UNIVERSIDAD NACIONAL DE INGENIERÍA



SI806U · Desarrollo Adaptativo e Integración de Software

CodeAudit Skills: una metodología para auditar código generado con inteligencia artificial mediante reglas, skills y supervisión humana

Informe de laboratorio

Autor: Diogo Fabricio Abregu Gonzales
Docente: Carlos Alberto Janhachi Toribio

Lima, Perú
Noviembre de 2026

Resumen

El uso de asistentes de inteligencia artificial (IA) para escribir código acelera el desarrollo, pero suele ir acompañado de una revisión sin método: el desarrollador prueba el caso feliz y consulta a la misma IA, que responde con seguridad y sin evidencia. Este informe propone *CodeAudit Skills*, una metodología de aseguramiento de la calidad que responde a tres preguntas: si el código fue generado por IA, si contiene errores o vulnerabilidades y si presenta malas prácticas. La metodología combina un analizador estático determinista de 17 reglas, cuatro *skills* que guían a un agente de IA mediante siete prompts estructurados, una etapa de verificación adversarial y la decisión final de una persona. Sobre un archivo de prueba de 35 líneas, las reglas identificaron 8 hallazgos (5 graves) y un indicio de procedencia por IA; un agente que siguió las skills confirmó esos hallazgos y reportó uno adicional que ninguna regla cubre. Tras aplicar las correcciones mínimas, el archivo superó el criterio de calidad. Los resultados muestran que la combinación de reglas e IA supervisada cubre más que cada enfoque por separado.

Palabras clave: aseguramiento de la calidad, análisis estático, código generado por IA, revisión de código, skills, ingeniería de prompts.

Abstract

AI coding assistants speed up development, but their output is often accepted without a review method. This report presents *CodeAudit Skills*, a quality assurance methodology that answers three questions: whether code was AI-generated, whether it contains defects or vulnerabilities, and whether it shows poor practices. It combines a deterministic static analyzer with 17 rules, four skills that guide an AI agent through seven structured prompts, an adversarial verification step and a final human decision. On a 35-line test file, the rules found eight issues and an AI-provenance indication; an agent following the skills confirmed them and reported one additional defect that no rule covers. After minimal fixes, the file passed the quality gate.

Keywords: software quality assurance, static analysis, AI-generated code, code review, agent skills.

1. Introducción

Los asistentes de IA generan en segundos fragmentos de código que antes tomaban horas. Esta rapidez desplaza el problema: la dificultad ya no está en escribir el código, sino en comprobar que se puede confiar en él. En la práctica es frecuente que el desarrollador ejecute una sola prueba con datos válidos y luego pregunte a la misma IA si el código es correcto. La respuesta suele ser afirmativa y no cita líneas concretas, por lo que no puede verificarse.

El laboratorio del curso plantea definir y fundamentar un mecanismo de aseguramiento de la calidad que resuelva tres pilares: (1) identificar si un código fue generado por IA, o estructurar un proceso que evite los patrones típicos del código generado; (2) verificar que no existan fallas, errores de ejecución ni vulnerabilidades; y (3) evaluar y corregir malas prácticas, como el código basura, las variables no inicializadas y las funciones innecesarias. Se permite el uso de herramientas de análisis y de prompts estructurados, siempre que estos se incluyan y se fundamente su funcionamiento.

El objetivo de este trabajo es proponer, implementar y evaluar una metodología que responda a los tres pilares con evidencia verificable: cada hallazgo debe indicar el archivo, la línea, el fragmento que lo origina y una corrección comprobable.

2. Marco conceptual

2.1 Análisis estático

El análisis estático examina el código sin ejecutarlo. Herramientas como SonarQube, Ruff o Bandit transforman el programa en un árbol de sintaxis abstracta (AST) y aplican reglas escritas previamente por especialistas [7], [8], [9]. Su ventaja es la reproducibilidad: una regla produce siempre el mismo resultado y nunca reporta un hallazgo inexistente. Su límite es la cobertura: solo detecta los patrones para los que fue programada.

2.2 Modelos de lenguaje como revisores

Un modelo de lenguaje puede razonar sobre la intención del código y detectar defectos que ninguna regla describe. Sin embargo, también puede producir hallazgos inexistentes, citar líneas incorrectas o variar entre ejecuciones. Por ello su salida debe anclarse en evidencia y someterse a verificación [3], [4].

2.3 Skills de agentes

Una *skill* es un archivo `SKILL.md` que contiene un nombre, una descripción de cuándo debe activarse, un procedimiento, reglas y un formato de salida. Un agente (por ejemplo, Codex o Claude Code) la carga cuando la tarea coincide con su descripción, lo que permite repetir el mismo procedimiento en cada auditoría [2].

2.4 Ingeniería agéntica supervisada

Karpathy describe a los agentes de IA como herramientas potentes pero falibles, cuya salida requiere supervisión y estándares por parte del ingeniero responsable [1]. Este principio orienta el diseño: la IA nunca decide sola.

2.5 Límites de la detección de código generado por IA

Clasificar un código como generado por IA a partir de su estilo es poco fiable: el estilo cambia entre modelos y se pierde cuando una persona edita el resultado. Por esta razón, la metodología prioriza la evidencia declarada y solo admite indicios técnicos localizables, sin emitir porcentajes de probabilidad.

3. Metodología propuesta

3.1 Principios

Cada regla de la auditoría se deriva de un principio de buen desarrollo. La Tabla 1 resume la relación.

Tabla 1. Principios de desarrollo vigilados por la auditoría.

Principio	Qué exige	Reglas o prompts que lo vigilan
Evidencia, no opinión	Todo hallazgo cita archivo, línea y fragmento.	P0
No confiar en la entrada	La entrada del usuario no se ejecuta ni se concatena en consultas.	SEC-SQL-INJECTION, SEC-DYNAMIC-EXEC, SEC-SHELL-INJECTION
Secretos fuera del código	Credenciales en el entorno o en un gestor de secretos.	SEC-HARDCODED-SECRET
Estado definido	Toda variable tiene valor en todos los caminos.	Q-UNINITIALIZED, BUG-UNDEFINED-NAME
Fallar de forma visible	No silenciar excepciones.	Q-SILENT-EXCEPT
Código mínimo	Sin imports, variables ni funciones sin uso.	Q-UNUSED-*, Q-UNREACHABLE
Dependencias reales	No importar módulos inexistentes.	BUG-MISSING-MODULE
Simplicidad	Baja complejidad y sin estado mutable compartido.	Q-COMPLEXITY, Q-MUTABLE-DEFAULT
Corrección probada	Cada arreglo con su prueba de regresión.	P2, P3
Procedencia declarada	El uso de IA se declara.	P1, P6
Decisión humana	Una persona aprueba el resultado.	P5

3.2 Arquitectura

La Figura 1 muestra la arquitectura. El agente carga la skill orquestadora *code-audit*, que ejecuta el analizador determinista y entrega su salida como evidencia a las tres skills de los pilares. Los hallazgos pasan por una etapa de refutación (P4) y se consolidan en un reporte (P5) que aplica el criterio de calidad. Si no se aprueba, el código se corrige y vuelve a medirse.

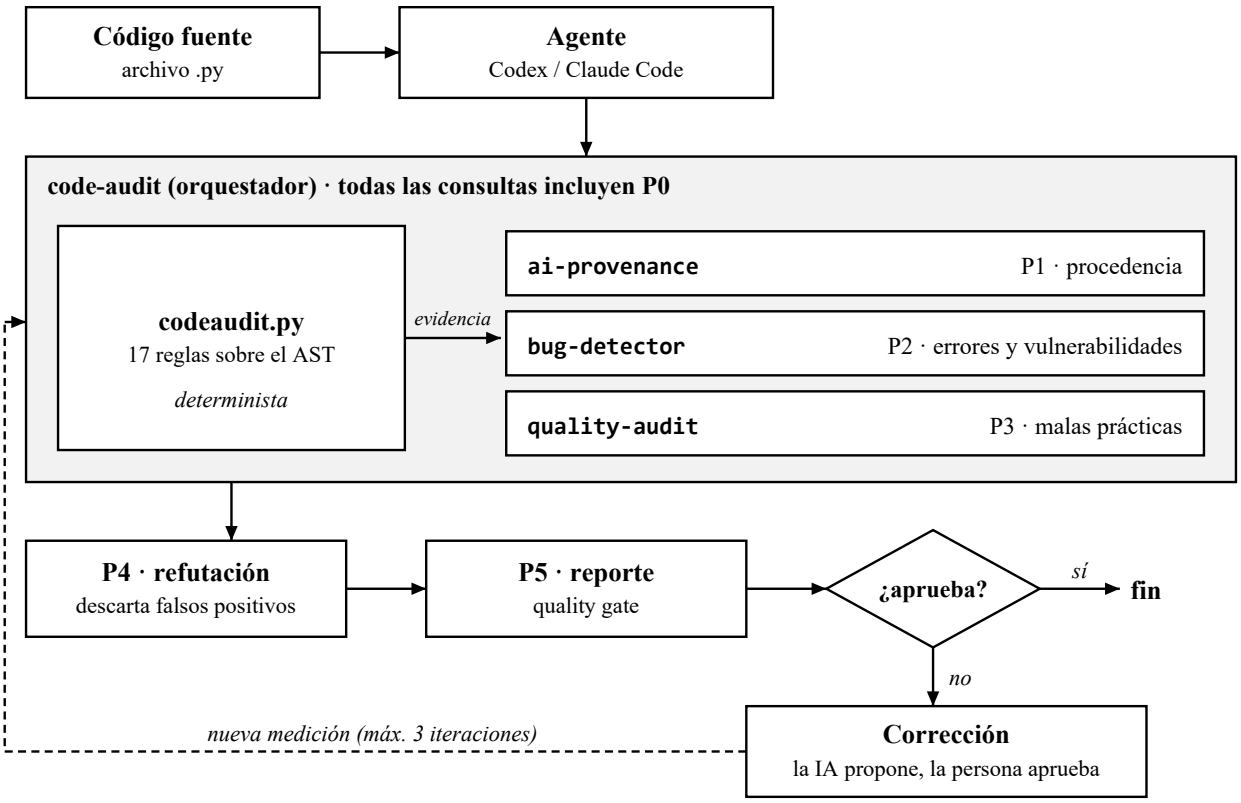


Figura 1. Arquitectura de CodeAudit Skills.

3.3 Skills

Tabla 2. Skills de la metodología.

Skill	Responsabilidad	Prompts	Reglas
`code-audit`	Orquestación, verificación y reporte.	P0, P4, P5	Usa las 17
`ai-provenance`	Pilar 1: procedencia e indicios de IA.	P1, P6	4 señales
`bug-detector`	Pilar 2: errores y vulnerabilidades.	P2	9
`quality-audit`	Pilar 3: malas prácticas.	P3	8

3.4 Prompts

Se diseñaron siete prompts. P0 fija reglas comunes que se anteponen a P1–P4: citar archivo, línea y fragmento; distinguir entre hallazgos ciertos, condicionales y por revisar; responder "indeterminado" cuando falta evidencia; no ejecutar el código; y devolver JSON. La Tabla 3 resume la finalidad de cada uno; el texto completo se incluye en el Anexo A.

Tabla 3. Prompts de la metodología.

Prompt	Finalidad	Entrada	Salida
P0	Reglas comunes.	Código, evidencia y contexto.	Hallazgos en JSON con línea y fragmento.
P1	Detectar procedencia por IA.	Código, historial y manifiesto.	Declarado, indicio o indeterminado.
P2	Encontrar errores y vulnerabilidades.	Código y evidencia.	Falla, entrada que la dispara y prueba.
P3	Detectar malas prácticas.	Código.	Práctica y corrección mínima.
P4	Refutar hallazgos.	Hallazgos de P1–P3 y código.	Confirmado, descartado o degradado.
P5	Consolidar el reporte.	Hallazgos confirmados.	Resumen, criterio de calidad y plan.
P6	Generar código sin patrones típicos de IA.	Requisito y ejemplo del repositorio.	Código, pruebas y uso de IA declarado.

3.5 Procedimiento

El procedimiento consta de ocho etapas: (1) el usuario entrega el código; (2) *code-audit* ejecuta el analizador; (3) *ai-provenance* evalúa la procedencia con P0 y P1; (4) *bug-detector* busca fallas con P0 y P2; (5) *quality-audit* busca malas prácticas con P0 y P3; (6) P4 intenta refutar cada hallazgo; (7) P5 genera el reporte y aplica el criterio de calidad, que no aprueba si existe algún hallazgo de severidad P0 o P1; y (8) si no aprueba, la IA propone correcciones mínimas que la persona revisa antes de volver a la etapa 2. Tras tres iteraciones sin aprobación, el caso pasa a revisión humana.

4. Implementación

El analizador `codeaudit.py` está escrito en Python con la biblioteca estándar. Convierte cada archivo en un AST mediante `ast.parse` y aplica reglas que recorren el árbol buscando patrones; el código auditado nunca se importa ni se ejecuta. Por ejemplo, la regla de inyección SQL marca toda llamada cuyo nombre termina en `execute` y cuyo primer argumento es una cadena construida con variables. Las Tablas 4 y 5 listan las reglas de los pilares 2 y 3; el pilar 1 se apoya en un manifiesto de procedencia declarada y en cuatro señales automáticas.

Tabla 4. Reglas del pilar 2.

Regla	Detecta	Sev.
`PY-SYNTAX`	Archivo que no se puede analizar.	P0
`BUG-UNDEFINED-NAME`	Nombre no definido (NameError).	P1
`BUG-MISSING-MODULE`	Módulo inexistente (ModuleNotFoundError).	P1
`BUG-ZERO`	División por cero literal.	P1
`BUG-LEN-EMPTY`	División por len() sin comprobar colección vacía.	P2
`SEC-SQL-INJECTION`	Consulta SQL con texto dinámico (CWE-89).	P1
`SEC-SHELL-INJECTION`	Comando de sistema con shell (CWE-78).	P1
`SEC-DYNAMIC-EXEC`	Uso de eval o exec (CWE-95).	P1
`SEC-HARDCODED-SECRET`	Credencial escrita en el código (CWE-798).	P1

Tabla 5. Reglas del pilar 3.

Regla	Detecta	Sev.
`Q-UNINITIALIZED`	Variable sin valor en algún camino.	P1
`Q-MUTABLE-DEFAULT`	Argumento mutable por defecto.	P2
`Q-SILENT-EXCEPT`	Excepción silenciada.	P2
`Q-UNREACHABLE`	Código inalcanzable.	P2
`Q-COMPLEXITY`	Complejidad ciclomática mayor que 10.	P2
`Q-UNUSED-IMPORT`	Import sin uso.	P3
`Q-UNUSED-VAR`	Variable asignada y no usada.	P3
`Q-UNUSED-FUNCTION`	Función nunca invocada.	P3

El paquete incluye además 13 pruebas unitarias del analizador, un script `armar_prompt.py` que compone el prompt completo de cada etapa para utilizarlo con un modelo de chat, y las cuatro skills en formato estándar.

5. Resultados

5.1 Conjunto de prueba

Se utilizaron cuatro archivos sintéticos. Sobre el conjunto, el analizador reportó 14 hallazgos (1 P0, 5 P1, 4 P2 y 4 P3) y el criterio de calidad no se aprobó. El archivo escrito y revisado por una persona no generó hallazgos, y las 13 pruebas unitarias se superaron.

5.2 Caso de estudio: demo_unknown.py

El archivo de 35 líneas que motiva este informe produjo 8 hallazgos, 5 de ellos graves, y una clasificación de *indicio de IA* basada en dos señales distintas: un comentario conversacional en la línea 1 y la importación de un módulo inexistente en la línea 3 (Tabla 6).

Tabla 6. Hallazgos del analizador sobre demo_unknown.py.

Línea	Regla	Sev.	Descripción
3	`BUG-MISSING-MODULE`	P1	Módulo no disponible: fastdb_helpers
5	`SEC-HARDCODED-SECRET`	P1	Secreto escrito en el código (DB_PASSWORD)
9	`SEC-DYNAMIC-EXEC`	P1	Uso de eval()
14	`SEC-SQL-INJECTION`	P1	Consulta SQL construida con texto dinámico
21	`Q-UNINITIALIZED`	P1	Variable posiblemente no inicializada: tasa
3	`Q-UNUSED-IMPORT`	P3	Import sin uso: fastdb_helpers
24	`Q-UNUSED-FUNCTION`	P3	Función nunca invocada en el módulo: formatear_legacy()
25	`Q-UNUSED-VAR`	P3	Variable asignada y nunca usada: resultado

5.3 Ejecución con un agente

El procedimiento completo se ejecutó con el agente Codex, al que se le entregaron las skills. El agente identificó y cargó *code-audit*, ejecutó el analizador, aplicó las tres skills de los pilares y la etapa de refutación. Confirmó los ocho hallazgos y reportó uno adicional: la función principal abre una base de datos en memoria y consulta la tabla `usuarios`, que nunca se crea, lo que produciría un error en tiempo de ejecución. Ninguna de las 17 reglas cubre este caso, porque detectarlo requiere comprender el comportamiento del programa.

5.4 Corrección

Tras aplicar las correcciones mínimas a los ocho hallazgos del analizador (consulta parametrizada, credencial leída del entorno, evaluación literal segura, inicialización de la variable y eliminación del código sin uso), el analizador reportó 0 hallazgos y el criterio de calidad se aprobó. El hallazgo adicional del agente requiere una corrección manual que crea la tabla antes de consultarla.

6. Discusión y limitaciones

Los resultados ilustran la complementariedad de los enfoques. Las reglas aportaron hallazgos seguros y repetibles; el agente guiado por las skills encontró un defecto fuera de su alcance. La etapa de refutación y la revisión humana reducen el riesgo de aceptar hallazgos inexistentes. El trabajo presenta limitaciones:

- El conjunto de prueba es pequeño y sintético; demuestra el funcionamiento del flujo, no mide precisión ni exhaustividad sobre código real.
- La procedencia por IA no puede demostrarse sin trazabilidad; la clasificación es orientativa y no debe usarse sola para sancionar.
- Las 17 reglas no sustituyen a herramientas como SonarQube o Checkmarx; las complementan.

- La ejecución con el agente es una única corrida; las respuestas de un modelo pueden variar entre ejecuciones.

7. Conclusiones

- La metodología responde a los tres pilares del laboratorio con un principio común: todo hallazgo debe ser localizable y verificable.
- Combinar un analizador determinista con un agente guiado por skills cubre más defectos que cada enfoque por separado, como mostró el hallazgo adicional del caso de estudio.
- Las skills convierten la revisión con IA en un procedimiento repetible, y la verificación adversarial y la decisión humana controlan sus errores.
- Como trabajo futuro se propone evaluar la metodología sobre repositorios reales, medir precisión y exhaustividad, e integrarla en la integración continua.

Referencias

[1] A. Karpathy, “Sequoia Ascent 2026,” 2026. [En línea]. Disponible: https://karpathy.bearblog.dev/sequoia-ascent-2026/

[2] Anthropic, “Agent Skills,” 2026. [En línea]. Disponible: https://github.com/anthropics/skills

[3] OpenAI, “Codex code review guidelines,” 2026. [En línea]. Disponible: https://github.com/openai/codex

[4] Anthropic, “Claude Code Security Review,” 2026. [En línea]. Disponible: https://github.com/anthropics/claude-code-security-review

[5] OWASP Foundation, “OWASP Top 10.” [En línea]. Disponible: https://owasp.org/Top10/

[6] MITRE, “CWE: Common Weakness Enumeration.” [En línea]. Disponible: https://cwe.mitre.org/

[7] SonarSource, “SonarQube documentation: Quality Gates.” [En línea]. Disponible: https://docs.sonarsource.com/sonarqube/

[8] Astral, “Ruff linter.” [En línea]. Disponible: https://docs.astral.sh/ruff/

[9] PyCQA, “Bandit.” [En línea]. Disponible: https://bandit.readthedocs.io/

[10] T. J. McCabe, “A complexity measure,” *IEEE Trans. Softw. Eng.*, vol. SE-2, no. 4, pp. 308–320, 1976.

Anexo A. Prompts

Texto completo de los prompts. Cuando un prompt comienza con `[P0]`, se antepone el texto de P0.

P0. Reglas comunes

```
Actúas como auditor senior de código Python. Tu trabajo es producir hallazgos VERIFICABLES, no opiniones.

<codigo> {pegar archivos con números de línea} </codigo>
<herramientas> {salida JSON de codeaudit.py, Ruff, Bandit o SonarQube; "ninguna" si no hay} </herramientas>
<contexto> {qué debe hacer el programa, versión de Python, entradas esperadas} </contexto>

Reglas obligatorias:
1. Cada hallazgo cita archivo y línea exactos y copia el fragmento como evidencia.
2. Distingue: "cierto" (falla siempre que se ejecuta la línea), "condicional" (falla con una entrada
   concreta que debes nombrar) y "a-revisar" (depende de contexto que no tienes).
3. Si no tienes evidencia suficiente, responde "indeterminado". No inventes líneas, APIs ni resultados.
4. No ejecutes ni supongas que ejecutaste el código.
5. Responde SOLO con el JSON del esquema indicado en cada paso.
```

P1. Detección de código generado por IA

```
[P0]
Tarea: evaluar la PROCEDENCIA del código. No evalúes calidad aquí.

Paso A — Evidencia declarada (peso alto): historial git, trailers "Co-Authored-By", archivo
  .codeaudit-provenance.json, encabezados que declaren una herramienta. Una declaración no autenticada
  es un indicio, no una prueba.
Paso B — Indicios técnicos (peso bajo, cada uno por separado NO basta):
  S1 residuos conversacionales ("Aquí tienes", "As an AI", bloques de markdown dentro del .py)
  S2 comentarios narrativos que repiten la línea siguiente ("# Paso 1: sumar")
  S3 imports o funciones de librerías que no existen (APIs alucinadas)
  S4 marcadores de plantilla sin completar (your_api_key, example.com, TODO: implement)
  S5 manejo genérico uniforme (except Exception en todas las funciones) y docstrings idénticas
  S6 incoherencia con el resto del repositorio (estilo, nombres, versión de Python distinta)
Paso C — Clasifica: declarado-ai | declarado-human | declarado-hybrid | indicio-de-ia (≥2 señales
  distintas) | indeterminado.
Prohibido: dar porcentajes de "probabilidad de IA" o recomendar sanciones.

Esquema: {"file", "classification", "evidence":[{"signal","line","fragment"}],
          "verified_authorship": false, "limits": "por qué esto no prueba autoría"}
```

P2. Errores y vulnerabilidades

```
[P0]
Tarea: encontrar FALLAS (no estilo). Recorre el código en este orden y documenta tu razonamiento
por función antes de emitir hallazgos:
1. Sintaxis e imports: ¿parsea? ¿existen todos los módulos y nombres usados (NameError)?
2. Ejecución: divisiones por cero o por len() vacío, índices fuera de rango, None no controlado,
   tipos incompatibles, variables usadas antes de asignarse, recursos sin cerrar.
3. Casos límite: lista vacía, cadena vacía, 0, negativos, None, archivo inexistente.
4. Seguridad (OWASP/CWE): inyección SQL (CWE-89), comandos de shell (CWE-78), eval/exec (CWE-95),
   secretos en el código (CWE-798), deserialización insegura (CWE-502), rutas sin validar (CWE-22).
Para cada hallazgo indica la ENTRADA CONCRETA que lo dispara y escribe una prueba unitaria mínima
que hoy falla y pasará tras la corrección.
Severidad: P0 no ejecuta · P1 falla o es explotable en uso normal · P2 falla en caso límite · P3 menor.

Esquema: {"id","category":"error|vulnerabilidad","severity","file","line","fragment",
          "trigger_input","impact","fix","regression_test","certainty":"cierto|condicional|a-revisar"}
```

P3. Malas prácticas

```
[P0]
Tarea: detectar malas prácticas que reducen el valor del código. No repitas fallas de P2.
Busca y cita:
- Código basura: imports, variables y parámetros sin uso; código inalcanzable; código comentado.
- Funciones innecesarias: nunca invocadas, duplicadas o envoltorios que no aportan.
- Variables no inicializadas en algún camino (asignadas solo dentro de un if/for).
- Excepciones silenciadas (except: pass) o demasiado genéricas.
- Argumentos mutables por defecto; estado global mutable.
- Complejidad: funciones > 40 líneas o complejidad ciclomática > 10; nombres poco descriptivos;
  números mágicos; duplicación.
Para cada hallazgo propone la corrección MÍNIMA (diff) que no cambie el comportamiento esperado.
Ignora preferencias puramente estéticas que ya corrige un formateador.

Esquema: {"id","category":"codigo-muerto|practica|mantenibilidad","severity","file","line",
          "fragment","why_it_matters","minimal_fix_diff"}
```

P4. Verificación adversarial

```
Recibes la lista de hallazgos de P1–P3 y el código original.
Para CADA hallazgo intenta refutarlo:
- ¿La línea citada existe y contiene el fragmento? Si no, descártalo.
- ¿Hay una guarda, validación o contexto que lo hace imposible? Si sí, descártalo o baja severidad.
- ¿La prueba de regresión realmente fallaría hoy?
Devuelve {"id","verdict":"confirmado|descartado|degradado","reason"} por hallazgo.
Mantén solo lo que sobreviva. Es preferible perder un hallazgo dudoso que reportar uno falso.
```

P5. Consolidación y reporte

```
Con los hallazgos confirmados de P4 genera el reporte final:
1. Resumen: archivos, hallazgos por pilar y por severidad, dictamen de procedencia por archivo.
2. Quality gate: FALLA si existe algún P0 o P1; si no, APRUEBA.
3. Tabla ordenada por severidad y luego por archivo:línea.
4. Plan de corrección en orden de riesgo/esfuerzo, con el test que demuestra cada arreglo.
5. Limitaciones: qué no se pudo verificar y por qué.
No agregues hallazgos nuevos en este paso.
```

P6. Generación preventiva

```
Genera {descripción de la función/módulo} en Python {versión} siguiendo estas restricciones:
- Respeta el estilo del repositorio que adjunto en <ejemplo> (nombres, docstrings, manejo de errores).
- Usa solo librerías estándar o las listadas en <requirements>; si necesitas otra, pregunta.
- Sin comentarios que narren el código; comenta solo el "por qué" no evidente.
- Sin marcadores (TODO, your_api_key, example.com) ni texto conversacional dentro del archivo.
- Valida entradas y casos límite; nada de except genéricos silenciosos ni defaults mutables.
- Código mínimo: no agregues funciones, parámetros o abstracciones que no se pidieron.
- Entrega también las pruebas unitarias (pytest) de los casos normales y límite.
- Antes de responder, revisa tu propio código con la lista de P2 y P3 y corrige lo que encuentres.
Declara el uso de IA en el commit (trailer "Co-Authored-By") y en .codeaudit-provenance.json.
```

Anexo B. Reproducción

Con Python 3.10 o superior, desde la carpeta del paquete:

```
python codeaudit.py examples/demo_unknown.py --out reporte.json
python -m unittest discover -s tests -v
python codeaudit.py corregido/demo_unknown.py
python armar_prompt.py P2 examples/demo_unknown.py
```

Para ejecutar el procedimiento con un agente, se copian las skills a `.codex/skills` o `.claude/skills` y se solicita la auditoría del archivo con la skill *code-audit*.