

Errores lógicos, vulnerabilidades y malas prácticas en código Python generado por inteligencia artificial: taxonomía y evaluación de técnicas de detección

Logical Errors, Vulnerabilities and Bad Practices in AI-Generated Python Code: A Taxonomy and an Evaluation of Detection Techniques

Errori logici, vulnerabilità e cattive pratiche nel codice Python generato dall'intelligenza artificiale: tassonomia e valutazione delle tecniche di rilevamento

Rodrigo José Pazmiño Pérez¹

Email: rjpazmino@uagraria.edu.ec

ORCID: <https://orcid.org/0009-0000-9507-6865>

Romulo Steev Vélez Martínez²

Email: rvelezm@unemi.edu.ec

ORCID: <https://orcid.org/0009-0009-6928-0016>

Elisa Flor Guamán Troya³

Email: fguamant@unemi.edu.ec

ORCID: <https://orcid.org/0009-0005-3898-5161>

Jhonny Roberto Pilco Llongo⁴

Email: ab.jhonny.pilco@gmail.com

ORCID: <https://orcid.org/0009-0008-6059-4046>

Francisco Paolo Espinel Obregoso⁵

Email: fespinelo@unemi.edu.ec

ORCID: <https://orcid.org/0000-0002-7666-057X>

Correspondencia: Rodrigo José Pazmiño Pérez¹ Email: rjpazmino@uagraria.edu.ec

Artículo de investigación

* **Recibido:** 15-09-2026 * **Aceptado:** 01-10-2026 * **Publicado:** 11-10-2026

1. Universidad Agraria del Ecuador, Guayaquil, Ecuador.
2. Facultad de Posgrado - Universidad Estatal de Milagro, Ecuador.
3. Facultad de Posgrado - Universidad Estatal de Milagro, Ecuador
4. Universidad de Guayaquil, Guayaquil, Ecuador.
5. Facultad de Posgrado - Universidad Estatal de Milagro, Ecuador

Resumen

Los asistentes de programación basados en modelos de lenguaje de gran tamaño se han incorporado con rapidez a la formación y a la práctica profesional del desarrollo de software, y Python es uno de los lenguajes en los que más se emplean. La evidencia empírica acumulada muestra, sin embargo, que el código que producen puede contener errores lógicos,

vulnerabilidades de seguridad y malas prácticas que no siempre son evidentes para quien lo recibe. Este artículo tuvo dos propósitos: organizar en una taxonomía los defectos documentados en código Python generado por inteligencia artificial y evaluar qué técnicas de verificación permiten detectarlos. Se siguió un enfoque mixto en dos fases. En la primera se realizó una revisión documental de estudios empíricos publicados entre 2022 y 2025. En la segunda se construyó un corpus de 35 casos, cada uno con una versión defectuosa y otra corregida, que reproducen los patrones identificados (10 errores lógicos, 15 vulnerabilidades asociadas a la clasificación CWE y 10 malas prácticas), y se analizó con tres herramientas de análisis estático (Bandit, Ruff y Pylint) y con pruebas derivadas de la especificación. Bandit detectó 13 de las 15 vulnerabilidades, pero ninguno de los errores lógicos; Ruff y Pylint detectaron 3 de los 10 errores lógicos; el recorrido de directorios y la ausencia de control de autorización escaparon a todas las herramientas estáticas y solo fueron evidenciados por pruebas. Se concluye que ninguna técnica aislada basta: la verificación del código generado por inteligencia artificial requiere combinar análisis estático de seguridad y de estilo con pruebas basadas en la especificación y con revisión humana, una combinación que debe enseñarse de forma explícita en los cursos de programación.

Palabras clave: inteligencia artificial generativa, Python, seguridad del software, análisis estático, calidad del código.

Abstract

Programming assistants based on large language models have quickly entered both software development education and professional practice, and Python is one of the languages in which they are most widely used. The accumulated empirical evidence shows, however, that the code they produce may contain logical errors, security vulnerabilities and bad practices that are not always evident to those who receive it. This article had two aims: to organize the defects documented in AI-generated Python code into a taxonomy and to evaluate which verification techniques can detect them. A two-phase mixed approach was followed. In the first phase, a documentary review of empirical studies published between 2022 and 2025 was carried out. In the second, a corpus of 35 cases was built, each with a defective and a corrected version, reproducing the identified patterns (10 logical errors, 15 vulnerabilities mapped to the CWE classification and 10 bad practices), and it was analyzed with three static analysis tools (Bandit, Ruff and Pylint) and with specification-based tests. Bandit detected 13 of the 15 vulnerabilities but none of the logical errors; Ruff and Pylint detected 3 of the 10 logical errors; path traversal and missing authorization escaped all static tools and were revealed only by tests. It is concluded that no single technique is sufficient: verifying AI-generated code requires combining security and style static analysis with specification-based testing and human review, a combination that should be explicitly taught in programming courses.

Keywords: generative artificial intelligence, Python, software security, static analysis, code quality.

Riassunto

Gli assistenti di programmazione basati su modelli linguistici di grandi dimensioni sono entrati rapidamente nella formazione e nella pratica professionale dello sviluppo software, e Python è uno dei linguaggi in cui vengono utilizzati di più. Le evidenze empiriche accumulate mostrano, tuttavia, che il codice che producono può contenere errori logici, vulnerabilità di sicurezza e cattive pratiche non sempre evidenti per chi lo riceve. Questo articolo ha avuto due obiettivi: organizzare in una tassonomia i difetti documentati nel codice Python generato dall'intelligenza artificiale e valutare quali tecniche di verifica consentono di rilevarli. È stato adottato un approccio misto in due fasi. Nella prima è stata svolta una revisione documentale di studi empirici pubblicati tra il 2022 e il 2025. Nella seconda è stato costruito un corpus di 35 casi, ciascuno con una versione difettosa e una corretta, che riproducono i modelli individuati (10 errori logici, 15 vulnerabilità ricondotte alla classificazione CWE e 10 cattive pratiche), analizzato con tre strumenti di analisi statica (Bandit, Ruff e Pylint) e con test derivati dalla specifica. Bandit ha rilevato 13 delle 15 vulnerabilità ma nessuno degli errori logici; Ruff e Pylint hanno rilevato 3 dei 10 errori logici; l'attraversamento di directory e l'assenza di controllo delle autorizzazioni sono sfuggiti a tutti gli strumenti statici e sono stati evidenziati solo dai test. Si conclude che nessuna tecnica isolata è sufficiente: la verifica del codice generato dall'intelligenza artificiale richiede di combinare l'analisi statica di sicurezza e di stile con test basati sulla specifica e con la revisione umana, una combinazione che deve essere insegnata esplicitamente nei corsi di programmazione.

Parole chiave: intelligenza artificiale generativa, Python, sicurezza del software, analisi statica, qualità del codice.

Introducción

La generación automática de código dejó de ser una promesa de laboratorio para convertirse en una práctica cotidiana. Los asistentes basados en modelos de lenguaje de gran tamaño completan funciones, proponen soluciones a ejercicios y escriben programas enteros a partir de una descripción en lenguaje natural. En la educación superior su llegada fue inmediata: Finnie-Ansley et al. (2022) mostraron que un modelo como Codex resolvía buena parte de los ejercicios de un curso introductorio de programación, y Becker et al. (2023) advirtieron que esta capacidad obliga a replantear qué se enseña y cómo se evalúa. Denny et al. (2024) describen el momento actual como el inicio de una nueva etapa para la enseñanza de la computación, en la que leer, evaluar y corregir código ajeno pesa tanto como escribirlo.

El entusiasmo por la productividad contrasta con lo que revelan los estudios que examinan la calidad de ese código. Pearce et al. (2022) diseñaron 89 escenarios orientados a debilidades conocidas y encontraron que alrededor del 40 % de los 1689 programas sugeridos por GitHub Copilot eran vulnerables. Fu et al. (2025), al analizar fragmentos generados por asistentes y efectivamente incorporados a proyectos de GitHub, identificaron debilidades de seguridad en el 29,5 % de los fragmentos en Python, distribuidas en 43 categorías de la enumeración CWE (Common Weakness Enumeration). En el plano funcional, Liu et al. (2024) evaluaron 4066 programas generados por ChatGPT para 2033 tareas en Java y Python: 1082 producían resultados incorrectos, 177 presentaban errores de compilación o de ejecución y 1930 fragmentos mostraban problemas de mantenibilidad. Tambon et al. (2025), por su parte, clasificaron 333 errores de código generado por tres modelos y propusieron una taxonomía de diez patrones que va desde la mala interpretación del enunciado hasta la invención de objetos o atributos inexistentes.

A estos riesgos técnicos se suma uno humano. Perry et al. (2023) compararon a programadores que trabajaban con y sin un asistente de inteligencia artificial y hallaron que quienes lo utilizaban escribían código significativamente menos seguro y, al mismo tiempo, tendían a creer que su código era seguro. La combinación de defectos frecuentes y confianza excesiva resulta especialmente delicada en contextos formativos, donde el estudiante todavía no dispone del criterio necesario para reconocer un error sutil. La revisión sistemática de Negri-Ribalta et al. (2024) concluye en la misma dirección: la literatura coincide en que los modelos de inteligencia artificial no producen código seguro por defecto y en que se necesitan procesos de verificación adaptados a este tipo de desarrollo.

Python merece una atención particular. Es el lenguaje con el que se inician muchos estudiantes universitarios, predominante en conjuntos de evaluación de modelos generadores de código como HumanEval (Liu et al., 2023) y uno de los más usados en ciencia de datos y automatización. Su sintaxis permisiva y su tipado dinámico facilitan que un programa con defectos se ejecute sin advertencias hasta que encuentra la entrada que lo hace fallar. A ello se añade un riesgo propio de su ecosistema de paquetes: Spracklen et al. (2025) analizaron 576 000

muestras generadas por 16 modelos y encontraron que, en promedio, el 5,2 % de los paquetes sugeridos por modelos comerciales y el 21,7 % de los sugeridos por modelos de código abierto no existían, lo que abre la puerta a que un atacante publique un paquete malicioso con el nombre inventado.

La literatura ofrece abundantes diagnósticos, pero estos se encuentran dispersos entre estudios de seguridad, de corrección funcional y de calidad, con taxonomías distintas y escasa orientación práctica sobre cómo detectar cada tipo de defecto. Para el docente o el desarrollador que recibe código generado por inteligencia artificial, la pregunta relevante no es solo qué puede fallar, sino con qué herramientas puede descubrirlo. En consecuencia, este estudio se planteó tres preguntas: (a) ¿qué tipos de errores lógicos, vulnerabilidades y malas prácticas documenta la literatura en código Python generado por inteligencia artificial?; (b) ¿en qué medida las herramientas de análisis estático de uso libre detectan cada tipo de defecto?, y (c) ¿qué combinación de técnicas permite una verificación más completa, y qué implicaciones tiene para la enseñanza de la programación?

Metodología

El estudio adoptó un enfoque mixto de alcance descriptivo, organizado en dos fases complementarias. La primera fue una revisión documental de tipo integrador, adecuada cuando se busca sintetizar hallazgos de estudios con diseños diversos y proponer un marco conceptual (Snyder, 2019). La segunda fue un estudio analítico reproducible en el que se evaluó la capacidad de detección de varias técnicas de verificación sobre un corpus controlado de casos. Ambas fases se articularon de modo que la taxonomía derivada de la revisión sirvió como base para construir los casos.

En la fase de revisión se consultaron trabajos publicados entre 2022 y 2025 en revistas y congresos de ingeniería de software, seguridad informática y educación en computación, localizados en las bibliotecas digitales de ACM e IEEE, en Springer, en Elsevier y en el repositorio arXiv. Se incluyeron estudios empíricos que evaluaran código generado por modelos de lenguaje en términos de corrección funcional, seguridad o calidad, así como revisiones sistemáticas sobre el tema. Se excluyeron opiniones sin datos y trabajos centrados exclusivamente en lenguajes de hardware. Cada referencia se verificó por su identificador digital o por su registro editorial. De cada estudio se extrajeron el tipo de defecto descrito, su frecuencia cuando se informaba y la técnica con la que había sido identificado.

En la fase analítica se construyó un corpus de 35 casos en Python, cada uno compuesto por una función breve con un defecto y por su versión corregida. Los casos se distribuyeron en tres categorías: 10 errores lógicos, basados en la taxonomía de Tambon et al. (2025) y en los errores simples descritos por Jesse et al. (2023); 15 vulnerabilidades, cada una asociada a una debilidad de la enumeración CWE, con prioridad para las incluidas en la lista de las 25 más peligrosas de 2024 (MITRE, 2024); y 10 malas prácticas, basadas en los problemas de calidad y

mantenibilidad informados por Liu et al. (2024) y en la guía de estilo PEP 8 (van Rossum et al., 2001). Es necesario precisar el origen de los casos. Se redactaron de forma controlada, con apoyo de un asistente de inteligencia artificial al que se le indicó en cada caso el patrón documentado que debía reproducir, y la presencia del defecto se comprobó al ejecutar el análisis y, cuando existía, la prueba correspondiente. No son, por tanto, respuestas espontáneas de un asistente a enunciados de programación ni pretenden estimar la frecuencia con que un modelo comete cada error; su propósito es aislar cada patrón para examinar qué técnica lo detecta.

Como instrumentos se emplearon tres analizadores estáticos de uso libre y un marco de pruebas, ejecutados con Python 3.13.16. Bandit 1.9.4 (PyCQA, 2026a) examina el árbol sintáctico en busca de patrones inseguros conocidos. Ruff 0.16.8 (Astral Software, 2026) se configuró con las familias de reglas E, F, W, B, S, A, SIM, C90 y PL, que cubren errores de estilo, defectos probables, reglas de seguridad equivalentes a las de Bandit, sombras de funciones integradas, simplificaciones, complejidad y reglas derivadas de Pylint. Pylint 4.1.2 (PyCQA, 2026b) se ejecutó con su configuración por omisión, excluyendo solo los avisos sobre cadenas de documentación ausentes, que no guardan relación con los defectos estudiados. Finalmente, con pytest 9.1.1 se escribieron pruebas derivadas de la especificación para los 14 casos cuyo comportamiento esperado podía expresarse como una aserción: los 10 errores lógicos, dos vulnerabilidades de lógica de acceso, la inyección de código mediante eval y la notificación de errores por pantalla.

Para que la medición fuera objetiva, la línea defectuosa de cada caso se marcó con un comentario y se consideró que una herramienta detectaba el defecto solo cuando emitía al menos un hallazgo en esa línea; los avisos en otras líneas no se contabilizaron. En el caso de las pruebas, se consideró detección que la prueba fallara con la versión defectuosa. Todas las herramientas se aplicaron también a las versiones corregidas, con el fin de comprobar que las pruebas pasaran y de registrar los avisos que persistían en código sin el defecto. El proceso completo se automatizó en un guion que genera los casos, ejecuta las cuatro técnicas y produce los resultados en formato JSON, de modo que el análisis puede repetirse íntegramente; los guiones y el corpus se ofrecen como material suplementario.

Resultados

Taxonomía de defectos en código Python generado por inteligencia artificial

La revisión permitió agrupar los defectos documentados en tres categorías que responden a preguntas distintas sobre el código. Los errores lógicos responden a si el programa hace lo que se le pidió; las vulnerabilidades, a si puede ser usado en contra de quien lo ejecuta; y las malas prácticas, a si podrá mantenerse, leerse y modificarse sin introducir nuevos fallos. La Tabla 1 resume los subtipos de cada categoría, la evidencia que los respalda y su expresión típica en Python.

Tabla 1. Taxonomía de defectos en código Python generado por inteligencia artificial

Categoría	Subtipos principales	Evidencia	Expresión típica en Python
Errores lógicos	Mala interpretación del enunciado, casos límite omitidos, errores por uno, tipo de entrada equivocado, objetos o atributos inventados, generación incompleta	Tambon et al. (2025); Jesse et al. (2023); Liu et al. (2024); Liu et al. (2023)	División para cero en listas vacías, rangos incompletos, métodos inexistentes como list.push
Vulnerabilidades	Inyección (SQL, comandos, código), deserialización insegura, credenciales embebidas, criptografía débil, aleatoriedad insuficiente, recorrido de directorios, autorización ausente	Pearce et al. (2022); Fu et al. (2025); Khoury et al. (2023); Negri-Ribalta et al. (2024)	Consultas con f-strings, shell=True, eval, pickle, hashlib.md5, random para tokens
Malas prácticas	Manejo deficiente de excepciones, código muerto, recursos sin liberar, nombres que ocultan funciones integradas, complejidad excesiva, estilo no idiomático	Liu et al. (2024); Moradi Dakhel et al. (2023); van Rossum et al. (2001)	except sin tipo, from módulo import *, open sin with, variables llamadas list o id
Riesgos de la cadena de suministro	Dependencias inexistentes o inventadas que pueden ser suplantadas	Spracklen et al. (2025)	Instrucciones import de paquetes que no existen en PyPI

Nota. Elaboración propia a partir de la revisión documental. Los riesgos de la cadena de suministro se registran en la taxonomía, pero no se incluyeron en el corpus porque su detección depende de consultar el índice de paquetes y no del análisis del código.

Dos rasgos atraviesan las tres categorías. El primero es que el código defectuoso suele ser sintácticamente correcto y de apariencia profesional, lo que dificulta su detección por inspección visual. El segundo es que muchos defectos son reproducciones de patrones abundantes en los datos de entrenamiento: Jesse et al. (2023) encontraron que los modelos pueden reproducir errores simples ya conocidos hasta dos veces más que la versión correcta del mismo código, y Asare et al. (2023) observaron que Copilot replicaba las vulnerabilidades introducidas originalmente por personas en alrededor de un tercio de los escenarios evaluados.

DetECCIÓN POR TÉCNICA Y CATEGORÍA

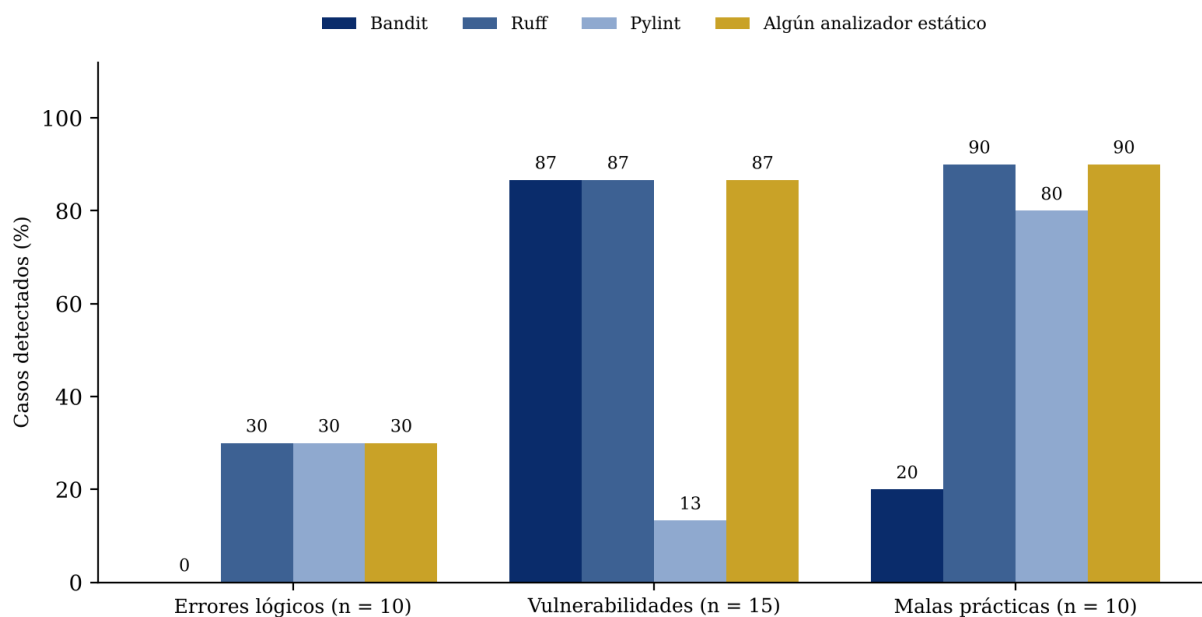
La Tabla 2 agrega por categoría el número de casos detectados por cada técnica, y la Figura 1 representa la parte correspondiente al análisis estático. El contraste más claro aparece entre los errores lógicos y las vulnerabilidades. Bandit, diseñado específicamente para la seguridad, detectó 13 de las 15 vulnerabilidades, pero ninguno de los 10 errores lógicos. Ruff y Pylint detectaron 3 errores lógicos cada uno, siempre los mismos: el argumento por defecto mutable, la captura tardía de la variable de un bucle dentro de una función lambda y la comparación de identidad con un literal. Los siete errores lógicos restantes, entre ellos el error por uno, la división entera en un porcentaje, la regla incompleta de los años bisiestos y la invocación de un método inexistente, solo fueron detectados por las pruebas.

Tabla 2. Casos detectados por técnica y categoría de defecto

Categoría	n	Bandit	Ruff	Pylint	Algún analizador estático	Pruebas	Alguna técnica
Errores lógicos	10	0/10	3/10	3/10	3/10	10/10	10/10
Vulnerabilidades	15	13/15	13/15	2/15	13/15	3/3	15/15
Malas prácticas	10	2/10	9/10	8/10	9/10	1/1	10/10
Total	35	15/35	25/35	13/35	25/35	14/14	35/35

Nota. Elaboración propia. Cada celda indica casos detectados sobre casos evaluados. Las pruebas se escribieron solo para los 14 casos cuyo comportamiento esperado podía expresarse como una aseveración; por ello su columna se expresa sobre los casos con prueba y no sobre el total de la categoría.

Figura 1. Porcentaje de casos detectados por los analizadores estáticos según la categoría de defecto



En la categoría de malas prácticas el panorama se invierte. Ruff detectó 9 de los 10 casos y Pylint 8, mientras que Bandit solo señaló dos: la excepción capturada y silenciada, y el uso de assert para validar entradas, ambos con implicaciones de seguridad. El único caso que escapó a los tres analizadores fue la notificación de errores mediante print, un patrón en el que la función devuelve None en lugar de lanzar una excepción y que solo pudo evidenciarse con una prueba que exigía el comportamiento correcto. En conjunto, al menos un analizador estático detectó 25 de los 35 casos (71,4 %), y todos los casos fueron detectados por alguna de las cuatro técnicas.

La Tabla 3 detalla el resultado de cada técnica en los 35 casos. En las columnas de los analizadores se indica el código de la regla que señaló la línea defectuosa; en la columna de pruebas, «Falla» indica que la prueba falló con la versión defectuosa, lo que equivale a una detección.

Tabla 3. Detección de cada caso del corpus por técnica de verificación

Caso	Defecto	Patrón o debilidad	Bandit	Ruff	Pylint	Pruebas
L01	Promedio de lista vacía	Caso límite omitido	—	—	—	Falla
L02	Suma 1..n con rango incompleto	Error por uno (off-by-one)	—	—	—	Falla
L03	Argumento por defecto mutable	Estado compartido entre llamadas	—	B006	W0102	Falla
L04	Porcentaje con división entera	Operador equivocado	—	—	—	Falla
L05	Año bisiesto sin regla de 400	Mala interpretación del requisito	—	—	—	Falla
L06	Igualdad exacta entre flotantes	Comparación numérica ingenua	—	—	—	Falla
L07	Eliminar elementos mientras se itera	Modificación de colección durante el recorrido	—	—	—	Falla
L08	Clausuras con enlace tardío	Captura de variable de bucle	—	B023	W0640	Falla
L09	Comparación de identidad con literal	Uso de «is» en lugar de «==»	—	F632	R0123	Falla
L10	Método inexistente (API alucinada)	Objeto o atributo alucinado	—	—	—	Falla
V01	Consulta SQL con f-string	CWE-89 Inyección SQL	B608	S608	—	s. p.
V02	subprocess con shell=True	CWE-78 Inyección de comandos del SO	B602	S602	—	s. p.
V03	os.system con entrada del usuario	CWE-78 Inyección de comandos del SO	B605	S605	—	s. p.
V04	eval sobre la entrada	CWE-95 Inyección de código	B307	S307	W0123	Falla
V05	pickle de datos no confiables	CWE-502 Deserialización insegura	B301	S301	—	s. p.
V06	yaml.load sin cargador seguro	CWE-502 Deserialización insegura	B506	S506	—	s. p.
V07	Credencial embebida en el código	CWE-798 Credenciales codificadas	B105	S105	—	s. p.
V08	MD5 para contraseñas	CWE-327/916 Hash débil	B324	S324	—	s. p.
V09	random para tokens	CWE-330 Valores insuficientemente aleatorios	B311	S311	—	s. p.
V10	Ruta de archivo sin validar	CWE-22 Recorrido de directorios	—	—	—	Falla
V11	Verificación TLS desactivada	CWE-295 Validación de certificado inadecuada	B501	S501	—	s. p.
V12	Petición HTTP sin tiempo de espera	CWE-400 Consumo de recursos no controlado	B113	S113	W3101	s. p.
V13	Flask en modo depuración	CWE-489/CWE-94 Depurador expuesto	B104, B201	S104, S201	—	s. p.
V14	Archivo temporal con mktemp	CWE-377 Archivo temporal inseguro	B306	S306	—	s. p.
V15	Operación sin control de autorización	CWE-862 Autorización ausente	—	—	—	Falla

Caso	Defecto	Patrón o debilidad	Bandit	Ruff	Pylint	Pruebas
P01	except desnudo que silencia errores	Manejo de excepciones deficiente	B110	E722, S110	W0702	s. p.
P02	Importación con comodín	Contaminación del espacio de nombres	—	F403	W0401, W0614, W0622	s. p.
P03	Importaciones y variables sin uso	Código muerto	—	F401, F841	W0611, W0612	s. p.
P04	Archivo abierto sin gestor de contexto	Fuga de recursos	—	SIM115	R1732	s. p.
P05	Sombra de funciones integradas	Nombres que ocultan builtins	—	A002	W0622	s. p.
P06	assert para validar entradas	Validación que desaparece con python -O	B101	S101	—	s. p.
P07	Estado global mutable	Acoplamiento por variables globales	—	PLW0603	W0603	s. p.
P08	Función con anidamiento excesivo	Complejidad ciclomática elevada	—	PLR0911, PLR0912	R0911, R0912	s. p.
P09	Errores reportados con print	Ocultamiento de fallos al llamador	—	—	—	Falla
P10	Comparación con None mediante ==	Estilo no idiomático (PEP 8)	—	E711, SIM201	C0121	s. p.

Nota. Elaboración propia con Bandit 1.9.4, Ruff 0.16.8, Pylint 4.1.2 y pytest 9.1.1 sobre Python 3.13.16. Un guion largo (—) indica que la técnica no señaló la línea defectuosa; «s. p.» indica que el caso no tenía una prueba de especificación.

Casos que escapan al análisis estático

El examen de los diez casos que ningún analizador detectó muestra un rasgo común: en todos, el defecto no está en una construcción sintáctica sospechosa, sino en la distancia entre lo que el código hace y lo que debería hacer. El Listado 1 reproduce el caso de eliminación de elementos durante el recorrido de una lista. El código es breve, legible y no contiene ninguna llamada peligrosa; sin embargo, al eliminar un elemento la lista se desplaza y el bucle salta el siguiente, de modo que con la entrada [2, 4, 6, 7] la función devuelve [4, 7] en lugar de [7].

Listado 1. Error lógico no detectado por los analizadores estáticos (caso L07)

```
def quitar_pares(numeros):
    for n in numeros:
        if n % 2 == 0:
            numeros.remove(n)    # modifica la lista mientras se recorre
    return numeros

quitar_pares([2, 4, 6, 7])    # devuelve [4, 7]; se esperaba [7]
```

Más preocupante resulta que dos vulnerabilidades escaparan a todas las herramientas estáticas. La primera es el recorrido de directorios (CWE-22), reproducido en el Listado 2: la función une un nombre de archivo recibido del usuario con un directorio base, pero no verifica que la ruta resultante permanezca dentro de él. Un nombre como ../../etc/hostname permite leer archivos del sistema. Ninguna de las tres herramientas emitió aviso, porque os.path.join y open

son operaciones legítimas y el defecto reside en la validación que falta, no en una instrucción presente. La segunda es la ausencia de control de autorización (CWE-862): la función que modifica una calificación no comprueba quién la invoca. Este tipo de defecto depende de las reglas del dominio y, por definición, no puede inferirse de la sintaxis.

Listado 2. Vulnerabilidad de recorrido de directorios no detectada (caso V10) y su corrección

```
# Versión defectuosa
def leer_archivo(nombre):
    ruta = os.path.join(BASE, nombre)
    with open(ruta, encoding="utf-8") as f:
        return f.read()

# Versión corregida
def leer_archivo(nombre):
    ruta = (BASE / nombre).resolve()
    if not ruta.is_relative_to(BASE):
        raise PermissionError("Ruta fuera del directorio permitido")
    return ruta.read_text(encoding="utf-8")
```

En contraste, las vulnerabilidades que sí fueron detectadas comparten una firma reconocible: una función o un argumento concretos, como `eval`, `pickle.loads`, `shell=True`, `verify=False` o `hashlib.md5`. El Listado 3 muestra el caso de inyección SQL, que Bandit y Ruff señalaron por la construcción de la consulta mediante una cadena formateada. La corrección, el uso de una consulta parametrizada, eliminó el aviso.

Listado 3. Inyección SQL detectada por Bandit (B608) y Ruff (S608), caso V01

```
def buscar_usuario(conexion, usuario):
    consulta = f"SELECT * FROM usuarios WHERE nombre = '{usuario}'"
    return conexion.execute(consulta).fetchall()

# Corrección: consulta parametrizada
# conexion.execute("SELECT * FROM usuarios WHERE nombre = ?", (usuario,))
```

Avisos persistentes en las versiones corregidas

Las 14 pruebas de especificación pasaron con todas las versiones corregidas, lo que confirma que las correcciones restablecen el comportamiento esperado. En cuanto a los analizadores, 32 de las 35 versiones corregidas quedaron sin avisos. Las tres restantes conservaron advertencias de baja severidad que no corresponden a defectos: la versión corregida de la ejecución de comandos, que usa `subprocess` con una lista de argumentos y sin intérprete de órdenes, siguió recibiendo avisos informativos de Bandit (B404, B603 y B607) y de Ruff (S603 y S607) que invitan a revisar cualquier uso de `subprocess`; la clase que sustituyó al estado global recibió de Pylint el aviso de tener pocos métodos públicos, y la función de clasificación reestructurada recibió de Ruff el aviso de usar valores numéricos literales. Estos resultados indican que los avisos de los analizadores deben interpretarse y no aplicarse de manera mecánica.

Discusión

Los resultados permiten responder a las preguntas de investigación desde una perspectiva práctica. La literatura documenta defectos de naturaleza muy distinta en el código Python generado por inteligencia artificial, y el análisis del corpus muestra que esa diversidad tiene una consecuencia directa: cada familia de defectos es visible para técnicas diferentes. Las herramientas de seguridad reconocen bien los patrones con firma sintáctica, y los analizadores de estilo y de defectos probables reconocen bien las malas prácticas, pero ninguna herramienta estática puede saber qué debía hacer el programa. Los errores lógicos, que en la evidencia de Liu et al. (2024) y de Tambon et al. (2025) constituyen la falla más frecuente del código generado, quedaron en su mayoría fuera del alcance del análisis estático.

Este hallazgo coincide con la observación de Liu et al. (2023), quienes al ampliar de forma sistemática las pruebas del conjunto HumanEval comprobaron que la tasa de aciertos de 26 modelos se reducía hasta en un 19,3 % a 28,9 %, es decir, que una parte del código considerado correcto solo lo era frente a pruebas insuficientes. En el presente estudio las pruebas detectaron todos los casos para los que se escribieron, pero esa cifra debe leerse con cautela: fueron redactadas conociendo el defecto, y en la práctica su eficacia depende de que quien las escribe anticipe los casos límite. Si las pruebas son generadas por el mismo asistente que produjo el código, existe el riesgo de que compartan la misma interpretación errónea del enunciado, que es precisamente el primer patrón de la taxonomía de Tambon et al. (2025).

En relación con la seguridad, el alto nivel de detección de Bandit es coherente con la naturaleza de las debilidades más frecuentes informadas por Pearce et al. (2022) y Fu et al. (2025), muchas de las cuales corresponden a inyecciones y a usos inseguros de bibliotecas. Sin embargo, el recorrido de directorios y la autorización ausente, ambos presentes en la lista de debilidades más peligrosas de 2024 (MITRE, 2024), no fueron detectados por ninguna herramienta estática. Esta limitación es relevante porque son defectos de omisión: lo que falta es una comprobación, y el análisis basado en patrones busca lo que está escrito, no lo que debió escribirse. Fu et al. (2025) informaron además que, al recibir los avisos de un analizador estático, Copilot Chat logró corregir hasta el 55,5 % de los problemas de seguridad, lo que sugiere que el análisis estático es útil también como retroalimentación para el propio asistente, siempre que exista quien valide la corrección resultante.

Un resultado práctico del estudio es que Ruff, configurado con un conjunto amplio de reglas, cubrió todos los casos detectados por Bandit y por Pylint, debido a que incorpora reimplementaciones de las reglas de ambas herramientas. Ello simplifica su adopción en cursos y equipos pequeños, aunque no elimina la necesidad de conocer qué regla se activa y por qué. La persistencia de avisos de baja severidad en código correcto refuerza esta idea: un estudiante que aprende a silenciar avisos sin comprenderlos reproduce, con otra herramienta, la misma confianza acrítica que Perry et al. (2023) observaron frente a los asistentes de inteligencia artificial.

Las implicaciones educativas son directas. Si la escritura de código básico puede delegarse en un asistente, como advirtieron Becker et al. (2023) y Denny et al. (2024), la competencia que gana valor es la verificación. Los resultados respaldan un protocolo en cuatro pasos que puede incorporarse en las asignaturas de fundamentos de programación y de lenguajes de programación: especificar el comportamiento esperado antes de solicitar el código, incluidos los casos límite; convertir esa especificación en pruebas; ejecutar un analizador estático con reglas de seguridad y de calidad e interpretar cada aviso; y revisar manualmente las reglas del dominio que ninguna herramienta conoce, como quién está autorizado a realizar una operación o qué rutas y datos de entrada son admisibles. Este protocolo traslada la atención del producto, el código, al proceso de verificación, y se alinea con la recomendación de Negri-Ribalta et al. (2024) de establecer procesos de revisión específicos para el desarrollo asistido por inteligencia artificial. En el plano institucional, Pazmiño Pérez et al. (2026) muestran que las políticas sobre inteligencia artificial generativa en la universidad solo se traducen en comportamiento ético cuando se acompañan de formación, participación y mecanismos de revisión; el protocolo propuesto puede entenderse como una concreción de esos mecanismos en el aula de programación, donde la verificación del código es la forma práctica de uso responsable.

El estudio presenta limitaciones que deben señalarse. La principal es que el corpus fue elaborado de forma dirigida para reproducir patrones documentados y no recogido de las respuestas espontáneas de un asistente; por ello, sus resultados describen la capacidad de detección de las técnicas frente a cada patrón, pero no permiten estimar la prevalencia de los defectos ni comparar modelos. En segundo lugar, cada caso aísla un único defecto en una función breve, mientras que el código real puede combinar varios defectos y distribuirlos entre archivos, lo que afecta de manera distinta a cada herramienta. En tercer lugar, las pruebas se escribieron con conocimiento del defecto, lo que sobreestima su eficacia respecto de una situación real. Finalmente, los resultados corresponden a versiones concretas de las herramientas, cuyas reglas evolucionan con el tiempo. Estas limitaciones orientan el trabajo futuro hacia la aplicación del mismo procedimiento sobre código efectivamente generado por varios asistentes a partir de enunciados de cursos universitarios.

Conclusiones

Este estudio organizó en una taxonomía los defectos documentados en código Python generado por inteligencia artificial y evaluó la capacidad de cuatro técnicas de verificación para detectarlos. En respuesta a la primera pregunta, la literatura revisada describe tres categorías principales, errores lógicos, vulnerabilidades y malas prácticas, a las que se suma un riesgo emergente en la cadena de suministro por la sugerencia de paquetes inexistentes. En todas ellas el código defectuoso suele ser sintácticamente válido y de apariencia correcta, lo que limita la eficacia de la inspección visual.

En respuesta a la segunda pregunta, las herramientas de análisis estático mostraron un desempeño muy desigual según la categoría. Bandit detectó 13 de las 15 vulnerabilidades y ninguno de los errores lógicos; Ruff y Pylint detectaron la mayoría de las malas prácticas y solo 3 de los 10 errores lógicos. Dos vulnerabilidades de omisión, el recorrido de directorios y la falta de control de autorización, escaparon a todas las herramientas estáticas.

En respuesta a la tercera pregunta, solo la combinación de técnicas permitió detectar todos los casos del corpus. La verificación del código generado por inteligencia artificial debe integrar análisis estático de seguridad y de calidad, pruebas derivadas de una especificación elaborada por una persona y revisión humana de las reglas del dominio. Para la educación superior, ello implica enseñar de forma explícita a especificar, probar y auditar código, y evaluar estas competencias con el mismo peso que la escritura de programas. Como líneas de trabajo futuro se proponen la aplicación del procedimiento a código generado por distintos asistentes en tareas reales de cursos universitarios y la evaluación del protocolo de verificación en el aula.

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), Artículo 129. <https://doi.org/10.1007/s10664-023-10380-1>
- Astral Software. (2026). *Ruff* (Versión 0.16.8) [Software]. <https://docs.astral.sh/ruff/>
- Becker, B. A., Denny, P., Finnie-Ansley, J., Luxton-Reilly, A., Prather, J., & Santos, E. A. (2023). Programming is hard – or at least it used to be: Educational opportunities and challenges of AI code generation. En *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1* (pp. 500–506). ACM. <https://doi.org/10.1145/3545945.3569759>
- Denny, P., Prather, J., Becker, B. A., Finnie-Ansley, J., Hellas, A., Leinonen, J., Luxton-Reilly, A., Reeves, B. N., Santos, E. A., & Sarsa, S. (2024). Computing education in the era of generative AI. *Communications of the ACM*, 67(2), 56–67. <https://doi.org/10.1145/3624720>
- Finnie-Ansley, J., Denny, P., Becker, B. A., Luxton-Reilly, A., & Prather, J. (2022). The robots are coming: Exploring the implications of OpenAI Codex on introductory programming. En *Proceedings of the 24th Australasian Computing Education Conference* (pp. 10–19). ACM. <https://doi.org/10.1145/3511861.3511863>
- 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), 1–34. <https://doi.org/10.1145/3716848>
- Jesse, K., Ahmed, T., Devanbu, P. T., & Morgan, E. (2023). Large language models and simple, stupid bugs. En *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)* (pp. 563–575). IEEE. <https://doi.org/10.1109/MSR59073.2023.00082>
- Khoury, R., Avila, A. R., Brunelle, J., & Camara, B. M. (2023). How secure is code generated by ChatGPT? En *2023 IEEE International Conference on Systems, Man, and Cybernetics (SMC)* (pp. 2445–2451). IEEE. <https://doi.org/10.1109/SMC53992.2023.10394237>
- Liu, J., Xia, C. S., Wang, Y., & Zhang, L. (2023). *Is your code generated by ChatGPT really correct? Rigorous evaluation of large language models for code generation* [Preprint]. arXiv. <https://doi.org/10.48550/arXiv.2305.01210>

- Liu, Y., Le-Cong, T., Widyasari, R., Tantithamthavorn, C., Li, L., Le, X.-B. D., & Lo, D. (2024). Refining ChatGPT-generated code: Characterizing and mitigating code quality issues. *ACM Transactions on Software Engineering and Methodology*, 33(5), Artículo 116. <https://doi.org/10.1145/3643674>
- MITRE. (2024). 2024 CWE Top 25 most dangerous software weaknesses. https://cwe.mitre.org/top25/archive/2024/2024_top25_list.html
- Moradi Dakhel, A., Majdinasab, V., Nikanjam, A., Khomh, F., Desmarais, M. C., & Jiang, Z. M. (2023). GitHub Copilot AI pair programmer: Asset or liability? *Journal of Systems and Software*, 203, Artículo 111734. <https://doi.org/10.1016/j.jss.2023.111734>
- 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, Artículo 1386720. <https://doi.org/10.3389/fdata.2024.1386720>
- Pazmiño Pérez, R. J., Lozano Alvarado, C. I., Solis Gomez, B., & León Pérez, C. (2026). Gobernanza institucional de la inteligencia artificial generativa y comportamiento ético en educación superior: efectos mediadores de la alfabetización en IA y la legitimidad percibida de las políticas. *Research Club Scientia*, 4(3), 1087–1096. <https://doi.org/10.65415/g4d29j67>
- 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. En *2022 IEEE Symposium on Security and Privacy (SP)* (pp. 754–768). IEEE. <https://doi.org/10.1109/SP46214.2022.9833571>
- Perry, N., Srivastava, M., Kumar, D., & Boneh, D. (2023). Do users write more insecure code with AI assistants? En *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security* (pp. 2785–2799). ACM. <https://doi.org/10.1145/3576915.3623157>
- PyCQA. (2026a). *Bandit* (Versión 1.9.4) [Software]. <https://bandit.readthedocs.io/>
- PyCQA. (2026b). *Pylint* (Versión 4.1.2) [Software]. <https://pylint.readthedocs.io/>
- Snyder, H. (2019). Literature review as a research methodology: An overview and guidelines. *Journal of Business Research*, 104, 333–339. <https://doi.org/10.1016/j.jbusres.2019.07.039>
- 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. En *Proceedings of the 34th USENIX Security Symposium* (pp. 3687–3706). USENIX Association. <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(3), Artículo 65. <https://doi.org/10.1007/s10664-025-10614-4>
- van Rossum, G., Warsaw, B., & Coghlan, A. (2001). *PEP 8 – Style guide for Python code*. Python Software Foundation. <https://peps.python.org/pep-0008/>