Empirical Analysis of Known Bug and Vulnerability Regeneration in Large Language Model-Driven Coding Assistants (2025–2026 Latest Trends)

Introduction

AI coding assistants based on large language models (LLMs) have established themselves as an indispensable infrastructure in the software development life cycle (SDLC), undergoing a paradigm shift from simple static code completion tools to autonomous agents capable of interpreting context and executing tasks. GitHub Copilot, which leads the market, is adopted as a standard tool in development environments worldwide. However, parallel to its widespread use, a crucial concern has been raised: "Do AI tools still tend to regenerate known bugs, deprecated code, and security vulnerabilities as they are?"

Based on a comprehensive review of the latest empirical data and academic research (2025–2026), the answer to this question is clearly affirmative. That is, cutting-edge AI coding tools, including GitHub Copilot, possess a strong tendency to regenerate known bugs and security flaws even today. This phenomenon is not so much a defect in the models themselves, but rather a structural issue stemming from the fundamental specifications of probability-based text generation algorithms, training data contamination, and architectural limits in contextual understanding. Because AI models are designed to optimize "statistical probability" rather than "correctness" or "security," they inherently carry the risk of outputting vulnerable patterns that have accumulated in public repositories over decades.[1]

This report provides a comprehensive, multifaceted analysis of the fundamental mechanisms behind bug regeneration in AI-generated code, quantitative assessments of vulnerabilities, the reality and limitations of the security defense mechanisms implemented by GitHub Copilot, the factors contributing to the decline in code quality reported by the developer community between 2025 and 2026, and the increase in technical debt caused by human cognitive biases.

Quality of AI-Generated Code and the Fundamental Mechanism of Bug Regeneration

To understand the mechanisms by which AI coding tools regenerate bugs, it is necessary to unravel the nature of their underlying training data and the tool's architectural pipeline.

LLMs ingest billions of lines of code from public repositories like GitHub as training data. The core of the problem is that these datasets are not necessarily based on best practices; they contain a massive amount of bugs, inefficient algorithms, and known security vulnerabilities

written by past developers.[3] AI models merely learn the contextual co-occurrence probabilities of these code patterns. While a human developer would use threat modeling to reason, "How should the system fail-safe if this endpoint is called out of order?" or "Should an authenticated user be granted permission to access this specific object?", AI lacks such critical thinking.[1] Consequently, AI generates logical bugs that appear syntactically perfectly correct and conform to common framework patterns, yet completely fail to consider edge cases and abuse scenarios.[1]

Furthermore, empirical studies on the quality of AI-generated code indicate that the linguistic proficiency of the generated code falls short of human developers' expectations. A study presented at the "Mining Software Repositories" international conference in 2026 analyzed 591 pull requests containing 5,027 Python files generated by three different AI agents using the AIDev dataset.[4] By mapping Python constructs to six proficiency levels corresponding to the Common European Framework of Reference for Languages (CEFR), from A1 (Basic) to C2 (Mastery), using the static analysis tool pycefr, the study revealed that over 90% of the code generated by AI agents was classified as A1 and A2 (basic-level syntax), while less than 1% was classified as C2 (mastery level).[4] This creates an asymmetrical relationship: while the AI-generated code itself is basic and easy to understand, human developers reviewing and maintaining the code are required to possess an advanced level of proficiency for certain complex tasks.[4]

Another prominent example of bug regeneration is the "hallucination" of deprecated code and non-existent APIs. According to analysis data from Ryz Labs, GitHub Copilot has a tendency to suggest non-existent npm packages or already deprecated libraries and functions with an approximate probability of 15%.[5] The combination of the cutoff of AI training data (lack of the latest information) and its inferential nature—which forcibly attempts to apply patterns frequently found in old codebases to the latest projects—creates a severe risk of introducing known security bugs inherent in older APIs into modern systems.[5]

Quantitative Assessment: Frequency of Vulnerabilities and Bugs

As of 2023, security evaluations of general-purpose models like ChatGPT were the primary focus of research. However, between 2025 and 2026, numerous large-scale, systematic empirical studies on code-generation-specific LLMs like GitHub Copilot were published, confirming that the risks posed by AI-generated code are not theoretical concerns but quantitatively proven actual harms.

The following table summarizes quantitative data regarding the frequency of vulnerabilities and bugs in AI-generated code, as revealed by recent major academic studies and industry research reports.

Research Organization / Researcher (Year)	Survey Prerequisites and Target Data	Key Quantitative Findings and Conclusions	Source
Pearce et al. (2025)	Generated 1,689 programs using GitHub Copilot across 89 scenarios based on MITRE's "Top 25 CWE."	Confirmed that in security-sensitive contexts, **approximately 40% of the generated code contained critical vulnerabilities**.	[8]
Veracode GenAI Report (2025)	Compared the output of over 100 LLMs with human-written code across 4 languages: Java, JavaScript, Python, and C#.	Demonstrated that across all languages, **AI-generated code contained an average of 2.74 times more vulnerabilities compared to human-written code**.	[14]
Fortune 50 Enterprise Research (2026)	Measured the impact and risk factors of adopting AI tools in real-world enterprise environments.	While AI adoption increased development speed by 4x, it generated **10 times the security risks** compared to humans and increased privilege escalation paths by 322%.	[15]
Elsisi et al. (2026)	Tracked and analyzed AI-assisted tool commits from	Over 15% of all AI tool commits introduced new bugs. Furthermore,	[13]

	repository histories over an extended period.	**24.2% of the bugs introduced by AI remained unfixed in the latest repository revisions**.	
Empirical Study of 3.8K Bugs (2026)	Manually analyzed over 3,800 bugs reported in the open-source repositories of Claude-Code, Codex, and Gemini CLI.	67% of all bugs were related to functionality. 37.3% of root causes originated from API, integration, or configuration errors, with 37.6% of errors occurring during the tool invocation phase.	[16]

These quantitative data highlight the trade-offs introduced by AI coding tools. While AI rapidly generates boilerplate syntax and dramatically accelerates development speed, it exponentially increases "semantic vulnerabilities." For example, an internal survey by a Fortune 50 enterprise reported a 6-fold surge in commits of vulnerable AI-generated code between January and March 2026.[15]

A specific example of vulnerability generation occurred when GitHub Copilot was instructed to implement a file upload feature using Python (Flask). Copilot generated barebones code completely devoid of security controls, proposing an implementation that permitted arbitrary file uploads (a vulnerability directly leading to remote code execution).[17] Furthermore, when generating code to set HSTS (HTTP Strict Transport Security) in an HTTP response header, it omitted the preload directive, which is extremely important for security. Thus, while the code may appear to function normally at first glance, outputs that completely deviate from security best practices are frequently produced.[17]

The Reality and Limitations of the Security Defense Mechanisms (Vulnerability Prevention System)

GitHub acknowledges Copilot's tendency to generate vulnerable code and has responded by adding multiple security layers within its architecture. The Copilot pipeline currently employs a multi-stage structure: a phase to gather context from the workspace and open tabs, a pre-model filter applying responsible AI and content exclusion policies, a proxy service routing to the appropriate model, inference by the AI model, and a post-model filter performing

duplicate code detection and safety checks.[18]

The "AI-based Vulnerability Prevention System" acts as the post-model filter in this pipeline. This system aims to analyze the code using LLMs right before it is returned to the developer's editor, blocking common and dangerous coding patterns in real-time, such as hardcoded credentials (CWE-798), SQL injection (CWE-89), and path traversal.[19]

However, empirical studies and third-party audits have expressed extreme skepticism regarding the effectiveness of this defense mechanism. An evaluation by Credo AI concluded that it is highly unlikely this vulnerability filter can identify and block all potential security vulnerabilities.[19] Moreover, a 2025 study presented at an IEEE conference, titled "Artificially Insecure: Examining GitHub Copilot's AI-based Vulnerability Prevention System," detailed how researchers successfully bypassed this AI-based vulnerability prevention system, forcing Copilot to generate vulnerable code.[25]

The fundamental limitation of filter-based defense mechanisms is that they cannot detect context-dependent, complex logic bugs solely through syntactic pattern matching. For instance, race conditions that occur only in a specific call order, or privilege escalation flaws in complex business logic, cannot be identified as vulnerabilities by merely looking at code fragments.[30]

Further complicating matters is the introduction of the "Copilot Code Review" feature, where the AI reviews its own generated outputs. An empirical study published in September 2025 evaluated the performance of Copilot Code Review using labeled vulnerable code samples extracted from various open-source projects.[31] The results revealed that the AI reviewer frequently missed critical vulnerabilities such as SQL injection, Cross-Site Scripting (XSS), and insecure deserialization.[31] Instead, the AI reviewer tended to focus almost exclusively on superficial issues of extremely low security importance, such as code styling conventions and typos.[31] This phenomenon proves that AI places excessive emphasis on parsing syntax (format and structure) over semantics (meaning), suggesting that if developers rely on AI review results, the risk of critical bugs slipping into production environments increases dramatically.

A New Threat Vector: Attacks Targeting the Copilot Infrastructure Itself

In addition to bugs in AI-generated code, between 2025 and 2026, a phenomenon emerged where the GitHub Copilot tool itself became a direct attack vector. This stems from the evolution of AI coding assistants from mere editor plugins into autonomous agents deeply integrated into CI/CD pipelines and GitHub infrastructure. The table below shows major Copilot-related CVEs and vulnerabilities reported during this period.

Vulnerability	Type and	CVSS / Severity	Source

Identifier / Attack Name	Mechanism of Vulnerability		
CVE-2025-64660	**Improper Access Control** A vulnerability allowing an authenticated attacker to bypass security features over a network in GitHub Copilot and VS Code.	5.7 (Moderate) EPSS: 0.134%	[32]
CVE-2025-59286	**Improper Access Control (Similar)** A flaw in access control within the Copilot infrastructure. Tracked as GHSA-hh8r-hx5c-8r8r, though detailed source code remains unpublished.	Moderate EPSS: 0.096%	[34]
CVE-2025-53773 (Rules File Backdoor)	**Prompt Injection** A supply chain attack that embeds hidden Unicode characters in configuration files to rewrite Copilot's rules, leading to Remote Code Execution (RCE).	High / Critical	[22]
CVE-2025-32711	**Zero-Click Prompt**	High / Critical	[36]

(EchoLeak)	**Injection** An attack method that injects prompts into Copilot via external emails or documents without user interaction, leading to data exfiltration.		
CamoLeak	**Data Exfiltration** An attack hiding malicious instructions in pull request descriptions, causing Copilot to read sensitive data within the codebase and transmit it externally via GitHub's infrastructure.	Advanced Threat	[37]

These cases clearly demonstrate that as AI becomes more autonomous, the risk is not just AI "writing bugs," but AI itself being exploited as a "proxy (pathway) for hacking." Particularly, prompt injection attacks like CamoLeak and EchoLeak exploit a fundamental architectural flaw (OWASP LLM01:2025) wherein LLMs cannot strictly differentiate between "legitimate system instructions" and "untrusted user or external inputs." This represents a fatal vulnerability when AI assistants process information across enterprise boundaries.[35]

Root Causes of Code Quality Degradation in 2025–2026 (From an Architectural Perspective)

In early 2026, an explosion of dissatisfaction occurred on developer forums such as GitHub Community and Stack Overflow, with complaints like "Copilot's suggestion quality has noticeably declined" and "It used to understand context, but now it suggests completely off-base code".[5] This sentiment is backed quantitatively: in the 2025 Stack Overflow Developer Survey, positive sentiment toward AI coding tools plummeted from over 70% in 2023–2024 to a

mere 60%.[5]

This decline in quality is not a user illusion, but a phenomenon brought about by the following structural architectural changes and shifts in business models.

The "Model Carousel" Phenomenon and Tuning Discrepancies

The biggest factor behind the quality decline is the "Model Carousel" phenomenon, where the underlying foundation models are frequently changed without clear notification to the users.[5] GitHub Copilot has repeatedly transitioned from the initial OpenAI Codex, through various GPT-4 variants, and since late 2025, dynamically routing to models in the GPT-5 series, as well as models from multiple providers like Claude 3.7 Sonnet and Gemini 2.0.[5]

Generally, newer, massive models (e.g., GPT-5) are assumed to be superior in language understanding, but this is not necessarily true in the context of software engineering. Prompt engineering techniques and context selection algorithms optimized for older models (like Codex) do not align with the characteristics of the new models, resulting in performance regressions.[5] The delay in tuning accompanying these model swaps directly correlates to higher bug regeneration rates and a worsened user experience.

Context Window Limitations and "Multi-File Blindness"

In modern enterprise software development, most bugs arise not from syntax errors within a single file, but from architectural inconsistencies across multiple files. Copilot's standard context window for inline completions is restricted to approximately 8,000 tokens.[5]

This 8,000-token limit is fatal in a massive Monorepo environment. The AI cannot reference project-specific conventions and interface definitions entirely, resulting in generic code that doesn't fit the project (or causes logical bugs).[5] According to Ryz Labs' analysis, in projects exceeding 10,000 lines, the probability of Copilot making an accurate suggestion drops to just 50%.[5] In tasks involving modifications to 10 or more files, the system lacks the requisite tokens to understand architectural interrelationships, falling into a state called "multi-file blindness," which drastically spikes the logical error rate.[5]

Decline in Proposal Acceptance Rates and Infrastructure Instability

Due to the lack of context and increased hallucinations, Copilot's proposal acceptance rate as of 2026 has dropped to 35–40%, falling below that of competitor Cursor (42–45%).[5] 75% of senior engineers report spending more time correcting erroneous code (bugs) generated by Copilot than they would writing code manually, offsetting the original goal of productivity enhancement.[5]

Furthermore, Copilot within IDEs (such as Visual Studio) has been widely reported in early 2026 to exhibit instability. When reaching the token limit during long sessions or complex threads, it loses context (becoming "scatty"), freezes the entire application for several minutes, or in the worst cases, crashes completely, leaving lingering doubts about its reliability as a tool.[41] Reports

in 2026 also pointed out issues with sluggishness (latency), where starting a web-based agent could take over 90 seconds.[5]

In March 2026, an incident known as the "PR Ads Controversy" occurred. Copilot inadvertently injected promotional "tips (ads)" into the descriptions of over 1.5 million pull requests, prompting Microsoft to disable the feature, citing it as a "programming logic issue." This incident highlighted the difficulty of controlling AI output while severely damaging developers' trust in the platform.[5]

Comparative Analysis with Competitor AI Coding Agents

Amidst discussions of GitHub Copilot's quality issues, the market for AI coding assistants has diversified, with competing tools emerging that surpass Copilot in specific functionalities. The table below compares the characteristics, bug reduction, and context understanding approaches of major AI coding tools as of 2026.

Tool Name	Approach to Context Understanding and Bug Reduction	Primary Strengths and Weaknesses	Source
GitHub Copilot	Editor integrated, transitioning to Agent Mode. Built-in vulnerability prevention system, but prone to losing context in large repositories due to the 8,000-token limit.	【Strengths】 Overwhelming enterprise adoption rate, ecosystem integration. 【Weaknesses】 High bug rate in multi-file editing, inconsistent quality due to model changes.	[5]
Cursor	Advanced context comprehension via deep indexing technology. Plan mode for designing complex tasks and visual diff display.	【Strengths】 High proposal acceptance rate (42-45%). Excellent in multi-file editing. 【Weaknesses】	[5]

		Requires using a proprietary forked editor.	
Claude Code	Utilizes a massive context window of 1 million (1M) tokens via Claude 3.7 Sonnet. Operates autonomously from the terminal.	【Strengths】Can reason across the entire massive codebase. Fewer bugs caused by missing context. 【Weaknesses】Specialized for terminal operations, offering a different experience from inline completions.	[5]
Cline	Fully autonomous open-source agent. Autonomously executes commands, edits files, and reads errors.	【Strengths】Inexpensive by bringing your own API key (BYOK). Powerful automation for repetitive tasks. 【Weaknesses】Requires monitoring (babysitting). Risk of unstable operation.	[7]
Amazon Q Developer	Specialized for AWS environments. Strong in security scanning for Infrastructure as Code (IaC) such as EC2, Lambda, and CloudFormation.	【Strengths】Accurate code generation optimized for AWS APIs. IP indemnification. 【Weaknesses】Lacks versatility in	[45]

		non-AWS environments.	
Tabnine	Prioritizes privacy, guaranteeing user code is not used as training data. Offers a local execution option.	【Strengths】High privacy standards. Meets enterprise compliance requirements. 【Weaknesses】Inferencing capabilities may fall behind the latest cloud-based LLMs.	[21]

As is evident from this comparison, Copilot's agent mode is cementing a reputation as "competent but not best-in-class".[42] Particularly when compared to Claude Code's 1-million-token context window or Cursor's deep indexing, Copilot suffers from an architectural weakness that makes it prone to generating bugs caused by contextual mismatches during complex multi-file refactoring.[5]

Changes in Licensing Models and Data Privacy in 2026

Behind the frequency of bugs and quality degradation lies the impact of GitHub's changing business models and resource allocations.

In the restructured pricing tiers of 2026 (Free, Pro, Pro+, Business, Enterprise), a cap was introduced to the individual Pro plan ($10/month), limiting "premium requests utilizing the latest models" to 300 times per month.[5] Since these premium requests are rapidly consumed when using Agent Mode or complex multi-step tasks, many developers find themselves hitting the cap after just a few days of heavy coding sessions.[42] Once the cap is reached, it automatically falls back to lower-performing base models, triggering the phenomenon where "code quality suddenly drops mid-month and becomes riddled with bugs".[5] Furthermore, an announcement in March 2026 restricted free tier students from opting into premium models like GPT-5.4 and Claude Opus/Sonnet.[48] To improve platform maintainability, support for older versions of JetBrains IDEs (2024.2 and 2024.3) was also terminated.[49]

Furthermore, from the perspective of data privacy and model improvement, the update to the "Interaction Data Usage Policy" effective April 24, 2026, carries profound significance.[50] With this update, interaction data collected from Copilot Free, Pro, and Pro+ users (input prompts, output results, code snippets, context around the cursor position, and accept/modify history) will be used for AI model training and improvement unless users explicitly opt out.[50]

GitHub explains that by leveraging this real-world data, they aim to compensate for the

shortcomings of initial models trained solely on static datasets, improving proactive bug detection capabilities and providing safer code patterns (Secure Code Patterns).[50] However, data from Copilot Business and Enterprise users, as well as data from enterprise-owned repositories, remain protected and excluded from this training program.[50]

Developer Psychological Biases (Automation Bias) and the Growth of Technical Debt

Beyond technical constraints, a more severe issue for the software development ecosystem is the psychological factor that arises in the interaction between humans and AI. The speed at which AI generates code—extremely fast and in a confident tone—creates a serious blind spot in the developer's cognitive process.

Over-reliance and the Hollowing Out of Code Reviews

Academic behavioral analyses (Perry et al., 2023; Sabouri et al., 2025; Ahuchogu et al., 2025) have highlighted the reality that developers place "Excessive Trust" in the quality of AI-generated code, blindly accepting it without proper validation.[8]

The proliferation of AI tools has fundamentally shifted the primary role of a software engineer from an "Author" who writes code from scratch to a "Reviewer" who verifies AI-generated code.[4] However, because the code presented by AI "looks syntactically beautiful and plausible," humans tend to lower their psychological guard and suspend critical thinking. Security flaws do not lie in obvious grammatical errors, but within the implicit assumptions and edge cases of logic that appears correct on the surface.[1]

The result is a phenomenon where unverified bugs and security vulnerabilities are integrated into the production codebase, accumulating as "Technical Debt" that will demand immense correction costs and time down the road.[8] He et al. (2025), through empirical study, proved that while the introduction of AI coding assistants provides a transient velocity boost, it leads to a persistent increase in code complexity in the long run.[8] This means that human developers are leaving unnecessary boilerplate and redundant logic generated by AI unattended without proper refactoring.

Widespread Propagation of Bugs via Autonomous Agents (Agent Mode)

In the era when inline completion was mainstream, bugs generated by AI were localized to a few lines near the cursor, making the "Blast Radius" easier for developers to visually control. However, the Copilot Coding Agent (cloud agent)—introduced in 2025 and popularized in 2026—runs asynchronously in the background, autonomously exploring codebases, making batch modifications across multiple files, running tests, and creating pull requests with completed commits.[22]

If the AI learns an "inappropriate premise" within this autonomous loop, the bug or vulnerability

does not remain confined to a single function; it consistently and silently propagates across multiple microservices and endpoints.[1] For instance, if the AI decides to implement an incorrect authentication flow or an insecure logging pattern, the same flaw will be duplicated across all new endpoints of that project, creating a "Uniform Exposure" that looks uniform but is actually widely exposed to attacks.[1] While Agent Mode can adequately handle "clear failures" like syntax errors or build errors through its Course-correction function [53], if the test cases themselves are missing or do not cover logical vulnerabilities, the AI will deem the execution a "test pass (success)" and complete the process while retaining critical bugs.

Conclusion and Strategic Recommendations

From the series of empirical data and architectural trend analyses, the answer to the user's inquiry is exceptionally clear: **As of 2026, GitHub Copilot retains a strong tendency to regenerate known bugs, deprecated code, and security vulnerabilities as they are.**

This tendency is rooted in the fundamental principle of LLMs, which attempt to generate "the statistically most frequent code (including past vulnerable patterns)," surpassing the capabilities of the AI-based Vulnerability Prevention System (real-time filter) built into Copilot. Furthermore, tuning discrepancies caused by frequent model switching (Model Carousel) and the exhaustion of the 8,000-token context window in large-scale projects are driving up the rates of logical bugs and hallucinations.

While the evolution of AI tools (multi-file autonomous editing via Agent Mode) has dramatically improved development speed, it has brought about the risk of bugs and vulnerabilities propagating instantly across broad architectures. Compounded by the psychological factor of developers "placing excessive trust in AI and accepting code without sufficient validation (Automation Bias)," severe technical debt is rapidly accumulating in today's software supply chains.

To counter the threat of bug regeneration accompanying AI-generated code, it is imperative to establish a new engineering paradigm and defense layers predicated on AI reliance:

1. **Mandatory AI-Driven SAST (Static Application Security Testing)** To monitor the massive amounts of code autonomously generated by Copilot, it is necessary to deeply integrate AI-based security scanners capable of semantically interpreting context and execution flow into CI/CD pipelines, going beyond traditional signature-based SAST.[1] If AI generates code at high speed, its validation must be automated with equivalent speed and precision.
2. **Prompt Partitioning and Context Control** To maximize Copilot's limited context window and prevent hallucinations, it is highly effective to artificially limit the AI's "field of vision (Blast Radius)" by breaking down complex tasks into smaller pieces and closing unnecessary files.[47]
3. **Elimination of Over-reliance on Copilot Code Review** The "closed echo chamber" of having AI (Copilot Code Review) review code written by AI (Copilot) must be avoided. As empirical studies show, AI reviews miss critical vulnerabilities such as SQL injections [31];

therefore, an audit by human experts or robust independent security tools must be enforced in security-critical paths.

4. **Building Defense-in-Depth against Prompt Injection** As demonstrated by cases like EchoLeak and the Rules File Backdoor, it is crucial to prepare for the risk of malicious code embedded in pull request comments or configuration files being executed via Copilot.[22] Applying strict Content Security Policies (CSP) when processing external inputs and implementing provenance-based access controls are strongly recommended.[36]

AI coding assistants are no longer just "typing aids" but "silent co-authors" that form the foundation of systems. Organizations as a whole must share the understanding that blindly trusting their output is tantamount to uncritically welcoming all the bugs and vulnerabilities accumulated in the open-source world over the past decades into their own projects.

引用文献

1. Vulnerabilities of Coding with GitHub Copilot: When AI Speed Creates Invisible Risk, 4月 18, 2026にアクセス、https://brightsec.com/blog/vulnerabilities-of-coding-with-github-copilot-when-ai-speed-creates-invisible-risk/
2. What is the Responsibility of Developers Using Generative AI? A, 4月 18, 2026にアクセス、https://learn.modernagecoders.com/blog/what-is-the-responsibility-of-developers-using-generative-AI
3. [2512.05239] A Survey of Bugs in AI-Generated Code - arXiv, 4月 18, 2026にアクセス、https://arxiv.org/abs/2512.05239
4. When is Generated Code Difficult to Comprehend? Assessing AI Agent Python Code Proficiency in the Wild - arXiv, 4月 18, 2026にアクセス、https://arxiv.org/html/2604.00299v1
5. Is GitHub Copilot Getting Worse in 2026? What Changed & Why ..., 4月 18, 2026にアクセス、https://nxcode.io/resources/news/github-copilot-getting-worse-2026-developers-switching
6. Choosing the Best AI for Coding in 2026 - Blog - Silk Data, 4月 18, 2026にアクセス、https://silkdata.tech/blog/article/best-ai-for-coding
7. I Tested 12 AI Coding Tools. Here's the Only Ranking That Matters. - Medium, 4月 18, 2026にアクセス、https://medium.com/lets-code-future/i-tested-12-ai-coding-tools-heres-the-only-ranking-that-matters-27c7c00e0e7f
8. Debt Behind the AI Boom: A Large-Scale Empirical Study of AI-Generated Code in the Wild, 4月 18, 2026にアクセス、https://arxiv.org/html/2603.28592v1
9. A Survey of Bugs in AI-Generated Code - arXiv, 4月 18, 2026にアクセス、https://arxiv.org/html/2512.05239v1
10. TypePilot: Leveraging the Scala type system for secure LLM-generated code - ACL Anthology, 4月 18, 2026にアクセス、https://aclanthology.org/2025.ommm-1.11.pdf

11. A Survey on Large Language Models in Software Security: Opportunities and Threats, 4月 18, 2026にアクセス、https://www.mdpi.com/2073-431X/15/4/226
12. Malicious Attack Challenges and Mitigation ... - ScholarSpace, 4月 18, 2026にアクセス、https://scholarspace.manoa.hawaii.edu/bitstreams/74c9194b-4b3f-4997-a15a-9acb295f8540/download
13. Asleep at the Keyboard? Assessing the Security of GitHub Copilot's ..., 4月 18, 2026にアクセス、https://www.researchgate.net/publication/388193053_Asleep_at_the_Keyboard_Assessing_the_Security_of_GitHub_Copilot's_Code_Contributions
14. AI-Generated Code Security Risks - Why Vulnerabilities Increase 2.74x and How to Prevent Them - SoftwareSeni, 4月 18, 2026にアクセス、https://www.softwareseni.com/ai-generated-code-security-risks-why-vulnerabilities-increase-2-74x-and-how-to-prevent-them/
15. The 23.7% Security Vulnerability Tax: Is AI-Generated Code Worth the Risk? - TianPan.co, 4月 18, 2026にアクセス、https://tianpan.co/forum/t/the-23-7-security-vulnerability-tax-is-ai-generated-code-worth-the-risk/3865
16. Engineering Pitfalls in AI Coding Tools: An Empirical Study of Bugs in Claude Code, Codex, and Gemini CLI - arXiv, 4月 18, 2026にアクセス、https://arxiv.org/html/2603.20847
17. Insecure coding workshop: Analyzing GitHub Copilot suggestions - Invicti, 4月 18, 2026にアクセス、https://www.invicti.com/blog/web-security/analyzing-security-github-copilot-suggestions
18. GitHub Copilot Workflow Cheatsheet — 2026 Edition, 4月 18, 2026にアクセス、https://sukurcf.github.io/resources/github-copilot-cheatsheet.html
19. Github Copilot - AI Vendor Risk Profile, 4月 18, 2026にアクセス、https://www.credo.ai/ai-vendor-directory/github-copilot
20. Week 2: Features & Data handling – Copilot Free learning & cert prep #151641 - GitHub, 4月 18, 2026にアクセス、https://github.com/orgs/community/discussions/151641
21. 16 Best AI Coding Assistants to Boost Your Engineering Productivity in 2025 | DigitalOcean, 4月 18, 2026にアクセス、https://www.digitalocean.com/resources/articles/best-ai-coding-assistant
22. A Developer's Guide to Writing Secure Code with GitHub Copilot - StackHawk, 4月 18, 2026にアクセス、https://www.stackhawk.com/blog/github-copilot-secure-coding-guide/
23. What is GitHub Copilot? - Pangea.app, 4月 18, 2026にアクセス、https://pangea.app/glossary/github-copilot
24. 1月 1, 1970にアクセス、https://github.blog/2023/02/14/github-copilot-now-has-a-vulnerability-prevention-system-to-block-insecure-code-patterns/
25. XOXO: STEALTHY CROSS-ORIGIN CONTEXT POI ... - OpenReview, 4月 18, 2026にアクセス、

https://openreview.net/pdf/6179550cf18cac2acd7b49b7e499ae47ac39270e.pdf
26. Artificially Insecure: Examining GitHub Copilot's AI-based Vulnerability Prevention System, 4月 18, 2026にアクセス、https://ieeexplore.ieee.org/document/11133629/
27. Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions, 4月 18, 2026にアクセス、https://www.computer.org/csdl/proceedings-article/sp/2022/131600a980/1FIQxERjKCs
28. Connected Papers | Find and explore academic papers, 4月 18, 2026にアクセス、https://www.connectedpapers.com/main/5de990a34c7ea1da2b3a98545422573ba6f44873
29. AI-assisted Collaboration: Exploring Developer Experience with GitHub Copilot and Windsurf - IEEE Computer Society, 4月 18, 2026にアクセス、https://www.computer.org/csdl/magazine/so/5555/01/11429684/2eNCjd5qusw
30. AI Code Security: Why Defenders Can't Afford to Fall Behind, 4月 18, 2026にアクセス、https://pluto.security/blog/ai-code-security-defenders/
31. GitHub's Copilot Code Review: Can AI Spot Security Flaws Before You Commit? - arXiv, 4月 18, 2026にアクセス、https://arxiv.org/html/2509.13650v1
32. Improper access control in GitHub Copilot and Visual... · CVE-2025-64660, 4月 18, 2026にアクセス、https://github.com/advisories/GHSA-j8xq-6qq7-vfv7
33. CVE-2025-64660 Detail - NVD, 4月 18, 2026にアクセス、https://nvd.nist.gov/vuln/detail/CVE-2025-64660
34. Copilot Spoofing Vulnerability · CVE-2025-59286 · GitHub Advisory Database, 4月 18, 2026にアクセス、https://github.com/advisories/GHSA-hh8r-hx5c-8r8r
35. Prompt Injection Attacks in Large Language Models and AI Agent Systems: A Comprehensive Review of Vulnerabilities, Attack Vectors, and Defense Mechanisms - Preprints.org, 4月 18, 2026にアクセス、https://www.preprints.org/manuscript/202511.0088?utm_source=chatgpt.com
36. EchoLeak: The First Real-World Zero-Click Prompt Injection Exploit in a Production LLM System - arXiv, 4月 18, 2026にアクセス、https://arxiv.org/html/2509.10540v1
37. CamoLeak: How GitHub Copilot Became An Exfiltration Channel | BlackFog, 4月 18, 2026にアクセス、https://www.blackfog.com/camoleak-how-github-copilot-became-an-exfiltration-channel/
38. Prompt Injection Attacks in Large Language Models: Vulnerabilities, Exploitation Techniques, and Defense Strategies | by Khmaïess Jannadi | Medium, 4月 18, 2026にアクセス、https://medium.com/@jannadikhemais/prompt-injection-attacks-in-large-language-models-vulnerabilities-exploitation-techniques-and-e00fe683f6d7
39. Is GitHub Copilot Getting Worse in 2026? What Changed & Why Devs Are Switching, 4月 18, 2026にアクセス、https://www.nxcode.io/resources/news/github-copilot-getting-worse-2026-developers-switching
40. Why Teams Use GitHub Copilot Wrong and Fix It in 2026 - ThoughtMinds, 4月 18, 2026にアクセス、

https://thoughtminds.ai/blog/why-teams-use-github-copilot-wrong-and-fix-it
41. Grrr Co-Pilot in Visual Studio 2026 is so impressive and frustrating ..., 4月 18, 2026にアクセス、https://github.com/orgs/community/discussions/187525
42. GitHub Copilot 2026: Complete Guide to Pricing, Agent Mode & Coding Agent | NxCode, 4月 18, 2026にアクセス、https://www.nxcode.io/resources/news/github-copilot-complete-guide-2026-features-pricing-agents
43. Cursor vs. GitHub Copilot: Which AI Coding Assistant Is Better? | DataCamp, 4月 18, 2026にアクセス、https://www.datacamp.com/es/blog/cursor-vs-github-copilot
44. Using AI Agent Cline to Remove Deprecated Code from SAP Fiori Applications, 4月 18, 2026にアクセス、https://community.sap.com/t5/technology-blog-posts-by-sap/using-ai-agent-cline-to-remove-deprecated-code-from-sap-fiori-applications/ba-p/14333374
45. CodeWhisperer (Amazon Q) vs. Copilot: Best AI Coding Assistant for Enterprise?, 4月 18, 2026にアクセス、https://www.augmentcode.com/tools/codewhisperer-vs-copilot-best-ai-coding-assistant-for-enterprise
46. GitHub Copilot · Your AI pair programmer, 4月 18, 2026にアクセス、https://github.com/features/copilot
47. What I've Learned About GitHub Copilot Agent Mode - DEV Community, 4月 18, 2026にアクセス、https://dev.to/anchildress1/what-ive-learned-about-github-copilot-agent-mode-4co2
48. Important Updates to GitHub Copilot for Students · community · Discussion #189268, 4月 18, 2026にアクセス、https://github.com/orgs/community/discussions/189268
49. Important updates: GitHub Copilot support ending for JetBrains 2024.2 and 2024.3, 4月 18, 2026にアクセス、https://devblogs.microsoft.com/java/important-updates-ghsupport-ending-for-jetbrains-2024-2-and-2024-3/
50. Updates to GitHub Copilot interaction data usage policy - The ..., 4月 18, 2026にアクセス、https://github.blog/news-insights/company-news/updates-to-github-copilot-interaction-data-usage-policy/
51. Usage, Effects and Requirements for AI Coding Assistants in the Enterprise: An Empirical Study - arXiv, 4月 18, 2026にアクセス、https://arxiv.org/html/2601.20112v1
52. (PDF) Usage, Effects and Requirements for AI Coding Assistants in the Enterprise: An Empirical Study - ResearchGate, 4月 18, 2026にアクセス、https://www.researchgate.net/publication/400178243_Usage_Effects_and_Requirements_for_AI_Coding_Assistants_in_the_Enterprise_An_Empirical_Study
53. Agent mode 101: All about GitHub Copilot's powerful mode - The ..., 4月 18, 2026にアクセス、https://github.blog/ai-and-ml/github-copilot/agent-mode-101-all-about-github-copilots-powerful-mode/