Research Club Scientia Vol. 4, No 3
Frecuencia: Julio - Septiembre
ISSN: 3103 - 1315
DOI: https://doi.org/10.65415/6mgvny75

Vulnerabilidades de seguridad y patrones de programación insegura en código fuente generado por modelos de inteligencia artificial generativa

Security Vulnerabilities and Insecure Programming Patterns in Source Code Generated by Generative Artificial Intelligence Models

Vulnerabilità di sicurezza e modelli di programmazione non sicura nel codice sorgente generato da modelli di intelligenza artificiale generativa

Ana Arellano Arcentales[1]
E-MAIL:aarellano@uagraria.edu.ec
aarellano@ecotec.edu.ec
ORCID: https://orcid.org/0009-0000-9507-6865

Verónica Adriana Freire Avilés
E-MAIL: vfreire@uagraria.edu.ec
ORCID: https://orcid.org/0000-0001-6509-6080

Jessica López Alvarado
E-MAIL: jlopeza21@unemi.edu.ec
ORCID: https://orcid.org/0009-0001-2364-9334

Correspondencia: **Ana Arellano Arcentales**· aarellano@uagraria.edu.ec

Artículo de Investigación

* Recibido: 14-07-2026 * Aceptado: 20-08-2026 * Publicado: 23-08-2026

I. Universidad Agraria del Ecuador, Ecuador.

II. Universidad Tecnológica Ecotec

III. Universidad estatal de Milagro

Resumen

La generación automática de código mediante modelos de inteligencia artificial generativa se ha integrado en editores, asistentes conversacionales y agentes de desarrollo. Su capacidad para producir soluciones funcionales no garantiza, sin embargo, que el código preserve propiedades de confidencialidad, integridad, disponibilidad y control de acceso. El objetivo de este estudio fue identificar las vulnerabilidades de seguridad y los patrones de programación insegura documentados en código fuente generado por modelos de lenguaje, así como valorar las condiciones que favorecen su propagación y las medidas de aseguramiento con respaldo empírico. Se realizó una revisión documental estructurada de estudios primarios publicados entre enero de 2022 y agosto de 2026 en fuentes académicas y técnicas reconocidas. El corpus final reunió 26 investigaciones empíricas, conjuntos de evaluación y estudios con usuarios; la síntesis se organizó mediante Common Weakness Enumeration, OWASP Top 10 y prácticas del Secure Software Development Framework. Los resultados muestran recurrencia de validación insuficiente de entradas, inyección, manejo inseguro de memoria y enteros, criptografía y credenciales débiles, uso incorrecto de APIs de seguridad, exposición de datos, configuraciones permisivas y dependencias inexistentes. También se identificaron patrones transversales: prioridad del camino feliz, reproducción del contexto vulnerable, confianza excesiva en respuestas plausibles y verificación limitada a pruebas funcionales. La evidencia no permite sostener que la IA sea siempre menos segura que el desarrollo humano, pero sí confirma que sus salidas deben tratarse como código no confiable. Se propone una estrategia de aseguramiento en profundidad que combina requisitos explícitos, análisis automatizado, pruebas adversarias, revisión humana y trazabilidad.

Palabras clave: inteligencia artificial generativa; seguridad del software; código fuente; vulnerabilidades; programación insegura.

Abstract

Automated code generation through generative artificial intelligence models has become embedded in editors, conversational assistants, and development agents. The ability to produce functional solutions does not guarantee that the resulting code preserves confidentiality, integrity, availability, and access-control properties. This study aimed to identify security vulnerabilities and insecure programming patterns documented in source code generated by language models, and to assess the conditions that facilitate their propagation and the assurance measures supported by empirical evidence. A structured documentary review was conducted on primary studies published between January 2022 and August 2026 in recognized academic and technical sources. The final corpus comprised 26 empirical investigations, evaluation datasets, and user studies; the synthesis was organized using Common Weakness Enumeration, the OWASP Top 10, and the Secure Software Development Framework. Results show recurring insufficient input validation, injection, unsafe memory and integer handling, weak cryptography and credentials, misuse of security APIs, data exposure, permissive configurations, and nonexistent dependencies. Cross-cutting patterns were also identified: prioritization of the happy path, reproduction of vulnerable context, excessive trust in plausible responses, and validation limited to functional tests. The evidence does not support the claim that AI is always less secure than human development, but it does confirm that its outputs must be treated as untrusted code. A defense-in-depth assurance strategy is proposed, combining explicit security requirements, automated analysis, adversarial testing, human review, and traceability.

Keywords: generative artificial intelligence; software security; source code; vulnerabilities; insecure programming.

Introducción

Los modelos generativos de código transformaron la asistencia al programador desde el autocompletado de fragmentos breves hacia la producción de funciones, módulos, pruebas, consultas, configuraciones e incluso cambios que abarcan varios archivos. Esta evolución amplía la superficie de uso, pero también la superficie de riesgo. Una respuesta puede compilar, superar pruebas unitarias y satisfacer el requisito visible, mientras conserva rutas de ejecución explotables, permisos excesivos, dependencias no verificadas o decisiones criptográficas débiles. La seguridad no es una consecuencia automática de la corrección funcional: constituye una propiedad adicional que depende del contexto, del adversario y de la forma en que el componente se integra con el sistema.

La diferencia es especialmente relevante porque los modelos generan secuencias plausibles a partir de regularidades aprendidas en grandes corpus. En esos corpus coexisten implementaciones robustas, ejemplos pedagógicos simplificados, código obsoleto y patrones inseguros. El sistema puede reproducir una práctica frecuente aunque la práctica no sea apropiada para producción. Además, la solicitud del usuario suele describir qué debe hacer el programa, pero omite activos, límites de confianza, amenazas y requisitos negativos. Cuando el prompt pide únicamente “crear un inicio de sesión”, el modelo puede priorizar el flujo feliz y dejar indeterminados el almacenamiento de contraseñas, la limitación de intentos, la gestión de sesiones o la protección frente a inyección.

La evidencia temprana hizo visible el problema. Pearce et al. (2022) evaluaron contribuciones de GitHub Copilot en escenarios sensibles y encontraron una proporción importante de soluciones vulnerables, con diferencias según lenguaje, debilidad y formulación del contexto. SecurityEval amplió la evaluación mediante 130 muestras asociadas con 75 tipos de vulnerabilidad (Siddiq & Santos, 2022). Posteriormente, CyberSecEval integró ocho lenguajes y 50 prácticas inseguras, y observó sugerencias vulnerables en todos los modelos estudiados (Bhatt et al., 2023). Estos trabajos establecieron dos ideas: el riesgo no pertenece a un único producto y la tasa observada depende fuertemente del instrumento de evaluación.

Los estudios con personas agregaron una dimensión socio-técnica. Perry et al. (2023) mostraron que la asistencia puede aumentar la producción de soluciones inseguras en determinadas tareas y, al mismo tiempo, elevar la confianza de quien programa. Sandoval et al. (2023), en cambio, no encontraron evidencia concluyente de un incremento general de errores críticos en una tarea de C. Asare et al. (2023) observaron que Copilot evitaba algunas vulnerabilidades humanas, aunque también reproducía fallos del contexto. La aparente contradicción no invalida la preocupación: revela que el efecto depende de la tarea, la experiencia, la métrica, el lenguaje y el modo de interacción.

La escala y el realismo de los datos modifican también la interpretación. FormAI reunió 112.000 programas compilables en C y los clasificó mediante verificación formal (Tihanyi et al., 2023); su ampliación comparó nueve modelos y 331.000 programas (Tihanyi et al., 2025). Fu et al. (2025) estudiaron fragmentos atribuidos a asistentes dentro de proyectos públicos de GitHub, mientras SecRepoBench trasladó la evaluación a 318 tareas sobre 27 repositorios C/C++ (Shen et al., 2026). El tránsito desde funciones aisladas hacia repositorios evidencia que comprender interfaces, dependencias y flujos entre archivos es parte del problema de seguridad.

La vulnerabilidad tampoco se limita a la línea producida. Los modelos pueden recomendar nombres de paquetes inexistentes, creando una oportunidad para que un tercero publique artefactos maliciosos y capture instalaciones posteriores. Spracklen et al. (2025) documentaron este fenómeno en 576.000 muestras de código. En otros casos, la salida usa correctamente la sintaxis de una API pero configura cifrado, certificados o generadores aleatorios de manera insegura; Mousavi et al. (2024) identificaron una elevada prevalencia de usos incorrectos de APIs de seguridad de Java. Estas fallas son peligrosas porque parecen especializadas y pueden superar una revisión superficial.

Las revisiones previas coinciden en que el código generado no ofrece garantía de seguridad, pero advierten heterogeneidad y limitaciones de comparabilidad (Negri-Ribalta et al., 2024). Desde entonces aparecieron nuevos benchmarks, herramientas y estudios de repositorios que obligan a actualizar la síntesis. Persisten tres vacíos: la tendencia a confundir defectos funcionales con vulnerabilidades; la presentación de listas de CWE sin explicar los patrones de programación que las producen; y la ausencia de una ruta operativa que conecte generación, verificación e integración dentro del ciclo de desarrollo seguro.

El objetivo general de esta investigación fue analizar las vulnerabilidades y los patrones de programación insegura documentados en código fuente generado por modelos de inteligencia artificial generativa. Se formularon tres preguntas: ¿qué categorías de debilidad aparecen de forma recurrente?; ¿qué condiciones técnicas y humanas explican su propagación?; y ¿qué controles permiten reducir el riesgo sin asumir que el propio modelo puede certificar su salida? La contribución consiste en una taxonomía orientada a causas, una lectura crítica de la evidencia y un modelo de aseguramiento en profundidad aplicable a entornos educativos y profesionales.

Metodología

Diseño y alcance

Se desarrolló una revisión documental estructurada con alcance descriptivo e interpretativo. El estudio no evaluó participantes ni ejecutó modelos propietarios en tiempo real; su unidad de análisis fue la publicación primaria que examinó seguridad en código generado o aceptado con asistencia de modelos de lenguaje. La decisión evita atribuir a una versión actual resultados obtenidos con versiones anteriores y permite comparar evidencia producida mediante pruebas estáticas, verificación formal, ejecución, revisión manual, repositorios y estudios con usuarios.

El protocolo se inspiró en las guías de revisión de ingeniería de software de Kitchenham y Charters (2007), la estrategia de búsqueda complementaria de Wohlin (2014) y los principios de transparencia de PRISMA 2020 (Page et al., 2021). No se presenta como metaanálisis porque los estudios utilizaron tareas, lenguajes, modelos, temperaturas, unidades y oráculos incompatibles. Los porcentajes publicados se conservaron como resultados específicos de cada diseño y no se combinaron en una estimación global.

La búsqueda se actualizó el 22 de agosto de 2026 y cubrió trabajos publicados desde enero de 2022. Se consultaron ACM Digital Library, IEEE Xplore, SpringerLink, ScienceDirect, USENIX y arXiv para localizar versiones abiertas o trabajos recientes todavía sin edición definitiva. Las cadenas combinaron términos equivalentes a “large language model”, “generative AI”, “code generation”, “insecure code”, “security vulnerability”, “CWE”, “secure code generation”, “code assistant” y “repository benchmark”. También se siguieron referencias hacia atrás y hacia adelante desde Pearce et al. (2022), Perry et al. (2023), Negri-Ribalta et al. (2024) y los benchmarks posteriores.

Se incluyeron estudios que: evaluaron código generado o completado por un modelo; midieron debilidades, vulnerabilidades, uso inseguro de APIs, dependencias o conducta del programador; describieron muestra, tareas y procedimiento; y publicaron resultados suficientes para interpretar su alcance. Se excluyeron opiniones sin método, análisis centrados exclusivamente en ataques contra el modelo, estudios de detección de código generado sin evaluación de seguridad y trabajos donde la generación de malware no guardaba relación con la seguridad del software producido. Cuando coexistían preprint y versión editorial, se priorizó la versión publicada.

El corpus final quedó conformado por 26 estudios primarios de evaluación o intervención y nueve documentos de apoyo metodológico, normativo o de síntesis. Se mantuvieron algunos preprints de 2024–2026 cuando presentaban benchmarks relevantes que todavía no contaban con versión editorial. Su condición se identifica en las referencias y sus hallazgos se interpretaron con menor fuerza que la evidencia revisada por pares. Esta decisión permite representar la evolución del campo sin equiparar disponibilidad reciente con validación definitiva.

Tabla 1. Protocolo de búsqueda, selección y extracción

Componente	Especificación aplicada
Periodo	Enero de 2022 a 22 de agosto de 2026.
Fuentes	ACM, IEEE, Springer, ScienceDirect, USENIX y arXiv; rastreo de referencias.
Población documental	Estudios de código generado, completado o aceptado con apoyo de modelos generativos.
Desenlaces	CWE, vulnerabilidades confirmadas, uso inseguro de APIs, dependencias, confianza y controles.
Criterio de síntesis	Codificación por presencia y mecanismo; sin agregación estadística de tasas heterogéneas.
Calidad	Claridad de tareas y versión; oráculo reproducible; revisión manual; contexto; separación función–seguridad.

Nota. El corpus primario se utilizó para la síntesis temática; las normas y revisiones apoyaron la interpretación y la propuesta de controles.

Codificación y evaluación de la evidencia

Para cada estudio se extrajeron año, tipo de publicación, modelo, lenguaje, unidad de análisis, tamaño de la muestra, tarea, oráculo de seguridad, principales debilidades, comparación y mitigación. Las debilidades se normalizaron con CWE cuando la publicación proporcionaba el identificador o permitía una correspondencia inequívoca. Después se agruparon en familias coherentes con OWASP Top 10 (OWASP Foundation, 2021), CWE Top 25 (MITRE, 2025) y la dimensión de seguridad del modelo de calidad ISO/IEC 25010:2023.

La codificación distinguió vulnerabilidad de patrón inseguro. Una vulnerabilidad es una debilidad concreta con posibilidad de explotación en un contexto. Un patrón inseguro es una regularidad de decisión que favorece una o varias vulnerabilidades, por ejemplo concatenar entradas en comandos, seleccionar un algoritmo criptográfico obsoleto, aceptar una dependencia sin verificar o comprobar solo el camino feliz. Esta distinción permite interpretar el mecanismo y no solo enumerar detectores activados.

La confianza de cada hallazgo se valoró mediante cuatro preguntas: si el código era funcional; si el oráculo de seguridad estaba descrito; si los positivos automáticos fueron confirmados o respaldados por ejecución; y si la tarea representaba un contexto realista. Se consideró evidencia más sólida la combinación de pruebas funcionales, análisis estático o formal, ejecución

adversaria y revisión humana. Una alerta de SAST se trató como indicio, no como vulnerabilidad confirmada, salvo que el estudio ofreciera validación adicional.

Resultados

Panorama de la evidencia

El corpus confirma una expansión metodológica. Los primeros estudios usaron escenarios autocontenidos y completaciones breves; los trabajos posteriores incorporaron conjuntos multilenguaje, verificación formal, repositorios y agentes. SecurityEval cubrió 75 tipos de vulnerabilidad (Siddiq & Santos, 2022), CyberSecEval empleó 189 reglas para 50 prácticas inseguras en ocho lenguajes (Bhatt et al., 2023) y SALLM propuso 100 prompts de Python centrados en seguridad (Siddiq et al., 2024). CodeLMSec automatizó la búsqueda de solicitudes capaces de inducir vulnerabilidades en modelos de caja negra (Hajipour et al., 2024).

Las tasas observadas son elevadas en varios diseños, pero no son intercambiables. Pearce et al. (2022) informaron alrededor de 40 % de soluciones vulnerables en su conjunto de escenarios. FormAI encontró debilidades en 51,24 % de 112.000 programas compilables generados en C (Tihanyi et al., 2023), y FormAI-v2 reportó al menos 62,07 % sobre 331.000 programas de nueve modelos (Tihanyi et al., 2025). Esos valores describen programas C generados con prompts y límites de verificación particulares; no constituyen una tasa universal aplicable a cualquier lenguaje o producto.

La evidencia de lenguajes de alto nivel tampoco autoriza confianza ciega. Hamer et al. (2024) compararon 108 fragmentos Java por fuente y detectaron 248 vulnerabilidades en código de ChatGPT frente a 302 en respuestas de Stack Overflow. El menor conteo relativo no convirtió a ninguna fuente en segura: ambas propagaron 25 tipos de CWE en conjunto. Rabbi et al. (2024) analizaron 1.756 fragmentos Python y mostraron la necesidad de inspección sistemática, mientras DeVAIC cubrió 35 CWE de OWASP y alcanzó 94 % de F1 y exactitud en su conjunto de evaluación (Cotroneo et al., 2025).

Los contextos reales revelan problemas que los ejercicios pequeños omiten. Fu et al. (2025) estudiaron 733 fragmentos atribuidos a asistentes en GitHub; las debilidades afectaron 29,5 % de los fragmentos Python y 24,2 % de JavaScript, distribuidos en 43 CWE. SecRepoBench incorporó 318 tareas de 27 repositorios C/C++ y observó una caída del desempeño conjunto de corrección y seguridad respecto de benchmarks autocontenidos (Shen et al., 2026). RealSec-bench trasladó la evaluación a repositorios Java y 19 tipos de CWE, reforzando la importancia de los flujos interprocedimentales y del contexto (Wang et al., 2026).

También emergieron riesgos que no encajan bien en una prueba de función. Spracklen et al. (2025) generaron 576.000 muestras y hallaron paquetes inexistentes en al menos 5,2 % de las recomendaciones de modelos comerciales y 21,7 % de modelos abiertos. El nombre alucinado puede ser registrado por un atacante y convertir una recomendación aparentemente inocua en compromiso de la cadena de suministro. Tambon et al. (2025) identificaron, además, 333 defectos y diez patrones de error propios de la generación, entre ellos objetos alucinados, generación incompleta, supuestos sesgados por el prompt y omisión de casos límite.

Figura 1. Ruta socio-técnica de propagación de una debilidad generada

Nota. El código vulnerable causa daño cuando supera las decisiones de revisión e integración y alcanza una entrada adversaria; cada transición ofrece una oportunidad de control.

Familias de vulnerabilidad y patrones inseguros

La síntesis produjo siete familias. La primera reúne fallas de validación, neutralización y construcción de consultas. Aparecen concatenación de entradas en SQL, comandos del sistema, rutas y HTML; ausencia de listas permitidas; escape incompleto; y confianza en parámetros externos. Los CWE-78, 79, 89 y 22 se repiten porque el modelo satisface rápidamente el requisito utilizando una interpolación directa, sin representar el límite de confianza. La presencia de un ejemplo vulnerable en el contexto puede incrementar la probabilidad de reproducción (Pearce et al., 2022; He & Vechev, 2023).

La segunda familia comprende seguridad de memoria y aritmética. Desbordamientos de búfer, accesos fuera de límites, uso después de liberar, punteros nulos y desbordamientos enteros predominan en C/C++. FormAI y FormAI-v2 confirman que compilar no elimina estos riesgos (Tihanyi et al., 2023, 2025). La dificultad aumenta en repositorios porque la seguridad de una operación depende del tamaño calculado en otra función, de la propiedad de la memoria y de invariantes que no caben en una ventana de contexto parcial.

La tercera familia agrupa criptografía, credenciales y APIs sensibles. Se observaron secretos codificados, algoritmos o modos obsoletos, semillas predecibles, validación deficiente de certificados y parámetros inseguros. En 48 tareas de cinco APIs Java, Mousavi et al. (2024) encontraron uso incorrecto en alrededor de 70 % de las instancias y 20 tipos de mal uso; en aproximadamente la mitad de las tareas la tasa alcanzó 100 %. La sintaxis válida y el nombre correcto de la biblioteca pueden ocultar una semántica criptográfica inadecuada.

La cuarta familia incluye exposición de datos, autenticación, autorización y configuraciones. Los patrones comprenden mensajes de error detallados, registros con datos sensibles, control de acceso ausente, tokens manejados sin protección, cookies o cabeceras inseguras, permisos amplios y modo de depuración. Estas decisiones suelen quedar fuera de la solicitud principal y el modelo completa valores por defecto. La salida puede funcionar en demostración, pero viola el principio de mínimo privilegio al desplegarse.

La quinta familia corresponde a dependencias y cadena de suministro. Además de paquetes inexistentes, se observaron importaciones no verificadas, ausencia de versión fijada y recomendación de componentes sin evaluación. La alucinación de paquetes es un patrón especialmente distintivo de los modelos: no requiere que la implementación contenga todavía una instrucción maliciosa; basta con que el flujo de instalación confíe en un nombre inventado (Spracklen et al., 2025).

La sexta familia reúne disponibilidad, excepciones y recursos. Incluye bucles o recursión sin límites, expresiones regulares costosas, lectura de archivos sin restricciones, ausencia de tiempo de espera y manejo de excepciones que ignora condiciones críticas. El modelo tiende a optimizar la respuesta para el caso nominal y a tratar los errores como detalles secundarios. Esa prioridad produce código breve y convincente, pero frágil ante cargas, entradas grandes o estados parciales.

La séptima familia es transversal: falsa seguridad derivada de plausibilidad. Se manifiesta cuando el modelo afirma que el código es seguro sin detectar todas las debilidades, cuando repara una alerta y crea otra, o cuando el usuario interpreta la explicación fluida como evidencia. Khoury et al. (2023) comprobaron que ChatGPT podía reconocer algunas vulnerabilidades después de ser interrogado, pero aun así generaba programas por debajo de estándares mínimos. La autorrevisión es útil como filtro auxiliar, no como oráculo independiente.

Tabla 2. Taxonomía de vulnerabilidades y patrones de programación insegura

Familia	CWE/manifestaciones	Patrón generativo recurrente	Control prioritario
Entradas e inyección	CWE-20, 22, 78, 79, 89, 94	Concatenar o interpolar datos no confiables; escape parcial.	Consultas parametrizadas, listas permitidas y pruebas adversarias.
Memoria y enteros	CWE-119, 125, 190, 416, 476, 787	Omitir límites, propiedad, signo, tamaño o ciclo de vida.	Lenguajes seguros, sanitizadores, verificación y fuzzing.
Criptografía y secretos	CWE-295, 327, 330, 798	Elegir defaults, algoritmos, semillas o credenciales inseguros.	Bibliotecas aprobadas, gestores de secretos y revisión experta.
Acceso y datos	CWE-200, 209, 269, 285, 306, 532	Asumir confianza, registrar datos o conceder permisos amplios.	Mínimo privilegio, pruebas de autorización y revisión de logs.
Dependencias	Paquetes inexistentes, sin fijar o no evaluados	Recomendar nombres plausibles sin comprobar el registro.	Allowlist, SCA, lockfiles, firmas y repositorio interno.
Disponibilidad	CWE-400, 770, 1333; timeouts y errores	Diseñar solo el camino feliz y omitir cuotas o cancelación.	Límites, timeouts, manejo seguro de fallos y pruebas de carga.
Falsa seguridad	Autorrevisión incompleta; reparación regresiva	Confundir explicación convincente con evidencia verificable.	Oráculos independientes, doble revisión y trazabilidad.

Nota. La tabla agrupa mecanismos, no estima prevalencias. Un fragmento puede pertenecer a varias familias y una misma CWE puede originarse por patrones diferentes.

Condiciones que modifican el riesgo

El lenguaje es un modificador importante. C y C++ exponen directamente memoria y aritmética, por lo que los estudios detectan más fallas de límites y punteros. Python reduce algunas clases de corrupción de memoria, pero conserva inyección, rutas, deserialización, secretos, dependencias y errores de lógica. Java ofrece administración de memoria, pero no evita el uso inseguro de APIs criptográficas ni problemas web. La comparación entre lenguajes no debe interpretarse como una clasificación absoluta: los conjuntos de tareas y los oráculos difieren.

El prompt modifica la salida, aunque no ofrece garantía. Incluir el término “seguro”, una debilidad específica o una política puede disminuir determinadas fallas; también puede degradar

corrección o desplazar el error. Tony et al. (2025) encontraron beneficios de estrategias iterativas de crítica y mejora, mientras CodeSecEval mostró que información explícita sobre vulnerabilidades favorece generación y reparación en parte de los casos (Wang et al., 2024). Sin embargo, CWEval advierte que un benchmark debe medir simultáneamente funcionalidad y seguridad para evitar premiar soluciones que se limitan a rechazar, eliminar o romper el comportamiento requerido (Peng et al., 2025).

La temperatura, el muestreo y la no determinación generan variación: una solicitud puede producir una alternativa segura y otra vulnerable. Por ello, seleccionar el primer resultado no es equivalente a evaluar la capacidad del modelo. Debe registrarse versión, fecha, parámetros, prompt, contexto, número de intentos y salida aceptada. Sin esa trazabilidad no es posible reproducir la evaluación ni identificar qué fragmento llegó a producción.

El método de detección condiciona el hallazgo. El análisis estático escala, pero produce falsos positivos y no siempre reconstruye flujo entre archivos; la verificación formal aporta contraejemplos dentro de límites; las pruebas dinámicas confirman comportamientos concretos pero dependen del oráculo; la revisión humana comprende intención y lógica de negocio, aunque es costosa y falible. CyberSecEval reportó alta precisión de su detector, pero una recuperación incompleta (Bhatt et al., 2023); la evolución hacia SecCodePLT y CWEval responde precisamente a la necesidad de oráculos ejecutables y evaluación conjunta (Yang et al., 2024; Peng et al., 2025).

El perfil del usuario introduce un riesgo de automatización. Perry et al. (2023) observaron peores resultados de seguridad en tareas específicas y una percepción elevada de seguridad entre usuarios asistidos. Sandoval et al. (2023) hallaron un efecto más pequeño y no concluyente en su tarea de C. La encuesta de Kudriavtseva et al. (2025) muestra que los desarrolladores reconocen beneficios y riesgos, pero sus prácticas de verificación no son uniformes. La conclusión común es que aceptar o modificar una sugerencia es una decisión humana que requiere competencia de seguridad.

El contexto del repositorio cambia la dificultad. Una función aislada puede parecer segura si se ignoran llamadas, invariantes, configuraciones y dependencias. SecRepoBench mostró que el rendimiento en programas autocontenidos no se traslada directamente a repositorios reales (Shen et al., 2026). Este resultado cuestiona los rankings basados en ejercicios breves y exige evaluar cambios sobre el sistema completo, incluidas pruebas de regresión y análisis de nuevas debilidades.

Controles y mitigaciones con respaldo empírico

Ningún control aislado elimina el riesgo. La mejora de prompts puede reducir fallas conocidas, pero depende de que el usuario conozca la amenaza. El análisis estático detecta patrones, pero no demuestra explotación. Pedir al mismo modelo que revise su salida puede corregir algunos hallazgos; Fu et al. (2025) lograron reparar hasta 55,5 % de las debilidades al proporcionar mensajes de herramientas, lo que deja un conjunto sustantivo sin resolver. La arquitectura más defendible distribuye la confianza entre especificación, herramientas independientes y personas.

La primera barrera es prevenir requisitos incompletos. La solicitud debe identificar entradas no confiables, activos, privilegios, dependencias autorizadas, requisitos de privacidad y condiciones de abuso. Esto no implica convertir cada prompt en una auditoría, sino trasladar al modelo los requisitos negativos que normalmente quedan implícitos: no registrar secretos, no construir SQL por concatenación, validar rutas, usar algoritmos aprobados y fallar de manera segura.

La segunda barrera es automatizar verificación en el flujo de integración. Deben ejecutarse compilación estricta, linters, SAST, análisis de composición de software, detección de secretos, pruebas unitarias y pruebas de seguridad. Para código nativo se agregan sanitizadores, fuzzing y comprobaciones de límites; para aplicaciones web, pruebas de autorización, inyección, gestión de sesiones y cabeceras. Los hallazgos deben bloquear la integración según severidad, no depender de una recomendación opcional.

La tercera barrera es la revisión humana informada. El revisor necesita conocer el requisito y el modelo de amenazas, no limitarse a estilo o legibilidad. Debe inspeccionar fronteras de confianza, transformación de datos, permisos, llamadas a red, criptografía, manejo de errores y procedencia de dependencias. El código generado se trata como una contribución externa no confiable, aunque provenga de una herramienta institucional.

La cuarta barrera es controlar la cadena de suministro. Los nombres de paquetes deben resolverse en registros autorizados; las versiones, hashes y licencias deben quedar fijados; las dependencias nuevas requieren justificación y análisis. Las organizaciones pueden aplicar listas permitidas, proxies internos y firmas. Esta barrera responde a una clase de riesgo que la revisión de la lógica local no detecta.

La quinta barrera es conservar trazabilidad y observar producción. Deben registrarse modelo, versión, prompt, archivos modificados, pruebas ejecutadas y revisor. En operación se mantienen mínimo privilegio, telemetría, alertas y capacidad de reversión. El Secure Software Development Framework recomienda integrar prácticas de seguridad a lo largo del ciclo y proteger los artefactos (NIST, 2022); la IA cambia la fuente de la propuesta, pero no reduce esas obligaciones.

Figura 2. Estrategia de aseguramiento en profundidad

Nota. Las capas se complementan. La salida del modelo no debe avanzar a producción solo por compilar, superar pruebas funcionales o recibir una autodeclaración de seguridad.

Protocolo mínimo de aceptación

La propuesta de aseguramiento puede traducirse en puertas verificables. La primera pregunta no es si el fragmento fue escrito por una persona o por un modelo, sino si existe evidencia suficiente para aceptar el riesgo residual. Una política que solo exige declarar el uso de IA aporta trazabilidad, pero no demuestra seguridad. El criterio de aceptación debe vincular cada riesgo con un artefacto: una prueba, un informe, una revisión o una decisión formal.

La severidad y la exposición determinan la profundidad. Un script local sin datos sensibles puede requerir revisión básica; una función que procesa autenticación, pagos, infraestructura o información personal necesita pruebas negativas, revisión especializada y control de dependencias. Cuando el requisito o el contexto son incompletos, la respuesta correcta es detener la integración y aclarar supuestos. La presión por aprovechar velocidad no debe convertir la ausencia de evidencia en evidencia de ausencia.

El protocolo de la Tabla 3 separa funcionalidad, seguridad y procedencia. Si una puerta falla, el equipo corrige el código, modifica el diseño o documenta una excepción aprobada; no se acepta la afirmación del modelo como sustituto. La evidencia queda asociada con el cambio para que una actualización de dependencia, de modelo o de requisito active una nueva revisión.

Tabla 3. Puertas mínimas para aceptar código asistido por IA

Puerta	Evidencia mínima	Criterio de bloqueo
Procedencia	Modelo/versión, prompt, fecha y fragmentos modificados.	Origen o alcance no identificable.
Funcionalidad	Compilación y pruebas positivas, negativas y de regresión.	Fallo, cobertura crítica ausente o comportamiento ambiguo.
Seguridad	SAST, secretos, SCA y pruebas según el modelo de amenazas.	Hallazgo alto/crítico no resuelto o sin evaluación.
Dependencias	Paquete existente, versión fijada, licencia y procedencia verificadas.	Nombre inexistente, paquete no aprobado o integridad desconocida.
Revisión	Aprobación humana independiente en cambios sensibles.	El autor y el único revisor son el mismo modelo.
Operación	Mínimo privilegio, observabilidad, reversión y responsable.	No existe contención ni respuesta frente a un fallo.

Nota. El umbral concreto debe adaptarse a la criticidad del sistema; las puertas no sustituyen una evaluación de riesgos, sino que la convierten en evidencia auditable.

Discusión

Interpretación central

La síntesis respalda una conclusión matizada: los modelos generativos producen código vulnerable con frecuencia no trivial, pero la magnitud no es una constante del modelo. Cambia con el lenguaje, la tarea, el contexto, la versión, el muestreo y el oráculo. Por esa razón, afirmar que la IA genera siempre más vulnerabilidades que las personas sería tan débil como asumir que un modelo reciente es seguro por superar HumanEval. La comparación relevante exige la misma especificación, el mismo presupuesto de intentos y una evaluación conjunta de funcionalidad y seguridad.

Los estudios con resultados divergentes ayudan a precisar el mecanismo. Perry et al. (2023) encontraron deterioro en varias tareas; Sandoval et al. (2023) observaron diferencias pequeñas y Asare et al. (2023) documentaron casos en que Copilot evitó fallos humanos. Las tres piezas son

compatibles si se entiende la asistencia como sistema humano–modelo. La herramienta puede aportar una implementación segura, una insegura o varias alternativas; el resultado final depende de qué sugiere, qué reconoce el usuario, qué acepta y qué verifica el proceso.

La corrección funcional es una condición necesaria, no suficiente. Un programa incapaz de cumplir el requisito no constituye una solución segura; un programa funcional puede conservar una ruta explotable. Los benchmarks más recientes avanzan hacia métricas conjuntas porque contar solo alertas premia código incompleto y medir solo pruebas funcionales ignora el adversario. CWEval, SecCodePLT y SecRepoBench sitúan esta tensión en el centro (Peng et al., 2025; Yang et al., 2024; Shen et al., 2026).

Las familias identificadas muestran que los modelos reproducen tanto vulnerabilidades clásicas como fallas propias de la generación probabilística. Inyección, memoria, secretos y control de acceso son problemas conocidos del desarrollo humano; la alucinación de dependencias, la variación entre respuestas y la falsa seguridad expresada con fluidez añaden mecanismos específicos. El aseguramiento debe conservar controles tradicionales y sumar controles de procedencia, trazabilidad y validación de recomendaciones.

La comparación de escalas revela otra lección. Un benchmark con prompts diseñados para activar una CWE mide conocimiento local; un repositorio mide comprensión de dependencias y contexto; un estudio con usuarios mide selección, confianza y revisión. Ningún diseño representa por sí solo la seguridad del software. Una agenda acumulativa necesita declarar con precisión qué capa estudia y evitar extrapolar una tasa de fragmentos a sistemas desplegados.

Las herramientas de detección dedicadas aportan una ruta práctica. DeVAIC demuestra que reglas adaptadas a fragmentos incompletos pueden mejorar cobertura en Python (Cotroneo et al., 2025); SALLM y CodeSecEval ofrecen tareas centradas en seguridad (Siddiq et al., 2024; Wang et al., 2024); CodeLMSec y CyberSecEval permiten comparar modelos a escala (Hajipour et al., 2024; Bhatt et al., 2023). Sin embargo, cada herramienta delimita un universo de debilidades. La ausencia de una alerta significa “no detectado bajo estas reglas”, no “seguro”.

Implicaciones para el desarrollo y la formación

En equipos profesionales, la política debería clasificar el código generado como insumo sujeto al mismo control que una contribución externa. Los fragmentos triviales no están exentos: una línea que arma una consulta, carga una dependencia o configura TLS puede concentrar el riesgo. La revisión basada en riesgo permite asignar mayor rigor a autenticación, pagos, datos personales, infraestructura, criptografía y software expuesto a red.

La productividad no debe medirse solo por tiempo de escritura. Si el ahorro inicial se desplaza hacia diagnóstico, auditoría o respuesta a incidentes, la organización recibe una ganancia aparente. Conviene medir proporción de cambios aceptados, defectos escapados, alertas válidas, retrabajo, tiempo de revisión y vulnerabilidades posteriores. La seguridad debe integrarse como guardarraíl del rendimiento y no como inspección tardía.

En educación superior, el uso de asistentes puede convertirse en una oportunidad para enseñar programación segura. Las actividades deberían exigir que el estudiante identifique el límite de confianza, explique la vulnerabilidad, construya una entrada adversaria, compare alternativas y justifique la corrección. La evaluación de un código no debe premiar únicamente que ejecute; debe valorar validación, manejo de errores, secretos, dependencias y pruebas. Así, la IA funciona como objeto de crítica y no como autoridad técnica.

La transparencia es igualmente formativa. Registrar el prompt, la salida original, las modificaciones y los controles ejecutados permite distinguir autoría, razonamiento y verificación. Este procedimiento reduce copia acrítica y prepara para prácticas reales de revisión. La competencia central no es obtener una respuesta del modelo, sino convertir una propuesta probabilística en un artefacto verificable.

Amenazas a la validez y trabajo futuro

La revisión tiene límites. Aunque se consultaron las principales bibliotecas del área y se aplicó rastreo de referencias, no pretende exhaustividad bibliométrica. El campo cambia con rapidez, algunos trabajos recientes permanecen como preprints y los modelos cerrados pueden modificarse sin conservar una versión accesible. Los resultados describen la evidencia disponible hasta el 22 de agosto de 2026 y deben actualizarse cuando aparezcan réplicas o versiones editoriales.

La validez de constructo está afectada por la definición de seguridad. Algunos estudios cuentan alertas, otros vulnerabilidades confirmadas, tasas por programa, instancias por CWE o respuestas correctas y seguras. La taxonomía reduce fragmentación conceptual, pero no elimina las diferencias. Por ello no se calculó una prevalencia combinada ni se ordenaron modelos con cifras obtenidas de tareas distintas.

Existe riesgo de sesgo de publicación hacia resultados negativos o llamativos. También hay concentración en Python, C/C++, Java y JavaScript, con menor evidencia en móviles, sistemas embebidos, infraestructura como código y lenguajes recientes. Los corpus públicos pueden estar presentes en datos de entrenamiento, y la contaminación puede inflar corrección sin demostrar razonamiento seguro.

La investigación futura debería ejecutar estudios preregistrados con versiones fijadas, prompts archivados y oráculos independientes. Se necesitan tareas multiarquivo con pruebas funcionales, pruebas de explotación y búsqueda de vulnerabilidades nuevas; comparaciones entre revisión humana, análisis automatizado y agentes; y mediciones longitudinales de mantenimiento. También conviene estudiar si la formación en seguridad reduce aceptación acrítica, qué información de una herramienta debe mostrarse al usuario y cuál combinación de controles ofrece la mejor relación entre cobertura y costo.

Conclusiones

La evidencia analizada demuestra que el código generado por modelos de inteligencia artificial puede contener vulnerabilidades clásicas y patrones específicos de la generación probabilística. Las debilidades más consistentes afectan validación de entradas, inyección, memoria, aritmética, criptografía, credenciales, APIs de seguridad, control de acceso, exposición de datos, manejo de recursos y dependencias. La apariencia profesional, la compilación y el cumplimiento de pruebas funcionales no certifican seguridad.

No existe una tasa universal ni una superioridad estable frente al código humano. El resultado depende del lenguaje, el escenario, el prompt, la versión, el usuario y el método de evaluación. Esa heterogeneidad no justifica relajar controles; obliga a tratar cada salida como código no confiable y a verificarla en su contexto de integración.

La mitigación más sólida es el aseguramiento en profundidad: requisitos de seguridad explícitos, selección controlada de dependencias, análisis estático y de composición, pruebas adversarias, revisión humana basada en amenazas, trazabilidad y operación con mínimo privilegio. El modelo

puede participar en la generación o en una primera crítica, pero no debe actuar como único autor, revisor y certificador de su propia salida.

Para la investigación, el reto es avanzar desde fragmentos aislados hacia repositorios, agentes y mantenimiento real. Para la formación y la práctica profesional, el reto es desarrollar criterio de verificación. El valor de la IA generativa dependerá menos de cuántas líneas produzca y más de la capacidad del proceso para demostrar que esas líneas son funcionales, mantenibles y seguras.

Referencias

Asare, O., Nagappan, M., & Asokan, N. (2023). Is GitHub's Copilot as bad as humans at introducing vulnerabilities in code? Empirical Software Engineering, 28(6), Article 129. https://doi.org/10.1007/s10664-023-10380-1

Bhatt, M., Chennabasappa, S., Nikolaidis, C., Wan, S., Evtimov, I., Gabi, D., Song, D., Ahmad, F., Aschermann, C., Fontana, L., Frolov, S., Giri, R. P., Kapil, D., Kozyrakis, Y., LeBlanc, D., Milazzo, J., Straumann, A., Synnaeve, G., Vontimitta, V., Whitman, S., & Saxe, J. (2023). Purple Llama CyberSecEval: A secure coding benchmark for language models. arXiv. https://doi.org/10.48550/arXiv.2312.04724

Cotroneo, D., De Luca, R., & Liguori, P. (2025). DeVAIC: A tool for security assessment of AI-generated code. Information and Software Technology, 177, 107572. https://doi.org/10.1016/j.infsof.2024.107572

Fu, Y., Liang, P., Tahir, A., Li, Z., Shahin, M., Yu, J., & Chen, J. (2025). Security weaknesses of Copilot-generated code in GitHub projects: An empirical study. ACM Transactions on Software Engineering and Methodology, 34(8), Article 218. https://doi.org/10.1145/3716848

Hajipour, H., Hassler, K., Holz, T., Schönherr, L., & Fritz, M. (2024). CodeLMSec benchmark: Systematically evaluating and finding security vulnerabilities in black-box code language models. 2024 IEEE Conference on Secure and Trustworthy Machine Learning, 684–709. https://doi.org/10.1109/SaTML59370.2024.00040

Hamer, S., d'Amorim, M., & Williams, L. (2024). Just another copy and paste? Comparing the security vulnerabilities of ChatGPT generated code and StackOverflow answers. 2024 IEEE Security and Privacy Workshops, 87–94. https://doi.org/10.1109/SPW63631.2024.00014

He, J., & Vechev, M. (2023). Large language models for code: Security hardening and adversarial testing. Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, 1865–1879. https://doi.org/10.1145/3576915.3623175

International Organization for Standardization. (2023). ISO/IEC 25010:2023 systems and software engineering—Systems and software Quality Requirements and Evaluation (SQuaRE)—Product quality model. https://www.iso.org/standard/78176.html

Khoury, R., Avila, A. R., Brunelle, J., & Camara, B. M. (2023). How secure is code generated by ChatGPT? 2023 IEEE International Conference on Systems, Man, and Cybernetics, 2445–2451. https://doi.org/10.1109/SMC53992.2023.10394237

Kitchenham, B., & Charters, S. (2007). Guidelines for performing systematic literature reviews in software engineering (EBSE Technical Report EBSE-2007-01). Keele University and Durham University.

Kudriavtseva, A., Hotak, N. A., & Gadyatskaya, O. (2025). My code is less secure with Gen AI: Surveying developers' perceptions of the impact of code generation tools on security. Proceedings of the 40th ACM/SIGAPP Symposium on Applied Computing, 1637–1646. https://doi.org/10.1145/3672608.3707778

MITRE. (2025). 2025 CWE Top 25 most dangerous software weaknesses. https://cwe.mitre.org/top25/archive/2025/2025_cwe_top25.html

Mousavi, Z., Islam, C., Moore, K., Abuadbba, A., & Babar, M. A. (2024). An investigation into misuse of Java security APIs by large language models. Proceedings of the 19th ACM Asia Conference on Computer and Communications Security, 1045–1059. https://doi.org/10.1145/3634737.3661134

National Institute of Standards and Technology. (2022). Secure Software Development Framework (SSDF) version 1.1: Recommendations for mitigating the risk of software vulnerabilities (NIST SP 800-218). https://doi.org/10.6028/NIST.SP.800-218

Negri-Ribalta, C., Geraud-Stewart, R., Sergeeva, A., & Lenzini, G. (2024). A systematic literature review on the impact of AI models on the security of code generation. Frontiers in Big Data, 7, 1386720. https://doi.org/10.3389/fdata.2024.1386720

OWASP Foundation. (2021). OWASP Top 10: The ten most critical web application security risks. https://owasp.org/Top10/

Page, M. J., McKenzie, J. E., Bossuyt, P. M., Boutron, I., Hoffmann, T. C., Mulrow, C. D., Shamseer, L., Tetzlaff, J. M., Akl, E. A., Brennan, S. E., Chou, R., Glanville, J., Grimshaw, J. M., Hróbjartsson, A., Lalu, M. M., Li, T., Loder, E. W., Mayo-Wilson, E., McDonald, S., ... Moher, D. (2021). The PRISMA 2020 statement: An updated guideline for reporting systematic reviews. BMJ, 372, n71. https://doi.org/10.1136/bmj.n71

Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., & Karri, R. (2022). Asleep at the keyboard? Assessing the security of GitHub Copilot's code contributions. 2022 IEEE Symposium on Security and Privacy, 754–768. https://doi.org/10.1109/SP46214.2022.9833571

Peng, J., Cui, L., Huang, K., Yang, J., & Ray, B. (2025). CWEval: Outcome-driven evaluation on functionality and security of LLM code generation. arXiv. https://doi.org/10.48550/arXiv.2501.08200

Perry, N., Srivastava, M., Kumar, D., & Boneh, D. (2023). Do users write more insecure code with AI assistants? Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, 2785–2799. https://doi.org/10.1145/3576915.3623157

Rabbi, M. F., Champa, A. I., Zibran, M. F., & Islam, M. R. (2024). AI writes, we analyze: The ChatGPT Python code saga. Proceedings of the 21st International Conference on Mining Software Repositories, 177–181. https://doi.org/10.1145/3643991.3645076

Sandoval, G., Pearce, H., Nys, T., Karri, R., Garg, S., & Dolan-Gavitt, B. (2023). Lost at C: A user study on the security implications of large language model code assistants. 32nd USENIX Security Symposium, 2205–2222. https://www.usenix.org/conference/usenixsecurity23/presentation/sandoval

Shen, C., Dilgren, C., Chiniya, P., Griffith, L., Ding, Y., & Chen, Y. (2026). SecRepoBench: Benchmarking code agents for secure code completion in real-world repositories. Proceedings of the 3rd International Workshop on Large Language Models for Code, 159–166. https://doi.org/10.1145/3786181.3788703

Siddiq, M. L., & Santos, J. C. S. (2022). SecurityEval dataset: Mining vulnerability examples to evaluate machine learning-based code generation techniques. Proceedings of the 1st International Workshop on Mining Software Repositories Applications for Privacy and Security, 29–33. https://doi.org/10.1145/3549035.3561184

Siddiq, M. L., Santos, J. C. S., Devareddy, S., & Muller, A. (2024). SALLM: Security assessment of generated code. Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering Workshops, 54–65. https://doi.org/10.1145/3691621.3694934

Spracklen, J., Wijewickrama, R., Sakib, A. H. M. N., Maiti, A., Viswanath, B., & Jadliwala, M. (2025). We have a package for you! A comprehensive analysis of package hallucinations by code generating LLMs. 34th USENIX Security Symposium, 3687–3706. https://www.usenix.org/conference/usenixsecurity25/presentation/spracklen

Tambon, F., Moradi Dakhel, A., Nikanjam, A., Khomh, F., Desmarais, M. C., & Antoniol, G. (2025). Bugs in large language models generated code: An empirical study. Empirical Software Engineering, 30, Article 65. https://doi.org/10.1007/s10664-025-10614-4

Tihanyi, N., Bisztray, T., Ferrag, M. A., Jain, R., & Cordeiro, L. C. (2025). How secure is AI-generated code: A large-scale comparison of large language models. Empirical Software Engineering, 30, Article 47. https://doi.org/10.1007/s10664-024-10590-1

Tihanyi, N., Bisztray, T., Jain, R., Ferrag, M. A., Cordeiro, L. C., & Mavroeidis, V. (2023). The FormAI dataset: Generative AI in software security through the lens of formal verification. Proceedings of the 19th International Conference on Predictive Models and Data Analytics in Software Engineering, 33–43. https://doi.org/10.1145/3617555.3617874

Tony, C., Díaz Ferreyra, N. E., Mutas, M., Dhif, S., & Scandariato, R. (2025). Prompting techniques for secure code generation: A systematic investigation. ACM Transactions on Software Engineering and Methodology, 34(8), 1–53. https://doi.org/10.1145/3722108

Wang, J., Luo, X., Cao, L., He, H., Huang, H., Xie, J., Jatowt, A., & Cai, Y. (2024). Is your AI-generated code really safe? Evaluating large language models on secure code generation with CodeSecEval. arXiv. https://doi.org/10.48550/arXiv.2407.02395

Wang, Y., Zhang, Z., Wang, C., Xu, X., Liu, M., Wang, Y., Chen, J., & Zheng, Z. (2026). RealSec-bench: A benchmark for evaluating secure code generation in real-world repositories. arXiv. https://doi.org/10.48550/arXiv.2601.22706

Wohlin, C. (2014). Guidelines for snowballing in systematic literature studies and a replication in software engineering. Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering, Article 38. https://doi.org/10.1145/2601248.2601268