Smarter, but Not Safer: An Empirical Analysis of the Functional-Security Gap in Evolving LLMs

Alfonso Cannavale*, Gilberto Recupito†, Coen De Roover†, Fabio Palomba*, Andrea De Lucia*
*Software Engineering Salerno (SeSa) Lab, University of Salerno, Fisciano, Italy
Email: acannavale@unisa.it, fpalomba@unisa.it, adelucia@unisa.it
†Software Languages Lab, Vrije Universiteit Brussel, Brussel, Belgium
Emails: gilberto.recupito@vub.be, coen.de.roover@vub.be

***Abstract*—Large Language Models (LLMs) have become tools for software development and maintenance, demonstrating impressive capabilities in automating tasks such as code generation and refactoring. However, it remains empirically unclear whether the rapid scaling of these capabilities is accompanied by proportional improvements in security, or whether a discrepancy, defined as the Security–Functionality Gap, persists across model generations. To address this question, we conducted an evolutionary study analyzing the trajectory of 12 state-of-the-art models across the GPT, Llama, and Claude Sonnet lineages. By leveraging an outcome-driven benchmark based on dynamic execution, we quantified the divergence between the models' ability to generate functionally correct code versus secure code over four sequential versions per lineage. Results reveal that, while functional correctness has effectively plateaued ($> 90\%$) across all families, functional-security correctness has evolved more slowly. Specifically, while proprietary models demonstrate marginal improvements, open-weight models exhibit stagnation, and the most advanced models occasionally introduce new regressions due to increased code complexity. These findings suggest that security alignment is not an intrinsic byproduct of model scaling, highlighting the urgent need for external guardrails in AI-assisted maintenance pipelines.**

***Index Terms*—Large Language Models, Software Evolution, Software Security, Empirical Software Engineering.**

I. Introduction

Large Language Models (LLMs) have rapidly transitioned from experimental tools to fundamental components of the software development lifecycle. They are now pervasive in critical software maintenance tasks, including automated refactoring, program repair, and code optimization [1]. Modern Integrated Development Environments (IDEs) increasingly embed these models directly into the workflow, enabling developers to delegate complex reasoning and code modification tasks to AI agents [2]. In selecting these agents, practitioners heavily rely on established leaderboards and benchmark reports, which predominantly emphasize the models' capabilities in logical reasoning and problem-solving [3].

However, these functional improvements may bring about costs in other quality dimensions. The Software Engineering (SE) community increasingly recognizes that pursuing raw coding efficiency may entail significant trade-offs in other software quality attributes. This tension was recently highlighted during the ICSE 2025 panel [4], which emphasized the need to distinguish between the *velocity* of code production and the *overall quality* of the generated artifacts. Indeed, while benchmarks show exceptional performance when using LLMs, recent studies confirmed the criticality of using LLM-generated code, with risk of introducing unsafe code during the development process [5], [6].

Among quality trade-offs, security represents one of the most critical concerns. Early empirical studies on commercial LLMs demonstrate that, despite their apparent proficiency, these models frequently generate insecure and vulnerable code [7], [8]. Following the first wave of ChatGPT adoption, subsequent evidence showed that insecure code suggestions were often incorporated into production systems via pull requests, partly due to developers' overtrust in model outputs [9].

While these studies provide crucial insights, they share a common limitation: they primarily adopt a *cross-sectional perspective*, evaluating different models or a single model version at a specific point in time. Such evaluations are valuable for assessing the immediate risk associated with a specific model version. However, by design, cross-sectional analyses do not consider temporal dynamics, hence failing to capture how model properties evolve over successive releases. As a result, they lack an *evolutionary perspective* [10], which is essential for understanding whether improvements in functional capabilities are accompanied by sustained, stagnant, or even regressive security guarantees over time. Such insights may contribute to the current body of knowledge by enabling a deeper understanding not only of how well LLMs perform at a given time, but also of how their functional and security properties co-evolve over time.

To address this gap, we introduce the notion of the **Security-Functionality Gap** (Δ_{Sec}). While its mathematical formulation is detailed in Section IV, we informally define this gap as the quantitative divergence between a model's *Functional Correctness* (its ability to generate code that successfully passes functional tests) and its *Functional-Security Correctness* (its ability to generate code that is simultaneously functional and successfully passes targeted security tests). By operationalizing this discrepancy, we can precisely measure whether the industry's drive for scaling reasoning capabilities is effectively closing the security gap or widening it.

In this paper, we present an evolutionary empirical study of intra-family concurrent progression between functional correctness and security. We analyze three model lineages (GPT, Llama, Claude Sonnet) across multiple generations, using both

functional benchmarks and security-oriented dynamic evaluation suites. This approach allows us to determine whether the industry's drive for scaling reasoning capabilities is closing the security gap or inadvertently widening it.

Our evolutionary analysis reveals a critical concern. While functional correctness has effectively plateaued ($> 90\%$) across all lineages, functional-security correctness evolves more slowly. Proprietary models (GPT, Claude) demonstrate a statistically significant but practically marginal narrowing of the Security-Functionality Gap ($\beta \approx -0.05$). Conversely, open-weight models (Llama) exhibit stagnation, with no significant improvement in security alignment across generations.

Furthermore, our granular vulnerability analysis uncovers three distinct evolutionary behaviors: (1) *Persistent Blind Spots*, where failure rates for certain flaws (e.g., Log Injection) remain near 100% across all models and generations, highlighting a systemic inability to override insecure training data idioms; (2) *Evolutionary Divergence*, where, for classic web vulnerabilities (e.g., Path Traversal), proprietary models gradually mitigate the risks while open-weights models completely fail to improve; and (3) *Regression Anomalies*, where the most advanced frontier models inadvertently introduce subtle memory safety regressions (e.g., in C/C++ pointers) in their attempt to handle complex logic and low-level optimizations, a phenomenon absent in earlier, more conservative versions.

This paper makes the following contributions:

- A comprehensive evolutionary empirical study that evaluates 12 state-of-the-art LLMs across three lineages (GPT, Llama, Claude Sonnet). We quantify the evolution of the *Security-Functionality Gap* leveraging a dynamic, outcome-driven evaluation framework.
- A granular analysis of how specific Common Weakness Enumerations (CWEs) persist or regress across model generations. This analysis uncovers three distinct common behaviors: *Persistent Blind Spots*, *Evolutionary Divergence*, and *Regression Anomalies* in frontier models.
- An available replication package [11] containing generated code samples, dynamic execution logs, automated parsing scripts, and statistical analysis workbooks.

Structure of the paper. The remainder of this paper is structured as follows. Section II provides background on LLMs in software engineering and discusses related work on security evaluation. Section III formulates our research questions, while Section IV details the empirical study design and methodology. Section V presents the quantitative and qualitative results of our analysis. Section VI discusses the implications of our findings for researchers and practitioners. Section VII outlines the threats to validity. Finally, Section VIII concludes the paper and outlines future research directions.

II. Background & Related Work

In this section, we review the use of LLMs in software engineering, focusing on maintenance, evolution, and security.

A. LLMs in Software Maintenance and Evolution

The integration of LLMs into the software development lifecycle affects software maintenance and evolution practices. Recent surveys [1], [12] highlight that a significant portion of LLM literature in software engineering is dedicated to automating complex maintenance activities.

Models are increasingly deployed as coding assistants to support a wide range of tasks, from automated program repair and bug fixing [13], [14] to code summarization [15] and test generation [16]. Moreover, LLMs have shown promise in automating refactoring tasks. For instance, Liu et al. [17] revealed that LLMs identify refactoring opportunities with high accuracy when guided by specific prompts. Similarly, Cordeiro et al. [18] investigated the refactoring capabilities of LLMs, noting their potential to improve code quality.

However, despite their reported capabilities, the reliability of generated artifacts remains a concern, as models often prioritize functional correctness over other quality attributes. Nunes et al. [6] reported that LLMs such as Llama 3.1 and Copilot often degrade maintainability or introduce new defects when applying real-world fixes. This exposes a critical gap: existing studies largely assess models as static artifacts, overlooking how their capabilities and associated risks evolve across successive generations.

B. Quality Assurance and Security of Generated Code

As LLMs become integral to development workflows, the research community has increasingly focused on Quality Assurance (QA) for AI-generated code. Beyond functional correctness, researchers have investigated various quality dimensions, including the accumulation of bugs [19] and the introduction of structural code quality issues [20]. Among these quality attributes, **security** has emerged as a central concern in current research.

To systematize evaluation, researchers have proposed benchmarks like *LLMSecEval* [21] and *CodeLMSec* [22]. However, these early efforts primarily relied on static analysis tools (e.g., CodeQL) to detect flaws, an approach that is prone to false positives and cannot verify actual runtime behavior. To address this limitation, more recent works have shifted towards outcome-driven evaluation—i.e., assessing models by dynamically executing the generated code against concrete test suites. For instance, Liu et al. [23] demonstrated with *EvalPlus* that standard benchmarks severely overestimate capabilities due to insufficient testing. Building on this dynamic execution for security, Lee et al. [24] introduced *SEC-bench*, a framework for evaluating LLM agents on real-world software-security tasks. It constructs vulnerable repositories, automatically verifies the reproducibility of vulnerabilities within isolated execution environments, and provides patches to assess both Proof-of-Concept generation and vulnerability remediation by LLMs.

These benchmarks enabled subsequent research to raise alarms about the risks posed by AI-generated code. Pearce et al. [25] conducted a large-scale evaluation of code generation tools (e.g., GitHub Copilot), finding that 40% of the generated

snippets were vulnerable, characterizing the risk as developers being "asleep at the keyboard". Khoury et al. [8] confirmed that even models like ChatGPT frequently generated insecure code unless explicitly constrained by the prompt.

Sajadi et al. [26] assessed whether LLMs detect insecure code and provide explanations. Results indicate consistently low vulnerability detection rates, ranging between 12.5% and 40%. However, other studies report mixed findings. Asare et al. [27] observed that GitHub Copilot did not consistently replicate human-introduced vulnerabilities and occasionally suggested corrective fixes. In contrast, Fu et al. [28] reported that 30% of Copilot-generated snippets in public GitHub projects exhibited vulnerabilities. Collectively, these studies provide valuable but fragmented evidence regarding the security properties of LLM-generated code, hence motivating the need for a more comprehensive analysis.

C. Research Gap

Despite recent advances in understanding the security of LLM-generated code, current research lacks a consolidated understanding of how LLMs generate insecure code. Existing studies analyze individual models or specific versions at a single point in time, following a *cross-sectional* approach. As in software evolution [10], such studies capture specific releases but miss a broader *evolutionary* perspective on how security properties change, persist, or regress across successive model generations. Such a perspective complements the existing fragmented evidence by contextualizing model capabilities within a temporal trajectory. Indeed, by tracking successive model generations within the same lineage, it allows (i) distinguishing between *systematic improvements and episodic fluctuations in security performance*, (ii) revealing *regression trends that remain invisible in single-snapshot evaluations*, and (iii) helping *disentangle the effects of general model scaling from targeted security alignment efforts*. Therefore, this evolutionary view supports a dynamic interpretation of security results, providing a principled basis for assessing when improvements are sustained over time and when model updates may introduce new security risks.

The Knowledge Gap. Prior work mainly relies on functional correctness benchmarks and assessments of individual model versions [24]. As a result, existing evidence remains fragmented, offering limited insight into how security properties evolve as models improve in capability, leaving the **Security–Functionality Gap** largely unexplored. This paper advances the state of the art by introducing an evolutionary, lineage-aware empirical study that explicitly traces the concurrent progression of functionality and security across successive LLM generations, enabling the identification of sustained trends, stagnation phases, and security regressions that remain invisible to cross-sectional evaluations.

III. Research Questions

The *goal* of this empirical study is to evaluate the Security-Functionality Gap within major LLM lineages. The *purpose* is to provide empirical evidence regarding whether model scaling and generational updates mitigate security risks at the same pace as they improve coding capabilities. The *perspective* of this study spans multiple stakeholders involved in the development and adoption of LLM-based coding assistants. It mainly targets LLM vendors and model architects, who are responsible for aligning training objectives with security requirements, as well as software maintenance engineers and practitioners who integrate these models into automated development workflows. More generally, the findings are relevant to researchers and tool builders interested in understanding the long-term security implications of deploying evolving LLMs in software engineering pipelines.

Unlike prior studies that evaluate models as static entities, we focus on the interplay and potential misalignment between these two dimensions over time. Given this goal, our study is guided by two research questions:

> **RQ$_1$.** *How does the Security-Functionality Gap evolve across successive generations of LLMs?*

This research question provides a quantitative assessment of the phenomenon. We investigate the evolutionary trajectory of the misalignment between functional correctness and security. By analyzing the trend of this gap across model generations, we aim to determine whether security compliance scales commensurately with reasoning capabilities or whether a systemic divergence emerges as models become more advanced.

> **RQ$_2$.** *What vulnerability classes characterize the Security-Functionality Gap throughout LLM evolution?*

While the previous research question measures the magnitude of the gap, **RQ$_2$** provides a qualitative explanation of its root causes. We identify which specific Common Weakness Enumerations (CWEs) persist despite generational upgrades and which emerge as new regressions in frontier models. With this mapping, we aim to understand the *anatomy* of the gap, and to highlight whether specific CWEs remain resistant to the general scaling effects of modern LLMs.

IV. Study Design

Following the empirical guidelines for using LLMs in software engineering research [29], we designed an evolutionary study to address our research questions. Rather than evaluating a disparate set of standalone models, we analyze multiple distinct LLM lineages across several successive generations. Our experimental protocol is designed to isolate the evolutionary variable—i.e., model updates within the same family—to observe how the *Security-Functionality Gap* shifts over time while keeping the evaluation environment constant. Figure 1 illustrates the overall workflow of our methodology,

detailing the pipeline from subject selection to the dual-oracle evaluation and final analysis.

A. Subject of the Study

The subjects of our study consist of 12 LLMs categorized into three lineages: **GPT** (OpenAI), **Llama** (Meta), and **Claude** (Anthropic). To enable an evolutionary analysis, we adopted an experimental design, selecting four sequential versions for each lineage.

The selection of these specific models and their chronological ordering (detailed in Table I) adheres to three criteria designed to maximize the scientific validity of the comparison:

1) *Relevance*: we selected families that are de facto standards in both industry and research [1], ensuring that our findings on the Security-Functionality Gap have direct implications for current software engineering practices.
2) *Diversity*: the inclusion of proprietary (GPT and Claude) and open-weights models (Llama) allows us to observe whether different development philosophies (closed vs. open) impact the alignment of security properties.
3) *Evolution*: we required at least four distinct, sequential versions per lineage. This granularity is essential for distinguishing between fluctuations in functional correctness and security.

TABLE I: Model Lineages and Evaluation Snapshots

Lineage	Model Version	Snapshot
Claude Sonnet Lineage*	Claude 3.5 Sonnet	2024-10-22
	Claude 3.7 Sonnet	2025-02-19
	Claude 4 Sonnet	2025-05-22
	Claude 4.5 Sonnet	2025-09-29
GPT Lineage*	GPT-4 Turbo	2024-04-09
	GPT-4o	2024-11-20
	GPT-4.1	2025-04-14
	GPT-5.2	2025-12-11
Llama Lineage†	Llama 3 (70B)	2024-04-18
	Llama 3.1 (70B)	2024-07-23
	Llama 3.3 (70B)	2024-12-06
	Llama 4 Scout (17B)	2025-01-28

*Access via API. †Local deployment (Open Weights).

Claude Sonnet Lineage. The Claude Sonnet lineage tracks the evolution of the *Sonnet* class across four consecutive versions (from `Claude 3.5` to `4.5`). We specifically targeted the Sonnet tier as it represents the lineage's balanced offering, optimized for both reasoning capability and computational efficiency. Maintaining a fixed tier across generations eliminates the variance associated with different model scales (e.g., *Haiku* vs. *Opus*) and provides a coherent, intra-tier trajectory of the provider's security alignment.

GPT Lineage. The GPT lineage traces the evolution of industry-standard proprietary models, following a chronological progression from `GPT-4 Turbo` to `GPT-5.2`, including `GPT-4o` and `GPT-4.1` as intermediate iterations. The latest version (`GPT-5.2`) was explicitly configured with the *reasoning effort* parameter set to *none*. This methodological constraint is essential to ensure that security performance is comparable across the entire sequence, preventing inference-time scaling from masking the underlying evolution of the model's weights and alignment.

Llama Lineage. The Llama lineage follows the evolution of the Llama family in an open-weight context. The sequence primarily uses the 70B parameter class (`Llama 3`, `3.1`, and `3.3`) to provide a controlled environment in which model scale remains constant. The lineage concludes with the `Llama 4 Scout` (17B) version. This model was selected over the `Behemoth` (288B) variant due to local computational constraints and `Maverick` (17B) variant due to its unavailability at the time of the running of experiments, while still allowing evaluation of the architectural advancements of the Llama 4 generation relative to its predecessors.

B. Object of the Study

The object of our study is the security quality of code generated by these models. To assess this, we employ the **CWEval** benchmark [30], which comprises 119 security-critical coding tasks covering 31 distinct CWEs.

CWEval was selected over other benchmarks (e.g., LLM-SecEval or CodeLMSec, specified in Section II-B) for two methodological reasons aligned with our study's goal.

First, CWEval implements a **Dual Oracle System**. Each task t is associated with a functional test suite (T_{func}) to verify correctness and a security test suite (T_{sec}) to verify the absence of vulnerabilities. This separation allows us to distinguish between code that simply works (functional success) and code that is secure. This distinction is a prerequisite for quantifying the *Security-Functionality Gap*: without a dual oracle, it would be impossible to identify the deceptive cases in which functionality improves while security regresses.

Second, CWEval utilizes **dynamic oracles** to assess these properties. As noted by Chess and McGraw [31], static analysis is effective at identifying *potential* defects but may not accurately reflect run-time behavior. Conversely, dynamic execution allows for the verification of *exploitable* vulnerabilities [32]. By executing attack payloads against the generated code in a sandboxed environment, we ensure that the measured security gap corresponds to confirmed vulnerabilities rather than theoretical warnings [33].

Each CWEval task is a self-contained function specification with a docstring and an associated test. For a vulnerability such as CWE-112 (HTTP Request/Response Splitting), the model receives a function signature and a natural-language description of the intended behavior, then must generate the function body. The model generates the function body, which is then dynamically validated by the dual oracle.[1]

C. Data Collection

For each of the 119 tasks in the benchmark, and for each of the 12 models, we generated $n = 10$ candidate solutions using

[1]This is a CWEval test for CWE-112: https://github.com/Co1lin/CWEval/blob/main/benchmark/core/c/cwe_113_0_c_test.py

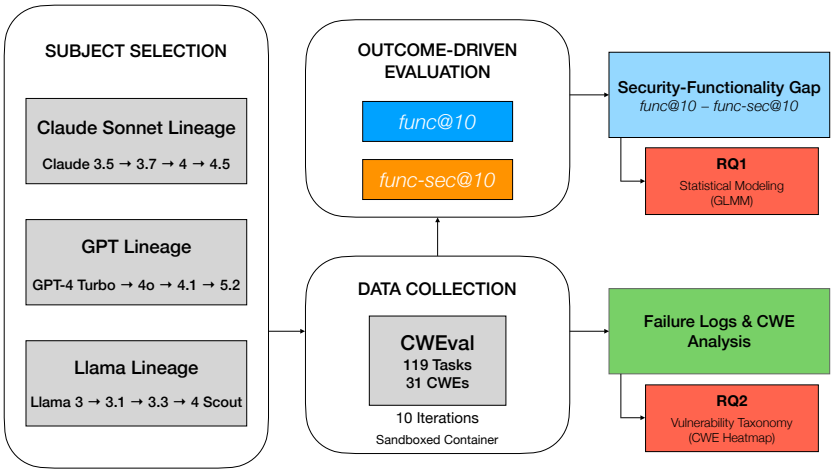


Fig. 1: Overview of the study. The study selects evolutionary lineages to generate samples in a sandboxed environment ($N = 10$, $T = 0.8$). The evaluation splits into a quantitative branch (blue), computing the Security-Functionality Gap (Δ_{Sec}) for statistical modeling (RQ_1), and a qualitative branch (green), analyzing test logs to observe vulnerability patterns (RQ_2).

a sampling temperature of $T = 0.8$. This configuration aligns with prior work on code generation limits [34], allowing us to estimate the model's potential capability ($pass@k$) rather than just its most likely output (greedy decoding).

The inference infrastructure was hybrid. Open-weights models (Llama lineage) were hosted locally using **LM Studio** in server mode, utilizing high-end consumer GPUs to ensure consistent quantization (BF16). Closed-source models were queried via **OpenRouter**[2], a unified API gateway ensuring access to specific model versions without silent updates. All generated code was executed within the official CWEval sandboxed Docker container (`co1lin/cweval`).[3] This environment isolates the host system from potentially malicious code (e.g., infinite loops, fork bombs) while running the functional and security test suites. We collected execution logs and test verdicts for all $119 \times 12 \times 10 = 14,280$ generated samples. The logs and automated parsing artifacts are available in our online appendix [11].

D. Variable Operationalization

To quantify evolutionary trends in code quality across model generations, we operationalize the construct of *secure code generation* into two primary metrics originally defined by the CWEval benchmark [30]. Because LLMs generate code non-deterministically, relying on a simple success percentage often underestimates a model's practical capability. Instead, it is standard practice to measure the probability that *at least one* out of k generated candidates is successful. To estimate this probability without bias from a total pool of n samples, we use the combinatorial $pass@k$ estimator proposed by Chen et al. [34]. Specifically, we evaluate:

- **Functional Correctness** ($func@k$): the average success rate that at least one of k sampled solutions passes the functional test suite T_{func}. Let c_{func} be the number of generated samples (out of n) that pass T_{func}. This metric is calculated as:

$$func@k = \mathbb{E}\left[1 - \frac{\binom{n-c_{func}}{k}}{\binom{n}{k}}\right] \tag{1}$$

  This metric serves as a proxy for the model's general programming capability, independent of security considerations.
- **Functional-Secure Correctness** ($func\text{-}sec@k$): the average success rate that at least one of k sampled solutions passes *both* the functional test suite T_{func} and the security test suite T_{sec}. Let $c_{func\text{-}sec}$ be the number of samples satisfying both oracles simultaneously. The metric is computed as:

$$func\text{-}sec@k = \mathbb{E}\left[1 - \frac{\binom{n-c_{func\text{-}sec}}{k}}{\binom{n}{k}}\right] \tag{2}$$

  This metric captures the model's ability to produce code that is simultaneously correct and secure.

The research object of this study is the relationship between these two dimensions. Motivated by a misalignment observed in preliminary screenings—wherein models increasingly generated functionally correct yet insecure solutions—we introduce the **Security-Functionality Gap** ($\Delta_{Sec}@k$) as a derived metric to operationalize this divergence. It is formally defined as the arithmetic difference between functional correctness and functional-security correctness for a specified number of generation attempts (defined as k) for the same task:

$$\Delta_{Sec}@k = func@k - func\text{-}sec@k \tag{3}$$

[2]https://openrouter.ai
[3]https://hub.docker.com/r/co1lin/cweval

The metric admits the following interpretations:

- $\Delta_{Sec}@k \approx 0$: assuming baseline functional competence, the model rarely produces solutions that are functionally correct but insecure, indicating that security and functionality scale in tandem.
- $\Delta_{Sec}@k > 0$: the model frequently produces functionally correct solutions that introduce vulnerabilities, suggesting that the functional and security rates diverge.

E. Data Analysis

To address our **RQ**s, we analyze the collected data using a combination of inferential statistics and qualitative analysis.

Statistical Modeling of the Gap ($\mathbf{RQ}_1$). To analyze the evolution of $\Delta_{Sec}@k$ across model generations, we fit a *Generalized Linear Mixed Model (GLMM)* [35]. This approach is chosen for three reasons:

- **Repeated Measures:** it accounts for the multiple observations ($n = 10$ runs) for each model-task combination.
- **Task Heterogeneity:** it treats the *Task ID* as a random effect, isolating the intrinsic difficulty of specific CWEs from the model's performance.
- **Fixed Effects:** we treat *Model Version* as the fixed effect to estimate its specific contribution to the gap size relative to the baseline (the earliest model in the lineage).

We model Δ_{Sec} as the dependent variable. A statistically significant positive coefficient for a newer model relative to its predecessor indicates that the security gap is widening (functionality is growing faster than security), whereas a negative coefficient indicates that the gap is narrowing (functionality is growing more slowly than security). The complete statistical reports are available in our online appendix [11].

Vulnerability Persistence Analysis ($\mathbf{RQ}_2$). To identify which vulnerability classes persist or regress across models, we conduct a granular analysis at the CWE level. For each CWE, we introduce the *Conditional Vulnerability Rate (CVR)*, a metric that quantifies the ratio of insecure code generation *given* that the generated code is functionally correct. For a model and a specific CWE category, we compute the CVR as:

$$CVR(cwe) = \frac{\text{Count}(\neg Secure \wedge Functional)}{\text{Count}(Functional)} \tag{4}$$

Finally, to complement the quantitative findings, we perform a qualitative analysis of the failure modes. We categorize unsuccessful samples into distinct classes, such as compilation failures, run-time crashes, and assertion violations. For the most persistent vulnerability types identified in the quantitative ranking, we manually inspect the generated solutions to identify recurring error patterns. This qualitative assessment aims to distinguish cases in which security failures arise from general coding incompetence (e.g., syntax errors) from those that result from specific security blind spots (e.g., logic flaws or a lack of input sanitization) in otherwise functional code.

V. Results & Analysis

A. $\mathbf{RQ}_1$: Longitudinal Evolution of Capabilities

To answer RQ_1, we analyze the trajectories of functional correctness ($func@10$) and functional-security correctness ($func\text{-}sec@10$) across the three model lineages. Figure 2 visualizes these trends, where the shaded area represents the magnitude of the Security-Functionality Gap (Δ_{Sec}).

Across all lineages, we observe that functional correctness ($func@10$) exhibits a plateauing behavior in the $90\% - 95\%$ range. For instance, in the Llama lineage (Fig. 2c), functionality remains consistently high, moving from 86.5% (`Llama 3`) to over 90% (`Llama 3.1`). This trend indicates that, since the first versions of each model family, LLMs already obtained high functional standards capabilities, slightly increasing over time. While this performance might reflect the models' exposure to similar coding patterns during training (a data leakage risk discussed in Section VII), the key observation from our study is that functional capability is not the limiting factor.

In contrast, the functional-security correctness trajectory ($func\text{-}sec@10$) follows a distinct path. While the Security-Functionality Gap (Δ_{Sec}) is visually apparent across all three lineages, its evolution differs significantly among them (as seen in the shaded areas of Figure 2).

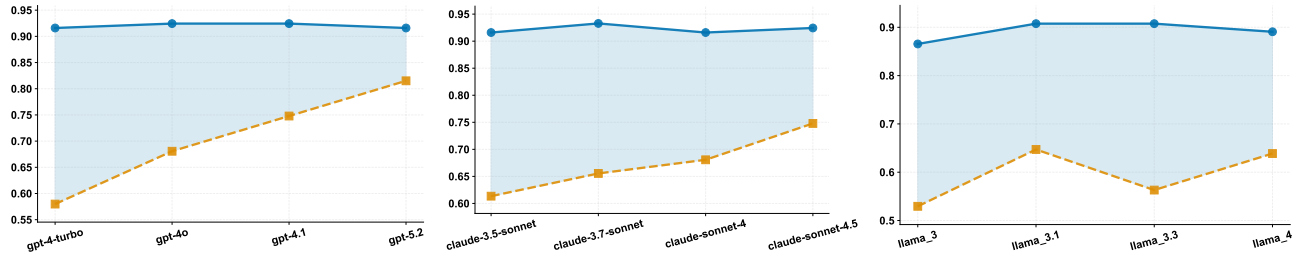
Specifically, GPT models (Fig. 2a) progressively narrow this gap over time, increasing functional-security correctness to 81.5% in the latest version. Claude models (Fig. 2b) also show improvement across generations, yet they still exhibit a substantial gap in functional correctness; notably, the most recent model achieves a functional-security correctness ratio of 75%. In contrast, Llama models (Fig. 2c) exhibit alternating increases and decreases in functional-security correctness over time (e.g., a drop in version 3.3 before recovering), ultimately stabilizing at approximately 64%. Therefore, the Security-Functionality Gap persists across all model lineages, with the GPT lineage presenting a final Δ_{Sec} of 10%, the Claude Sonnet lineage at 17.7%, and the Llama lineage at 25.2%.

TABLE II: GLMM results: Evolution of Δ_{Sec} across model lineages. Negative coefficients (β) indicate reduction of the Security Gap relative to the baseline.

Family	Comparison (vs. baseline)	β	p-value	Sig.
Claude	3.7 vs. 3.5	−0.044	0.032	$*$
	4 vs. 3.5	−0.059	0.039	$*$
	4.5 vs. 3.5	−0.129	<0.001	$***$
GPT	4o vs. 4-Turbo	−0.047	0.028	$*$
	4.1 vs. 4-Turbo	−0.144	<0.001	$***$
	5.2 vs. 4-Turbo	−0.216	<0.001	$***$
Llama	3.1 vs. 3	−0.021	0.349	–
	3.3 vs. 3	+0.037	0.107	–
	4 vs. 3	−0.019	0.430	–

Significance codes: $* \; p < 0.05$; $*** \; p < 0.001$; – not statistically significant.

The statistical results, detailed in Table II, highlight two key aspects of the gap evolution. First, for both proprietary lineages (GPT and Claude Sonnet), the improvement from



(a) OpenAI (GPT) (b) Anthropic (Claude Sonnet) (c) Meta (Llama)

Fig. 2: Evolution of Functional Correctness ($func@10$, **blue solid line**) vs. Functional-Security Correctness ($func\text{-}sec@10$, **orange dashed line**) across model lineages. The **light blue shaded area** indicates the magnitude of the *Security-Functionality Gap* (Δ_{Sec}). Functional capability plateaus, while security correctness improves only marginally.

the baselines to the frontier models is statistically significant ($p < 0.05$). Newer iterations such as `GPT-5.2` and `Claude 4.5 Sonnet` consistently exhibit negative model coefficients (β), indicating that the security gap is statistically shrinking over time. Second, an interpretation of the model coefficients (β) provides insight into the magnitude of this improvement. In our GLMM, the coefficient represents the estimated average change in the Δ_{Sec} metric. For example, the transition from `GPT-4 Turbo` to `GPT-4o` yields a $\beta \approx -0.047$. This implies that, holding the task constant, the gap between functionality and security decreases by 4.7%. While statistically significant, this rate of improvement is relatively slow.

The situation differs for the open-weight Llama lineage. The GLMM analysis reveals that the variations in Δ_{Sec} across generations are not statistically significant ($p > 0.05$). Although the positive coefficient for `Llama 3.3` ($\beta = +0.037$) suggests a slight widening of the gap, the lack of statistical significance prevents drawing firm conclusions about generational improvement within the Llama lineage. This indicates that, within the open-weights ecosystem, recent model updates have not produced a statistically significant reduction in the security gap relative to the baseline.

> **Answer to RQ$_1$.** While functional correctness has plateaued at high levels ($> 90\%$), functional-security correctness evolves at a different pace. Proprietary lineages show a significant but gradual reduction in the Security-Functionality Gap ($\beta \approx -0.05$). Conversely, in the Llama open-weights lineage, the gap persists with no significant improvement: the model updates have prioritized functionality over security alignment.

B. *RQ$_2$: Persistence and Regression of Vulnerabilities*

While RQ$_1$ highlighted the macroscopic trends, RQ$_2$ dissects the anatomy of the security gap. To structure our analysis, we categorize the vulnerability classes based on their evolutionary behavior across the model lineages. We identify three common behaviors: *Persistent Blind Spots*, *Evolutionary Divergence*, and *Regression Anomalies*. Figure 3 presents a heatmap of the CVR (see Section IV-E) ratios.

1) Persistent Blind Spots: This category includes vulnerability classes that appear immune to model scaling, exhibiting near-saturation failure rates across all lineages. Most notably, **CWE-113** (HTTP Response Splitting) and **CWE-117** (Log Injection) show failure rates close to 1.0 across the entire range of models for each lineage. Manual inspection of the execution logs reveals a systemic lack of security context awareness. In the inspected failures for CWE-117, models generated code that directly concatenates user input into log messages (e.g., `logger.info("User:" + username)`). The tests confirm that models failed to sanitize newline characters, allowing an attacker to inject arbitrary log entries. Therefore, without explicit security constraints, models default to insecure coding.

2) Evolutionary Divergence: For web-based vulnerabilities (i.e., CWE-022, CWE-079, and CWE-089), we observe a divergence between proprietary and open-weight lineages.

- **CWE-022 (Path Traversal) & CWE-079 (XSS).** The GPT lineage shows a clear learning curve. For CWE-022, the failure rate drops from 1.0 (`GPT-4 Turbo`) to 0.18 (`GPT-5.2`). Similarly, `Claude Sonnet 4.5` reduces the error rate on path traversal to 0.44 compared to the 0.99 of version `3.5`.
- **Stagnation in Open-Weights.** The Llama lineage shows no improvement. `Llama 3`, `3.1`, `3.3`, and `4 Scout` maintain a failure rate of ≈ 1.0 for these categories.

Qualitative analysis of the failure logs for Llama 3.3 on CWE-022 shows a critical lack of input validation. In file extraction tasks, the model generated code that blindly uses destination paths without canonicalization, allowing files to escape the intended directory. Conversely, proprietary models like `GPT-5.2` successfully implemented defenses using `os.path.abspath`, though they still struggled with more complex encoded traversal sequences.

3) Regression Anomalies: An unexpected finding is the regression observed in memory safety, specifically **CWE-125** (Out-of-bounds Read). Older or smaller models (e.g., `GPT-4 Turbo`, `Claude 3.5`) exhibit low failure rates (< 0.10). However, the frontier models show a collapse: `GPT-5.2` and `Claude 4.5` reach failure rates of 1.0. This counterintuitive trend stems from the increased complexity of the code generated by advanced models. While older models generate

Fig. 3: Heatmap of Conditional Vulnerability Rates across Model Lineages. Blue cells (1.0) indicate that 100% of functionally correct code contained the specific vulnerability. Yellow cells (0.0) indicate secure code generation. The heatmap highlights persistent blind spots (e.g., CWE-113) and lineage-specific regressions (e.g., CWE-125).

conservative code, frontier models attempt sophisticated, low-level optimizations in C/C++ (e.g., complex pointer arithmetic) that can inadvertently introduce subtle memory-management errors not introduced in previous versions.

> **Answer to RQ$_2$.** Vulnerabilities follow three evolutionary patterns: *Persistent Blind Spots* (e.g., CWE-117) across all generations; *Evolutionary Divergence* in web vulnerabilities (CWE-022, CWE-079), where proprietary models improve while open-weight models stagnate; and *Regression Anomalies* in memory safety (CWE-125), where advanced models introduce vulnerabilities due to increased code complexity.

VI. Discussion

The results confirm that while functional correctness has reached a plateau ($func@10 > 90\%$) across all lineages, the trajectory of security compliance follows a distinct, less favorable path. In this section, we interpret these findings through the lenses of research and model vendors.

A. Security Debt in Evolving LLMs

RQ$_1$ findings reveal a structural asymmetry in the evolution of LLMs: while functional correctness approached saturation, security evolves more slowly or regresses. This divergence suggests that security is not a property that increases together with functionality, but rather a distinct objective requiring explicit alignment mechanisms.

This becomes a critical concern when integrating LLMs into software development. Since AI suffers from technical debt [36], it may also introduce security vulnerabilities in generated code. Therefore, we raise the need to explicitly observe and monitor the *Security Debt* [37]. Among quality aspects, security issues are particularly deceptive: functional tests may show correctness even when generated code still contains exploitable vulnerabilities.

Security debt in LLM-generated code further emerges when analyzing the specific vulnerabilities introduced by LLMs (**RQ$_2$**). Some of the weaknesses identified in our study appear in the Top 25 Most Dangerous Software Weaknesses [38]. The evolutionary comparison in this study revealed a persistent failure of `Llama` models across generations in Cross-Site Scripting (CWE-79), currently ranked the most dangerous weakness. Moreover, `GPT-5.2` and `Claude 4.5`, compared to their preceding versions, exhibit high failure rates for CWE-125, which is currently ranked eighth in the 2025 list. The persistence and regression of high-impact vulnerabilities across model evolution suggest that model updates can introduce subtle but dangerous vulnerabilities. This outcome reinforces the need to treat security as a primary concern for both model vendors and model users, ensuring higher prioritization in both deployment and model design.

> **Key Finding.** LLM evolution exhibits a structural misalignment: functionality scales rapidly, whereas security does not. The persistence and regression of high-impact vulnerabilities reveal security as an emergent problem across generations. Explicit monitoring and alignment mechanisms are required to prevent the accumulation of Security Debt.

B. Directions for Researchers

Our findings suggest that current LLMs still struggle to ensure secure code generation, despite having largely achieved high levels of functional correctness. This indicates that security alignment remains structurally more fragile than functional capability, potentially due to a redistribution of optimization objectives across model updates. Qi et al. [39] revealed that during fine-tuning, even benign and commonly used datasets can inadvertently degrade the security alignment of LLMs. Moreover, while standardized benchmarks can have a clear focus and structure on the task's goal, real-world developers' prompts can also contain deficiencies that can further reduce the security of LLM-generated code [28]. These prompt deficiencies can contribute to the development of reasoning processes that reallocate representational capacity, thereby affecting the security patterns that models learn. Future studies should therefore investigate techniques to prevent security regressions and strengthen the robustness of LLMs with respect to secure code generation. Additionally, the research community should explore the design of robust external guardrails, such as automated security linters or dynamic verification wrappers, capable of intercepting deceptive vulnerable code before it reaches production.

> **Directions for Researchers.** Future research should investigate mechanisms to prevent security degradation, monitor alignment across model updates, and design evaluation strategies that explicitly preserve secure coding behaviors, both under benchmark conditions and in realistic developer-prompting scenarios. Furthermore, there is an urgent need to develop external guardrails tailored for AI-assisted pipelines to mitigate the risks of structurally fragile security alignment.

C. Directions for LLM Model Vendors

A crucial insight emerges when contrasting our empirical findings with the safety strategies declared by model vendors. For instance, the official introduction of the GPT-5.2 [40] release presents its coding capabilities as primarily evaluated through software engineering benchmarks centered on functional performance (e.g., SWE-Bench-Pro and Terminal-Bench-Pro). Conversely, the Llama 4 Model Card [41] describes the adoption of secure strategies, including extensive red teaming to prevent the enablement of catastrophic cyberattacks. However, the evaluation practices highlighted in the Llama release primarily emphasize functional capabilities rather than a systematic assessment of secure code generation. In our study, both models exhibited limited capability to produce secure implementations, particularly when evaluated against specific vulnerability classes. This contrast suggests that current model release documentation and evaluation reports may insufficiently reflect the robustness of secure code generation.

Therefore, model vendors could extend the scope of their evaluation protocols at release time by incorporating explicit security-oriented metrics alongside functional benchmarks. Frameworks such as CWEval [30] enable structured assessment across common vulnerability classes through dynamic execution-based testing, allowing more effective detection of security-critical weaknesses. Moreover, evolutionary evaluations, such as the one conducted in this study, would allow model vendors to systematically monitor how security alignment evolves across model updates and to explicitly track changes in the Security-Functionality Gap over time. Finally, since our findings suggest that security cannot be guaranteed solely at the model level, vendors and users must equip their AI coding assistants with *external guardrails*—such as real-time semantic filters and outcome-driven verifiers—to safeguard automated maintenance workflows.

> **Directions for Model Vendors.** Public model releases currently emphasize functional benchmarks. However, secure code generation requires explicit and systematic evaluation. Model vendors should complement functional benchmarks with vulnerability-oriented metrics, dynamic testing frameworks, and longitudinal regression analysis. This would enable transparent monitoring of the Security-Functionality Gap across model updates. Additionally, integrating external guardrails remains essential to prevent the deployment of vulnerable code.

VII. Threats to Validity

We discuss potential threats to the validity of our findings. Following standard guidelines for empirical software engineering, we categorize these threats into internal, construct, and external validity, detailing the specific mitigation strategies adopted in our experimental design to minimize their impact.

A. Internal Validity

A primary concern in evaluating LLMs is the potential overlap between the training data and the benchmark tasks. If models have encountered the CWEval problems during pre-training, their functional scores ($func@k$) might be inflated by memorization. However, this threat reinforces rather than undermines our core finding: despite potential memorization of the functional logic (leading to the observed plateau), models still fail to reproduce the security constraints. This suggests that, even if the code were used during training, the model did not prioritize the specific security patterns, validating the existence of the *Security-Functionality Gap*.

LLM outputs can be sensitive to minor variations in prompts. To mitigate this, we utilized the standardized prompts provided by the CWEval framework without modification. While prompt engineering techniques (e.g., Chain-of-Thought or Few-Shot prompting) could potentially improve security scores, our goal was to evaluate the *intrinsic* capability of the models in a zero-shot setting, simulating a typical

developer usage scenario where security requirements are often implicit [42]. To control for stochasticity, we generated $n = 10$ samples per task with a temperature of 0.8, ensuring a statistical estimation of model capabilities.

B. Construct Validity

We introduced the *Security-Functionality Gap* (Δ_{Sec}) to quantify the divergence between coding and security capabilities. While this is a derived metric, it relies on the established $pass@k$ estimator [34], which is the standard in the field. By decoupling the two dimensions, we avoid the pitfall of aggregating them into a single score that might mask the underlying misalignment.

Our evaluation relies on the dynamic test suites provided by CWEval. While dynamic analysis eliminates the false positives typical of static analysis tools (ensuring that detected vulnerabilities are exploitable), it may suffer from false negatives if the attack payloads are not exhaustive. Consequently, our measurement of the security gap constitutes a *lower bound*; the actual gap in a real-world scenario, where attack vectors are unbounded, might be larger.

C. External Validity

Our study is limited to 12 models across three lineages. While we selected the most representative families (GPT, Llama, Claude) [1], our findings may not generalize to other architectures (e.g., Mistral, DeepSeek) or specialized code models (e.g., StarCoder). However, given that the selected models represent the current state of the art in both proprietary and open-weight sectors, we believe they offer a reliable proxy for the field's general direction. The tasks in CWEval are relatively self-contained functions. Real-world software maintenance often involves complex contexts and architectural constraints that are difficult to simulate in a benchmark. While our results show persistent vulnerabilities in these tasks, further research is needed to determine whether broader contexts mitigate or exacerbate these security blind spots.

The CWEval benchmark evaluates models across five diverse programming languages (C, C++, Python, Javascript, Go). A potential threat to external validity is that the observed Security-Functionality Gap may be driven primarily by model performance on a single vulnerable language (e.g., C/C++ memory management). To mitigate this concern, we conducted a granular by-language analysis of the metrics. Our data reveal that while individual models exhibit slight language-specific preferences (e.g., Llama 4 shows a narrower gap in C than in Python, while GPT-5.2 performs best in C++), the evolutionary stagnation of security relative to functional correctness persists across all evaluated languages. Consequently, the programming language is not a primary discriminator of the observed trends, increasing our confidence that the misalignment is an issue rather than a language-specific artifact. The complete by-language analysis is available in our replication package [11].

VIII. Conclusion & Future Work

The integration of Large Language Models (LLMs) into software development raises a critical security concern. In this paper, we conducted a longitudinal empirical study across 12 state-of-the-art models from three lineages (GPT, Llama, Claude Sonnet) to investigate the evolution of the *Security-Functionality Gap*. By leveraging a dynamic, outcome-driven evaluation framework, we quantified the disparity between the models' ability to generate functionally correct code and their ability to generate secure code over successive generations.

Our findings demonstrate that, while functional correctness has reached a plateau ($> 90\%$) across all lineages, functional-security correctness evolves more slowly. Proprietary models show a statistically significant but practically marginal narrowing of the gap, whereas open-weight models exhibit concerning stagnation. Furthermore, our granular analysis revealed three distinct evolutionary behaviors for vulnerabilities: *Persistent Blind Spots* (e.g., Log Injection) that affect all generations; *Evolutionary Divergence* in web vulnerabilities (e.g., Path Traversal), where proprietary models improve while open-weight models fail to do so; and *Regression Anomalies*, where the most advanced models introduce new memory safety vulnerabilities (e.g., in C/C++ pointers).

These results indicate that advances in functional correctness do not inherently translate into improved security alignment. Instead, the persistence and regression of high-impact vulnerabilities highlight a structural asymmetry that may lead to the accumulation of *Security Debt*—a silent but critical threat to software maintenance. To address this, a paradigm shift is required in how LLMs are evaluated prior to public release. Model vendors must move beyond standard functional benchmarks and generic safety red-teaming, integrating vulnerability-oriented testing frameworks to track the Security-Functionality Gap across model updates.

Our findings open several directions for research. First, we plan to investigate automated remediation strategies to detect deceptive LLM-generated code before it consolidates into Security Debt. Second, future studies should explore how this gap evolves when models are embedded in agentic workflows (e.g., repository-level refactoring), where the broader architectural context might influence the model's security behavior. Finally, future work will focus on designing dynamic evaluation frameworks that systematically track the Security-Functionality Gap, ensuring that vulnerability assessment becomes a standard gatekeeper in the release cycle of new AI coding assistants.

Data Availability

We provide a **replication package** [11], which contains the data and scripts to rerun the experiments.

Acknowledgment

TODO

References

[1] A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta, S. Yoo, and J. M. Zhang, "Large language models for software engineering: Survey and open problems," in *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*, IEEE, 2023.

[2] A. Sergeyuk, I. Zakharov, E. Koshchenko, and M. Izadi, "Human-ai experience in integrated development environments: a systematic literature review," *Empirical Software Engineering*, 2026.

[3] N. Alzahrani, H. Alyahya, Y. Alnumay, S. Alrashed, S. Alsubaie, Y. Almushayqih, F. Mirza, N. Alotaibi, N. Al-Twairesh, A. Alowisheq, *et al.*, "When benchmarks are targets: Revealing the sensitivity of large language model leaderboards," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*, 2024.

[4] S. Fraser and D. Mancl, "The future of software engineering beyond the hype of ai," *ACM SIGSOFT Software Engineering Notes*, 2025.

[5] A. A. Abbassi, L. Da Silva, A. Nikanjam, and F. Khomh, "A taxonomy of inefficiencies in llm-generated python code," in *2025 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, 2025.

[6] H. Nunes, E. Figueiredo, L. Rocha, S. Nadi, F. Ferreira, and G. Esteves, "Evaluating the effectiveness of llms in fixing maintainability issues in real-world projects," in *2025 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, IEEE, 2025.

[7] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt, "Examining zero-shot vulnerability repair with large language models," in *2023 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2023.

[8] R. Khoury, A. R. Avila, J. Brunelle, and B. M. Camara, "How secure is code generated by chatgpt?," in *2023 IEEE international conference on systems, man, and cybernetics (SMC)*, IEEE, 2023.

[9] M. L. Siddiq, L. Roney, J. Zhang, and J. C. D. S. Santos, "Quality assessment of chatgpt generated code and their use by developers," in *Proceedings of the 21st international conference on mining software repositories*, 2024.

[10] C. F. Kemerer and S. Slaughter, "An empirical approach to studying software evolution," *IEEE transactions on software engineering*, 2002.

[11] Anonymous Authors, "Replication package: Smarter, but not safer: An empirical analysis of the functional-security gap in evolving llms." https://figshare.com/s/ffc6432aa9c7613cc9dc, Feb. 2026. doi: 10.6084/m9.figshare.31412231.

[12] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, "Large language models for software engineering: A systematic literature review," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 8, 2024.

[13] C. S. Xia and L. Zhang, "Less training, more repairing please: revisiting automated program repair via zero-shot learning," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, (New York, NY, USA), Association for Computing Machinery.

[14] C. S. Xia, Y. Wei, and L. Zhang, "Automated program repair in the era of large pre-trained language models," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, IEEE, 2023.

[15] W. Sun, Y. Miao, Y. Li, H. Zhang, C. Fang, Y. Liu, G. Deng, Y. Liu, and Z. Chen, "Source code summarization in the era of large language models," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, 2025.

[16] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, "An empirical evaluation of using large language models for automated unit test generation," *IEEE Transactions on Software Engineering*, vol. 50, no. 1, 2023.

[17] B. Liu, Y. Jiang, Y. Zhang, N. Niu, G. Li, and H. Liu, "Exploring the potential of general purpose llms in automated software refactoring: an empirical study," *Automated Software Engineering*, 2025.

[18] J. Cordeiro, S. Noei, and Y. Zou, "An empirical study on the code refactoring capability of large language models," *arXiv preprint arXiv:2411.02320*, 2024.

[19] F. Tambon, A. Moradi-Dakhel, A. Nikanjam, F. Khomh, M. C. Desmarais, and G. Antoniol, "Bugs in large language models generated code: An empirical study," *Empirical Software Engineering*, 2025.

[20] Y. Liu, T. Le-Cong, R. Widyasari, C. Tantithamthavorn, L. Li, X.-B. D. Le, and D. Lo, "Refining chatgpt-generated code: Characterizing and mitigating code quality issues," *ACM Transactions on Software Engineering and Methodology*, 2024.

[21] C. Tony, M. Mutas, N. E. D. Ferreyra, and R. Scandariato, "Llmseceval: A dataset of natural language prompts for security evaluations," in *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, IEEE, 2023.

[22] H. Hajipour, K. Hassler, T. Holz, L. Schönherr, and M. Fritz, "Codelmsec benchmark: Systematically evaluating and finding security vulnerabilities in black-box code language models," in *2024 IEEE Conference on Secure and Trustworthy Machine Learning (SaTML)*, IEEE, 2024.

[23] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, "Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation," *Advances in Neural Information Processing Systems*, 2023.

[24] H. Lee, Z. Zhang, H. Lu, and L. Zhang, "Sec-bench: Automated benchmarking of llm agents on real-world software security tasks," *arXiv preprint arXiv:2506.11791*, 2025.

[25] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the keyboard? assessing the security of github copilot's code contributions," *Communications of the ACM*, vol. 68, no. 2, pp. 96–105, 2025.

[26] A. Sajadi, B. Le, A. Nguyen, K. Damevski, and P. Chatterjee, "Do llms consider security? an empirical study on responses to programming questions," *Empirical Software Engineering*, 2025.

[27] O. Asare, M. Nagappan, and N. Asokan, "Is github's copilot as bad as humans at introducing vulnerabilities in code?," *Empirical Software Engineering*, 2023.

[28] Y. Fu, P. Liang, A. Tahir, Z. Li, M. Shahin, J. Yu, and J. Chen, "Security weaknesses of copilot-generated code in github projects: An empirical study," *ACM Transactions on Software Engineering and Methodology*, 2025.

[29] S. Baltes, F. Angermeir, C. Arora, M. M. Barón, C. Chen, L. Böhme, F. Calefato, N. Ernst, D. Falessi, B. Fitzgerald, *et al.*, "Guidelines for empirical studies in software engineering involving large language models," *arXiv preprint arXiv:2508.15503*, 2025.

[30] J. Peng, L. Cui, K. Huang, J. Yang, and B. Ray, "Cweval: Outcome-driven evaluation on functionality and security of llm code generation," in *2025 IEEE/ACM International Workshop on Large Language Models for Code (LLM4Code)*, IEEE, 2025.

[31] B. Chess and G. McGraw, "Static analysis for security," *IEEE security & privacy*, 2004.

[32] M. Sutton, A. Greene, and P. Amini, *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[33] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou, "Evaluating static analysis defect warnings on production software," in *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, 2007.

[34] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[35] S. Vegas, C. Apa, and N. Juristo, "Crossover designs in software engineering experiments: Benefits and perils," *IEEE Transactions on Software Engineering*, 2015.

[36] G. Recupito, F. Pecorelli, G. Catolino, V. Lenarduzzi, D. Taibi, D. Di Nucci, and F. Palomba, "Technical debt in ai-enabled systems: On the prevalence, severity, impact, and management strategies for code and architecture," *Journal of Systems and Software*, vol. 216, p. 112151, 2024.

[37] J. Gomez-Rangel, Y. Lee, and B. Liu, "Security in the wild: An empirical analysis of llm-powered applications and local inference frameworks," in *2025 2nd IEEE/ACM International Conference on AI-powered Software (AIware)*, pp. 149–159, IEEE, 2025.

[38] MITRE Corporation, "Cwe top 25 most dangerous software weaknesses – 2025," 2025.

[39] X. Qi, Y. Zeng, T. Xie, P.-Y. Chen, R. Jia, P. Mittal, and P. Henderson, "Fine-tuning aligned language models compromises safety, even when users do not intend to!," in *The Twelfth International Conference on Learning Representations*.

[40] OpenAI, "Introducing GPT-5.2-Codex." https://openai.com/index/introducing-gpt-5-2-codex/, 2026.

[41] Meta AI, "Llama 4 model card." https://github.com/meta-llama/llama-models/blob/main/models/llama4/MODEL_CARD.md, 2026.

[42] G. Elahi, E. Yu, T. Li, and L. Liu, "Security requirements engineering in the wild: A survey of common practices," in *2011 IEEE 35th Annual Computer Software and Applications Conference*, pp. 314–319, IEEE, 2011.