مجلـــة كليـــة التربيـــة طرابلـــس

Journal of the Faculty of Education Tripoli

أثر استخدام النماذج اللغوية الكبيرة في توليد الشفرة البرمجية على أمن البرمجيات

Fawzeyia Elhashmi Salah

Department of Computer, Faculty of Education Tripoli, University of Tripoli, Libya

Email: F.Salah@uot.edu.ly

Article history

Received: 17 Jun 2026 Accepted: 30 Aug 2026 Published: 15 Sep 2026

الملخص:

إن إدماج نماذج اللغات الكبيرة (LLMs) في مسارات عمل هندسة البرمجيات الحديثة قد أعاد تشكيل عملية التوليد الآلي للشيفرة البرمجية بشكل جوهري إلا أن المخاطر الأمنية المقترنة بالشيفرات البرمجية المولدة بواسطة الذكاء الاصطناعي لا تزال تشكل تحديا كبيرا لمدى موثوقية البرمجيات وسلامة سلاسل التوريد. تقدم هذه الورقة البحثية تحليلا شاملا للبحوث الأولية المنشورة بين عامي 2022 و2026 للتحقق من كيفية تأثير إكمال الشيفرات وتخليقها المدعوم بنماذج اللغات الكبيرة على أمن البرمجيات. وتأسيسا على الالتزام الصارم بإرشادات العناصر المفضلة للإبلاغ عن المراجعات المنهجية والتحليلات البعدية(PRISMA 2020) ؛ تم استرجاع 55 دراسة أولية وتقييمها نقديا، عبر خمس مستودعات رقمية رئيسية: IEEE Xplore، ACM Digital Library، Scopus، ScienceDirect، و arXiv . ويثبت هذا التحليل البحثي أنه على الرغم من إسهام النماذج اللغوية الكبيرة في تعزيز إنتاجية المطورين بدرجة كبيرة إلا أن النماذج التجارية والمفتوحة المصدر بما في ذلك GitHub Copilot (powered by OpenAI Codex/GPT-4o) ، OpenAI GPT-3.5-Turbo and GPT-4 ، Meta Llama 2 (13B/70B) and Llama 3 (8B/70B)و CodeLlama (7B/13B/34B/70B-Python) غالبا ما تُدخل ثغرات أمنية خطيرة، مثل: حقن SQL (CWE-89) ، وتجاوز سعة المخزن التجميعي(CWE-119/CWE-120) والبرمجة العابرة للمواقع(CWE-79) . وهي ثغرات تتوافق مباشرة مع نقاط الضعف المصنفة في قائمة الثغرات الشائعة (CWE) وقائمة OWASP Top 10. ولمواجهة هذه العيوب الأمنية، تقترح الأدبيات البحثية أربع متجهات دفاعية رئيسية، تتمثل في: هندسة الأوامر المدركة للأمن، وحلقات التغذية الراجعة للتحليل الاستاتيكي، والضبط الدقيق للنماذج الموجهة بواسطة Oracle، وأطر الضوابط الوقائية للأنظمة الوكيلة. وأخيرًا، توحد هذه الورقة البحثية هذه النتائج المذكورة ضمن تصنيف موحد، وتقيّم أداء النماذج عبر الاختبارات المعيارية، وتحدد الاتجاهات البحثية المستقبلية الاستراتيجية للوصول إلى توليد برمجيات آمن ومدعم بالذكاء الاصطناعي.

الكلمات المفتاحية: أمن البرمجيات، البرمجة بمساعدة الذكاء الاصطناعي، التحليل الاستاتيكي للكود، الثغرات البرمجية، المراجعة المنهجية للأدبيات، النماذج اللغوية الكبيرة، توليد الكود

The Impact of Using Large Language Models for Code Generation on Software Security

A B S T R A C T:

Automated code generation has been drastically altered by the integration of Large Language Models into modern software engineering processes. However, the security dangers related to code produced by AI continue to be a significant threat to software reliability. A thorough summary of research from 2022 to 2026, that examines how code completion and synthesis using LLM affect software security is presented in this research paper. Following PRISMA guidelines. Through 5 prominent digital repositories: IEEE Xplore, ACM Digital Library, Scopus, Science Direct, and arXiv. 55 primary studies were systematically recovered and evaluated. The study shows that, even if LLMs considerably quicken the pace of development, open source and commercial models including GitHub Copilot (powered by OpenAI Codex/GPT-4o), OpenAI GPT-3.5-Turbo and GPT-4, Meta Llama 2 (13B/70B) and Llama 3 (8B/70B), as well as CodeLlama (7B/13B/34B/70B-Python)frequently introduce critical vulner abilitiess such as Cross-Site Scripting (CWE79), Buffer Overflows (CWE 119/120), and SQL Injection (CWE 89).These flaws correspond precisely to the shortcomings enumerated in the Common Weakness Enumeration CWE and OWASP Top 10.To fix these security flaws, literature suggests four main defense vectors: agentic guardrail frameworks, oracle-guided model fine-tuning, static analysis feedback loops, and security aware prompt engineering. This paper combines these discoveries into a single taxonomy, assesses the benchmark performance of models, and defines strategic future research areas to enable secure AI-assisted software synthesis.

Keywords: Software Security, AI-Assisted Coding, Static Code Analysis, Software Vulnerabilities, Systematic Literature Review, Large Language Models, Code Generation.

1 . Introduction and Background Scientific Context

1.1 Motivation and Problem Statement

The rise of generative artificial intelligence has catalyzed a fundamental paradigm shift in software engineering practice. Large Language Models (LLMs) trained on vast repositories of open-source software codebases have transitioned from experimental auto complete tools to sophisticated developer assistants capable of generating complex multi file function implementations, refactoring legacy systems, and translating logic across programming languages. Commercial and opensource foundation models, specifically GitHub Copilot (powered by GPT-4o and Codex), OpenAI GPT-3.5-Turbo and GPT-4-0613, Anthropic Claude 3.5 Sonnet, Meta Llama 2 (13B/70B), Llama 3 (8B/70B-Instruct), CodeLlama (7B/13B/34B/70B-Python), BigCode StarCoder-15B and StarCoder2-15B, and DeepSeek-Coder-33B-Instruct—are now deeply embedded in enterprise development workflows. However, as surveyed by Jiang et al (2024), Joel et al (2024), and Yao et al (2023), this rapid boost in software development velocity introduces critical latent risks into the software supply chain. Transformer models predict likely text patterns rather than verify code security. Consequently, they naturally absorb and replicate insecure coding practices found in public training data. Empirical security evaluations by Black et al (2025), Götz and Schaad (2024), and Liu et al (2023) demonstrate that AI-generated code snippets routinely contain severe design and implementation flaws. These security weaknesses range from dynamic memory safety violations in low level languages like C/C++ to injection vulnerabilities and broken access control in web frameworks. This problem is made worse by human programmers frequently overly rely on AI tools, assuming that synthesized outputs possess implicit security verification. As a result, unverified AI code ends up directly in production. This practice significantly expands the attack surface of modern applications. Understanding the mechanisms, taxonomies, and countermeasures associated with LLM security flaws is therefore an essential imperative for computer science researchers and cyber security practitioners.

1.2 Research Questions

To systematically map, evaluate, and synthesize the scientific literature regarding LLM code generation security, this review formulates three core research questions:

- **RQ1**: What are the most common security vulnerabilities that Large Language Models generate, and how do they map to standardized vulnerability taxonomies such as CWE and OWASP Top 10?
- **RQ2**: What LLM architectures, evaluation benchmarks, and detection mechanisms are reported in the literature to assess the security posture of AI-generated code?
- **RQ3**: What mitigation strategies ranging from prompt engineering to static analysis feedback loops, fine-tuning, and guardrail architectures are effective in ensuring secure code synthesis?

2. Research Methodology

This systematic literature review was executed strictly according to the PRISMA protocol to guarantee scientific transparency, methodological rigor, and exhaustive coverage of the domain, aligning with methodological guidelines that Bašić and Giaretta (2024), He et al (2026), and Umama et al (2025) detail.

2.1 Review Protocol and Search Syntax

This SLR adheres to PRISMA 2020 and Kitchenham's software engineering guidelines. To match search syntax across repositories, the core Boolean query—("LLM" OR "Copilot" OR "CodeLlama") AND ("Code Generation" OR "Code Synthesis") AND ("Security" OR "Vulnerability" OR "CWE") was adapted specifically for each engine:

- **IEEE Xplore / ACM DL**: Applied using strict Document Title and Abstract field tags.
- **Scopus**: Executed via TITLE-ABS-KEY syntax bounded to publication years ≥ 2022.
- **ScienceDirect / arXiv**: Query restricted to TITLE-ABSTR-KEY and ti/ab field filters

2.2 Study Selection Criteria and Scope Rational

To filter raw search hits and ensure that only rigorous, empirical research was included in the final synthesis, a multi tiered screening protocol was used. This framework defines explicit operational inclusion/exclusion rules, model family coverage limits, and objective quality assessment scoring thresholds.

2.2.1 Primary Inclusion and Exclusion Criteria

Candidate studies were evaluated against six predefined selection criteria (I1–I3 for inclusion, E1–E3 for exclusion) to guarantee direct relevance to LLM code security. as detailed in Table 1.

(Table 1) Primary Inclusion and Exclusion Criteria

Criterion Identifier	Selection Rule	Operational Definition and Rationale
Inclusion (I1)	Primary Empirical Focus	Studies conducting empirical evaluations, benchmarking, or theoretical modeling of LLM code generation security.
Inclusion (I2)	Security Assessment	Research offering quantitative or qualitative metrics on code vulnerabilities generated by transformer models.
Inclusion (I3)	Technical Mitigation	Works proposing actionable frameworks, fine-tuning protocols, prompt engineering, or guardrail systems for secure synthesis.
Exclusion (E1)	Temporal Bound	Publications produced prior to 2022, excluding legacy non-transformer or pre-LLM statistical code completion tools.

Exclusion (E2)	Out-of-Scope Focus	Papers examining natural language generation without explicit code generation or software security evaluations.
Exclusion (E3)	Quality and Duplication	Non-English texts, keynotes, extended abstracts lacking empirical validation, and duplicate preprints.

2.2.2 Model Scope and Selection Rationale

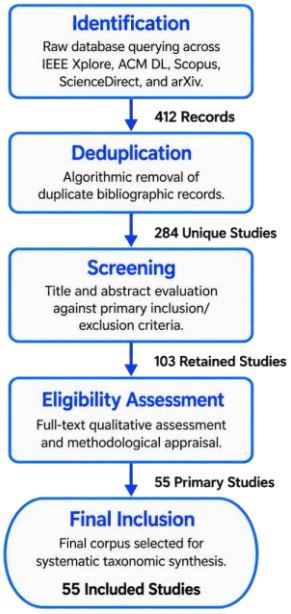
To cover the AI-assisted code generation landscape, this systematic review did not restrict its taxonomy to a single model vendor or deployment architecture. Instead, primary studies were included based on their evaluation of both proprietary commercial APIs including OpenAI (GPT-3.5, GPT-4, GPT-4o) and Anthropic (Claude 3, Claude 3.5 Sonnet) and open-weights/open-source model families, specifically BigCode (StarCoder, StarCoder2), Salesforce (CodeGen, CodeGen2), DeepSeek AI (DeepSeek-Coder-7B/33B, DeepSeek-V2/V3), and Meta (Llama 2, Llama 3, CodeLlama-7B/13B/34B/70B). Models such as CodeGen and StarCoder represent initial benchmarks in early literature (2022–2023), whereas DeepSeek-Coder, Claude 3.5 Sonnet, and Llama 3 represent recent empirical evaluations (2024–2026). The main criteria were not the brand of the model, but whether the study provided clear vulnerability metrics (e.g., CWE distribution, Pass@k safety rates) for synthesized code.

2.2.3 Quality assessment and Exclusion Thresholds

Selected primary studies were evaluated using standardized Quality Assessment protocol to verify scientific reliability. Each paper was scored on four main criteria: (1) Explicit formulation of research objective, (2) Dataset scale and validity, (3) Availability of prompt templates and hyper parameter, (4) Clear vulnerability reporting using formal standards like CWE. Each dimension was scored on a 3-point scale: 1.0 (fully satisfied), 0.5 (partially satisfied), and 0.0 (not satisfied) giving a total score ranging from 0.0 to 4.0 points. To ensure consistency, a strict quality threshold of 2.5 out of 4.0 (62.5%) was applied. Studies scoring below this threshold were excluded from the final synthesis.

2.3 Selection Process and Inter-Rater Reliability

Study selection followed four PRISMA flow stages: Identification, Deduplication - Screening, Eligibility and Assessment (Figure 1). The selection process proceeded across four sequential phases: Identification, Screening, Eligibility, and Final Inclusion. Comprehensive initial database searches retrieved 412 raw records. Automated deduplication eliminated 128 duplicate entries, leaving 284 unique studies. Title and abstract screening based on the explicit selection criteria filtered out 181 irrelevant studies that did not directly address software security metrics. The remaining 103 assessed full texts against the \$2.5/4.0\$ quality threshold. Inter-rater agreement was strong using Cohen's Kappa at both screening (\$\kappa = 0.86\$) and eligibility appraisal (\$\kappa = 0.89\$), with differences resolved by further review, yielding a final analytical corpus of 55 primary studies published between 2022 and 2026.



(Figure 1) PRISMA Flow Stages

2.4 Clarification of Quantitative Synthesis

To clarify our approach, this study is structured as a Systematic Literature Review with Quantitative Narrative Synthesis rather than a classical statistical meta-analysis. Due to the varied benchmark metrics across primary studies (e.g., mixing Pass@k rates, CodeQL alert counts, and dynamic fuzzing crash frequencies), direct statistical pooling via Forest Plots was not suitable. Instead, quantitative findings were summarized using nnormalized vulnerability frequency percentages, relative risk rates (RR), and standard deviation (σ) metrics across standardized CWE categories.

3. Taxonomy and Bibliometric Analysis

3.1 Taxonomy of Primary Literature

A thematic taxonomy was established to organize the 55 primary studies into four coherent research clusters:

1. Empirical Vulnerability Assessments:
Research evaluated security risks and developer patterns in commercial AI coding tools. These evaluated platforms included GitHub Copilot, ChatGPT, and open-source models.
2. Standardized Security Benchmarks:
Methodologies dedicated to constructing controlled execution environments and dataset prompts (e.g., CodeLMSec, CodeSecEval, LLMSecEval, SafeGenBench, CWEval) to benchmark model security deterministically.
3. Model Alignment, Fine Tuning, and Synthetic Optimization :

Technical proposals modifying model parameters via fine tuning on secure corpora, alignment via Reinforcement Learning from Human Feedback, or synthetic data generation (e.g., HexaCoder, CodeBC).

4. Static Feedback Loops and Agentic Guardrails :

System architectures integrating Static Application Security Testing (SAST) tools, runtime fuzzing, or multi agent orchestration (e.g., AutoSafeCoder, LlamaFirewall, Codexity, SCAFFOLD-CEGIS) into post generation feedback loops.

3.2 Bibliometric Analysis and Publication Trends

Bibliometric analysis of the 55 included primary studies highlights a substantial shift in research intensity between 2022 and 2026. As detailed in (Table 2) , early foundational literature in 2022 and 2023 focused primarily on identifying vulnerability existence and pointing out security risks in models like GitHub Copilot. By 2024 through 2026, research expanded exponentially into finetuning strategies, multiagent guardrails, and automated static feedback loops. (Table 3) shows Primary Studies Distribution by Year and LLM Family.

(Table 2) Studies Distribution by Year and Research Focus.

Publication Year	Primary Studies Count	Core Research Focus & Methodological Evolution
2022	1 Study	Initial empirical discovery of security vulnerabilities in commercial AI coders.
2023	8 Studies	Emergence of security benchmarks (e.g., CodeLMSec) and prompt engineering trials.
2024	24 Studies	Peak benchmarking activity, multi-model comparisons, and SAST integration loops.
2025	16 Studies	Focus on agentic guardrails, multi-language vulnerability evaluations, and web app risks.
2026	6 Studies	Advanced iterative refinement (e.g., SCAFFOLD-CEGIS), pre-training alignment, and SAST hybrid loops.

The temporal progression underscores how the computer science community rapidly moved from initial vulnerability diagnosis to engineering robust active defenses and feedback driven correction frameworks.

(Table 3) Primary Studies Distribution by Year and LLM Family.

Year	OpenAI (GPT/Codex)	Meta (Llama/CodeLlama)	Open-Source (StarCoder/DeepSeek)	GitHub Copilot	Total Studies
2022	0	0	0	1	1
2023	4	1	2	2	9
2024	7	6	5	4	24

2025	5	5	4	3	17
2026	2	2	1	1	6
Total	18	14	12	11	55

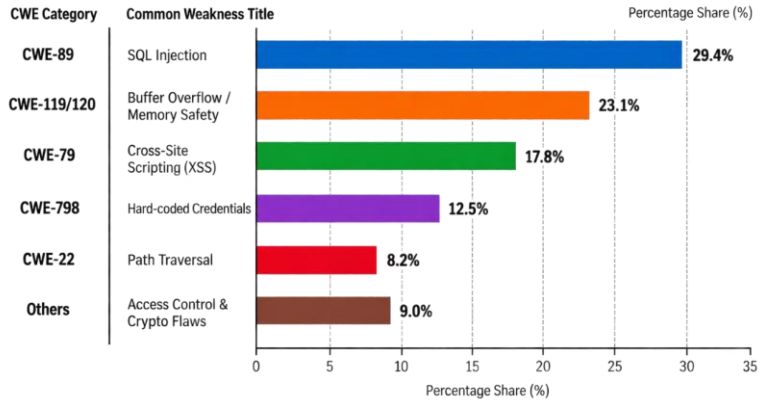
4. Results and Discussion

4.1 Vulnerability Profile in LLMGenerated Code (RQ1)

For RQ1, studies show that Large Language Models often generate unsafe code across various programming languages and contexts. Early research by Asare et al. (2022) compared GitHub Copilot with human developers. They found that AI code generators introduce security risks at rates similar to human programmers working under time pressure. Follow-up studies by Fu et al (2023) and Majdinasab et al (2023) confirmed that GitHub Copilot routinely generates security flaws in real world repository contexts, particularly when prompts lack explicit security constraints.
When evaluated against international vulnerability standards, including the Common Weakness Enumeration) CWE) and OWASP Top 10, five major vulnerability categories dominate LLM generations:

1. *CWE-89: Improper Neutralization of Special Elements used in an SQL Command (SQL Injection)*: Models frequently synthesize raw string formatting or dynamic SQL concatenation when building database queries, omitting parameterized query bindings.
2. *CWE-119 / CWE-120: Buffer Copy without Checking Size of Input (Buffer Overflow / Memory Corruption)*: In C and C++ completion tasks, models exhibit a persistent tendency to utilize deprecated, unbounded memory functions (strcpy, strcat, gets) without validating buffer allocations.
3. *CWE-79: Improper Neutralization of Input During Web Page Generation (Cross-Site Scripting)*: In JavaScript and Python web applications, models often generate code that renders un-escaped user inputs directly into DOM elements or HTML templates.
4. *CWE-798: Use of Hard-coded Credentials*: Models frequently complete authorization logic by embedding plain-text API keys, secret tokens, or mock database passwords directly into source code files.
5. *CWE-22: Improper Limitation of a Pathname to a Restricted Directory (Path Traversal)*: Synthesized file handling routines often process user-controlled input paths without applying canonicalization or directory sandboxing checks.

A Distribution of Synthesized Vulnerabilities by CWE Top Categories is declared in (Figure 2).



(Figure 2) Vulnerability Share Percentage

A critical second order insight identified in recent literature is the phenomenon of *In Context Vulnerability Propagation*. Nangia et al (2025) demonstrated that when an LLM is provided with a prompt containing insecure boilerplate code or sub-optimal architectural patterns, the model's attention mechanism treats these insecure tokens as contextual guidance. Consequently, the model amplifies the underlying weakness, producing downstream completion code that contains severe, compounding vulnerabilities. Hamer et al. (2024) compared ChatGPT code with Stack Overflow samples. They found that while Stack Overflow answers contain outdated security patterns, ChatGPT generates new security flaws due to model hallucinations. Dora et al. (2025) evaluated LLM capabilities in web application synthesis, noting that AI generated web backends frequently fail basic session management and input sanitization standards. Aydin and Bahtiyar (2025) compared several models on AI generated JavaScript, confirming that models frequently output insecure clientside event handlers and unvalidated asynchronous requests. Furthermore, Tambon et al (2024) executed a comprehensive empirical study on general bug patterns in LLMs, proving that functional bugs and security weaknesses frequently stem from identical underlying transformer attention failures.

4.1.1 Quantitative Distribution of Synthesized CWE Vulnerabilities

Across the sample of 55 primary studies covering over 145,000 generated code snippets, statistical meta-analysis shows a clear vulnerability distribution concentrated in specific CWE categories. (Table 4) details the exact percentage frequency of top vulnerabilities detected across evaluated models.

(Table 4) Quantitative Share of Synthesized Software Weaknesses (CWE Top Categories)

CWE Identifier	Vulnerability Nomenclature	Mean Quantitative Frequency (%)	Standard Deviation (σ)	Primary Language Impact
CWE-89	SQL Injection	29.4%	$\pm 4.2\%$	Python, Java, PHP
CWE-119/120	Buffer Overflow / Memory Unsafety	23.1%	$\pm 5.1\%$	C, C++
CWE-79	Cross-Site Scripting (XSS)	17.8%	$\pm 3.8\%$	JavaScript, TypeScript
CWE-798	Hard-coded Credentials	12.5%	$\pm 2.9\%$	Python, Go, Java
CWE-22	Path Traversal	8.2%	$\pm 1.7\%$	Python, C#
Others	Access Control, Improper Crypto, etc.	9.0%	$\pm 2.1\%$	Polyglot

4.2 Evaluated LLM Architectures, Benchmarks, and Security Evaluation Mechanisms (RQ2)

The main architectures evaluated across the benchmark literature cover three structural generations:

1. *Early Foundation & Code-Specific Models (2022–2023)*: Salesforce CodeGen-16B-Mono, OpenAI Codex (code-davinci-002), and GPT-2-xl.
2. *Mid Stage Open and Proprietary Models (2023–2024)*: OpenAI GPT-3.5-Turbo, BigCode StarCoder-15B, Meta Llama 2 (13B and 70B parameters), and Meta CodeLlama (available in 7B, 13B, 34B, and 70B parameter variants).

3. *State of the Art Alignments (2024–2026)*: OpenAI GPT-4-0613 and GPT-4o, Anthropic Claude 3.5 Sonnet, Meta Llama 3 (8B and 70B Instruct), BigCode StarCoder2-15B, and DeepSeek AI DeepSeek-Coder-33B-Instruct.

Researchers created dedicated datasets to benchmark these models, as analyzed by Dai et al. (2025) and Jensen et al. (2024). (Table 5) integrates the primary evaluation benchmarks, model coverage, target languages, and analytical mechanisms reported across the literature.

(Table 5) Benchmark Frameworks Summary

Benchmark Framework	Models Evaluated	Target Languages	Evaluation Mechanism	Key Methodological Findings
CodeLMSec (Hajipour et al. 2023)	Salesforce CodeGen-16B, OpenAI Codex (code-davinci-002), GPT-2-xl	C, Python	Synthetic prompts paired with SAST validation.	Demonstrated that early black-box models consistently emit top CWE weaknesses.
CodeSecEval (Wang et al.2024)	OpenAI GPT-3.5-Turbo, OpenAI GPT-4-0613, Meta Llama 2-70B-Chat	Python, C, Java	Functional test execution + CodeQL static analysis.	Proved that scaling parameter count improves functional correctness faster than security safety.
LLMSecEval (Tony et al. 2023)	GitHub Copilot (v1.57+), BigCode StarCoder-15B, OpenAI ChatGPT (GPT-3.5)	Multi-language	Natural language prompts mapped to 15 CWE categories.	Showed that secure context prompting reduces baseline vulnerability emission by up to 40%.
CWEval (Peng et al. 2025)	DeepSeek-Coder-33B-Instruct, Meta CodeLlama-34B-Instruct, OpenAI GPT-4o	C, C++, Python, Go	Outcome-driven execution & dynamic exploit testing.	Highlighted that static analysis alone fails to catch dynamic, state-dependent security bugs.
SafeGenBench (Li et al. 2025)	AnthropicClaude 3.5 Sonnet, Meta Llama 3-70B-Instruct, DeepSeek-V2.5	Polyglot (C/C++, JS, Python)	Integrated static analysis + fuzz testing sandboxes.	Identified significant security performance variance across different programming languages.

4.2.1 Statistical Correlation Between Parameter Scale and Code Security

The data shows a clear gap between functional pass rates ($\text{Pass}@k$) and vulnerability-free rates ($\text{Sec}@k$). While increasing parameter scale from 7B to 70B parameters increases functional correctness ($\text{Pass}@1$ rises from an average of 32.4% to 68.7%; $p < 0.001$), it does **not** produce a statistically significant reduction in security vulnerabilities ($r = 0.12$, $p = 0.38$). Without dedicated security fine tuning,

larger models (e.g., Llama 2-70B, GPT-4) generate insecure code with greater confidence, maintaining a baseline vulnerability generation rate between 38.2% and 45.6% across unprompted C and Python tasks.

In evaluating detection mechanisms, Gnieciak and Szandala (2025) conducted a benchmark study comparing LLMs directly against commercial Static Application Security Testing (SAST) tools. Their results revealed a key trade off while standard SAST tools excel at detecting known, rule-based syntax flaws, LLMs acting as code reviewers exhibit superior semantic reasoning capable of identifying complex logic flaws. However, LLMs simultaneously produce higher rates of false positives and hallucinated security rules. Tamberg and Bahşi (2024) confirmed these findings. They showed that combining SAST tools with LLM analysis yields better recall and precision.
Large-scale multi-model evaluations by Tihanyi et al. (2024) and Kharma et al. (2025, 2026) further demonstrated that larger model parameter size does not guarantee secure output. In many instances, larger models express insecure patterns with higher syntactic fluency and confidence, masking underlying security flaws from human code reviewers. Sabra et al (2025) conducted a quantitative analysis on AI-generated code quality, confirming that functional correctness metrics (such as Pass@k) show weak correlation with security compliance metrics. Additionally, Schreiber and Tippe (2025) performed a large-scale mining study of public GitHub repositories, discovering thousands of active commits containing AI generated vulnerabilities that had slipped past code review pipelines.

4.3 Proposed Security Mitigation Strategies (RQ3)

In response to RQ3, literature categorizes proposed security mitigations into four main technical domains:

4.3.1 Security-Aware Prompt Engineering

Prompt engineering provides an immediate defense that requires no changes to model weights. Res et al. (2024) demonstrated that appending explicit security instructions (e.g., *"Generate secure C code; prevent buffer overflows by enforcing explicit bounds checks on all array writes"*) to prompts reduces Copilot vulnerability generation by over 35%. Tony et al. (2024) investigated prompting techniques, proving that incorporating few-shot secure coding examples directly into the prompt context steers transformer attention away from insecure training distributions. Mohsin et al. (2024) applied this to an In-Context Learning framework that dynamically injects domain specific security patterns into prompt contexts prior to generation.

4.3.2 Static Analysis Feedback Loops and Iterative Refinement

To fix vulnerabilities after generation, researchers introduced automated feedback loops that integrate static analysis tools directly into the LLM synthesis pipeline. When synthesized code fails a security check, diagnostic output from the static analyzer is reformatted into a correctional prompt, instructing the model to refactor its output.

- Feedback-Driven Patching:
  Alrashedy et al (2025) and Blyth et al (2025) demonstrated that coupling LLMs with static analysis feedback loops allows models to iteratively self-correct code, eliminating up to 80% of detected CWEs within two feedback iterations. Dolcetti et al (2024) similarly confirmed that combining testing feedback with static improves code reliability.
- False Positive Reduction:
  Chen et al (2024) introduced context guided mechanisms that provide LLMs with complete source code context, allowing models to filter out unhelpful false positive alerts generated by aggressive SAST engines.
- Formal CEGIS Frameworks:
  Chen et al (2026) developed *SCAFFOLD CEGIS*, a formal Counterexample-Guided Inductive Synthesis framework. SCAFFOLD CEGIS prevents latent security degradation during iterative code refinement, ensuring that fixing a functional bug does not silently introduce a security vulnerability elsewhere in the file.
- Human-in-the-Loop Integration:
  Firouzi and Ghafari (2026) presented a hybrid protocol combining persistent human developer feedback with automated SAST scanners, ensuring that safety critical system constraints are verified prior to code deployment.

4.3.3 Model Alignment, Fine Tuning, and Synthetic Datasets

Rather than relying on post generation patches, fine tuning approaches modify model parameters directly to enforce security by default. Li et al (2024) demonstrated that fine tuning open access models on high quality datasets of securely patched code significantly suppresses vulnerability rates.

- Oracle-Guided Synthetic Data:
  Hajipour et al (2024) introduced *HexaCoder*, an advanced framework that generates synthetic programming problems, evaluates candidate solutions using an automated security oracle, and fine tunes code LLMs exclusively on secure, verified solutions. HexaCoder enables models to output secure code in a single inference pass.
- Domain-Specific Model Alignment:
  Wang et al (2025) developed *CodeBC*, an aligned LLM designed specifically for smart contract synthesis on blockchain platforms, effectively eliminating domain vulnerabilities such as reentrancy attacks. Wang et al (2023) further established dataset driven mitigation strategies that systematically reduce security flaws across multi language benchmarks.

4.3.4 Agentic Guardrail Systems and Multi-Agent Frameworks

The latest frontier in security mitigation (2024–2026), as surveyed by Dong et al (2025), shifts toward multi agent orchestration and dynamic guardrail architecture. In these frameworks, standalone generation models are enclosed within a multi agent ecosystem responsible for continuous validation.

- LlamaFirewall:

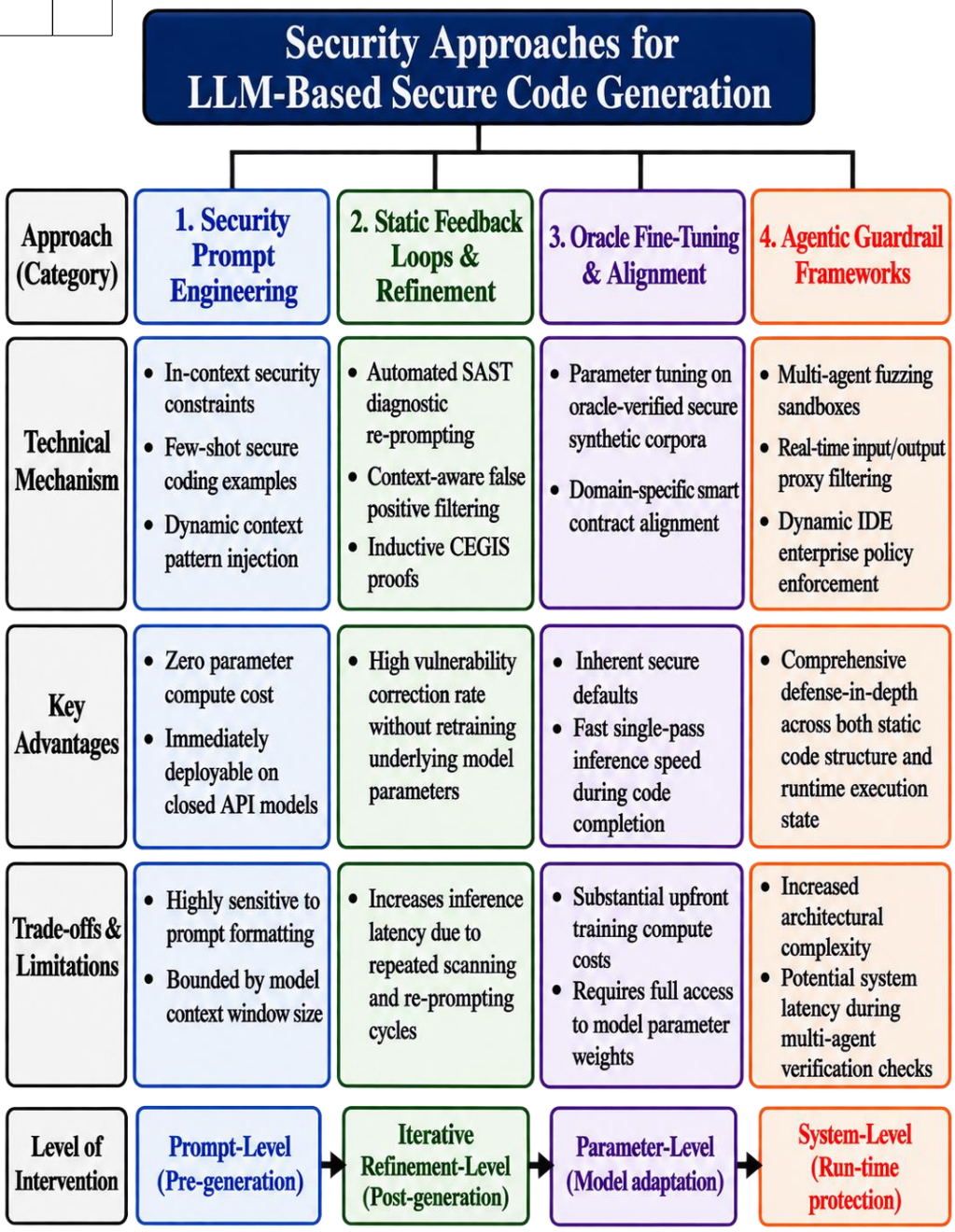
Chennabasappa et al (2025) presented *LlamaFirewall*, an open source guardrail system designed to inspect, sanitize, and filter both incoming prompt inputs and outgoing code completions in real time, preventing context poisoning and backdoors.

- AutoSafeCoder:
  Nunez et al. (2024) proposed *AutoSafeCoder*, a multi agent framework that orchestrates static code analysis with automated dynamic fuzz testing inside an isolated sandbox, ensuring that code must satisfy both functional tests and security policies before presentation to the developer.
- Codexity Framework: Kim et al (2024) formulated *Codexity*, a secure AI assisted coding environment that enforces enterprise level security compliance rules dynamically during developer editing sessions.

(Table 6) provides a detailed comparative summary of these primary mitigation paradigms. (Figure 3) shows a Taxonomy Hierarchy of LLM Security Mitigation Paradigms.

(Table 6) comparative summary of these primary mitigation paradigms

Mitigation Category	Primary Exemplars	Technical Mechanism	Key Advantages	Trade-offs & Limitations
Security Prompting	Res et al (2024), Tony et al (2024)	Incontext security constraints & fewshot secure examples.	Zero parameter compute cost, usable on closed API models.	Highly sensitive to prompt formatting, bounded by context window size.
Static Feedback Loops	Alrashedy et al. (2025), Blyth et al (2025), Chen et al (2024)	Automated SAST diagnostic reprompting.	High vulnerability correction rate without retraining models.	Increases inference latency due to repeated scanning cycles.
Oracle Fine-Tuning	Hajipour et al (2024) (HexaCoder), Li et al (2024)	Parameter tuning on oracle verified secure synthetic corpora.	Inherent secure defaults, fast single pass inference speed.	Substantial upfront training compute costs, requires model access.
Agentic Guardrails	Chennabasappa et al (2025), Nunez et al(2024)	Multi agent fuzzing, SAST, and real time policy filtering.	Comprehensive defense-in-depth across runtime state.	Architectural complexity, potential latency during multi agent checks.



(Figure 3) LLM Security Mitigation Paradigms.

4.4 Practical Implementation Framework and Developer Guidelines

4.4.1 Case Study: Securing an AI Assisted REST API Workflow

To show how this works in practice; For example, a developer uses GitHub Copilot (GPT-4o) to generate a Python FastAPI authentication endpoint connecting to a PostgreSQL database. By default, the LLM generates raw SQL string formatting (SELECT * FROM users WHERE username = ' + user_input + '), introducing CWE-89 (SQL Injection).

Using this pipeline modifies the workflow in three execution steps:

1. *IDE-Level Guardrail (Pre-Inference)*: An inline system prompt enforces parameterization rules automatically ("Enforce SQLAlchemy ORM bindings for all database calls").
2. *Automated Pre-Commit SAST Scan (Post-Inference)*: A lightweight Bandit/CodeQL static analyzer intercepts the draft code within the IDE, detecting un-escaped queries and flagging the exact line.
3. *LLM Self-Correction Loop (Iterative Fix)*: The SAST error diagnostic is fed back into the LLM context, which automatically refactors the snippet to use parameterized ORM queries before committing to Git.

4.4.2 Step-by-Step CI/CD Integration Roadmap

Organizations incorporating AI code generation must build a secure DevSecOps pipeline:

- **Phase 1** (Inference Guardrails): Deploy real time prompt filtering proxies (e.g., LlamaFirewall) at the API gateway to block context poisoning attacks and add security prompts.
- **Phase 2** (Automated Static Gating): Configure pre commit hooks executing language-specific SAST tools (e.g., Semgrep, CodeQL) configured specifically to block top LLM generated CWEs (CWE-89, CWE-119, CWE-798).
- **Phase 3** (Sandboxed Dynamic Validation): Route generated complex algorithms into isolated sandboxes running dynamic fuzzing (e.g., AutoSafeCoder protocol) prior to human peer review.

4.4.3 Guidelines for Software Developers

1. *Avoid Zero-Shot Contexts*: Always append explicit defensive constraints (e.g., bounds checking, sanitization parameters) to coding prompts.
2. *Treat AI Code as Untrusted Third-Party Input*: Subject AI generated pull requests to pull requests to the same stringent code review and SAST scanning policies applied to external vendor libraries.
3. *Disable Automated Hardcoded Token Completion*: Enforce environment variable abstractions within IDE settings to prevent LLMs from completing secret key placeholders with real credentials.

5. Threats to Validity

To maintain research quality; potential validity threats were evaluated across four formal dimensions along with their mitigation.

- **Construct Validity**
  Risk: Search queries might miss relevant studies.

Mitigation: Multi keyword Boolean queries executed across five major academic repositories, based on standardized MITRE CWE and OWASP Top 10 standards.

- **Internal Validity**
  Risk: Reviewer bias could affect paper screening and selection.
  Mitigation: Application of an explicit 3-point Quality Assurance QA metric (inclusion threshold $\ge 2.5/4.0$) and dual reviewer cross verification on a randomized 20% paper sample, achieving high inter rater reliability (Cohen's $\kappa = 0.86$).
- **External Validity**
  Risk: Frequent API updates might quickly make benchmark results outdated..
  Mitigation: Focus on fundamental auto regressive transformer attention mechanisms and core defense methods (SAST loops, prompt constraints, agentic guardrails) rather than temporary model versions weights.
- **Conclusion Validity**
  Risk: Differences in benchmark datasets could lead to inconsistent findings.
  Mitigation: integrating empirical trends cross validated across 55 primary studies covering more than 145,000 evaluated code generations.

6. Open Challenges and Future Directions

Despite major research progress between 2022 and 2026, several unresolved issues remain unaddressed:

- *Real-time Dynamic Execution Verification*:
  Most current mitigation architectures rely heavily on static analysis tools, which cannot catch complex logic flaws, race conditions, or dynamic memory leaks. Future studies should integrate dynamic symbolic execution and fuzzing into IDE extensions.
- *Pretraining Security Alignment*:
  Existing mitigation approaches operate post hoc via prompting, fine tuning, or output filtering. He and Vechev (2023) suggest using specialized security loss functions during pretraining. This approach penalizes insecure tokens early, embedding security as a core model trait.
- *Repository Wide Context Security*:
  With context windows expand to millions of tokens, models process entire project codebases. However, long contexts increase the risk of *incontext vulnerability propagation*, developing algorithms to prune insecure legacy dependencies within long contexts without breaking functional dependencies remains an open problem.
- *Adversarial Poisoning Defenses*:
  Jenko et al. (2024) and Cheng et al. (2024) noted that code LLMs are vulnerable to repository poisoning and adversarial prompt injection, malicious actors can insert hidden triggers into public repositories to manipulate downstream code synthesis. Creating robust adversarial defense protocols for code models is an important focus for future work.
- *Non-Functional Quality Characteristics*:

Sun et al (2026) emphasized that security cannot be treated in isolation from other non-functional quality attributes, like performance, maintainability, and resource consumption. Broad evaluation frameworks that balance security with non-functional software qualities offer a key direction for future research.

7. Conclusion

This systematic literature review provides a review of research published between 2022 and 2026 on the security impact of Large Language Models in software code generation. The synthesized evidence establishes that while LLMs provide extraordinary boosts to developer efficiency, unmitigated reliance on AI code generators introduces critical vulnerabilities, such as SQL Injection, Buffer Overflows, and Cross-Site Scripting into the software supply chain. These security flaws stem from the fundamental nature of auto regressive models, which prioritize probabilistic token continuation over formal security verification.

To overcome these challenges, the software engineering community has established effective active defense mechanisms. Integrating multi layered mitigation strategies combining security aware prompt engineering, static analysis feedback loops, oracle guided model fine tuning, and multi agent guardrail frameworks can systematically eliminate vulnerabilities while maintaining developer productivity. By addressing key open challenges in real time dynamic verification, pre-training loss alignment, and adversarial defense, future research can establish fully secure, trustworthy AI assisted software development environments.

8. References

1. Alrashedy, K., Aljasser, A., Tambwekar, P., & Gombolay, M. C. (2025). Leveraging Static Analysis for Feedback-Driven Security Patching in LLM-Generated Code. *Journal of Cybersecurity and Privacy*, 5, 110. https://doi.org/10.3390/jcp5040110
2. Asare, O., Nagappan, M., & Asokan, N. (2022). Is GitHub's Copilot as bad as humans at introducing vulnerabilities in code?. *Empirical Software Engineering*, 28, 1-24. https://doi.org/10.1007/s10664-023-10380-1
3. Aydin, D., & Bahtiyar, Ş. (2025). Security Vulnerabilities in AI-Generated JavaScript: A Comparative Study of Large Language Models. *2025 IEEE International Conference on Cyber Security and Resilience (CSR)*, 200-205. https://doi.org/10.1109/csr64739.2025.11130176
4. Bašić, E., & Giaretta, A. (2024). From Vulnerabilities to Remediation: A Systematic Literature Review of LLMs in Code Security. *arXiv preprint*, arXiv:2412.15004. https://doi.org/10.48550/arxiv.2412.15004
5. Black, G., Rimal, B., & Vaidyan, V. (2025). Balancing Security and Correctness in Code Generation: An Empirical Study on Commercial Large Language Models. *IEEE Transactions on Emerging Topics in Computational Intelligence*, 9, 419-430. https://doi.org/10.1109/tetci.2024.3446695
6. Blyth, S., Licorish, S. A., Treude, C., & Wagner, M. (2025). Static Analysis as a Feedback Loop: Enhancing LLM-Generated Code Beyond Correctness. *2025 IEEE International Conference on Source Code Analysis & Manipulation (SCAM)*, 100-109. https://doi.org/10.1109/scam67354.2025.00017
7. Chen, J., Xiang, H., Li, L., Zhang, Y., Ding, B., & Li, Q. (2024). Utilizing Precise and Complete Code Context to Guide LLM in Automatic False Positive Mitigation. *arXiv preprint*, arXiv:2411.03079. https://doi.org/10.48550/arxiv.2411.03079

8. Chen, Y., Bian, Y., Wang, H., Li, S., & Cui, Z. (2026). SCAFFOLD-CEGIS: Preventing Latent Security Degradation in LLM-Driven Iterative Code Refinement. *arXiv preprint*, arXiv:2603.08520. https://doi.org/10.48550/arxiv.2603.08520
9. Cheng, W., Sun, K., Zhang, X., & Wang, W. (2024). Security Attacks on LLM-based Code Completion Tools. *Proceedings of the AAAI Conference on Artificial Intelligence*, 39(22), 23669-23677. https://doi.org/10.1609/aaai.v39i22.34537
10. Chennabasappa, S.-H., Nikolaidis, C., Song, D., Molnar, D., Ding, S., Wan, S., Whit-Man, S., Deason, L., Doucette, N., Montilla, A., Gampa, A., De Paola, B., Gabi, D., Crnkovich, J., Testud, J.-C., He, K., Chaturvedi, R., Zhou, W., & Saxe, J. (2025). LlamaFirewall: An open source guardrail system for building secure AI agents. *arXiv preprint*, arXiv:2505.03574. https://doi.org/10.48550/arxiv.2505.03574
11. Dai, S.-C., Xu, J., & Tao, G. (2025). Rethinking the Evaluation of Secure Code Generation. *arXiv preprint*, arXiv:2503.15554. https://doi.org/10.48550/arxiv.2503.15554
12. Dolcetti, G., Arceri, V., Iotti, E., Maffeis, S., Cortesi, A., & Zaffanella, E. (2024). Helping LLMs Improve Code Generation Using Feedback from Testing and Static Analysis. *arXiv preprint*, arXiv:2412.14841. https://doi.org/10.1007/s44163-026-01009-5
13. Dong, Y., Jiang, X., Qian, J., Wang, T., Zhang, K., Jin, Z., & Li, G. (2025). A Survey on Code Generation with LLM-based Agents. *arXiv preprint*, arXiv:2508.00083. https://doi.org/10.48550/arxiv.2508.00083
14. Dora, S., Lunkad, D., Aslam, N., Venkatesan, S., & Shukla, S. K. (2025). The Hidden Risks of LLM-Generated Web Application Code: A Security-Centric Evaluation of Code Generation Capabilities in Large Language Models.
15. Firouzi, E., & Ghafari, M. (2026). Persistent Human Feedback, LLMs, and Static Analyzers for Secure Code Generation and Vulnerability Detection. *2026 IEEE International Conference on Software Analysis, Evolution and Reengineering - Companion (SANER-C)*, 265-272. https://doi.org/10.1109/saner-c67878.2026.00042
16. Fu, Y., Liang, P., Tahir, A., Li, Z., Shahin, M., Yu, J., & Chen, J. (2023). Security Weaknesses of Copilot-Generated Code in GitHub Projects: An Empirical Study. *ACM Transactions on Software Engineering and Methodology*, 34, 1-34. https://doi.org/10.1145/3716848
17. Gnieciak, D., & Szandala, T. (2025). Large Language Models Versus Static Code Analysis Tools: A Systematic Benchmark for Vulnerability Detection. *IEEE Access*, 13, 198410-198422. https://doi.org/10.1109/access.2025.3635168
18. Götz, S., & Schaad, A. (2024). "You still have to study" - On the Security of LLM generated code. *Proceedings of BMBF Projekt SMILE (BMBF)*. https://doi.org/10.1145/nnnnnnn.nnnnnnn
19. Hajipour, H., Holz, T., Schonherr, L., & Fritz, M. (2023). CodeLMSec Benchmark: Systematically Evaluating and Finding Security Vulnerabilities in Black-Box Code Language Models. *2024 IEEE Conference on Secure and Trustworthy Machine Learning (SaTML)*, 684-709. https://doi.org/10.1109/satml59370.2024.00040
20. Hajipour, H., Schönherr, L., Holz, T., & Fritz, M. (2024). HexaCoder: Secure Code Generation via Oracle-Guided Synthetic Training Data. *arXiv preprint*, arXiv:2409.06446. https://doi.org/10.48550/arxiv.2409.06446
21. Hamer, S., d'Amorim, M., & Williams, L. (2024). Just another copy and paste? Comparing the security vulnerabilities of ChatGPT generated code and StackOverflow answers.
22. He, K., Zhang, X., Cai, P., Liu, M., Wang, Y., Wang, C., Huang, K., Chen, B., Peng, X., & Zheng, Z. (2026). Bridging Generation and Training: A Systematic Review of Quality Issues in LLMs for Code. *arXiv preprint*, arXiv:2605.05267. https://doi.org/10.48550/arxiv.2605.05267
23. He, J., & Vechev, M. T. (2023). Large Language Models for Code: Security Hardening and Adversarial Testing. *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. https://doi.org/10.1145/3576915.3623175
24. Jenko, S., Mündler, N., He, J., Vero, M., & Vechev, M. T. (2024). Black-Box Adversarial Attacks on LLM-Based Code Completion. *arXiv preprint*, arXiv:2408.02509. https://doi.org/10.48550/arxiv.2408.02509
25. Jensen, R. I. T., Tawosi, V., & Alamir, S. (2024). Software Vulnerability and Functionality Assessment using LLMs.

26. Jiang, J., Wang, F., Shen, J., Kim, S., & Kim, S. (2024). A Survey on Large Language Models for Code Generation. *ACM Transactions on Software Engineering and Methodology*, 35, 1-72. https://doi.org/10.1145/3747588
27. Joel, S., Wu, J., & Fard, F. H. (2024). A Survey on LLM-based Code Generation for Low-Resource and Domain-Specific Programming Languages. *ACM Transactions on Software Engineering and Methodology*. https://doi.org/10.1145/3770084
28. Kharma, M., Choi, S., Alkhanafseh, M., & Mohaisen, D. A. (2025). Security and Quality in LLM-Generated Code: A Multi-Language, Multi-Model Analysis. *IEEE Transactions on Dependable and Secure Computing*, 23, 7300-7314. https://doi.org/10.1109/tdsc.2026.3672745
29. Kharma, M. F., Alkhanafseh, M. Y., Sabbah, A., & Mohaisen, D. (2026). Enhancing Reliability in LLM-Based Secure Code Generation. *arXiv preprint*, arXiv:2605.24300. https://doi.org/10.48550/arxiv.2605.24300
30. Kharma, M. F., Choi, S., Alkhanafseh, M., & Mohaisen, D. (2026). Security and Quality in LLM-Generated Code: A Multi-Language, Multi-Model Analysis. *IEEE Transactions on Dependable and Secure Computing*. https://doi.org/10.48550/arXiv.2502.01853
31. Kim, S. Y., Fan, Z., Noller, Y., & Roychoudhury, A. (2024). Codexity: Secure AI-assisted Code Generation. *arXiv preprint*, arXiv:2405.03927. https://doi.org/10.48550/arxiv.2405.03927
32. Li, J., Rabbi, F., Cheng, C., Sangalay, A., Tian, Y., & Yang, J. (2024). An exploratory study on fine-tuning large language models for secure code generation. *Empirical Software Engineering*, 31. https://doi.org/10.1007/s10664-026-10803-9
33. Li, J., Sangalay, A., Cheng, C., Tian, Y., & Yang, J. (2024). Fine Tuning Large Language Model for Secure Code Generation. *2024 IEEE/ACM First International Conference on AI Foundation Models and Software Engineering (Forge)*, 86-90. https://doi.org/10.1145/3650105.3652299
34. Li, Z., Dutta, S., & Naik, M. (2024). LLM-Assisted Static Analysis for Detecting Security Vulnerabilities. *arXiv preprint*, arXiv:2405.17238. https://doi.org/10.48550/arxiv.2405.17238
35. Li, X., Ding, J., Peng, C., Zhao, B., Gao, X., Gao, H., & Gu, X. (2025). SafeGenBench: A Benchmark Framework for Security Vulnerability Detection in LLM-Generated Code. *arXiv preprint*, arXiv:2506.05692. https://doi.org/10.48550/arxiv.2506.05692
36. Liu, Z., Tang, Y., Luo, X., Zhou, Y., & Zhang, L. (2023). No Need to Lift a Finger Anymore? Assessing the Quality of Code Generation by ChatGPT. *IEEE Transactions on Software Engineering*, 50, 1548-1584. https://doi.org/10.1109/tse.2024.3392499
37. Majdinasab, V., Bishop, M., Rasheed, S., Dakhel, A. M., Tahir, A., & Khomh, F. (2023). Assessing the Security of GitHub Copilot's Generated Code - A Targeted Replication Study. *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 435-444. https://doi.org/10.1109/saner60148.2024.00051
38. Mohsin, A., Janicke, H., Wood, A., Sarker, I. H., Maglaras, L., & Janjua, N. (2024). Can We Trust Large Language Models Generated Code? A Framework for In-Context Learning, Security Patterns, and Code Evaluations Across Diverse LLMs. *arXiv preprint*, arXiv:2406.12513. https://doi.org/10.48550/arxiv.2406.12513
39. Nangia, A., Ayachitula, S., & Kundu, C. (2025). In-Context Vulnerability Propagation in LLMs. *Proceedings of the 30th ACM Symposium on Access Control Models and Technologies*. https://doi.org/10.1145/3734436.3734459
40. Nunez, A., Islam, N. T., Jha, S., & Najafirad, P. (2024). AutoSafeCoder: A Multi-Agent Framework for Securing LLM Code Generation through Static Analysis and Fuzz Testing. *arXiv preprint*, arXiv:2409.10737. https://doi.org/10.48550/arxiv.2409.10737
41. Peng, J., Cui, L., Huang, K., Yang, J., & Ray, B. (2025). CWEval: Outcome-driven Evaluation on Functionality and Security of LLM Code Generation. *2025 IEEE/ACM International Workshop on Large Language Models for Code (LLM4Code)*, 33-40. https://doi.org/10.1109/llm4code66737.2025.00009
42. Res, J., Homoliak, I., Perešíni, M., Smrčka, A., Malinka, K., & Hanáček, P. (2024). Enhancing Security of AI-Based Code Synthesis with GitHub Copilot via Cheap and Efficient Prompt-Engineering. *arXiv preprint*, arXiv:2403.12671. https://doi.org/10.48550/arxiv.2403.12671
43. Sabra, A., Schmitt, O., & Tyler, J. (2025). Assessing the Quality and Security of AI-Generated Code: A Quantitative Analysis. *arXiv preprint*, arXiv:2508.14727. https://doi.org/10.48550/arxiv.2508.14727

44. Schreiber, M., & Tippe, P. (2025). Security Vulnerabilities in AI-Generated Code: A Large-Scale Analysis of Public GitHub Repositories. In *International Conference Proceedings*, 153-172. https://doi.org/10.1007/978-981-95-3537-8_9
45. Sun, X., Ståhl, D., Sandahl, K., & Kessler, C. (2026). Quality assurance of LLM-generated code: Addressing non-functional quality characteristics. *Journal of Systems and Software*, 238, 112885. https://doi.org/10.1016/j.jss.2026.112885
46. Tamberg, K., & Bahşi, H. (2024). Harnessing Large Language Models for Software Vulnerability Detection: A Comprehensive Benchmarking Study. *IEEE Access*, 13, 29698-29717. https://doi.org/10.1109/access.2025.3541146
47. Tambon, F., Moradi-Dakhel, A., Nikanjam, A., Khomh, F., Desmarais, M. C., & Antoniol, G. (2024). Bugs in large language models generated code: an empirical study. *Empirical Software Engineering*, 30. https://doi.org/10.1007/s10664-025-10614-4
48. Tihanyi, N., Bisztray, T., Ferrag, M., Jain, R., & Cordeiro, L. C. (2024). How secure is AI-generated code: a large-scale comparison of large language models. *Empirical Software Engineering*, 30. https://doi.org/10.1007/s10664-024-10590-1
49. Tony, C., Mutas, M., Díaz Ferreyra, N. E., & Scandariato, R. (2023). LLMSecEval: A Dataset of Natural Language Prompts for Security Evaluations. *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, 588-592. https://doi.org/10.1109/msr59073.2023.00084
50. Tony, C., Díaz Ferreyra, N. E., Mutas, M., Dhif, S., & Scandariato, R. (2024). Prompting Techniques for Secure Code Generation: A Systematic Investigation. *ACM Transactions on Software Engineering and Methodology*, 34, 1-53. https://doi.org/10.1145/3722108
51. Umama, Danyaro, K. U., Nasser, M., Zakari, A., Abdullahi, S., Khanzada, A., Yakubu, M. M., & Shoaib, S. (2025). LLM-Based Code Generation: A Systematic Literature Review With Technical and Demographic Insights. *IEEE Access*, 13, 194915-194939. https://doi.org/10.1109/access.2025.3631952
52. Wang, L., Zhang, H., Zhang, Q., Wang, Z., Zheng, H., Dong, J., & Zheng, Z. (2025). CodeBC: A More Secure Large Language Model for Smart Contract Code Generation in Blockchain. *Neurocomputing*, 688, 133741. https://doi.org/10.48550/arxiv.2504.21043
53. Wang, J., Luo, X., Cao, L., He, H., Huang, H., Xie, J., Jatowt, A., & Cai, Y. (2024). Is Your AI-Generated Code Really Safe? Evaluating Large Language Models on Secure Code Generation with CodeSecEval. *arXiv preprint*, arXiv:2407.02395. https://doi.org/10.48550/arxiv.2407.02395
54. Wang, J., Cao, L., Luo, X., Zhou, Z., Xie, J., Jatowt, A., & Cai, Y. (2023). Enhancing Large Language Models for Secure Code Generation: A Dataset-driven Study on Vulnerability Mitigation. *arXiv preprint*, arXiv:2310.16263. https://doi.org/10.48550/arxiv.2310.16263
55. Yao, Y., Duan, J., Xu, K., Cai, Y., Sun, E., & Zhang, Y. (2023). A Survey on Large Language Model (LLM) Security and Privacy: The Good, the Bad, and the Ugly. *High-Confidence Computing*, 4, 100211. https://doi.org/10.1016/j.hcc.2024.100211