajo presión de reloj**. El mecanismo cognitivo subyacente presenta una diferencia decisiva: el programador no juega a solas, sino que recurre al **sesgo de automatización (*automation bias*)** (Parasuraman et al., 2000), pasando del análisis deliberado a un sub-análisis superficial donde, para escapar del agotamiento cognitivo y cumplir con los plazos, suspende el juicio crítico y asume pasivamente que la propuesta de la IA es correcta.

En The Wharton School, Steven Shaw y Gideon Nave (2026) comprobaron este colapso en tareas de razonamiento asistidas por modelos de lenguaje: al imponer **presión de tiempo**, las **probabilidades de aceptar sin cuestionar respuestas erróneas de la IA se multiplicaron por más de tres** (Odds Ratio = 3,19), mientras que la verificación humana crítica cayó del 19,7% al 6,2%.

5. Los Límites del Vibe-Coding

5.1. Programar por "vibras"

Recientemente se popularizó el término *vibe-coding*: la práctica de construir software pidiéndole código a la IA y aceptando sus sugerencias mientras la aplicación parezca funcionar, sin detenerse a diseñar la arquitectura, consultar documentación ni verificar qué ocurre por debajo.

5.2. El costo de la asistencia continua

A. LA PÉRDIDA DEL ESFUERZO

Un estudio experimental de doce semanas realizado en The Wharton School con más de doscientos alumnos de ajedrez (Poulidis, Bastani y Bastani, 2025) comparó dos métodos de uso de la IA:

- **Grupo con IA libre:** Tenían un botón para pedir la solución a la IA cada vez que se trababan.
- **Grupo con IA regulada:** La plataforma bloqueaba la ayuda inmediata y los obligaba a intentar resolver el problema por su cuenta antes de mostrarles la respuesta.
- **El resultado al evaluarlos sin asistencia externa:** Los alumnos con ayuda libre mejoraron apenas un **30%**, mientras que los alumnos con ayuda regulada **mejoraron un 64% (más del doble)**.
- **La explicación:** Tener la respuesta a un clic de distancia destruye el esfuerzo necesario para consolidar conocimientos, ese período incómodo pero imprescindible donde el cerebro ensaya hipótesis, se equivoca y construye esquemas en la memoria de largo plazo. En el desarrollo de software, la adopción indiscriminada del *vibe-coding* corre el riesgo de atrofiar este mismo esfuerzo analítico.

B. DOMINIOS CERRADOS FRENTE A DOMINIOS ABIERTOS

Suele argumentarse que las nuevas generaciones alcanzan altos niveles de maestría en ajedrez a edades tempranas apoyándose casi exclusivamente en el entrenamiento algorítmico continuo en pantallas, prescindiendo del estudio formal de la teoría clásica. Sin embargo, trasladar este fenómeno formativo a la ingeniería de software constituye una extrapolación apresurada entre dos dominios de naturaleza cognitiva divergente:

- **El ajedrez como dominio cerrado:** Posee reglas formales invariables, información perfecta y un espacio discreto de 64 casillas. En este entorno, la exposición masiva a miles de patrones tácticos entrena de forma automática el reconocimiento visual, permitiendo resolver problemas locales sin necesidad de elaborar una teoría formal exhaustiva.
- **El software como dominio abierto:** La ingeniería de software no es un acertijo determinista ni cerrado. Los requerimientos de negocio son ambiguos, las regulaciones externas cambian, la concurrencia introduce comportamientos no deterministas y el sistema interactúa con entornos imperfectos. Como señaló **Fred Brooks (1987)**, esta complejidad es esencial e irreducible: no puede resolverse mediante mero reconocimiento de patrones sintácticos, sino que exige abstracción, modelado conceptual y la formulación explícita de teorías de dominio (**Peter Naur, 1985**).
- Incluso en el ajedrez, el reconocimiento intuitivo de patrones encuentra un límite estricto: resulta eficaz para partidas a ritmo relámpago, pero resulta insuficiente para la alta competencia a ritmo clásico, donde la deliberación estratégica de largo plazo y el dominio de los principios posicionales siguen siendo indispensables.

C. EL LÍMITE DE ASIMILACIÓN HUMANA

Un macroestudio publicado en el *British Journal of Psychology* analizó **más de 11,6 millones de decisiones tomadas por ajedrecistas de élite a lo largo de 40 años** (Bilalić et al., 2026). Los investigadores comprobaron que la llegada de motores de IA con un nivel muy superior al humano, como AlphaZero, no produjo ningún salto brusco en la precisión de los jugadores humanos: la mejora fue de apenas un 2% en cuatro décadas.

Sin embargo, investigaciones recientes muestran que los ajedrecistas sí pueden aprender de sistemas como AlphaZero, pero solo cuando las jugadas de la máquina se traducen a conceptos comprensibles para el ser humano (Schut et al., 2025). Esto refuerza nuestra idea central: ver lo que genera la IA no transmite conocimiento automáticamente; entenderlo exige frenar y hacer el esfuerzo de analizar qué hay detrás de cada decisión.

No pretendemos trasladar el resultado de Bilalić et al. (2026) de forma literal al código fuente, pero el dato deja una conclusión clara: la mente humana tiene sus propios tiempos para incorporar conocimientos. Que un modelo genere cientos de líneas de código en un par de segundos no hace que el programador las entienda más rápido. Comprender un sistema, revisarlo y hacerse cargo de él sigue demandando tiempo y análisis.

---

6. Principios para una Velocidad Sostenible

Para resolver este problema y aprovechar la potencia de la IA sin perder el control sobre el sistema, formalizamos cinco principios de trabajo adaptados del ajedrez:

Principio en Ajedrez	Regla Práctica en Software
1. Control de ritmo	En componentes críticos (bases de datos, lógica fiscal y reglas de negocio), el desarrollo debe ser reflexivo. No se debe medir la productividad por cantidad de tickets cerrados o líneas generadas.
2. Incremento de reloj (Regla del 10 + 10)	Se establece una reserva mínima obligatoria para leer y asimilar el código, calculada sobre el esfuerzo previo de desarrollo manual para respetar el ritmo de inspección.
3. Evaluación socrática (Peaje cognitivo)	Antes del *commit* en código crítico, la IA frena la generación de código e interroga al programador: este debe justificar las decisiones de diseño e invariantes por su cuenta, sin explicaciones predigeridas.
4. Pensamiento profiláctico	No alcanza con que el código compile y pase los tests: antes del despliegue hay que realizar una auditoría preventiva de fallas (concurrencia, datos inconsistentes o casos límite).
5. Modelo centauro (Ley de Kaspárov)	El proceso de trabajo importa más que la herramienta: un equipo con un método riguroso de revisión asistido por IA debería producir software más seguro y robusto que expertos usando IA sin control.

6.1. Control de ritmo en componentes críticos

FUNDAMENTO EN AJEDREZ

Aunque el ajedrez rápido y *bullet* tienen sus propios campeonatos mundiales, ningún Gran Maestro se prepara jugando a un minuto cuando tiene que competir a ritmo clásico. La razón es simple: el juego ultra rápido acostumbra a la mente a reaccionar por reflejo y a conformarse con soluciones superficiales. En el ajedrez clásico, donde un solo descuido cuesta la partida, las decisiones de fondo exigen cálculo profundo, paciencia y tiempo para pensar.

EL PROBLEMA EN EL DESARROLLO DE SOFTWARE

Ante la rapidez de los nuevos agentes de IA, es habitual que en la industria se intente aplicar el mismo ritmo de entrega a todos los componentes del sistema. Sin embargo, evaluar el avance únicamente por la velocidad de cierre de tickets o el volumen de código generado lleva, casi sin notarlo, a trabajar siempre a ritmo *bullet*.

Este enfoque pasa por alto la criticidad del código:

- Un script descartable de migración o un prototipo de interfaz gráfica toleran un desarrollo rápido y exploratorio.
- En cambio, el núcleo del sistema —motores de facturación, lógica fiscal, modelos relacionales de base de datos y algoritmos transaccionales concurrentes— no puede construirse a las apuradas.

Cuando el código crítico se genera en minutos y se aprueba sin un análisis pausado, el equipo ahorra tiempo al principio, pero siembra fallas silenciosas que más adelante explotan en producción: bloqueos de base de datos, corrupción de datos o comportamientos impredecibles.

LA REGLA DE TRABAJO

Para evitar esta degradación, el equipo debe aplicar una política estricta de control de ritmo basada en dos medidas:

1. **Clasificación explícita de componentes:** Antes de iniciar el desarrollo, el equipo debe clasificar los módulos del sistema en dos categorías:
   - *Módulos periféricos o exploratorios:* Permiten asistencia intensiva de IA a ritmo acelerado.
   - *Módulos de misión crítica:* Tienen vedado el desarrollo apresurado. La generación de código con IA puede utilizarse para explorar alternativas sintácticas, pero la arquitectura y la lógica transaccional deben diseñarse a ritmo clásico (con un análisis riguroso de invariantes y precondiciones).
2. **Prohibición de métricas de velocidad en módulos críticos:** En los componentes centrales queda prohibido medir el rendimiento de los programadores por cantidad de Pull Requests aprobados, tickets cerrados o líneas generadas por jornada. La productividad en estos módulos se mide exclusivamente por la estabilidad del código en producción y la claridad de los modelos mentales del equipo.

6.2. Incremento de reloj: la regla del 10 + 10

FUNDAMENTO EN AJEDREZ

En 1988, Bobby Fischer patentó el reloj con incremento (habitualmente 30 segundos adicionales tras cada jugada) para evitar que las partidas se desvirtuaran por el apuro de tiempo, asegurando que la partida se defina por la calidad de las ideas y no por una carrera física contra el reloj.

EL PROBLEMA EN EL DESARROLLO DE SOFTWARE

Al incorporar agentes de IA en el desarrollo de software, aparece una inercia comprensible hacia la reinversión inmediata del tiempo: si la herramienta permite escribir en 4 horas lo que antes llevaba 40 horas manuales, lo más intuitivo en la planificación es destinar las 36 horas restantes a tomar de inmediato la siguiente tarea.

Esta práctica ignora décadas de investigación sobre revisión e inspección de software: la investigación empírica ha demostrado que el rol esencial del *code review* trasciende la búsqueda mecánica de defectos; su verdadero valor reside en la transferencia de conocimiento y el entendimiento compartido del sistema (Bacchelli & Bird, 2013). Sin embargo, esta comprensión se desmorona cuando el ritmo de lectura supera los límites cognitivos humanos:

- **El límite de velocidad de revisión (Cohen et al., 2006 / Cisco Systems):** En un estudio sobre 2.500 revisiones y 3,2 millones de líneas de código en Cisco Systems, Cohen y sus colaboradores observaron que los revisores que leyeron a menos de **400 líneas por hora (LOC/h)** detectaron muchos más defectos que el promedio. En cambio, por encima de las **450–500 LOC/h**, la detección de errores cayó drásticamente (el 87% de las revisiones por encima de ese umbral detectó menos defectos que la media), lo que suele atribuirse al paso de un análisis deliberado a un escaneo superficial. El estudio recomendó además no superar los 60 minutos seguidos de revisión y limitar los lotes a un rango de **200 a 400 líneas por sesión**.
- **El ritmo de inspección profunda (Fagan, 1976):** En las inspecciones formales de IBM, Michael Fagan documentó que analizar lógica compleja requiere ritmos pausados de **100 a 150 LOC/h** (en desarrollo de aplicaciones suele situarse entre **150 y 200 LOC/h**).
- **La densidad de la lógica:** No es lo mismo revisar 200 líneas de código repetitivo o declarativo (como modelos o maquetación) que 50 líneas de algoritmos concurrentes. El ritmo de lectura debe medirse sobre

líneas de lógica de negocio real, no sobre el conteo ciego de texto.

Suele argumentarse que la solución a esta sobrecarga es dividir el código en Pull Requests pequeños. Si bien acotar el tamaño del lote es una buena práctica recomendada por la literatura (Cohen et al., 2006), en la era de la IA el problema crítico deja de ser el tamaño y pasa a ser el **ritmo de llegada**: recibir veinte PRs pequeños al día puede generar una fatiga similar y desembocar en la misma aprobación pasiva. La regla del 10 + 10 no compite con los PRs pequeños; complementa el tamaño del lote fijando un tope al ritmo de trabajo.

Si el tiempo ahorrado por la IA se reinvierte en generar más código sin respetar este tiempo de lectura, la memoria de trabajo se satura, el cambio se aprueba sin entenderlo y las fallas latentes pasan directo a producción.

LA REGLA DE TRABAJO: LA HEURÍSTICA DEL 10 + 10

En alusión conceptual a los controles con incremento del ajedrez (donde cada movimiento garantiza tiempo adicional para evaluar el tablero), formulamos para la planificación de software la **regla del 10 + 10**. Esta pauta opera como una **heurística de estimación para la gestión de sprints**, calculada sobre el tiempo que habitualmente insumía desarrollar la funcionalidad de forma manual (T_{manual}):

- Si una funcionalidad de complejidad media llevaba tradicionalmente una semana laboral de desarrollo manual ($\boldsymbol{T_{\text{manual}}}$ **= 40 horas**), la IA permite comprimir la escritura de código al **10% del tiempo manual de referencia (4 horas de generación asistida)**.
- La regla establece que debe reservarse de forma obligatoria, **como cota mínima**, al menos otro **10% del tiempo manual de referencia (otras 4 horas)** asignado con exclusividad a leer, inspeccionar y asimilar el código generado.
- En total, en este ejemplo de desarrollo, el equipo completa la tarea en **8 horas** (10% + 10% = 20% del esfuerzo previo). A modo ilustrativo, este esquema plantea que es viable acelerar la entrega técnica sin sacrificar la comprensión del sistema antes del despliegue.

Un desarrollador responsable podría intentar tomarse este tiempo de asimilación por iniciativa propia. Ahora bien, si las métricas de seguimiento se mantienen atadas a los esquemas tradicionales de cantidad de tickets cerrados, quien frena a estudiar el código puede parecer menos veloz que quien avanza sin revisar en profundidad. Por eso, esta reserva de tiempo funciona mejor cuando se acuerda como una pauta explícita del equipo y no queda librado solo al esfuerzo individual.

Para coordinar los tiempos del equipo, el marco articula dos niveles operativos y una pauta de resolución:

1. **Nivel Macro (Planificación y Sprints):** La regla del **10 + 10** se aplica en la estimación global del sprint, garantizando que las horas ganadas con la IA no se comprometan en abrir más tareas sin tiempo de asimilación.
2. **Nivel Micro (Auditoría del commit en el IDE):** Al momento de enviar un cambio concreto, la herramienta de Peaje Cognitivo audita el código real aplicando las tasas de inspección de la literatura (100 a 200 LOC/h para lógica densa, según Fagan y Cohen) y ponderando los puntos de decisión críticos agregados.
3. **Criterio de resolución y prioridad operativa:** Si un commit puntual es demasiado extenso o complejo para asimilarse en una sola sesión (por ejemplo, ante 800 líneas generadas de golpe), no se puede aprobar de forma continua: el sistema exige dividir el cambio en unidades más chicas o pausar la revisión en bloques antes de permitir el commit. En caso de discrepancia, la auditoría a nivel micro en el IDE tiene prioridad operativa sobre la estimación macro del sprint: si un cambio puntual concentra una densidad lógica inusitada, el tiempo real de asimilación debe ajustarse a las líneas efectivas de lógica de negocio (Fagan, 1976; Cohen et al., 2006) y no a un porcentaje administrativo fijo.

6.3. Evaluación socrática: el peaje cognitivo pre-commit

FUNDAMENTO EN AJEDREZ

En las competencias oficiales de ajedrez, el uso de computadoras está terminantemente prohibido. El maestro se sienta a solas frente al tablero y debe justificar cada jugada con su propio análisis y cálculo deliberado. Toda la preparación previa asistida por motores de ajedrez resulta inútil si el jugador no internalizó los planes estratégicos y la lógica posicional: quien se limita a memorizar jugadas sugeridas por el motor sin comprender sus fundamentos colapsa ante la primera respuesta imprevista del rival fuera del repertorio estudiado.

EL PROBLEMA EN EL DESARROLLO DE SOFTWARE

La revisión tradicional de código en la era de la IA padece la trampa de la revisión pasiva y la ilusión de fluidez (Bjork, 1994). Cuando un desarrollador observa en el entorno de desarrollo cientos de líneas de código generadas por la IA, la presentación ordenada y la sintaxis limpia le generan una falsa sensación de entendimiento (*fluency illusion*). El cerebro confunde la facilidad de lectura superficial con la comprensión profunda del problema.

En un estudio preliminar del MIT Media Lab con registro electroencefalográfico (EEG) sobre tareas de redacción asistida por IA (*"Your Brain on ChatGPT"*), Kosmyna et al. (2025, con una muestra de 54 participantes) observaron indicios de que la conectividad neuronal y el compromiso cognitivo decaen en función del grado de asistencia del modelo de lenguaje, advirtiendo sobre la acumulación de deuda cognitiva cuando el modelo asume la estructuración del pensamiento.

En el ámbito específico de la ingeniería de software, en un ensayo controlado con desarrolladores aprendiendo una librería asíncrona desconocida (*Trio*), Shen y Tamkin (2025, Anthropic, *"How AI Impacts Skill Formation"*) observaron que los ingenieros que delegaron la generación de código obtuvieron **17 puntos porcentuales menos en comprensión conceptual** que quienes codificaron manualmente (50% vs. 67%), manifestándose la brecha más severa precisamente en la lectura de código y la capacidad de depuración (*debugging*). El estudio constató que los desarrolladores con menor desempeño se limitaron a usar la IA como un atajo para que escribiera por ellos, mientras que los de mejor rendimiento la usaron para hacer preguntas y entender cómo funcionaba la librería.

Por su parte, el estudio de Shaw y Nave (2026) observó el impacto concreto de forzar una revisión activa: cuando a las personas se les exigió verificar deliberadamente las respuestas de la IA antes de aceptarlas, la tasa de detección y corrección de errores se duplicó (pasando del 20% al 42,3%).

LA REGLA DE TRABAJO: EL PEAJE COGNITIVO PRE-COMMIT

Para llevar las ideas de Naur (1985) a la práctica, proponemos el **Peaje Cognitivo Pre-Commit**. Formulamos como hipótesis que los equipos que validan activamente la comprensión del código antes de persistir los cambios tendrán una tasa significativamente menor de fallas en producción (la fase de operación o "Día 2") y evitarán el retrabajo por pérdida de contexto, en comparación con los equipos que dependen solo de las revisiones de código tradicionales.

El mecanismo opera bajo tres pasos:

1. **Inversión de roles y evaluación socrática:** Antes de consolidar un commit o integrar el cambio a la rama principal, se detiene la generación de código. Inspirado en el método de Sócrates —que no daba respuestas elaboradas, sino que formulaba preguntas punzantes para obligar a sus interlocutores a pensar por sí mismos y justificar sus razonamientos—, la herramienta frena la IA y la pone a interrogar al desarrollador. Este giro fuerza el paso de una lectura pasiva a la auto-explicación activa (Chi et al., 1994).
2. **Las tres preguntas sobre el código real:** A partir del código recién modificado, la herramienta inspecciona los cambios locales (*git diff*) y formula tres preguntas técnicas sobre las líneas concretas:

- *Trazabilidad y flujo de datos:* Se interroga sobre el recorrido exacto de los datos en las funciones modificadas y las invariantes de negocio que garantiza la solución.
- *Bordes, fallas y concurrencia:* Se pregunta cómo reacciona el código ante situaciones límite (caídas de red, accesos simultáneos o valores nulos).
- *Mantenimiento y evolución («Día 2»):* Se plantea un cambio realista en las reglas del negocio para verificar si el desarrollador sabe exactamente en qué archivos y funciones debería intervenir.

3. **Condición de avance y semáforo de evaluación:** Solo cuando el desarrollador responde con precisión técnica a estas preguntas, el cambio queda habilitado para ingresar al repositorio. La comprensión humana se convierte en un requisito previo a la persistencia del código.

IMPLEMENTACIÓN PRÁCTICA: LA HERRAMIENTA EN EL IDE

Para que esto no quede en una recomendación teórica ni dependa únicamente de la voluntad del programador —que ante la presión de tiempo suele ceder—, implementamos este mecanismo como una herramienta ejecutable en el entorno de desarrollo (*skill* de **Peaje Cognitivo**).

La herramienta se ejecuta en la terminal o en el IDE antes de enviar los cambios (`git commit`) y opera en cuatro pasos:

1. **Inspección del *diff*:** La herramienta analiza los cambios locales (`git diff`) y calcula dos valores: la cantidad de líneas de lógica real modificadas y los puntos de decisión agregados (reglas del negocio, consultas a bases de datos o manejo transaccional). Para evitar la rendición cognitiva, la herramienta no genera explicaciones automáticas predigeridas del cambio ni responde por el usuario: el desarrollador puede consultar dudas conceptuales, pero debe inspeccionar las líneas modificadas y formular sus propias justificaciones.
2. **Cálculo del tiempo recomendado:** A partir de las tasas de inspección de la literatura (100 a 200 líneas por hora para lógica densa, según Fagan y Cohen), la herramienta calcula y muestra el tiempo mínimo que el desarrollador debería dedicar a revisar el cambio según el volumen y la complejidad del código modificado.
3. **Evaluación de comprensión:** Formula tres preguntas técnicas que apuntan directamente a los archivos, métodos y líneas modificadas en ese *diff*.
4. **Semáforo de avance:** Evalúa las respuestas del desarrollador con tres posibles resultados:
   - **Verde (Comprensión sólida):** Confirma que el programador entiende los cambios y habilita el *commit* o Pull Request.
   - **Amarillo (Comprensión parcial):** Detecta puntos ciegos (por ejemplo, efectos colaterales no contemplados), señala las líneas exactas en el código y pide una aclaración puntual antes de aprobar.
   - **Rojo (Falta de comprensión):** Bloquea el avance y obliga a revisar en detalle los bloques críticos del *diff*.

6.4. Pensamiento profiláctico: anticipar fallas en producción

FUNDAMENTO EN AJEDREZ

En ajedrez, el pensamiento profiláctico (desarrollado por Aron Nimzowitsch y popularizado por campeones como Tigrán Petrosián) consiste en anticipar y desactivar los planes del rival antes de que aparezcan en el tablero. El jugador profiláctico no solo responde a las amenazas inmediatas: mira su propia posición, busca qué debilidades podría explotar el adversario más adelante y toma medidas preventivas mucho antes de que el peligro sea evidente.

EL PROBLEMA EN EL DESARROLLO DE SOFTWARE

La IA suele ser optimista por defecto: resuelve el camino feliz (*happy path*). Escribe código asumiendo que los datos de entrada siempre son válidos, que la red nunca tiene latencia, que no hay accesos simultáneos y que la base de datos responde rápido.

Con la presión por entregar rápido, el programador acepta ese código sin dudar. La solución pasa las pruebas unitarias básicas —que suelen evaluar ese mismo camino feliz— y se envía a producción. El problema aparece cuando el código se choca con la realidad: datos históricos corruptos, microcortes de red o picos de concurrencia. El sistema se cae y el equipo tarda horas en diagnosticar la falla porque nunca pensó en esos escenarios durante el desarrollo.

LA REGLA DE TRABAJO

En componentes críticos, antes de desplegar a producción el equipo debe realizar una auditoría preventiva en tres frentes:

1. **Concurrencia e idempotencia:** Verificar que las operaciones sobre datos compartidos sean atómicas. Si una llamada de red falla y se reintenta, el código debe ser idempotente (no duplicar cobros ni registros). Asegurar además que las consultas no provoquen bloqueos mutuos (*deadlocks*) en la base de datos bajo carga real.
2. **Tolerancia a datos reales:** Probar el código contra la suciedad del mundo real: campos nulos, registros históricos incompletos o caracteres especiales. Si algo falla, el sistema debe registrar el error de forma clara y predecible, nunca romperse en silencio ni corromper información.
3. **Aislamiento de la lógica de negocio:** Separar las reglas del negocio de los detalles de infraestructura (bases de datos, colas o librerías externas). Mantener la lógica central en funciones aisladas permite verificar las reglas del sistema y contemplar los casos límite sin depender de servicios externos.

6.5. El modelo centauro: la ley de Kaspárov

FUNDAMENTO EN AJEDREZ: EL TORNEO FREESTYLE DE 2005

Tras perder con Deep Blue en 1997, Garry Kaspárov creó el *Ajedrez Avanzado* (equipos formados por humanos con computadoras). En 2005, el torneo *Freestyle* dejó una lección que Kaspárov convirtió en ley: dos aficionados que usaban computadoras comunes bajo un **método de trabajo riguroso** vencieron tanto a Grandes Maestros como a supercomputadoras operadas sin un proceso claro. Kaspárov observó que un proceso disciplinado puede superar a la potencia pura de la máquina si esta se usa sin una guía clara.

Aunque hoy los motores jueguen solos en el ajedrez por tratarse de un dominio cerrado, en software el modelo centauro cobra pleno sentido: en un entorno abierto, con reglas de negocio cambiantes y requerimientos ambiguos, la IA no garantiza una visión estratégica integral. Solo un proceso deliberado, conducido por el programador, permite transformar la velocidad de la IA en sistemas sólidos.

EL PROBLEMA EN EL DESARROLLO DE SOFTWARE

El avance constante de los modelos de IA suele generar la expectativa natural de que una versión más potente resolverá por sí sola los desafíos del desarrollo de software.

Planteamos que, sin reglas de trabajo compartidas, existe el riesgo plausible de que ocurra lo contrario: la IA puede amplificar el desorden a gran velocidad. Si cada programador usa prompts con criterios dispares, incorpora dependencias incompatibles o acepta soluciones desconectadas entre sí, la velocidad de la herramienta termina acelerando el caos arquitectónico.

LA REGLA DE TRABAJO

La ventaja de un equipo no está en qué modelo de lenguaje contrata, sino en las reglas con las que trabaja:

1. **La arquitectura y el diseño quedan en manos del equipo:** La IA actúa como soporte táctico para escribir código o resolver dudas puntuales, pero las decisiones de fondo, los modelos de datos y la arquitectura del sistema son responsabilidad exclusiva del programador humano.
2. **Reglas de trabajo comunes:** El uso de IA debe tener pautas compartidas: componentes críticos identificados, uso obligatorio del peaje cognitivo en módulos sensibles y revisión humana pausada antes de cada despliegue.
3. **El programador como conductor:** El objetivo no es competir en velocidad con la IA ni aceptar sus respuestas a ciegas, sino aprovechar su rapidez para ganar tiempo y pensar con calma el diseño y la solidez del sistema.

6.6. Limitaciones del marco

Para ser realistas con la propuesta, reconocemos cinco limitaciones prácticas:

1. **Engaño a la herramienta y respuestas automáticas:** Si la evaluación la hace un agente de IA en el IDE, un programador experimentado podría responder con frases hechas o explicaciones superficiales que convenzan al evaluador sin haber entendido el código de verdad. Para evitar esto, las preguntas deben apuntar a líneas y variables específicas del *diff*, complementándose con revisiones periódicas entre compañeros de equipo.
2. **Pérdida de tiempo en tareas simples:** Exigir este control en maquetas visuales, tareas menores o scripts descartables agrega una molestia innecesaria que solo lograría que el equipo busque saltearse la herramienta. Por eso, el peaje cognitivo y la regla del 10 + 10 deben aplicarse únicamente a los módulos críticos y a las reglas centrales del negocio.
3. **La aceleración real varía según el proyecto:** El ejemplo de reducir 40 horas a 4 horas con IA es un caso ideal para ilustrar la regla. En proyectos reales con sistemas heredados (*legacy*) o arquitecturas complejas, la ganancia de tiempo puede ser bastante menor. El tiempo de asimilación debe ajustarse según la dificultad del código y la experiencia del equipo en ese sistema.
4. **Alineación con los objetivos del proyecto:** Ninguna regla técnica funciona de forma aislada si los indicadores de seguimiento solo contemplan el volumen de entrega inmediato. Cuando el apuro del calendario desplaza el espacio de análisis, cualquier control corre el riesgo de convertirse en un paso formal que se completa solo por cumplir.
5. **Parámetros a calibrar en la práctica:** La psicología del ajedrez sirve como analogía para entender qué le pasa a la mente bajo presión de tiempo, pero no reemplaza la medición real en desarrollo de software. Los parámetros propuestos en este trabajo (como las proporciones de la regla del 10 + 10) son estimaciones iniciales que deberán medirse, calibrarse y ajustarse con datos empíricos de la industria.

6.7. Protocolo experimental futuro

Para validar este marco en la práctica y poner a prueba la hipótesis del peaje cognitivo, proponemos un diseño experimental que pueda llevarse a cabo con equipos de desarrollo en empresas reales a lo largo del tiempo. Aunque el protocolo define requisitos metodológicos agnósticos de la plataforma (cualquier integración en el IDE o pipeline que implemente las reglas de la sección 6.3), se declara de forma transparente que el autor ha desarrollado la implementación de referencia ejecutable (*skill* de Peaje Cognitivo) evaluada en este diseño, cuyos detalles de arquitectura, código abierto y paquete de replicación se publicarán en un repositorio público abierto. Asimismo, para asegurar la máxima integridad metodológica, el protocolo experimental debe preregistrarse en una plataforma pública como el Open Science Framework (OSF), fijando la tasa de incidentes en producción como desenlace primario único antes de iniciar la recolección de datos.

ESTRUCTURA FACTORIAL DEL EXPERIMENTO

Para que los resultados sean comparables metodológicamente, **los cuatro grupos comparten la misma base común de buenas prácticas de ingeniería**: clasificación de componentes críticos (6.1), anticipación de fallas (6.4), uso de Pull Requests pequeños, pruebas automatizadas y revisión asíncrona tradicional. Sobre esta base común, el protocolo implementa un **diseño factorial 2 × 2** que cruza dos factores independientes:

- **Factor 1: Reserva de tiempo de asimilación (Regla 10 + 10)** [Sí / No].
- **Factor 2: Evaluación socrática en el IDE (Peaje cognitivo pre-commit)** [Sí / No].

Este cruce define cuatro grupos de intervención:

- **Brazo A (Control con base común):** Opera exclusivamente con las buenas prácticas tradicionales, sin reserva formal de tiempo (10 + 10) ni peaje socrático en el IDE.
- **Brazo B (Solo reserva de tiempo / 10 + 10):** Aplica la base común e incorpora la reserva obligatoria de tiempo de asimilación en el sprint (6.2), sin evaluación socrática en el IDE.
- **Brazo C (Solo peaje cognitivo):** Aplica la base común e incorpora las preguntas socráticas en el IDE antes de cada commit (6.3), sin reserva formal de tiempo en la planificación.
- **Brazo D (Intervención combinada):** Aplica la base común y la presencia conjunta de ambos factores (reserva 10 + 10 y peaje socrático en el IDE).

Como diseño exploratorio preliminar, proponemos una muestra orientativa de 4 a 6 equipos independientes por brazo (entre 16 y 24 equipos en total, sumando de 80 a 150 desarrolladores) con un seguimiento mínimo de seis meses. Dado que este tamaño muestral es de carácter exploratorio y preliminar, no se asume potencia estadística suficiente para detectar efectos específicos, en particular interacciones entre factores; el preregistro formal en OSF deberá incluir una simulación de potencia calibrada sobre la tasa base histórica de incidentes de las organizaciones participantes, y la evaluación fina de los términos de interacción se deja para replicaciones a mayor escala. El recuento de incidentes en producción se modela mediante regresiones para eventos discretos (Poisson o binomial negativa, según la sobredispersión), incorporando efectos aleatorios por equipo y empresa.

VARIABLES A MEDIR

1. **Calidad y estabilidad en producción:**
   - *Incidentes críticos:* Fallas funcionales, problemas de concurrencia o corrupción de datos tras el despliegue. Para determinar el origen del fallo (código de la IA o cambios manuales del programador), el análisis de causa raíz rastrea el commit causante en el historial de Git (`git blame`), complementado con registros de eventos del entorno de desarrollo (telemetría de autoría) y revisiones manuales a ciegas sobre una muestra de incidentes.
   - *Tiempo medio de recuperación (MTTR):* Cuánto tarda el equipo en diagnosticar y solucionar una falla en producción. Constituye un indicador indirecto de qué tan bien entendían el sistema (reconociendo que también influyen las herramientas de observabilidad y la severidad del incidente).
   - *Retrabajo temprano (*code churn*):* Porcentaje de código modificado o descartado dentro de las dos semanas posteriores al despliegue.
2. **Velocidad de entrega (estándar DORA — *DevOps Research and Assessment*):**
   - *Tiempo de entrega (*lead time*):* Tiempo desde el primer commit hasta la puesta en producción. Permite comprobar si el tiempo invertido en asimilar el código se recupera al tener revisiones más rápidas y menos errores que corregir.
   - *Funcionalidades entregadas (*throughput*):* Cantidad de requerimientos terminados y estables por sprint, para comparar la velocidad real y sostenida frente a la velocidad superficial que genera retrabajo.
3. **Comprensión del sistema:**
   - *Retención de la teoría del programa:* Evaluación socrática a ciegas a los desarrolladores de los cuatro grupos, 30 días después del despliegue, sobre la lógica interna, el flujo de datos y los casos límite del

código puesto en producción.

AMENAZAS A LA VALIDEZ

El protocolo contempla dos precauciones metodológicas:

1. **Efecto Hawthorne y novedad de las herramientas:** Saberse observado o empezar a usar una herramienta nueva puede alterar transitoriamente la conducta del programador en las primeras semanas. Para amortiguar ambos sesgos, el estudio contempla un período previo de adaptación de 30 días donde las herramientas se usan con normalidad pero sin registrar métricas.
2. **Diferencias entre equipos y proyectos:** Encontrar equipos con experiencia similar y sistemas comparables en la industria es difícil. Por eso, el análisis estadístico debe controlar las diferencias de contexto entre empresas y la complejidad de cada código base mediante modelos de efectos mixtos.

Conclusión: Comprender la partida

El ajedrez y la ingeniería de software comparten una verdad fundamental formulada con claridad por Peter Naur: el valor del trabajo técnico no reside en los registros que quedan al final de la jornada —la planilla donde se anotaron las jugadas o el historial de commits en el repositorio—, sino en los modelos mentales que se construyeron en la cabeza de las personas durante el proceso. Cuando un maestro gana una partida, la planilla es apenas el registro visible de una comprensión estratégica mucho más rica y profunda. Del mismo modo, cuando un equipo construye un sistema, el código fuente es únicamente la expresión externa de una teoría que debe vivir y actualizarse en la mente de quienes tienen que mantenerlo.

La Inteligencia Artificial es una de las herramientas más potentes en la historia de la computación. Su capacidad para generar código sintácticamente correcto en segundos reduce de forma drástica las tareas mecánicas de la programación. Sin embargo, la rapidez para generar código no equivale por sí misma a comprender el sistema. Cuando la dinámica de estas herramientas nos empuja a trabajar a ritmo *bullet* —guiándonos solo por la velocidad de generación o el cierre apresurado de tareas—, corremos el riesgo de debilitar la capacidad de análisis de los equipos, frenar el aprendizaje de los desarrolladores y trasladar la fragilidad del software directamente a producción.

Frente a este riesgo, este trabajo propone un marco operativo basado en principios concretos: reservar tiempo obligatorio de asimilación técnica mediante una cuota mínima de inspección (la regla del 10 + 10), exigir pruebas activas de comprensión antes de cada commit mediante evaluación socrática en el IDE, aplicar pensamiento profiláctico sobre los componentes críticos y adoptar el modelo centauro para que el programador ejerza la conducción estratégica y no sea un espectador pasivo. Estas reglas no buscan frenar la tecnología ni volver a un desarrollo artesanal; su objetivo es recuperar el equilibrio entre la velocidad de la IA y el criterio del profesional para construir una productividad real y sostenible en el tiempo.

En la era de los agentes de Inteligencia Artificial, la excelencia técnica no se mide por la cantidad de líneas generadas por segundo ni por la velocidad con la que se mueven las piezas en el tablero. La verdadera victoria en la ingeniería de software consiste, hoy más que nunca, en comprender la partida.

Referencias Bibliográficas

Bacchelli, A., & Bird, C. (2013). Expectations, outcomes, and challenges of modern code review. In *Proceedings of the 35th International Conference on Software Engineering (ICSE '13)* (pp. 712–721). IEEE / ACM.

Baddeley, A. (1992). Working memory. *Science*, 255(5044), 556–559.

Becker, J., Rush, N., Barnes, E., & Rein, D. (2025). *Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity*. METR (Model Evaluation and Threat Research). https://arxiv.org/abs/2507.09089

Bilalić, M., Graf, M., & Vaci, N. (2026). Computers and chess masters: The role of AI in transforming elite human performance. *British Journal of Psychology*, 117(2), 585–609. https://doi.org/10.1111/bjop.12750

Bjork, R. A. (1994). Memory and metamemory considerations in the training of human beings. En J. Metcalfe & A. Shimamura (Eds.), *Metacognition: Knowing about knowing* (pp. 185–205). MIT Press.

Brooks, F. P. (1987). No Silver Bullet: Essence and Accidents of Software Engineering. *IEEE Computer*, 20(4), 10-19.

Brynjolfsson, E., Li, D., & Raymond, L. R. (2023). *Generative AI at work*. National Bureau of Economic Research (NBER Working Paper No. 31161).

Chase, W. G., & Simon, H. A. (1973). Perception in chess. *Cognitive Psychology*, 4(1), 55-81.

Chi, M. T. H., De Leeuw, N., Chiu, M. H., & LaVancher, C. (1994). Eliciting self-explanations improves understanding. *Cognitive Science*, 18(3), 439-477.

Cohen, J., Brown, E., DuRette, B., & Teleki, S. (2006). *Best Kept Secrets of Peer Code Review*. SmartBear Software & Cisco Systems.

Cowan, N. (2001). The magical number 4 in short-term memory: A reconsideration of mental storage capacity. *Behavioral and Brain Sciences*, 24(1), 87–114.

De Groot, A. D. (1946/1965). *Thought and choice in chess*. Mouton De Gruyter.

Dreyfus, H. L. (1972/1992). *What Computers Still Can't Do: A Critique of Artificial Reason*. MIT Press.

Dreyfus, H. L., & Dreyfus, S. E. (1986). *Mind over machine: The power of human intuition and expertise in the era of the computer*. Free Press.

Ericsson, K. A., Krampe, R. T., & Tesch-Römer, C. (1993). The role of deliberate practice in the acquisition of expert performance. *Psychological Review*, 100(3), 363–406.

Fagan, M. E. (1976). Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3), 182-211. DOI: 10.1147/sj.153.0182.

Faros AI. (2026). *The Speed Trap: How Generative AI Impacts Software Quality and Production Incidents*. Faros Engineering Report.

He, H., & Vasilescu, B. (2026). *AI Writes Faster Than Humans Can Review: A Longitudinal Study of an Enterprise "2x" Mandate*. Carnegie Mellon University & Stanford University. https://arxiv.org/abs/2607.01904

Hermans, F. (2021). *The Programmer's Brain: What every programmer needs to know about cognition*. Manning Publications.

Kahneman, D. (2011). *Thinking, Fast and Slow*. Farrar, Straus and Giroux.

Kaspárov, G. (2007). *Cómo la vida imita al ajedrez*. Debate / Random House Mondadori.

Kosmyna, N., Hauptmann, E., & Maes, P. (2025). *Your Brain on ChatGPT: Accumulation of Cognitive Debt when Using an AI Assistant for Essay Writing Task*. MIT Media Lab. https://arxiv.org/abs/2506.08872

Kotov, A. (1971). *Think Like a Grandmaster*. Batsford Chess.

Naur, P. (1985). Programming as Theory Building. *Microprocessing and Microprogramming*, 15(5), 253-261.

Osmani, A. (2026). *Comprehension Debt: The Hidden Cost of AI-Generated Code*. O'Reilly Media. https://addyosmani.com/blog/comprehension-debt/

Parasuraman, R., Sheridan, T. B., & Wickens, C. D. (2000). A model for types and levels of human interaction with automation. *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, 30(3), 286–297.

Peng, S., Kalliamvakou, E., Cihon, P., & Demirer, M. (2023). *The impact of AI on developer productivity: Evidence from GitHub Copilot*. https://arxiv.org/abs/2302.06590

Pennington, N. (1987). Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, 19(3), 295–341.

Poulidis, S., Bastani, H., & Bastani, O. (2025). *Self-Regulated AI Use Hinders Long-Term Learning*. The Wharton School, University of Pennsylvania. https://ssrn.com/abstract=5604932

Rowson, J. (2000). *The Seven Deadly Chess Sins*. Gambit Publications.

Schut, L., Tomašev, N., McGrath, T., Hassabis, D., Paquet, U., & Kim, B. (2025). Bridging the human–AI knowledge gap through concept discovery and transfer in AlphaZero. *Proceedings of the National Academy of Sciences (PNAS)*, 122(13), e2406675122. https://doi.org/10.1073/pnas.2406675122

Shaw, S. D., & Nave, G. (2026). *Thinking—Fast, Slow, and Artificial: How AI is Reshaping Human Reasoning and the Rise of Cognitive Surrender*. The Wharton School, University of Pennsylvania. https://ssrn.com/abstract=6097646

Shen, J. H., & Tamkin, A. (2025). *How AI Impacts Skill Formation*. Anthropic Research Report. https://arxiv.org/abs/2601.20245

Storey, M. A. (2026). *From Technical Debt to Cognitive and Intent Debt: Rethinking Software Health in the Age of AI*. University of Victoria. https://arxiv.org/abs/2603.22106

Sweller, J. (1988). Cognitive load during problem solving: Effects on learning. *Cognitive Science*, 12(2), 257–285.