

Mitigação de Vulnerabilidades de Segurança em Código Gerado por Inteligência Artificial: um Guia Técnico e uma Arquitetura de Análise Automatizada

Robson D. Marchesan¹, Raul Ceretta Nunes²

¹Curso de Ciência da Computação – Universidade Federal de Santa Maria (UFSM)
Av. Roraima, 1000 – 97.105-900 – Santa Maria – RS – Brasil

²Departamento de Computação Aplicada – Universidade Federal de Santa Maria (UFSM)

robsondaniel2444@gmail.com, ceretta@inf.ufsm.br

Abstract. *Code produced by AI programming assistants frequently reproduces known weakness patterns. This paper presents two complementary artifacts: a technical mitigation guide that organizes recurring vulnerabilities in AI-generated code into seven categories (GIA-001 to GIA-007), mapped to CWE and OWASP and enriched with actionable recommendations; and a language-independent pipes-and-filters architecture combining static analysis, AST-based heuristics, classification and recommendation, evaluated through a Python prototype. On externally labeled corpora it ties with a consolidated SAST tool in detection, while 98% of its findings receive a guide category and 100% of the classified ones receive CWE/OWASP mapping and a structured mitigation recommendation.*

Resumo. *Código produzido por assistentes de programação baseados em inteligência artificial reproduz, com frequência, padrões conhecidos de falhas. Este artigo apresenta dois artefatos complementares: um guia técnico de mitigação que organiza em sete categorias (GIA-001 a GIA-007) as vulnerabilidades recorrentes em código gerado por IA, com mapeamento para CWE e OWASP e recomendações acionáveis; e uma arquitetura de análise em filtros e dutos, independente de linguagem, que integra análise estática, heurísticas sobre AST, classificação e recomendação, avaliada por um protótipo em Python. Em corpora rotulados por terceiros, ela empata com uma ferramenta SAST consolidada na detecção, enquanto 98% dos achados recebem categoria do guia e 100% dos classificados recebem mapeamento CWE/OWASP e recomendação estruturada.*

1. Introdução

Assistentes de programação baseados em modelos de linguagem foram amplamente incorporados ao desenvolvimento de software, inclusive de aplicações web, APIs e serviços de rede, nos quais uma falha de validação de entrada, autenticação ou controle de acesso fica diretamente exposta a ataques remotos. O ganho de produtividade, contudo, vem acompanhado de uma preocupação de segurança: esses modelos podem gerar implementações funcionalmente corretas que reproduzem falhas conhecidas e padrões inseguros. Pearce et al. [Pearce et al. 2022] observaram que uma fração relevante das completções do GitHub Copilot, em cenários derivados do CWE Top 25, é

vulnerável; Fu et al. [Fu et al. 2025] confirmaram o fenômeno em projetos reais e Tihanyi et al. [Tihanyi et al. 2025] o mediram em larga escala.

A literatura já caracteriza esse risco, mas de forma dispersa. Ferramentas de análise estática tradicionais enfrentam barreiras de adoção pela dificuldade de interpretação dos alertas [Wadhams et al. 2024]: o desenvolvedor recebe uma regra técnica, não uma orientação de mitigação. Na frente corretiva, trabalhos exploram reparo *zero-shot* [Pearce et al. 2023] e intervenções diretas nos modelos [He and Vechev 2023].

Este artigo assume o código já gerado e trata o problema no ponto de decisão do desenvolvedor, com dois artefatos complementares: (i) um **guia técnico de mitigação** que organiza em sete categorias (GIA-001 a GIA-007) as vulnerabilidades recorrentes relatadas na literatura, com mapeamento CWE/OWASP e recomendações em três frentes — o que verificar, corrigir e validar; e (ii) uma **arquitetura de análise automatizada**, independente de linguagem, que integra análise estática, heurísticas complementares, classificação e recomendação, usando o guia como base de conhecimento. A avaliação, sobre corpora externos de aplicações web geradas por assistente de programação, mostra que a contribuição verificável não está em detectar mais, e sim em transformar alertas técnicos em orientação categorizada, rastreável e acionável. O guia descritivo completo, a base JSON, o protótipo, a suíte de testes e os *scripts* de avaliação estão disponíveis publicamente¹.

2. Construção do Guia Técnico

O guia foi construído por um mapeamento bibliográfico estruturado, orientado por cinco questões: quais vulnerabilidades são recorrentes em código gerado por IA (Q1); como associá-las a CWE e OWASP (Q2); quais impactos e contextos são descritos (Q3); quais são detectáveis por análise estática, heurísticas ou revisão apoiada (Q4); e quais recomendações de mitigação derivam delas (Q5).

CWE e OWASP Top 10 forneceram o vocabulário inicial de classes. As buscas em ACM Digital Library, IEEE Xplore, SpringerLink, arXiv, USENIX e Google Scholar retornaram cerca de 1.100 registros (890 sem duplicatas). Os critérios de inclusão exigiram estudos empíricos relatando vulnerabilidades no código *gerado* por IA e que permitissem extrair mitigações; foram excluídos textos opinativos ou de uso de IA para detecção. A triagem selecionou sete estudos centrais [Pearce et al. 2022, Fu et al. 2025, Dora et al. 2025, Zhao et al. 2025, Spracklen et al. 2025, Sajadi et al. 2025, Tihanyi et al. 2025]. O *snowballing*, em uma iteração por direção, incorporou sete fontes complementares (quatro normativas — OWASP, NIST — e três corretivas [Pearce et al. 2023, He and Vechev 2023]). O processo atingiu saturação quando as fontes identificadas passaram a repetir as sete categorias já emergentes do agrupamento inicial ou a tratar de escopos alheios.

3. O Guia Técnico GIA-001..GIA-007

Cada entrada do guia reúne identificador, categoria, descrição, evidências na literatura, mapeamento CWE/OWASP, exemplos de ocorrência, impacto potencial, forma de detecção, recomendações de mitigação e prioridade. As recomendações são organizadas

¹https://github.com/robissu/Arquitetura_Seg_IA

em três frentes — *o que verificar, como corrigir e como validar* —, o que distingue o artefato de uma lista genérica de vulnerabilidades. A Tabela 1 sintetiza as categorias.

Tabela 1. Categorias do guia técnico e mapeamento.

ID	Categoria	Mapeamento principal
GIA-001	Validação inadequada de entrada e injeção	CWE-20, 78, 79, 89; A03:2021
GIA-002	Falhas de autenticação, autorização e controle de acesso	CWE-306, 862, 863; A01:2021, A07:2021
GIA-003	Uso inseguro de bibliotecas, APIs e dependências	CWE-829, 1104; A06:2021
GIA-004	Exposição de dados sensíveis e segredos no código	CWE-798, 200; A02:2021
GIA-005	Tratamento inadequado de erros e exceções	CWE-703, 209; A09:2021
GIA-006	Configurações inseguras e proteções ausentes	CWE-16; A05:2021
GIA-007	Implementação incompleta de controles de segurança	Variável, conforme o controle ausente

O guia tem representação descritiva e operacional (JSON); o identificador GIA é a chave de rastreabilidade entre achado, categoria e recomendação.

4. Arquitetura de Análise Automatizada

A arquitetura adota o estilo filtros e dutos (*pipes and filters*), no qual cada estágio encapsula uma responsabilidade com contratos explícitos de entrada e saída (Figura 1).

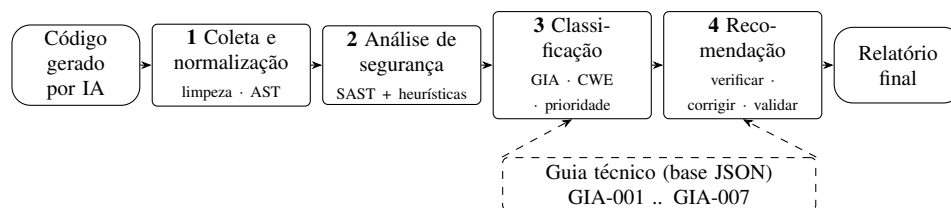


Figura 1. Arquitetura de análise automatizada de segurança proposta.

O **módulo de coleta e normalização** remove artefatos textuais típicos de saídas de LLMs (cerca de *markdown*, explicações em linguagem natural), uniformiza formatação e codificação e valida a sintaxe, bloqueando a análise quando o artefato não é interpretável. O **módulo de análise de segurança** opera em duas camadas paralelas sobre o código normalizado. A primeira é SAST, instanciada no protótipo pelo Bandit. A segunda reúne nove heurísticas (H001–H009) implementadas sobre a Árvore Sintática Abstrata (AST) — e não por expressões regulares, evitando falsos positivos em *strings* e comentários. Cada heurística deriva de um padrão inseguro mapeado no guia e possui limitações específicas, resumidas na Tabela 2. Todo achado, qualquer que seja sua origem, é convertido a um formato intermediário unificado (origem, regra, arquivo, linha, severidade, confiança, descrição, contexto).

Tabela 2. Heurísticas baseadas em AST e suas limitações.

ID	Indício detectado (disparo)	Limitação	GIA
H001/2	<code>except</code> cujo único comando é <code>pass</code>	<code>except:</code> continue não é detectado	005
H003	TODO/FIXME em comentário de função com nome sensível	Depende do nome da função	007
H004	Função de acesso (<code>is_admin</code>) com <code>return True</code> fixo	Só nomes da lista; <code>return</code> não literal impede	007
H005	Chamada que recebe <code>request</code> , <code>args</code> ou <code>form</code>	Sinaliza presença, não a falha; não checa validação	001
H006/7	Rota sensível sem decorador de <code>auth</code> . (H006); rota com <code><id></code> sem verificação de dono (H007)	<code>Auth.</code> por <i>middleware</i> gera falso positivo; H007 exige marcador explícito	002
H008	<code>import</code> ausente do <code>requirements.txt</code>	Nome do pacote $\neq$ módulo (PyYAML/yaml): falso positivo	003
H009	Segredo por formato de chave ou alta entropia	Valores simples e <i>f-strings</i> escapados	004

O **módulo de classificação** mapeia cada achado a uma categoria (via regra SAST ou heurística) e recupera do guia CWE, OWASP e prioridade; achados sem correspondência são marcados para `requires_manual_review`. O **módulo de recomendação** consulta a base de conhecimento e anexa ao achado o impacto potencial e as três frentes de mitigação, sem gerar texto novo e sem alterar o código. O resultado é uma arquitetura de apoio à decisão, não um mecanismo de correção automática.

5. Avaliação

A avaliação separou duas dimensões. A primeira, *detecção*, compara a linha de base A (Bandit isolado) com a configuração B (arquitetura completa), medindo se a vulnerabilidade foi encontrada. A segunda e principal contribuição, *enriquecimento semântico*, mede se o achado recebeu categoria, mapeamento e recomendação estruturada. Para evitar circularidade, usaram-se dois corpora externos rotulados por terceiros. O SecurityEval [Siddiq and Santos 2022] reúne funções Python isoladas rotuladas por CWE (121 amostras, 21 em escopo); o conjunto de Pearce et al. [Pearce et al. 2022] reúne 339 completções de aplicações web Python geradas pelo Copilot (112 vulneráveis e 227 seguras), rotuladas pela ferramenta CodeQL e, no caso do CWE-306, manualmente pelos autores originais. O protótipo foi previamente validado por uma suíte de 41 testes automatizados.

5.1. Detecção

No benchmark genérico o ganho é marginal: o *recall* passa de 33% (A) para 43% (B) nas 21 amostras em escopo, pois são funções isoladas, sem entrada web nem rotas. No corpus web, as métricas agregadas sugerem um ganho maior — o *recall* sobe de 69% para 83%, com queda de precisão de 64% para 52% —, mas a análise por categoria (Tabela 3) revela que esse ganho não corresponde a detecção real.

Em GIA-001 e GIA-004, A e B empatam: o SAST já detecta essas vulnerabilidades de fluxo de dados. Todo o ganho aparente em GIA-002 resume-se a um falso positivo de 100%: a heurística H006 marcou o *endpoint* de *login* em todas as completções, sem

Tabela 3. Recall e falsos positivos por categoria no corpus de Pearce et al.

Categoria	Rec. A	Rec. B	FP A	FP B
GIA-001 – injeção ($n = 53$)	91%	91%	7%	12%
GIA-002 – autenticação ausente ($n = 16$)	0%	100%	0%	100%
GIA-004 – segredos no código ($n = 43$)	67%	67%	44%	44%

capturar a falha real (falta de verificação de permissões em rota GET já autenticada). O *recall* é espúrio e foi descartado; na prática, a arquitetura empata em *recall* e perde em precisão.

Esse resultado indica que os corpora não exercitam plenamente as heurísticas. H005 e H006 sinalizam a presença do padrão estrutural e não a vulnerabilidade ativa; com precisão de 52% na configuração completa, seus alertas (confiança Baixa/Média) servem de apoio à revisão humana, com custo inerente de revisão.

5.2. Classificação e recomendação

A contribuição consistente da arquitetura aparece após a detecção. Considera-se enriquecido o achado que recebe categoria do guia, associação a CWE/OWASP quando aplicável, impacto potencial, prioridade e recomendação composta pelas três frentes de mitigação. No corpus de Pearce et al., 857 dos 871 achados (98%) receberam categoria do guia, as 124 detecções corretas receberam todas mapeamento CWE e os 857 achados classificados receberam todos recomendação completa (Tabela 4).

Tabela 4. Exemplo de enriquecimento estruturado para o achado H001 (exceção silenciosa).

Alerta Técnico	Regra H001: corpo do <code>except</code> <code>Exception</code> possui apenas <code>pass</code> .
Classificação	GIA-005, CWE-703, Prioridade Média.
O que verificar	Localizar blocos de captura genérica e silenciosa.
Como corrigir	Tratar exceção específica, registrar (ex: <code>logging.error</code>) e repropagar.
Como validar	Escrever testes para caminhos de erro e reexecutar a análise.

5.3. Ameaças à validade

Parte do *ground truth* provém do CodeQL, que possui falsos negativos próprios: o rótulo *seguro* significa não detectado por aquele oráculo. Os corpora são pequenos e restritos a Python, de modo que os resultados são ilustrativos de viabilidade e não um *benchmark* generalizável; além disso, não contêm em quantidade suficiente os padrões estruturais que a maioria das heurísticas detecta. Adicionalmente, as questões orientadoras (Q1–Q5) do guia não foram validadas por especialistas em segurança.

6. Conclusão

Este artigo apresentou um guia técnico de mitigação para código gerado por IA e uma arquitetura que o emprega como base de conhecimento em um fluxo automatizado de análise, classificação e recomendação. A avaliação em corpora externos mostra que a arquitetura não detecta mais que o SAST convencional — ela orienta melhor.

Como trabalhos futuros, destacam-se: a construção de um corpus rotulado para as categorias do guia, capaz de exercitar as heurísticas hoje avaliadas apenas qualitativamente; a generalização para linguagens além da Python, dado que o guia descreve categorias independentes de linguagem e é diretamente portátil; e a validação com especialistas e desenvolvedores.

Agradecimentos

Os autores declaram o uso de modelos de linguagem (Claude/Anthropic e Gemini/Google) como apoio à redação e à revisão do texto; o conteúdo e os resultados foram verificados pelos autores, que assumem a responsabilidade por eles.

Referências

- Dora, S. et al. (2025). The hidden risks of LLM-generated web application code: A security-centric evaluation of code generation capabilities in large language models. arXiv:2504.20612.
- Fu, Y. et al. (2025). Security weaknesses of Copilot-generated code in GitHub projects: An empirical study. *ACM Transactions on Software Engineering and Methodology*, 34(8):1–34.
- He, J. and Vechev, M. (2023). Large language models for code: Security hardening and adversarial testing. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 1865–1879. ACM.
- Pearce, H. et al. (2022). Asleep at the keyboard? Assessing the security of GitHub Copilot’s code contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 754–768. IEEE.
- Pearce, H. et al. (2023). Examining zero-shot vulnerability repair with large language models. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2339–2356. IEEE.
- Sajadi, A. et al. (2025). Do LLMs consider security? An empirical study on responses to programming questions. *Empirical Software Engineering*, 30:101.
- Siddiq, M. L. and Santos, J. C. S. (2022). SecurityEval dataset: mining vulnerability examples to evaluate machine learning-based code generation techniques. In *Proceedings of the 1st International Workshop on Mining Software Repositories Applications for Privacy and Security*, pages 29–33. ACM.
- Spracklen, J. et al. (2025). We have a package for you! A comprehensive analysis of package hallucinations by code generating LLMs. In *34th USENIX Security Symposium*, pages 3687–3706.
- Tihanyi, N. et al. (2025). How secure is AI-generated code: a large-scale comparison of large language models. *Empirical Software Engineering*, 30(2):47.
- Wadhams, Z. D., Reinhold, A. M., and Izurieta, C. (2024). Barriers to using static application security testing (SAST) tools: A literature review. In *39th IEEE/ACM International Conference on Automated Software Engineering Workshops*. ACM.
- Zhao, S. et al. (2025). Is vibe coding safe? Benchmarking vulnerability of agent-generated code in real-world tasks. arXiv:2512.03262.