

Security in LLM-Assisted Coding: A Systematic Literature Review

LIANG YUAN*, MARYAM KHANIAN NAJAFABADI, MASHUD RANA, and VARVARA VETROVA, Australian Catholic University, Australia

Large Language Models accelerate software development but introduce severe security vulnerabilities. This systematic review investigates security risks in automated coding, benchmark limitations, and the role of human developers. Findings indicate that models frequently generate insecure code and faulty patches due to training data biases. Additionally, the models themselves are susceptible to direct attacks. Current evaluation methods suffer from data contamination and lack robust security metrics. Consequently, human oversight remains essential. Safer integration of automation will require more security-aware model architectures, stronger security-specific benchmarks, and more effective human-AI collaboration frameworks to mitigate these systemic risks.

CCS Concepts: • **Security and privacy** → **Software and application security**.

Additional Key Words and Phrases: LLM, Software security, AI safety

ACM Reference Format:

Liang Yuan, Maryam Khanian Najafabadi, Mashud Rana, and Varvara Vetrova. 2026. Security in LLM-Assisted Coding: A Systematic Literature Review. *J. ACM* 37, 4, Article 111 (July 2026), 35 pages. <https://doi.org/XXXXXXX.XXXXXXX>

1 Introduction

The advent of Large Language Models (LLMs) marks a transformative period in software engineering, offering unprecedented capabilities in automating and assisting various coding tasks. From code generation and completion [31, 54, 57, 101] to automated program repair (APR) and vulnerability patching [42, 112, 140, 157], LLMs promise to enhance developer productivity and accelerate software development lifecycles [9]. However, this rapid integration is paralleled by a growing array of security concerns that demand systematic investigation and proactive mitigation strategies [50].

A central challenge in securing LLM-assisted coding is that functional correctness does not establish software security. Generated code may satisfy its intended functionality while containing exploitable vulnerabilities, and an automatically produced patch may leave the underlying security issue incompletely resolved or introduce additional weaknesses [46, 69, 111]. Assessing these risks is further complicated by limitations in evaluation datasets and pipelines, including benchmark contamination, incomplete vulnerability coverage, and differences between isolated coding tasks and realistic development settings. Human review provides an additional safeguard, but its effectiveness

*Corresponding author

Authors' Contact Information: Liang Yuan, liang.yuan@acu.edu.au; Maryam Khanian Najafabadi, maryam.khanian@acu.edu.au; Mashud Rana, mashud.rana@acu.edu.au; Varvara Vetrova, varvara.vetrova@acu.edu.au, Australian Catholic University, North Sydney, New South Wales, Australia.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM 1557-735X/2026/7-ART111

<https://doi.org/XXXXXXX.XXXXXXX>

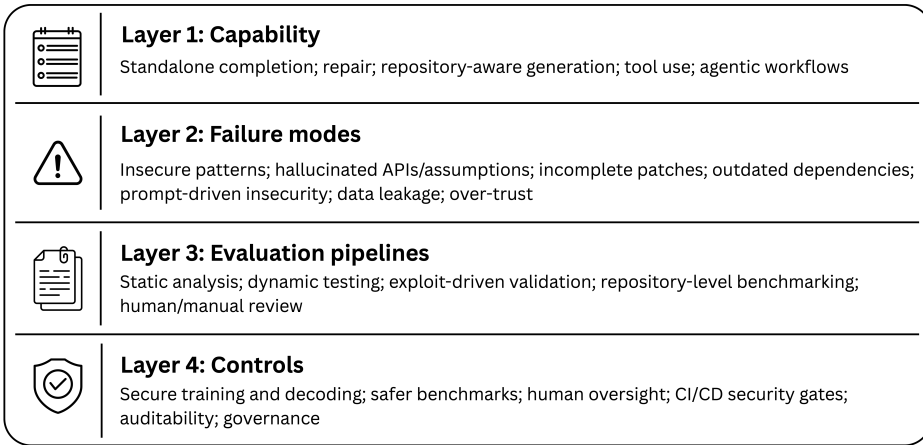


Fig. 1. A layered framework for analyzing security in LLM-assisted coding.

depends on developers' security expertise and their ability to critically assess model outputs [100, 114].

These challenges involve two related but distinct concepts: *software security* and *AI safety*. Software security concerns the confidentiality, integrity, and availability of software artifacts and systems, including vulnerabilities introduced through code generation, insecure repairs, and adversarial manipulation. AI safety encompasses broader concerns about the dependable, controllable, and socially responsible operation of LLM-based systems, including misuse resistance, human oversight, trust calibration, accountability, and governance [23]. The distinction matters because a coding system's security depends both on the properties of its outputs and on the conditions under which those outputs are produced, evaluated, and accepted into software systems.

The transition from passive code-completion tools to *agentic coding systems* makes these interactions more consequential. Autonomous coding agents, multi-agent frameworks, and tool-using LLMs can search repositories, invoke external tools, execute tests, edit files, and iteratively revise their outputs [24, 70, 106, 153]. These capabilities expand the security problem beyond individual code snippets. Errors can propagate across multiple steps, tool invocation and environment access introduce additional attack surfaces, and increasing autonomy can make intermediate actions less visible to human reviewers. Secure deployment therefore requires attention to workflow-level risks, the limits of human oversight, and the controls governing an agent's actions.

Existing studies provide substantial coverage of individual parts of this problem. Reviews of automated program repair examine repair techniques and their limitations [4, 53, 150], while code-security surveys address vulnerabilities in generated code and the use of LLMs for vulnerability detection and repair [11, 122, 156]. Broader reviews of AI safety and agentic systems examine oversight, controllability, and governance [23, 24]. However, these strands provide limited synthesis of how code-level weaknesses, evaluation limitations, human interaction, and increasing autonomy jointly affect security in coding workflows. In particular, the relationship between what current evaluations measure and what is needed to support secure deployment remains insufficiently integrated with analyses of human oversight and agentic risk.

To address this gap, this survey provides a systematic review of security in LLM-assisted coding. We focus on five connected issues: the security weaknesses introduced by LLM-based generation and repair, the limitations of current evaluation datasets and pipelines, the changing threat model created by agentic and tool-integrated coding systems, the dual role of human oversight as both safeguard and risk factor, and the governance conditions required for trustworthy deployment.

Thus, this review makes three main contributions. First, it synthesises security issues across code generation and repair, human interaction, and adversarial attacks, extending the analysis to the workflow-level risks of agentic and tool-integrated systems. Second, it critically examines evaluation datasets, methodologies, and metrics, with particular attention to benchmark contamination, the limitations of individual analysis methods, and the gap between benchmark settings and deployment conditions. Third, it analyses the role and limitations of human oversight and develops research directions connecting technical safeguards, evaluation practices, and deployment governance.

These dimensions are organised through the four-layer framework shown in Fig. 1, which links system capabilities, failure modes, evaluation pipelines, and controls. The framework provides a common structure for examining how different coding capabilities expose particular risks and how those risks can be assessed and mitigated.

The remainder of the paper is organised as follows. Section 2 reviews the existing literature and positions this work relative to surveys on software engineering, code security, AI safety, and agentic systems. Section 3 outlines the review methodology. Section 4 analyses the security issues associated with LLM-assisted coding, including the risks introduced by more autonomous agentic workflows. Section 5 examines current evaluation methods and benchmarks, and Section 6 discusses human oversight and human-factor risks in LLM-assisted coding. Finally, Section 7 offers practical insights for academia and industry and lays out directions for future work, before Section 8 concludes the paper.

Table 1. Comparison of closely related surveys and the scope of the present review. ✓ indicates substantial coverage; ~ indicates partial coverage; × indicates that the topic is not a central focus.

Survey	Year	Primary scope	Sec	Eval.	Human	Agentic	Gov.	Fac- tor	Main gap
Huang et al. [53]	2023	APR broadly	×	×	×	×	×		Not LLM-security focused.
Anand et al. [4]	2024	APR+code gen.	~	~	×	×	×		Broad overview; limited security synthesis.
Zhou et al. [156]	2024	Vulnerability detect/repair	✓	~	×	×	×		Task-specific rather than integrated.
Zhang et al. [150]	2024	LLMs for APR	~	~	×	×	×		Repair-focused; narrow security scope.
Basic and Giaretta [11]	2025	LLMs in code security	✓	~	×	×	×		Limited on deployment and governance.
Sheng et al. [122]	2025	Vulnerability detection	✓	✓	×	×	×		Detection-oriented only.
Li and Chow [23]	2025	AI safety/governance	~	×	✓	~	✓		Not coding-task specific.
Li and Chow [24]	2026	Agentic AI systems	~	×	✓	✓	✓		Not code-security centered.
This survey	2026	Security in LLM-assisted coding	✓	✓	✓	✓	✓		Integrated security-centered view.

2 Related Works

Existing surveys explore the integration of LLMs into software engineering tasks like Automated Program Repair (APR) and code generation, increasingly focusing on security implications. Anand

et al. [4] review 27 recent papers on LLMs in APR and code generation, noting their transformative impact in reducing manual debugging and improving code generation. It contrasts methodologies, identifies trends like using LLMs and feedback loops, and discusses challenges, including functional correctness and security. Similarly, Huang et al. [53] provide a broad overview of APR techniques and discuss the opportunities introduced by learning-based methods, including LLMs. More recently, Zhang et al. [150] present a systematic literature review focused specifically on LLM applications in APR, analysing repair strategies, utilisation patterns, and repair scenarios that include security vulnerabilities.

Other surveys narrow their focus specifically to the intersection of LLMs and code security. Basic and Giarretta [11] investigate security vulnerabilities introduced by LLM-generated code, evaluate LLM effectiveness in detecting and fixing vulnerabilities, and explore data poisoning impacts. It aims to provide an in-depth analysis across these three areas. Likewise, Sheng et al. [122] survey LLM-based software vulnerability detection, examining problem formulations, models, datasets, and evaluation metrics, while Zhou et al. [156] review the use of LLMs for vulnerability detection and repair and outline future research directions. These studies provide useful foundations, but they often concentrate on specific security tasks, such as detection or repair, without fully integrating broader issues such as benchmark reliability, human-AI trust, deployment governance, or the security implications of increasing model autonomy.

A growing body of broader LLM safety research examines issues that intersect closely with secure code generation, including controllability, misuse resistance, human oversight, and deployment governance. Li and Chow [23] help frame risks that extend beyond insecure code artifacts, such as trust miscalibration, weak oversight, and insufficient accountability. These perspectives complement code-centric security analysis by explaining how flawed outputs become operational and organizational risks. Another emerging direction concerns agentic AI systems. Coding systems are increasingly evolving from passive assistants into tool-using agents that can plan, retrieve context, execute tests, and revise files. In another work, Li and Chow [24] show that this shift expands the risk landscape from single-output vulnerabilities to workflow-level risks such as tool misuse, persistent memory, compromised retrieval, and compounding multi-step errors.

Table 1 summarizes the main scope and limitations of closely related surveys. Prior reviews tend to focus on APR, vulnerability detection and repair, or broader AI safety questions separately. By contrast, this survey provides an integrated security-centered perspective on LLM-assisted coding across generation, repair, evaluation, human oversight, agentic risk, and deployment governance. We also place particular emphasis on evaluation limitations, including benchmark contamination, the partial visibility of static or dynamic analysis in isolation, and the mismatch between function-level benchmarks and real-world deployment settings.

Accordingly, this review is intended not only to summarize vulnerabilities and defences, but also to clarify how technical, human, and organizational factors jointly shape the secure use of LLM-assisted coding systems.

3 Review Methodology

To provide a comprehensive overview of the security landscape of LLM-based code tasks, a systematic literature review (SLR) was conducted. The methodology was designed following established guidelines for systematic reviews in software engineering and consists of three main stages: defining the research questions, executing a structured search and selection strategy, and performing a final snowballing search to ensure completeness.

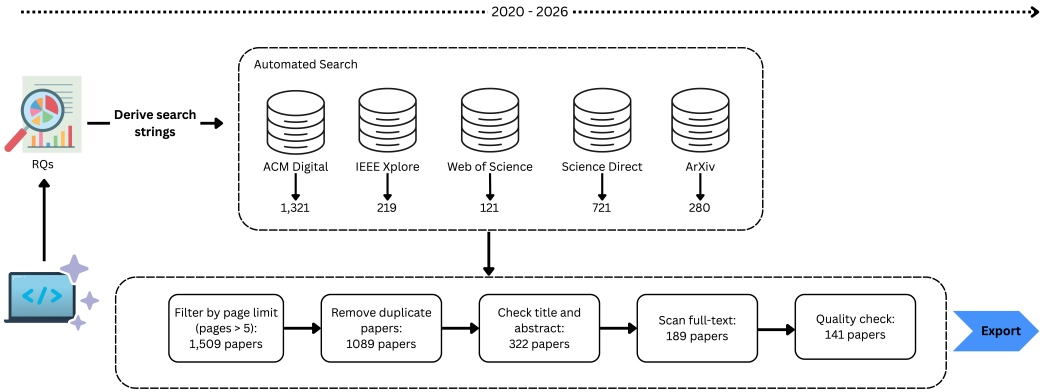


Fig. 2. Search process

3.1 Research Questions

This review is guided by the following research questions (RQs), which address the core objectives identified in the introduction:

- **RQ1:** What are the primary security issues introduced by the use of LLMs in code-related tasks?
- **RQ2:** What are the capabilities and limitations of current evaluation methods and benchmarks for assessing LLM-assisted coding tasks?
- **RQ3:** What is the role of human oversight and interaction in mitigating security risks in LLM-assisted software development?

3.2 Search Strategy

A systematic search strategy was designed to identify relevant literature, as shown in Fig. 2. The search was conducted across several major digital libraries and preprint archives, including ACM Digital Library, IEEE Xplore, SpringerLink, ScienceDirect, Web of Science, and arXiv, reflecting common sources used in similar comprehensive reviews. The search keywords were formulated to capture the intersection of LLMs, code-related tasks, and security. Boolean operators were used to combine terms from three main categories:

- LLM terms: "Large Language Model", "LLM", "Foundation Model", "Generative AI".
- Code task terms: "code generation", "program synthesis", "code completion", "program repair", "vulnerability detection", "vulnerability fixing", "automated program repair", "APR", "code analysis", "software development".
- Security terms: "security", "vulnerability", "secure code", "insecure code", "exploit", "attack", "threat", "prompt injection", "jailbreak", "data poisoning", "adversarial attack", "data leakage", "confidentiality", "integrity", "robustness".
- Human-AI interaction terms: "human-in-the-loop", "human oversight", "human-AI collaboration", "cognitive bias", "feedback", "human review".

The timeframe for the search initially focused on publications from 2020 to early 2025, aligning with the emergence and rapid proliferation of LLMs in software engineering tasks, particularly the significant surge in research from 2023 onwards. To ensure comprehensive coverage, both forward and backward snowballing techniques were applied to the reference lists of selected key papers, a method also employed in other systematic reviews [150].

3.3 Study Selection Criteria

The initial search results were subjected to a rigorous, multi-stage filtering process to determine inclusion in the final review.

First, titles, abstracts, and keywords were screened for relevance. Subsequently, the full texts of the remaining articles were reviewed against the predefined inclusion and exclusion criteria.

Inclusion Criteria:

- The paper must discuss security aspects directly related to the use of LLMs in tasks involving source code (generation, analysis, modification, repair, etc.).
- The paper evaluates LLMs for security-related tasks performed on source code (e.g., vulnerability detection, patch generation).
- The paper proposes or analyses adversarial attacks against LLMs specifically within the context of code-related tasks, or defences against such attacks.
- The study must be a peer-reviewed conference paper or journal article. Exceptionally, recent high-quality preprints were retained only when they addressed fast-moving topics not yet represented by an archival version and when their methodology and evidence were sufficiently detailed for assessment.

Exclusion Criteria:

- Papers where LLMs are not a central component of the discussed code task or where the security aspect is trivial or out of scope.
- Papers not written in English.
- Editorials, opinion pieces, abstracts, posters, or very short workshop papers lacking substantial methodological detail or findings.
- Studies focusing on the security of LLMs in purely natural language processing tasks (e.g., general text generation, chatbots for non-technical domains) unless the principles or findings are directly and significantly applicable to code security.
- Studies focusing solely on ethical concerns (e.g., job displacement, copyright) without a direct technical security component.

3.4 Quality Assessment of Preprints

Because this field is evolving rapidly, a limited number of recent preprints were retained when no peer-reviewed counterpart was available and when the work addressed important emerging topics. To reduce the risk of relying on unstable evidence, we applied additional screening to these studies, considering author credibility, methodological transparency, evidence quality, and the availability of later archival versions where applicable.

4 LLM-Assisted Coding Task Security Issues

Integrating LLMs into software development introduces diverse security issues, including inherent technological flaws, hallucination-related semantic failures, human-driven vulnerabilities, and direct attacks on the models (summarised in Table 2). This section explores these categories to address **RQ1**.

Understanding this threat landscape is necessary because weaknesses originate from multiple sources, ranging from the underlying technology to specific deployment and usage configurations. Consequently, mitigation requires a layered approach. For example, a model replicating insecure training data, a malicious data poisoning attack, and a poorly written user prompt are distinct problems that require separate defensive strategies.

Table 2. A Taxonomy of LLM Security Issues

Category	Specific Threat	Potential Impact	References
Inherent Security Challenges	Trained with risk datasets	Introduction of exploitable flaws into applications.	[45, 46, 58, 100, 104, 111, 131]
	Lack of security principle understanding	Subtle, hard-to-detect vulnerabilities; insecure design patterns.	[5, 46, 47, 51, 58, 100, 135]
	Insecure/incomplete patches	Regression of security posture; false sense of security.	[5, 12, 46, 69, 131]
	Propagation of outdated/vulnerable dependencies	System compromise through known exploits in third-party components.	[56, 100, 111, 118, 123]
	Hallucinated APIs / semantic misassumptions	Incorrect security assumptions; broken validation logic; misuse of non-existent or misunderstood protections.	[67, 100, 129]
Human-Related Security Issues	Insecure prompt engineering	Generation of vulnerable code; exposure of sensitive information.	[20, 38, 45, 51, 61, 104]
	Over-reliance and reduced scrutiny	Insecure code integrated into production systems.	[12, 49, 114, 130]
	Lack of developer security training for LLM usage	Increased likelihood of insecure code generation and deployment.	[58, 59, 100]
Attack-Related Security Issues	Adversarial Attacks/-Data Poisoning	Incorrect security assessments; introduction of exploitable flaws or backdoors.	[2, 10, 26, 37, 50, 107, 119, 143, 144, 149]
	Data Leakage (from training data or prompts)	Exposure of proprietary algorithms, API keys, personal data.	[44, 46, 48–50, 61, 69, 104, 116, 131, 154]

4.1 Inherent Security Challenges

A primary concern is that LLMs frequently generate vulnerable code. Because they replicate patterns from large training datasets, they often reproduce insecure practices, common weaknesses (CWEs), and outdated library usages. Lacking a semantic understanding of security principles [29], LLMs produce code that may be functionally correct but structurally insecure, simply mirroring flaws present in their training data [10].

Additionally, the confident presentation of LLM outputs can cause developers to accept generated code with insufficient review [100], overlooking subtle, exploitable bugs [144]. This risk also applies to automated program repair (APR) and vulnerability patching. While an LLM might resolve a targeted functional error, the resulting patch can introduce new security flaws or provide an incomplete fix that attackers can still bypass [3, 105].

Training Data Contamination. Several studies highlight that LLMs have an inherent tendency to introduce security vulnerabilities, often due to being trained on massive open-source code corpora, which may contain insecure practices [100, 104]. Some research suggests this is partly due to LLMs' lack of awareness regarding scenario-relevant security risks, as they primarily prioritise functional requirements [46, 58, 111]. Consequently, LLMs replicate these security flaws from their training

data, leading to the generation of vulnerable code. This creates significant concerns about the reliability of LLMs for secure code generation.

The empirical evidence for this is compelling. Götz and Schaad [45] found that up to 65% of code generated by some LLMs was initially insecure, attributing this partly to a lack of best-practice examples in training data. Similarly, Pearce et al. [111] notably found that approximately 40% of code snippets generated by GitHub Copilot were susceptible to vulnerabilities. This was linked to Copilot being trained on public GitHub code known to contain insecure patterns. Mohsin et al. [100] demonstrate that LLMs trained on public data consistently reproduce software vulnerabilities. Their zero-shot evaluations expose a widespread lack of baseline security knowledge across multiple models. Notably, systems like ChatGPT and Copilot produced insecure code in roughly 40% of test cases, and baseline Prompt-Driven Code Generators (PDCGs) similarly failed to meet acceptable security standards.

More comprehensive evaluations by Tihanyi et al. [131] across nine state-of-the-art models, showing that at least 62% to 63% of generated C programs were vulnerable when using neutral prompts. They concluded that all examined models notoriously introduce vulnerabilities at unacceptable rates, attributing this to training on human-generated code containing vulnerabilities. **Limited Security Principle Understanding.** Another critical inherent challenge is rooted in the design and training objectives of LLMs, which almost universally prioritise functional correctness and linguistic fluency over security awareness [100]. These models are engineered to predict the most probable next token based on statistical patterns, not to reason abstractly about security implications or to evaluate potential attack vectors [5]. This architectural limitation means that the security quality of generated code is often inconsistent and unpredictable.

Gong et al. [46] observed that models often produce functional code while overlooking hidden vulnerabilities, falling into scenario-specific "traps." The authors attribute this failure to a general lack of security awareness and an inability to identify risks not explicitly detailed in the prompt. Studies show that even security-aware instructions in prompts may not significantly enhance security [51] or reliably [47, 58] reduce vulnerabilities. Tóth et al. [135] highlighted that security risks can be highly situational, making it difficult for AI to grasp context and distinguish safe from risky practices.

Insecure or Incomplete Patches. The application of LLMs to automated program repair (APR) and vulnerability patching introduces another vector of risk. Fixes generated by LLMs may be functionally correct on the surface but can introduce new security flaws or leave the original vulnerability only partially mitigated.

For example, Belozarov et al. [12] demonstrate that ChatGPT and GPT-4o correctly identified and fixed only 17 of 32 tested vulnerabilities. The models often produced incomplete patches, relied on superficial mitigations such as simple input validation, and misclassified vulnerability types (e.g., confusing CWE-78 [94] with CWE-77 [93]), thereby applying inappropriate defenses that left the original flaws unresolved.

Tihanyi et al. [131] suggest that current LLMs are not yet ready for program repair because they still struggle to generate functionally correct fixes. Although some studies report promising results, particularly when combined with other tools, substantial further research is still needed [46, 131].

Asare et al. [5] show that Copilot may reintroduce vulnerabilities rather than produce secure repairs, while scanner-guided iterative repair eventually plateaus. Likewise, Liu et al. [69] report that repair success varies by vulnerability type, with incomplete fixes persisting for classes such as Input Validation and Information Leakage.

Outdated Dependency Propagation. LLMs often recommend deprecated libraries, obsolete APIs, and cryptographically broken implementations that persist in training data despite safer alternatives. Studies report risks from outdated libraries [56] and deprecated APIs [111, 118],

indicating that obsolete practices can continue to appear in generated code as the security landscape evolves [111]. One example is MD5, a broken hash algorithm still suggested by models such as GitHub Copilot [118] and also identified in code analyzed by the Sallm framework under CWE-328 [123]. Because LLMs are trained on public repositories, they may reproduce outdated or vulnerable third-party dependencies in both prompt-driven models and coding assistants [100, 147]. While security quality varies across models, languages, and domains, severe vulnerabilities still recur under simple prompts.

Hallucinated APIs and Security-Relevant Semantic Misassumptions. Another inherent risk in LLM-assisted coding is hallucination: models may generate nonexistent APIs, invalid signatures, fabricated library behavior, or incorrect assumptions about security mechanisms. Although often framed as a functional correctness issue, such hallucinations can also have direct security consequences. For example, a model may invent a sanitization routine, misuse an authentication or cryptographic API, assume that a library enforces checks it does not provide, or produce plausible code that silently violates security requirements [100, 129].

These errors are particularly dangerous because they can appear superficially convincing. Developers may fix syntax or compilation issues while missing deeper semantic errors involving permissions, validation, state handling, or security boundaries. Recent work shows that current models still struggle to detect hallucinated code and to repair it reliably once introduced [67]. In more autonomous or agentic settings, such errors can also propagate across steps when hallucinated dependencies, fabricated tool capabilities, or mistaken execution assumptions are reused as context. Hallucination should therefore be treated not just as a coding nuisance, but as a security-relevant failure mode linking model uncertainty, plausible output, and exploitability.

4.2 Human-Related Security Issues

Beyond the inherent limitations of the models themselves, the human element represents a significant source of security vulnerabilities in LLM-assisted coding. The interaction between the developer and the model, from the initial prompt to the final code review, is a critical stage where misunderstandings, biases, and a lack of security awareness can lead to the introduction of serious flaws.

Insecure Prompt Engineering. User prompts can inadvertently steer LLMs toward insecure code or bypassed safeguards. The security of generated code depends heavily on prompt quality and specificity: vague prompts often yield insecure outputs [45], and models may fail to infer or prioritize risks that are not explicitly stated [46]. Consequently, non-expert users may obtain code that is functionally correct yet vulnerable [104], and even clearly security-sensitive tasks may not trigger warnings or protective measures unless these are explicitly requested [12, 104].

Crafting effective prompts is therefore a non-trivial skill, requiring both clear communication and engineering judgment [38]. Novice users often struggle to specify security requirements precisely enough to elicit secure code [20]. However, simply asking for “secure” code is usually insufficient; zero-shot security directives may still fail to prevent vulnerabilities such as path traversal [45], and generic security boilerplate alone yields little improvement [51]. More detailed security-aware prompts can help, but only when users can accurately identify and describe the relevant requirements [61]. Figure 3 illustrates how underspecified prompts can produce plausible code that omits essential security controls.

Over-Reliance and Reduced Scrutiny. Developers excessively trust LLM outputs, leading to insufficient review of generated code.

Perry et al. [114] indicate that programmers using AI assistants produced less secure code than those working independently in four of five security-focused tasks. Yet these users were more

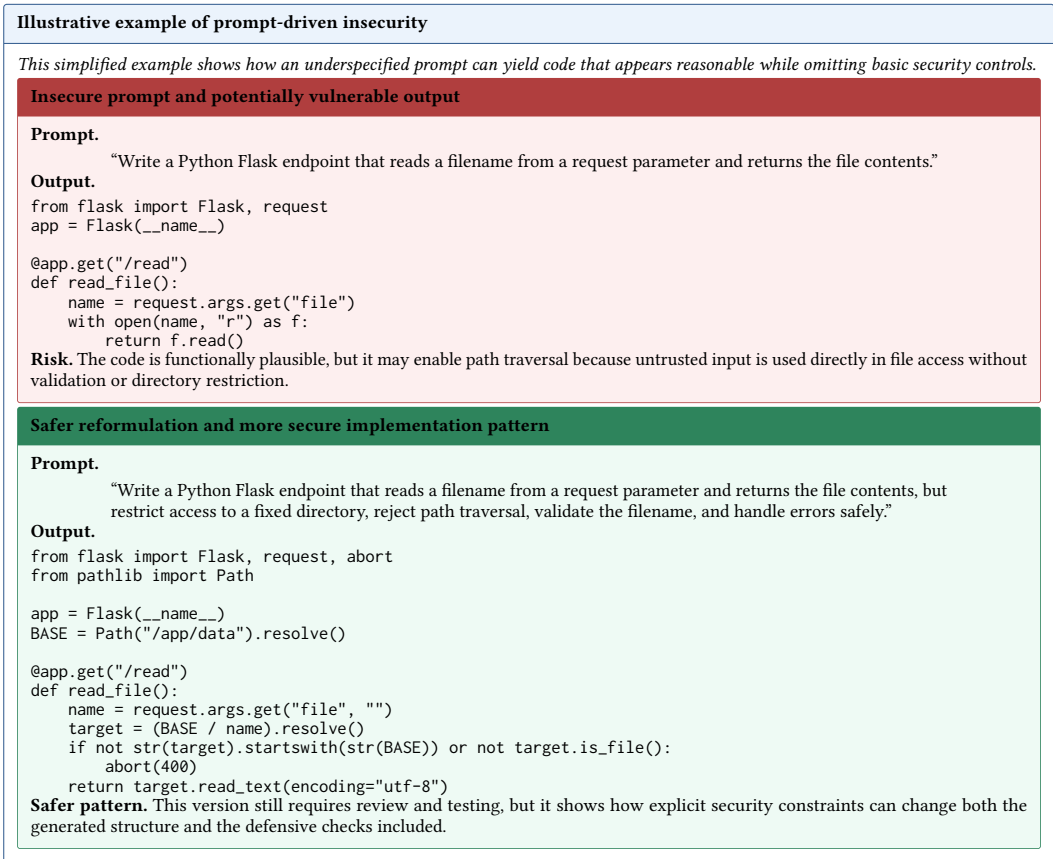


Fig. 3. Illustrative example showing how weak prompt specification can lead to insecure but plausible code, and how a more security-aware reformulation can elicit safer implementation patterns. The prompts were manually written, and the sample code was generated using Gemini 3.1 Pro and subsequently reviewed by the authors for illustrative accuracy.

likely to believe their code was secure, indicating overconfidence. The resulting code often omitted secure library use, proper input sanitisation, and edge-case handling for databases and file systems.

Belozarov et al. [12] caution against over-reliance on AI code generation, arguing that manual review and security testing remain essential. Because LLMs can create a false sense of safety, all generated output requires rigorous inspection. Hamer et al. [49] also warn against blindly trusting external code, whether AI-generated or human-written.

Tihanyi et al. [130] argue that relying on LLMs for production code generation without expert oversight is unsafe. Their evaluation of eight major models found vulnerabilities in at least 63.47% of generated C programs, covering 42 CWEs, including six from MITRE’s Top 25; dereference failures and buffer overflows were the most frequent. Even the best models produced vulnerable code in roughly half of the cases, underscoring the need for rigorous manual validation.

Insufficient Security Training for LLM Usage. Developers with limited security expertise are particularly vulnerable to misusing LLMs. Mohsin et al. [100] show that developer experience strongly affects the security of LLM-assisted coding: inexperienced programmers, especially in complex languages such as C++, often overlook critical practices, leading to vulnerabilities such

as buffer overflows and improper input validation. Limited understanding of prompt security and subtle flaws in LLM output further increase the risk of exploitable code entering production.

Khoury et al. [58] reinforce this concern by simulating novice programmers, finding that ChatGPT produced initially secure code in only 5 of 21 cases without explicit security instructions. The model frequently generated severe vulnerabilities, including SQL injection, path traversal, and insecure handling of sensitive operations such as unhashed passwords, and typically added basic safeguards only when asked explicitly.

Accordingly, software engineering education must adapt as LLMs automate routine coding [59]. Developers need stronger skills in prompt engineering, critical evaluation, and artifact verification, while curricula should emphasize review, testing, validation, and verification, together with professional ethics, responsible AI, and secure development practices [59, 100]. Systematic security testing and review therefore remain essential when using LLM-generated code.

4.3 Attack-Related Security Issues

Beyond inherent weaknesses and user-related errors, LLMs are themselves direct targets for attacks designed to compromise the integrity, confidentiality, and availability of the code they generate. These attacks exploit vulnerabilities in the LLM's architecture, its training data, or its input-processing mechanisms to maliciously influence its behaviour.

Adversarial Attacks and Data Poisoning. A primary attack vector is adversarial input, where carefully crafted prompts bypass an LLM's safety mechanisms [120]. Successful prompt injection can induce harmful code generation, confidential data disclosure, or unintended actions in integrated environments. As in adversarial attacks on image models [37, 143], subtle input modifications can also make coding LLMs misclassify vulnerable code as safe, suggest flawed patches, or miss existing vulnerabilities. More broadly, poisoning the data used for training or fine-tuning can implant backdoors or systemic weaknesses that later reappear in generated code [10, 26, 126].

Zeng et al. [149] demonstrate this risk through HACKODE, which hides adversarial sequences in external code references, often within comments. Although these sequences appear meaningless to humans, they can alter model outputs while leaving the infected examples functionally correct, allowing them to spread through highly ranked online sources and into coding assistants. As a result, models may generate unsafe code from seemingly legitimate inputs, making the attack difficult to detect.

Poisoning attacks are also a serious training-phase threat. He et al. [50] show that poisoned data or malicious fine-tuning can bias code models toward insecure cryptographic recommendations and distinguish these attacks from SVEN, their post-training evaluation method. Schuster et al. [119] further classify poisoning threats into model poisoning, which compromises the model or its supply chain, and data poisoning, which inserts malicious files into training repositories.

Some poisoning attacks inject explicit insecure payloads into training data [26, 107, 119, 144], but such payloads may be detectable by static analysis. To evade this, Aghakhani et al. [2] propose COVERT, which hides malicious content in ignored regions such as comments and docstrings, and TROJANPUZZLE, which teaches the model token substitution patterns without including the full payload directly. When triggered, the model can then reconstruct and emit the malicious payload automatically.

Data Leakage and Confidentiality Breaches. LLM-generated code often contains vulnerabilities that can expose, alter, or destroy sensitive data, thereby compromising confidentiality and sometimes availability. Attacker-induced flaws such as buffer overflows can also enable unauthorised access or modification of critical information [104, 149].

Improper input validation (CWE-20 [80]) and related injection flaws, including SQL injection (CWE-89 [98]), cross-site scripting (CWE-79 [96]), OS command injection (CWE-78 [94]), and

code injection (CWE-94 [99]), are common [44, 46, 116, 131]. These vulnerabilities can enable data theft, tampering, deletion, or even system compromise [104]. A common cause is missing input validation or filtering, for example, failing to sanitize file names and thereby enabling path traversal (CWE-22 [82]) [69, 135].

Memory-safety flaws, such as buffer overflows (CWE-120 [77], CWE-787 [95], CWE-119 [76]), out-of-bounds reads or writes (CWE-125 [78], CWE-787 [95]), null pointer dereferences (CWE-476 [88], CWE-153 [79], CWE-346 [84]), and memory leaks or use-after-free errors (CWE-401 [86], CWE-416 [87]) are also frequent, especially in C/C++ [48, 61, 69, 130]. These may cause crashes, data leakage, or arbitrary code execution, and missing NULL checks after allocation are a recurring weakness [69].

Information exposure vulnerabilities (e.g., CWE-200 [81], CWE-352 [85]) are likewise reported [44, 100]. A common pattern is hard-coded credentials, where models generate passwords or API keys as clear-text strings [46]. Invalid validation can also lead to unsafe operations that leak sensitive information [69].

Further risks include API misuse, such as insecure cryptographic choices (CWE-327 [83], CWE-759 [92]) or unsafe deserialization (CWE-502 [89]), as well as resource leaks [44, 50, 154]. Other reported issues include path traversal, unrestricted file upload, incorrect permissions, unsafe dynamic execution, and infinite loops that may trigger denial-of-service conditions [46, 49, 69, 82, 90, 91, 97, 99, 149].

4.4 Security Implications of Agentic Coding Systems

Early studies often evaluate LLM-assisted coding as a largely standalone task, but recent research and practice are shifting toward *agentic coding systems*: autonomous or semi-autonomous LLM-based systems that can decompose tasks, invoke tools, retrieve repository context, execute tests, revise files, and iterate on intermediate results [24, 70, 106, 153]. This shift changes the threat model from a single-output artifact problem to a multi-step socio-technical workflow problem.

First, agentic systems expand the attack surface because they interact with shells, package managers, testing frameworks, retrieval systems, external APIs, and local development environments [70, 106]. Prompt injection may therefore arise not only from user input, but also from retrieved documentation, repository comments, malicious tool output, poisoned test artifacts, or adversarial external resources [24, 153]. As these components multiply, so do the trust boundaries that must be secured.

Second, agentic systems can amplify errors across steps. Unlike a standalone insecure completion, an early hallucination, false assumption, or unsafe action may cascade by selecting a vulnerable dependency, generating flawed tests, approving an incomplete patch, or overwriting secure code with insecure revisions [24]. Because later stages may reuse earlier outputs as context, the system can recursively propagate its own mistakes.

Third, agentic systems can reduce human visibility and controllability. Developers may see only the final artifact rather than the intermediate reasoning, tool calls, retrieved evidence, and edits that produced it, making it harder to audit vulnerabilities, assign responsibility, or identify whether failures stem from the base model, a tool interaction, a retrieval source, or workflow design [70]. Secure deployment therefore requires not only code inspection but also traceability across the broader execution process.

Finally, agentic coding systems blur the boundary between software security and broader AI safety concerns, including controllability, misuse resistance, and operational governance [106, 153]. Accordingly, evaluating future LLM-assisted coding systems requires workflow-aware, repository-level, and action-oriented assessment rather than only single-turn code generation benchmarks [24, 70]. Taken together, these observations help explain why recurring vulnerability families appear

Table 3. Recurring vulnerability families in LLM-assisted coding studies.

Vul. Family	Representative CWEs	Representative Works	Recurring Pattern
Input validation and injection	CWE-20, 22, 78, 79, 89, 94	[44, 46, 58, 69, 104, 116, 131, 135]	Models often generate functionally plausible code that omits sanitization, boundary checks, or command/query validation.
Memory safety	CWE-119, 120, 125, 416, 476, 787	[48, 61, 69, 130, 131]	Memory-unsafe languages show recurring low-level errors, especially when prompts do not explicitly encode safety constraints.
Weak crypto and insecure dependencies	CWE-327, 328, 759	[50, 57, 100, 111, 118, 123]	Models reuse outdated algorithms, deprecated APIs, and vulnerable dependencies inherited from training corpora.
Incomplete or insecure repair	CWE-dependent	[3, 5, 12, 69, 105, 112, 131]	Generated patches may satisfy local tests while leaving root causes unresolved or introducing new flaws.
Data leakage and exposure	CWE-200 and related patterns	[44, 46, 48–50, 61, 100, 104, 149, 154]	Insecure handling of secrets, credentials, and sensitive data remains a recurrent failure mode.
Hallucination-related semantic failure	Non-CWE-specific	[67, 100, 129]	Fabricated APIs or false assumptions about security behavior can produce plausible yet unsafe code and may propagate across workflows.

across models, languages, prompting settings, and evaluation designs, as summarized in Table 3; this complements Table 2 by shifting attention from broad risk categories to recurrent security weaknesses.

5 Evaluating the Security of LLM-assisted Coding

Assessing the security of LLM-assisted coding remains difficult because current benchmarks and evaluation pipelines capture different aspects of risk [40, 65, 71, 156]. Some focus on functional correctness, others probe known vulnerability patterns, and still others evaluate runtime behavior, repository context, or human review [10, 17, 19, 63, 121]. In addressing **RQ2**, this section examines the main datasets, metrics, and evaluation methods used in the literature, with particular attention to benchmark contamination, limited realism, and cross-framework incomparability.

5.1 Datasets

A crucial component of any evaluation framework is the dataset used for assessment. In the context of LLMs for code generation, these datasets can be broadly divided into those that measure functional correctness and those specifically designed to probe for security vulnerabilities.

5.1.1 Functional Correctness. The initial wave of benchmarks for LLM-based coding tasks focused primarily on functional correctness, whether the generated code produces the correct output for a given set of inputs. A foundational benchmark in this area is HumanEval [18], a collection of hand-written Python programming problems where the model must synthesise a function from its docstring. This benchmark has been extended to enhance its rigour and scope; HumanEval+ [68] augments the original with 80 times more test cases, while HumanEval-X [152] adapts the problems for multiple languages, including C++, Java, and Go.

Austin et al. [6] introduced two complementary datasets for code synthesis: MBPP, with 974 entry-level Python tasks including a hand-verified subset of 426, and MathQA-Python, with 23,914 mathematical word problems adapted from MathQA. MBPP targets imperative algorithmic construction, whereas MathQA-Python emphasizes language understanding through straight-line code generation. Hendrycks et al. [52] proposed APPS, a 10,000-task benchmark for Python code generation that spans introductory to competition difficulty and evaluates functional correctness through test execution.

Du et al. [36] introduced ClassEval, the first benchmark for class-level code generation, with 200 Python tasks requiring complete classes generated from natural language descriptions. Unit-test results show that class-level generation is substantially harder than function-level generation.

For more realistic and domain-specific assessment, DS-1000 provides 1,000 Stack Overflow data-science tasks requiring major Python libraries [60], BigCodeBench offers 1,140 practical Python tasks involving multi-library function use [158], and CoderEval evaluates 460 repository-level Python and Java tasks requiring full-project context [148]. Together, these benchmarks show that practical and context-dependent generation remains challenging for current models.

5.1.2 Security-Specific. As the functional limitations of LLMs became better understood, a new class of benchmarks emerged to address the critical issue of security in the code they generate. These security-specific datasets are not designed to evaluate if the code works, but rather if it is safe and free from known vulnerabilities.

Early evaluation methods relied on curated prompt sets to test for known insecure patterns. For instance, SecurityEval [124] provides 130 Python prompts mapped to 75 Common Weakness Enumeration (CWE) categories. These samples directly assess whether a model's generated output introduces specific security vulnerabilities. Similarly, LLMSecEval[134] offers 150 natural language prompts to assess code generation in vulnerability-prone scenarios. Covering 18 of MITRE's Top 25 CWEs [75], these prompts were derived from secure code snippets and manually refined to be language-agnostic. To support robust comparative evaluations, the dataset includes a secure, CodeQL-verified [25] Python baseline for each prompt.

Wang et al. [138] proposed CodeSecEval for evaluating both code generation and repair. The dataset contains 180 Python samples spanning 44 critical vulnerability types, each with executable secure and insecure versions and test cases for automated functional and security assessment. It comprises SecEvalBase, which extends SecurityEval [124], and SecEvalPlus, which targets the most common and severe weaknesses.

FormAI [132] instead combines large-scale code generation with formal verification. It contains 112,000 C programs generated by GPT-3.5-turbo and labeled with ESBMC, which identifies both vulnerability type and location. The study reported vulnerabilities in 51.24% of generated programs.

Tihanyi et al. [130] later introduced FormAI-v2, a larger set of 331,000 C programs generated by models such as GPT-4 and Gemini-1.0-Pro. Evaluating code from neutral prompts, they found with ESBMC that at least 63.47% of programs were vulnerable, suggesting that such risks are inherent to current code-generation models.

5.2 Metrics

To quantify the performance of LLMs in coding tasks, researchers employ a range of metrics. These can be divided into two main categories: those that assess the functional correctness of the generated code and those specifically designed to measure its security posture.

5.2.1 Functional correctness. **Pass@k:** A widely adopted metric for functional correctness is pass@k, which measures the probability that at least one of k code samples generated for a given task successfully passes all of its unit tests [18, 20, 30, 39, 43, 73, 74, 123, 127, 138]. This metric

provides an indication of the likelihood that a developer will find a correct solution within a set of k generated options [123, 138]. To ensure statistical robustness and reduce variance in results, an unbiased estimator of $\text{pass}@k$ is commonly utilised in evaluations [50, 133, 138]. Some frameworks have also introduced variations, such as the ranked $\text{pass}@k$ ($\text{pass}@k@m$) used by TiCoder [39], which denotes correctness after a specified number of user queries and subsequent pruning of options.

Compilation and Unit Tests: At a fundamental level, functional correctness is determined by whether the generated code compiles successfully and subsequently passes a predefined suite of unit tests [18, 43, 56, 68]. The percentage of code snippets that are merely compilable can also serve as a basic indicator of a model's output quality.

Text Similarity Metrics: Metrics that assess the token-level similarity between generated code and a reference solution—such as BLEU, ROUGE-L, CodeBLEU, METEOR, ChrF, and RUBY—have also been applied [18, 117]. However, these metrics are often considered unreliable for evaluating code. Functionally equivalent programs can exhibit significant syntactic variations, meaning a high similarity score does not necessarily indicate functional correctness, nor does a low score confirm its absence [133].

Task-Specific Criteria: Generic metrics are often inadequate for specialised tasks, which require tailored evaluation criteria. For example, path-planning code is assessed by whether it finds valid paths and satisfies task constraints [20], while hallucination detection uses dedicated metrics such as Valid Rate (VR), hallucination existence accuracy (Accrec), and hallucination type accuracy (Acctype) [67].

5.2.2 Security-Specific. Security Rate (SVEN-SR): An early metric proposed for security is the SVEN-SR, defined as the percentage of secure programs among all unique and compilable generated programs [43]. While this metric provides a baseline indication of security, it has been criticised for potentially overestimating a model's practical security performance. By not accounting for functional correctness, a trivially safe but entirely useless piece of code could achieve a high SVEN-SR score.

Secure-pass@k: To provide a more realistic assessment, the Secure-pass@ k metric measures the probability of generating code that is both secure and functionally correct within k attempts [30, 43]. It is considered a more practical measure of an LLM's secure code generation capability because it integrates functional correctness as an essential prerequisite for a successful outcome.

Secure@kpass: A related but distinct metric is Secure@ $k\text{pass}$, which assesses how likely any one out of k functionally correct generations is also secure [43]. This helps to decouple the security evaluation from the model's ability to generate working code in the first place.

func-sec@k: Similarly, the func-sec@ k metric evaluates functionality and security simultaneously, indicating the likelihood that an LLM-generated implementation (out of k options) is both correct and secure [113]. This metric is particularly effective at highlighting the significant gap that often exists between a model's ability to generate functional code and its ability to generate code that is both functional and secure.

vulnerable@k: Approaching the problem from the opposite perspective, the vulnerable@ k metric measures the probability that at least one code snippet out of k generated samples contains a detectable vulnerability [123]. In this case, a lower score for vulnerable@ k signifies a better performance in generating secure code.

False Refusal Rate (FRR): To quantify the inherent safety-utility tradeoff in models that have been fine-tuned for safety, the FRR measures the percentage of benign prompts that an LLM incorrectly refuses because it mistakes them for unsafe or malicious requests [13].

Table 4. Representative families of evaluation frameworks for LLM-assisted coding security.

Framework Family	Representative Works	Primary Oracle	Key Use&limitation
Function-level correctness	[6, 18, 52]	Unit tests	Useful for baseline synthesis ability, but weak in security realism and contextual coverage.
Security benchmarks	[123, 124, 134, 138]	Curated prompts + static rules	Effective for probing known vulnerability classes, but limited in runtime realism and sensitive to analyzer imprecision.
Dynamic/exploit-driven security	[30, 106, 113, 137, 146]	Runtime tests, fuzzing, exploit execution	Better reflects exploitability in practice, but depends heavily on test quality, environment fidelity, and setup cost.
Repository/context-aware evaluation	[64, 148, 151]	Project tests + contextual reasoning	Captures integration and cross-file failures, but is harder to standardize and compare across models.
Human-centered/manual evaluation	[39, 74, 114]	Expert review/user study	Reveals review burden, usability, and trust effects, but is costly, subjective, and hard to scale.

Static Security Analysis Tools: Underlying many of these security metrics is the use of static security analysis tools. Tools such as CodeQL, SonarQube, Bandit, ESLint, and Infer are widely employed to automatically detect security flaws and known vulnerabilities directly within the source code [14, 32, 43, 44, 47, 56, 106, 111, 113, 123, 134]. These tools are instrumental in identifying specific vulnerability classes and flagging security hotspots. It is important to note, however, that static analysers are known to suffer from imprecision, including both false positives and false negatives [64, 138].

5.3 Evaluation Tools

Security evaluations use complementary oracles. Static analyzers scale to many samples and vulnerability classes but have false positives, false negatives, and limited access to runtime or repository context [47, 64, 123, 138]. Dynamic tests, fuzzing, and exploit execution provide behavioral evidence, but their conclusions depend on test coverage, reproducible environments, and the fidelity of the attack scenario [30, 106, 113, 137, 146]. Repository-aware evaluation exposes integration failures, while expert or developer review captures aspects of usability and oversight that automated tests cannot measure [39, 64, 74, 148, 151]. For this reason, results from different tools should be interpreted in terms of their task, oracle, context, and security objective; a fuller catalogue of the surveyed frameworks appears in Supplementary Section 2.

5.4 Cross-Framework Comparison and Evaluation Gaps

The surveyed evaluation frameworks differ not only in implementation but also in what they treat as evidence of security. Some benchmarks focus on function-level correctness under predefined tests, while others probe for vulnerability patterns, repository integration, exploitability, or the interaction between human reviewers and generated code. As a result, two studies can evaluate the same model yet reach different conclusions because they observe different failure surfaces rather than because one of them is necessarily incorrect.

A useful way to compare the literature is along six dimensions: (i) task granularity, ranging from isolated functions to repository-level workflows; (ii) oracle type, including unit tests, static analyzers, exploit execution, and manual review; (iii) evaluation objective, such as correctness,

vulnerability detection, secure repair, or attack helpfulness; (iv) context availability, from prompt-only generation to cross-file and temporal repository context; (v) security realism, ranging from rule-based checks to exploit-driven validation; and (vi) human involvement, from fully automated pipelines to expert review. These dimensions help explain why benchmark results are often not directly comparable. Table 4 summarizes these framework families and highlights the distinct trade-offs that make cross-study results difficult to compare directly.

Static-analysis-centered frameworks such as SALLM [123], CodeLMsec [47], and parts of CodeSecEval [138] scale well and provide broad CWE coverage, but they inherit the incompleteness and context limitations of the underlying analyzers. Dynamic and exploit-driven approaches such as SecRepoBench [30], CWEVAL [113], AutoSafeCoder [106], and BAXBENCH [137] better capture runtime failures and real exploitability, but they depend heavily on test quality, environment setup, and scenario design. Repository-level and context-aware frameworks such as CORRECT [64], CoderEval [148], and HumanEvo [151] reveal integration failures that function-level benchmarks often miss, yet they are harder to standardize and compare across models. Manual and human-centered evaluations capture readability, reviewability, trust effects, and workflow realism, but they are expensive and difficult to scale [39, 74, 114].

Taken together, the literature suggests that no single framework is sufficient for evaluating secure LLM-assisted coding. Static analysis is useful for breadth, dynamic testing is useful for behavioral evidence, repository-level benchmarks are necessary for contextual realism, and human-centered evaluation is necessary for deployment realism. The main gap in the current landscape is therefore not simply the lack of benchmarks, but the lack of an integrated evaluation pipeline that combines these complementary signals into a more trustworthy picture of security performance.

6 Human Oversight and Human-Factor Risks

The integration of LLMs into coding tasks has significantly advanced software development, yet their inherent limitations necessitate human involvement to ensure code quality, correctness, and security [33, 55, 70, 129]. Human oversight is often treated as a key safeguard, whether through direct review, feedback loops, or human-in-the-loop validation [41, 62, 108, 142, 153]. However, human participation is not uniformly protective: the same workflows that enable developers to detect flaws and clarify intent can also introduce vulnerabilities through over-trust, automation bias, weak prompt engineering, and insufficient review practices.

Accordingly, this section treats human-AI interaction as a dual-role factor in secure LLM-assisted coding. Human developers and reviewers remain critical for interpreting requirements, auditing outputs, and identifying flaws that automated systems miss, yet empirical evidence also shows that LLMs can reduce scrutiny, create false confidence, and distort responsibility in ways that degrade security [49, 58, 100, 104]. To address **RQ3**, we therefore review the literature not only on human oversight as a mitigation mechanism, but also on human-AI collaboration as a source of security and accountability risk.

6.1 Limitation of Automated Feedback

A common approach to improving LLM-generated code is to design models capable of providing their own feedback and attempting self-correction, a process often referred to as self-repair or self-criticism [33, 108]. This typically involves the model generating an initial code snippet, executing it against test cases, and then using the resulting error messages or diagnostics as execution feedback to identify and rectify its own errors [28, 153]. For instance, Agents4PLC [70] incorporates a dedicated debugging agent that provides fixing advice back to the coding agent when code fails, drawing inspiration from existing automated code repair methods. It can also leverage previously generated patches in a self-RAG context during the debugging phase to improve repair effectiveness.

Similarly, frameworks like OpenCodeInterpreter [153] involve prompting LLMs (e.g., GPT-4) to intentionally generate incorrect code, then using the resulting error messages as cues for correction. This process mimics a real-life debugging cycle. StepCoder [34] also uses reinforcement learning with compiler feedback to guide the exploration of the output space for LLMs, aiming to improve code generation quality.

Despite recent progress, models lack the formal correctness guarantees and specialized support required for industrial applications. In Programmable Logic Controller (PLC) programming, for example, they often struggle to satisfy the rigorous logical and safety constraints essential for industrial control systems [70]. Moreover, LLMs have difficulty executing complex coding tasks and frequently produce suboptimal algorithms, resulting in issues like timeout errors [21, 33, 108]. Even when directed to use efficient solutions like dynamic programming, LLMs can still create inefficient code [115].

Additionally, LLMs can misunderstand programming specifications, which leads to generated code that does not align with the user's original intent. When the generated code deviates significantly from the ground truth, the coarse-grained error messages produced during automated feedback cycles often prove insufficient for effective improvement [128].

When LLMs are employed in coding assistance, they might inadvertently bring about new risks. For instance, feedback-driven efforts to rectify identified security flaws may result in the LLM inserting defects into previously flawless code [22]. Additionally, code produced by LLMs frequently omits defensive programming techniques, rendering it less secure and susceptible to vulnerabilities such as buffer and integer overflows [66]. Furthermore, this machine-generated code tends to be more intricate per line compared to code crafted by humans, which might complicate subsequent maintenance and analysis [33]. The perceived advantages of LLMs can engender an unfounded sense of productivity and security, given that LLMs might create flawed or weak code and misleading recommendations. Excessive dependence on these models can result in time wastage and suboptimal tool utilization [129].

Alongside security issues, LLM-assisted coding tasks also face computational challenges. The cost of enabling self-repair might lead to only slight improvements, which are inconsistent across various tasks or datasets and can occasionally be negligible [109]. This is mainly due to the limited capacity of the model to introspect and reliably generate effective feedback on its own mistakes [21, 109].

Finally, LLMs show decreased efficiency in processing feedback aimed at rectifying multiple errors at once, indicating that such feedback might be more effective if it is broken down into simpler steps to enhance model performance [16].

6.2 Incorporating Human Feedback

In addressing the limitations inherent in code generation by LLMs, human feedback is essential. This is commonly implemented through methods such as Reinforcement Learning from Human Feedback (RLHF) and Human-in-the-Loop (HITL) systems [55, 110, 141]. Such strategies employ human insight, proficiency, and subtle comprehension to steer LLMs toward producing more accurate, secure, and coherent code. It's crucial that users verify their code's security and integrity [44, 111], while developers should remain alert and exercise caution with generative AI [135]. The reliability of LLM-generated code, including suggested improvements, is typically variable and necessitates human oversight to ensure consistency [12, 38]. Developers are warned against placing complete trust in either AI-generated or AI-revised code fragments [49].

At the same time, incorporating human feedback does not automatically eliminate risk. The security benefits of RLHF, HITL, and related feedback-driven workflows depend heavily on reviewer expertise, the quality and consistency of the feedback provided, and the surrounding workflow design [12, 41, 55, 62, 100, 129]. Poorly calibrated trust, superficial review, or misplaced confidence

in AI-revised outputs can still allow insecure code to pass through development pipelines, while responsibility for failures may become blurred across model designers, tool providers, and human reviewers [12, 49, 58, 114]. Human feedback should therefore be understood not as a guaranteed safeguard, but as a socio-technical control whose effectiveness depends on how the collaboration is structured and audited.

Reinforcement Learning from Human Feedback (RLHF). A prominent method for aligning model behaviour is RLHF, a technique that fine-tunes LLMs using human preferences as a reward signal [7, 15, 110, 139]. The standard pipeline involves supervised fine-tuning on human demonstrations, followed by the training of a reward model using human-ranked output comparisons. In the final stage, the model is refined via Proximal Policy Optimization (PPO) [110], using the reward model as a signal to guide the model toward preferred responses.

RLHF has significantly improved the performance of models such as Codex and GPT across various tasks [110, 141]. However, the method is constrained by the high cost and inconsistency of human feedback. Relying on individual evaluators introduces subjectivity and bias, which can be counterproductive in uncertain domains where input may be unreliable. Furthermore, the substantial labor required and the limited scale of human-labeled datasets often restrict the efficiency and measurable impact of this alignment strategy.

Crowd-Sourced RLHF (cRLHF). To mitigate the inherent biases and constraints associated with individual evaluators, Wong et al. [141] introduced cRLHF, a framework that mitigates the biases of individual evaluators by aggregating feedback from multiple annotators. Using Bayesian inference, the system combines multi-source assessments—including line-by-line accuracy—into a "coordinated score" between 0 and 1. This score serves as a direct reward for model refinement, simplifying the reward modeling process while ensuring outputs align with diverse human expectations. Empirical results demonstrate that this collaborative approach effectively reduces model bias and enhances the quality of generated code.

Crowdsourced helps reduce bias and improve evaluation objectivity by aggregating diverse human inputs. However, its effectiveness is often constrained by small sample sizes and the logistical challenge of recruiting expert annotators for large-scale studies [141].

Human-in-the-Loop (HITL) Another crucial approach is HITL testing, which integrates human oversight at critical junctures to supplement automated evaluations [55]. This approach addresses the non-deterministic nature of model outputs by using automated tools to flag high-impact or low-confidence results for expert review. Human feedback on the accuracy and safety of these outputs is then used to refine prompts and triage criteria. This iterative process, which may include RLHF for response optimization [110], leverages models to generate adversarial test cases and summarize large datasets, ultimately improving both the reliability and precision of the system.

HITL improves the reliability and safety of model-integrated systems, especially in high-stakes domains, by combining large-scale automated analysis with targeted human judgment [55]. Human reviewers can catch nuanced errors, biases, and contextual issues that automated tools often miss, although the approach remains selective and is limited by the scale of expert feedback available.

Conversational Automated Program Repair (Conversational APR). Conversational APR iteratively generates patches and incorporates validation feedback, such as test results, outputs, and error messages, to refine subsequent repairs [142]. By combining prior patches with feedback in each round, it helps LLMs avoid repeating mistakes and better understand program semantics. Successful patches can also be varied through additional conversational turns.

This approach can substantially improve repair quality by leveraging natural-language hints and human clarification; indeed, a brief human correction may be more effective than greatly increasing the number of generated samples. However, the method can be computationally expensive because it often requires many LLM interactions [128].

Conversational Semi-Automatic Repair Framework (Cref). This system employs LLMs specifically designed to assist programming tutor [145]. It employs interactive processes involving tutor insights, solution narratives, and test cases that fail. Cref handles everything in one dialogue session and advances through specific stages: initially supplying LLMs with programming problems, erroneous code, and tutor assistance; if the code is still erroneous, it revisits all prior data and solution outlines; and if further correction is needed, it utilizes historic entries alongside failing test cases of the most recent code fix. This methodology notably enhances code generation efficiency via natural language prompts. Conversational APR [142] strategically circumvents the replication of past errors in patches and effectively uses previously overlooked test case details. Cref has achieved substantial progress in Average-5 scores for code repair, cutting debugging times by 71.2% and costs by 69.9% in real-world corporate scenarios. It also boosts tutoring success and enriches students' learning experiences. The achievements of Cref indicate its potential for wider use cases, such as automating code review processes by refining code based on reviewer feedback.

Socratic Human Feedback (SoHF). SoHF [21] is a form of expert feedback in which programmers guide LLMs through iterative natural-language prompts, much like a tutoring session. Using strategies such as test reiteration, error highlighting, and requirement clarification, experts provide step-by-step guidance across multiple debugging rounds. This approach can substantially improve performance: with expert Socratic guidance, LLMs were able to solve up to 74% of problems they initially failed. However, existing studies focus mainly on expert programmers, and some do not account for the time required for debugging, leaving its broader applicability unclear.

Imitation Learning from Language Feedback (ILF). Chen et al. [16] introduced ILF, a framework that incorporates natural language feedback during the training phase rather than solely at inference. The process begins by identifying code that fails unit tests and obtaining human-written feedback for those errors. A specialized refinement model, π_{Refine} , uses this feedback to generate corrected versions of the code, which are then used to fine-tune the primary model. This iterative approach, which effectively minimizes the KL divergence from the target distribution, requires minimal human-authored feedback and remains independent of such feedback during testing. Evaluation on the MBPP benchmark showed that ILF increased the pass@1 rate of a 6.1B parameter model by 38% relatively and 10% absolute. Despite these gains, the framework is limited by the high cost of human feedback and the difficulty π_{Refine} faces when addressing multiple, simultaneous errors.

Expert review is vital for auditing and refining code produced by automated systems to ensure functional correctness and security [28, 70, 141]. Professional developers are uniquely equipped to implement defensive programming and guardrails that automated tools frequently overlook, particularly in security-critical environments [22]. By leveraging deep domain knowledge and context awareness, human reviewers can resolve ambiguous requirements and adhere to industrial standards more reliably than current computational models [66]. Despite being resource-intensive and difficult to scale, manual oversight remains a necessary safeguard against the structural complexity and subtle vulnerabilities common in machine-authored software [22, 155].

Overall, human expertise continues to be crucial for addressing the shortcomings of LLMs, particularly in handling intricate, security-sensitive, or unclear tasks where LLMs face challenges related to efficiency, ambiguous requirements, and the creation of robust, secure, and maintainable code [22, 66, 145]. It is most effective to regard LLMs as an enhancement to human programmers rather than a replacement. The cooperation between human skills and LLM capabilities, especially through iterative and focused feedback, is essential for the progress of AI-supported programming [66].

Taken together, the literature suggests that human oversight is both indispensable and fragile in LLM-assisted coding. Human expertise remains essential for clarifying intent, evaluating security-sensitive outputs, and correcting model failures, but the introduction of LLMs can also distort

developer behavior through over-reliance, reduced scrutiny, and miscalibrated trust [12, 49, 58, 100, 114, 130]. The practical challenge is therefore not simply to keep a human in the loop, but to design human-AI workflows in which oversight is informed, accountable, and appropriately matched to the risks of the task.

7 Open Challenges and Research Directions

The reviewed literature points to three persistent challenges for secure LLM-assisted coding: a mismatch between probabilistic generation and deterministic security requirements, unreliable or incomplete evaluation pipelines, and fragile human-AI collaboration shaped by over-trust, weak prompting, and limited accountability. This section synthesizes these challenges into a research and deployment agenda, then outlines governance, practical recommendations, and broader implications for academia and industry.

7.1 Core Challenges

Our study indicates that the security concerns associated with LLM-assisted coding do not merely consist of isolated bugs but rather merge into three interrelated, major meta-challenges. These three vulnerabilities form the core of the issue and require a comprehensive approach for resolution.

7.1.1 Probabilistic Models vs. Deterministic Security. The core issue is a fundamental discrepancy between the function of LLMs and the demands of secure software development. LLMs operate as probabilistic, pattern-driven models designed to predict the next probable token based on the statistical trends found in their training data. They do not possess a profound, semantic comprehension of security principles, threat vectors, or the intricate context that differentiates secure from insecure code. Their key aim is functional plausibility and fluency in language, which means they aim to generate working code without an inherent focus on security vulnerabilities.

This fundamental limitation manifests through several significant issues. Firstly, LLMs are trained on extensive publicly available code, such as the open-source collections on GitHub, which tend to contain vulnerabilities, outdated dependencies, and insecure coding standards. As a result, the models learn to emulate these issues. Research repeatedly shows that a significant proportion of code produced by LLMs is inherently insecure; for example, studies reveal that around 40% of snippets from models such as GitHub Copilot contain vulnerabilities. This happens because the model's objective is to replicate patterns, and insecure patterns are abundant in the training data. Secondly, when LLMs are used for APR and patching vulnerabilities, they may produce corrections that seem functionally sound for a limited set of tests but remain vulnerable to more advanced or unforeseen threats. The patch may fix the current issue but inadvertently create a new, subtler vulnerability or only partially mitigate the risk, leaving it exploitable. Third, LLMs often suggest or use deprecated libraries, outdated APIs, or cryptographically vulnerable algorithms because such insecure patterns remain within their training sets and are suggested without recognising their compromised state.

Although the common explanation often blames bad training data, a more detailed analysis uncovers a deeper architectural flaw. The central problem is a semantic gap between statistical association and logical deduction. LLMs, as designed, do not reason about security attributes; they determine statistical probabilities. Even with security-focused prompts, models struggle because they are not equipped for abstract reasoning about possible threat vectors or security invariants. This indicates that merely cleaning the training data, while necessary, will not fully solve the problem. It is similar to teaching someone to write grammatically by exposing them to a large library without explicitly teaching grammatical rules. They might mimic style and common expressions, but cannot reason from grammatical fundamentals. Thus, to achieve genuinely secure code generation requires

more than just data refinement; it points to a necessity for profound changes in model design, training goals, or the incorporation of symbolic reasoning abilities.

7.1.2 The Unreliability of Ground Truth. The second significant obstacle is the inadequacy of current methods for assessing the security of code generated by LLMs. This leads to an evaluation crisis, where benchmark scores can be deceptive, potentially inflating a model's genuine security capabilities and creating an illusion of progress. Without reliable measurement, improvements cannot be accurately made. This crisis stems from various shortcomings in current evaluation practices. Firstly, many primary benchmarks for code generation, such as HumanEval [18], focus on functional correctness for small, self-contained programming tasks. However, these benchmarks often have a limited range and size, insufficient for detecting subtle logical mistakes or complex security weaknesses, resulting in exaggerated performance metrics that don't represent true robustness. Besides, the security evaluation of code heavily relies on SAST tools like CodeQL [25] and Bandit [8], which are known to be neither sound nor comprehensive, displaying considerable rates of false positives and false negatives. Moreover, these tools typically assess code in isolation, lacking the broader repository context (like data flows across multiple files) necessary to uncover many intricate real-world vulnerabilities. Although dynamic analysis can surpass static analysis in certain areas, it is inherently limited by the quality and breadth of the test cases employed. Most benchmarks fail to mimic the complexity of real-world software development, where newly created code must be correctly and securely integrated into a large, existing, and complex repository.

There is a clear progression in evaluation paradigms, shifting the primary question from "Does the code work?" to "Can the code be exploited?". Initial benchmarks like HumanEval [18] and MBPP [6] concentrated on functional correctness, utilising a cooperative set of predefined inputs. In contrast, a new wave of benchmarks is surfacing that adopts a more adversarial approach. Frameworks such as SecRepoBench [30], CodeLMSec [47], and BAXBENCH [137] are crafted to test vulnerabilities under more authentic conditions. They focus on repository-level context, dynamic execution of exploits, and resilience against adversarial inputs. This transition is significant as it redefines a good model not merely as one that passes tests but as one that withstands attempts at compromise. The future of effective LLM security evaluation lies in advancing and utilising these more demanding, adversarial, and context-sensitive benchmarking frameworks.

7.1.3 Human-AI Trust-Vulnerability. The third significant challenge emerges at the complex intersection where human developers engage with AI assistants. While human oversight is generally considered the most reliable defence against the weaknesses of LLMs, it also presents significant vulnerabilities. This forms a trust-vulnerability nexus, where cognitive biases, different expertise levels, and misaligned trust can compromise the security of the entire development process. This connection presents itself in several concerning ways. First, a dangerous bias has surfaced where developers, especially those using AI coding assistants, generate code with lower security standards, even while being overly confident about its safety. This situation generates a misguided sense of security, which diminishes thorough examination and leads to the premature integration of flawed or insecure code into operational systems. Furthermore, the security of output from LLMs is largely dependent on the user's skill in creating precise, security-conscious prompts and critically assessing the generated code. Inexperienced programmers or developers without specific security training are especially likely to create vague prompts and accept superficially suitable but insecure code, thereby adding new vulnerabilities to the software supply chain. Additionally, in the realm of software engineering, the notion of trust in LLMs is currently vague, inconsistently evaluated, and poorly comprehended. This leads to a precarious miscalibration, where a developer's trust level does not match the model's actual effectiveness for a task. This could lead to either over-trust,

resulting in uncritical acceptance of flawed code, or under-trust, inhibiting the use and advantages of the technology.

The common advice of simply training developers more is necessary, but not enough to tackle this challenge. While it is crucial to educate on security, the main issue is not merely a lack of knowledge but one of human-computer interaction (HCI). The essential challenge is around calibrating trust. Developers need efficient ways to discern when and how much to trust the LLM's output, as well as what aspects require closer examination. This shifts the focus from merely training to navigating a complex design challenge. Future research and tool development should aim to build Human-AI collaboration systems that actively aid in this calibration. Potential solutions include AI systems that communicate their uncertainty, provide verifiable evidence for their code suggestions, or automatically flag high-risk code segments that demand mandatory, thorough human assessment. This approach seeks to shift the human role from a passive, potentially susceptible reviewer to an empowered, critical, and effective guardian of code quality and security.

7.2 A Research Roadmap for Trustworthy LLM-Assisted Coding

Tackling the triad of challenges presented earlier demands a collaborative, cross-disciplinary research approach. We organize this roadmap around three core pillars: constructing inherently secure models, developing more reliable and realistic security benchmarks, and improving the calibration of human-AI collaboration. A fourth, cross-cutting requirement (accountability, auditability, and governance) is discussed separately in Section 7.3 because it conditions how progress in all three pillars can be translated into trustworthy real-world deployment.

7.2.1 Towards Inherently Secure Models. The key aim is to evolve from a reactive, post-incident security stance towards developing LLMs that are inherently secure. Achieving this goal involves integrating security principles into the model's fundamental architecture and training frameworks, rather than considering security as an afterthought. Security-aware architectural design and training are crucial for upcoming research, directing focus towards innovative model structures and training goals that surpass mere next-token forecasting. The prevailing approach, which chiefly values functional accuracy and fluency, falls short for applications critical to security. Future inquiries should explore incorporating security-driven loss functions that specifically discourage code creation with established vulnerability patterns (CWEs). Another promising path is crafting hybrid architectures that merge the pattern-detection capabilities of neural networks with symbolic reasoning elements adept at grasping and imposing security properties and constants.

Advanced Security-Oriented Fine-Tuning provides a practical solution, given the formidable costs associated with initiating foundation model pre-training. Techniques for Parameter-Efficient Fine-Tuning (PEFT), like Low-Rank Adaptation (LoRA), have proven effective for tailoring pre-trained models to secure code generation without the extensive computational demands of full retraining. A pivotal area of research focuses on developing and curating extensive, high-quality datasets featuring confirmed vulnerability corrections to instill secure coding norms in models. Additional study is warranted to comprehend how varying granularities of data—such as file-level, function-level, or detailed block-level modifications—affect the security of the refined model's outputs. A notable concern is safety degradation, where fine-tuning a model even with benign, task-specific data may unintentionally jeopardize its overall safety alignments. This underlines the necessity for Safety-Aware Fine-Tuning (SAFT) frameworks designed to automatically identify and exclude potentially harmful or vulnerability-inducing data from the refinement set or to guide gradient updates away from hazardous parameter territories via safety-aware probes.

Formal Methods for Verifiable Security and Correctness represent the most thorough approach to ensuring software security through formal validation. Although directly producing formally

verified code via LLMs is currently unsuitable except for the simplest programs, a highly promising area merges LLMs with formal methods tools. In this context, the LLM serves as a verification assistant rather than the definitive code author. For instance, frameworks like Marmaragan [27] utilise an LLM to produce formal annotations, such as pre- and post-conditions, loop invariants, and contracts, for existing code in languages like SPARK. These annotations are then subjected to checks by a dedicated formal verifier. The LEMUR calculus lays down a theoretical base for this interaction, conceptualising a coherent loop where the LLM suggests properties and an automated reasoner seeks to validate them. Similarly, LLMLift employs an LLM to generate program invariants necessary for proving the equivalence of code translated between languages. This hybrid approach, blending verifier-in-the-loop, significantly harnesses the pattern-recognition and creative-generation capacities of LLMs while rooting them in the mathematical exactness of formal verification, paving the way for constructing more reliable systems.

7.2.2 Next Generation of Security Benchmarks. To advance and monitor progress effectively within the community, it is essential to develop a new generation of evaluation benchmarks that are more trustworthy, realistic, and thorough than those currently widespread.

Holistic, Contextual, and Repository-Level Evaluation: The field must move decisively beyond evaluating isolated, function-level code snippets created in isolation. Future benchmarks should be designed at the repository level to be impactful. This requires furnishing the LLM with the relevant context—such as dependency structures, data-flow across files, and existing API definitions—needed to produce code that integrates accurately and securely into a larger, pre-existing codebase, not just correct in isolation. SecRepoBench [30] is an innovative initiative in this aspect, formulating code completion tasks from actual vulnerability patches in C/C++ projects and employing the projects' own unit test suites for validation. The HumanEvo [151] benchmark enhances this by offering a historically accurate repository context, restoring a project to its state just before a target function was initially developed, thereby creating a more realistic evaluation scenario.

Dynamic and Exploit-Driven Security Assessment: Sole reliance on static analysis to identify vulnerabilities is inadequate due to its inherent lack of precision. Future security evaluations must be dynamic, monitoring code during execution, and, where feasible, driven by the execution of real exploits.

Benchmarking the Human-AI System: A crucial step for future evaluation research is to transition from assessing the AI alone to benchmarking the performance of the entire human-AI system. This would entail devising evaluations that mimic a real-world developer workflow. Metrics could encompass the security of code generated by a developer with an AI assistant, the time and effort needed for a human reviewer to identify and amend AI-induced flaws, or the efficiency of AI-aided review tools in enhancing a developer's security performance. This comprehensive approach would deliver a more accurate depiction of these tools' impact on software security.

7.2.3 Calibrating Human-AI Collaboration. Given that human oversight will remain an essential component of secure software development for the foreseeable future, research must focus on making this collaborative process as effective and secure as possible. The goal is to transform the human from a potential point of failure into an empowered and critical partner.

Modelling and Measuring Trust to Mitigate Over-reliance: The well-documented risk of automation bias and developer over-trust is a primary human-related threat in LLM-assisted coding. To counter this, research drawing heavily from the field of Human-Computer Interaction (HCI) is needed to formally define, model, and measure the concept of trust in the specific context of software engineering. The objective is not simply to maximise trust, but to develop systems that can actively calibrate a user's trust to align with the model's actual capabilities and reliability for a given task. This could be achieved through novel user interfaces that visualise a model's uncertainty

about its own output, provide verifiable evidence for its suggestions, or automatically flag high-risk or low-confidence code segments that require mandatory, heightened human scrutiny.

Developing Tools for Secure Prompting and AI-Assisted Review: The quality of the interaction between the developer and the AI is paramount. Research should focus on creating tools and interaction paradigms that facilitate more secure and effective collaboration. Promising directions include:

- **Interactive and Conversational Repair:** Frameworks that use a dialogue-based process of iterative feedback and refinement have shown significant promise. Systems like Cref [145] and conversational APR [142] allow a developer to guide the LLM toward a correct and secure solution through a natural conversation, providing error messages or clarifications that help the model understand the program's semantics and avoid repeating mistakes.
- **Guided Reasoning and Socratic Feedback:** Techniques like Socratic Human Feedback (SoHF) [21], where an expert programmer guides the LLM with a series of probing questions and step-by-step debugging hints, can enable models to solve complex problems that they would fail at on their own. This approach turns the interaction into a collaborative problem-solving session rather than a simple request-response transaction.
- **AI-Powered Review Tools:** The development of tools that integrate LLM-based vulnerability analysis directly into the IDE can augment the developer's abilities. Tools like CODE-GUARDIAN [35] have been shown to make developers both faster and more accurate at identifying security flaws in code.

7.3 Accountability, Auditability, and Governance

A recurring theme across the literature is that the risks of LLM-assisted coding cannot be reduced to model quality alone [23, 24, 55, 62, 102, 125, 136]. Even if a model is improved through better data curation, safer decoding, or stronger evaluation pipelines, real-world deployment still raises unresolved questions of responsibility, traceability, and governance [22, 23, 55]. These issues become especially acute when insecure code is generated collaboratively by humans and AI systems or when agentic systems perform multi-step actions with limited human intervention [23, 62]. Accordingly, accountability, auditability, and governance should be treated not as peripheral policy concerns, but as cross-cutting conditions for the trustworthy deployment of LLM-assisted coding systems.

Responsibility for insecure outputs. A fundamental challenge concerns responsibility allocation when insecure or harmful code is introduced through AI-assisted development [22–24, 62]. In practice, responsibility may be distributed across several actors: the model developer, the provider of the coding assistant, the organization deploying the tool, the engineer who accepted the output, and the reviewer who approved it. This diffusion of responsibility can create accountability gaps, especially when AI-generated code is treated merely as a productivity aid rather than as an active decision-shaping component of the development process [23, 24, 72]. For secure deployment, organizations must therefore adopt explicit policies stating that AI assistance does not eliminate human accountability for the final artifact.

Auditability and traceability. Secure use of LLM-assisted coding also requires the ability to reconstruct how a code artifact was produced [24, 62, 103, 136]. This includes preserving relevant prompts, retrieved context, intermediate model outputs, tool invocations, validation results, and subsequent human edits. Without such traceability, it becomes difficult to investigate incidents, reproduce failures, or determine whether a vulnerability emerged from the model, the prompt, the retrieval source, the toolchain, or the human review process [24, 62]. Auditability is thus not only a compliance concern; it is also a practical prerequisite for debugging, incident response, and continuous security improvement.

Liability in regulated and high-stakes environments. The accountability problem becomes more pressing in regulated domains or safety-critical contexts, where software faults may trigger legal, financial, or societal harm [1, 23, 24]. In such settings, organizations cannot rely on opaque AI-generated artifacts without rigorous documentation, testing, and approval procedures. The introduction of LLM-assisted coding into domains such as healthcare, finance, public infrastructure, or industrial control raises difficult questions about due diligence, certification, and evidentiary standards for demonstrating that adequate safeguards were applied.

Governance as a socio-technical control layer. These concerns suggest that governance must be treated as a first-class layer of defence rather than as an afterthought [1, 22–24]. Effective governance includes policies for prompt and data handling, restrictions on the use of external tools and proprietary context, model and plugin vetting, mandatory review requirements for security-critical code, and documentation of human oversight and approval pathways. In the case of agentic systems, governance should additionally include action boundaries, escalation rules, logging of tool-mediated behavior, and explicit mechanisms for human interruption and override [24, 62]. Overall, accountability, auditability, and governance connect technical security properties with real deployment conditions and help ensure that human-AI collaboration remains reviewable, contestable, and aligned with institutional responsibility.

Overall, accountability, auditability, and governance should be understood as integral components of secure LLM-assisted coding. They connect technical security properties with real deployment conditions and help ensure that human-AI collaboration remains reviewable, contestable, and aligned with institutional responsibility.

7.4 Practical Insights and Actionable Recommendations

Organisations and software development teams have the opportunity to promptly implement specific measures to reduce security risks linked with incorporating LLMs into their processes. Here are seven practical recommendations that offer a comprehensive defence strategy for both leveraging the advantages of LLMs and addressing their potential risks.

Adopt a Zero-Trust Policy for AI-Generated Code. Treating all code generated by an LLM as untrusted third-party input is the most indispensable rule, no matter how advanced the model or how specific the prompt. Such code should undergo stringent security assessments, peer review, and extensive testing akin to what is applied to code from external or junior sources. It is imperative that LLM-generated code never be deployed directly to production without exhaustive human supervision and a series of automated security evaluations.

Invest in Continuous Developer Training for Secure AI Interaction. Human developers serve as the ultimate safeguard, with their knowledge being a crucial resource. Organisations need to prioritise ongoing training initiatives to enhance developers' understanding of the inherent limitations and particular security challenges associated with LLMs. This educational effort should go beyond standard secure coding practices to encompass AI-centric skills, including secure prompt engineering (e.g., crafting clear, precise, and security-conscious prompts), understanding common vulnerabilities linked to LLMs, and methods for critically assessing AI-generated code, including identifying indicators of model overconfidence or factual inaccuracies.

Integrate Multi-Layered, Automated Security Analysis into the CI/CD Pipeline. Utilizing only one security tool or method is inadequate for addressing the variety of possible issues in code generated by AI. An effective approach necessitates establishing a pipeline that integrates multiple levels of automated analysis that complement each other. This should encompass: Static Application Security Testing (SAST) to detect frequent vulnerability patterns within the source code prior to execution; Dynamic Application Security Testing (DAST) to discover runtime errors and vulnerabilities in the operational application; and Software Composition Analysis (SCA) to

pinpoint and control vulnerable or outdated third-party libraries and dependencies, as LLMs are often known to recommend such components.

Mandate Rigorous, Context-Aware Human Review for Security-Critical Code. Although all AI-produced code necessitates human oversight, code that is to be implemented in security-sensitive functions must undergo a rigorous and obligatory review process. This encompasses code involved in authentication, authorisation, encryption activities, session management, and input verification. Such evaluations should be performed by senior developers or specialised security professionals capable of analysing the code not only independently, but also within the broader context of the application, taking into account its interactions with various components and data pathways.

Develop and Enforce Strict Organisational Policies for Prompt Engineering and Data Handling. Organisations need to create and implement well-defined governance policies outlining how developers should engage with LLMs, especially those from third parties. Such policies should clearly forbid embedding sensitive or proprietary data—like API keys, internal algorithms, trade secrets, or Personally Identifiable Information (PII)—into prompts given to external models. In the case of applications utilising Retrieval-Augmented Generation (RAG), these policies must guarantee that internal knowledge bases used for retrieval are securely protected, access-controlled, and sanitised to avoid unintended data breaches and prevent malicious RAG poisoning attacks.

Prioritise and Vet the AI/ML Component Supply Chain. Before introducing a new tool, plugin, or open-source model based on LLMs, it is essential for organisations to perform comprehensive security due diligence. This process includes assessing the vendor's security practices, data privacy and retention policies, and model training procedures.⁵ In the case of open-source models obtained from public repositories, a standardised intake process should be adopted. This process should involve initial red-teaming to explore adversarial vulnerabilities and bias evaluations to highlight any potential ethical risks prior to integrating the model into internal development processes.

Establish Clear Lines of Accountability for AI-Assisted Development. The legal and regulatory framework concerning liability for mistakes and weaknesses in AI-generated code is unclear and continues to develop. In such a scenario, organisations must implement well-defined internal policies that delineate accountability. The human developer and engineering team should bear the final responsibility for the security, accuracy, and quality of the completed software, rather than the AI tool utilised. Keeping comprehensive documentation of the human oversight, amendment, and approval processes for all AI-generated code is vital for showing thoroughness and mitigating legal liabilities.

7.5 Implications for Academia and Industry

The rise of LLM-assisted programming brings about profound implications extending beyond technical challenges alone. It is reshaping research directions in academia and demanding new strategic methods for risk management and governance within the industrial sector. Navigating this new era effectively requires a collaborative effort between these communities to balance the pursuit of innovation with the crucial requirement for security and reliability.

7.5.1 Implications for Academia. The issues highlighted by this survey indicate the necessity for a pivotal change in academic approaches to software engineering and artificial intelligence research.

Traditionally, success for code-generating LLMs has been measured by their performance on tests like HumanEval [18], which largely focus on solving algorithmic puzzles in controlled environments. It is critical for the academic field to shift its focus toward emphasising security as a primary research goal. This includes fostering research on model architectures designed with security in mind, developing robust defences against adversarial threats, and creating realistic evaluation benchmarks that emphasise context-rich security scenarios. Adopting this new focus may require

accepting compromises, such as models that are more secure but show reduced performance on specific functional tasks being deemed more valuable scientific contributions.

The challenge of ensuring the security of LLM-assisted coding is not one that can be tackled by computer science in isolation. For instance, the Human-AI Trust-Vulnerability Nexus is inherently a Human-Computer Interaction (HCI) issue, demanding insights from psychology and cognitive science to effectively address and mitigate biases. Additionally, tackling concerns related to legal liability, data privacy, intellectual property, and algorithmic fairness requires collaboration with experts in law, ethics, and social sciences. The most significant future research efforts in this field will be fundamentally interdisciplinary, blending technical solutions with a detailed understanding of their human, societal, and legal implications to develop systems that are not only technically sound but also genuinely trustworthy.

Besides the practical aspect of constructing and testing more secure systems, there is an urgent need for foundational research aimed at understanding why these systems fail. This involves interpretability studies using techniques like representation engineering to explore how LLMs internally comprehend security concepts such as vulnerability or sanitisation. It also includes theoretical research seeking to outline the connection between a model's structural features, its training data, and the security assurances of its outputs. Gaining a deeper, foundational understanding of these systems is essential for advancing beyond empirical trial-and-error toward the systematic design of verifiably secure AI.

7.5.2 Implications for Industry. For industries, the primary challenge is to strategically manage the inherent conflict between leveraging the substantial productivity gains offered by LLMs and addressing the considerable security, legal, and reputational risks they pose.

LLMs are fundamentally redefining the economic dynamics of cybersecurity. For attackers, LLMs commoditise intelligence, significantly reducing the cost to uncover vulnerabilities and engineer custom, long-tail assaults on a broad spectrum of less common software, rendering previously unattractive targets financially worthwhile. This generates a compelling market-driven incentive for organisations to bolster their AI security measures. On the flip side, there is a rising positive economic incentive. As the market for specialised LLM agents and tools evolves, aspects like security, reliability, and trustworthiness will become vital competitive advantages. Vendors capable of proving that their products deliver enhanced security and produce safer code will be able to demand a premium, encouraging a market that benefits those investing in secure AI advancements.

The global regulatory framework for AI is quickly transitioning from broad ethical guidelines to specific, enforceable legal standards. Regulations such as the EU AI Act, which classifies AI systems based on risk, the NIST AI Risk Management Framework, and new standards like ISO 42001, are setting a new compliance landscape for organisations deploying AI [1]. These rules will necessitate organisations to develop strong governance, risk management, and compliance strategies for their AI systems. Moreover, the ongoing uncertainty regarding legal accountability for damages created by flaws in AI-produced code will push companies to adopt stringent internal validation, oversight, and documentation procedures to display due diligence and reduce legal liabilities. In the complex and dynamic environment, industry cannot afford to await perfectly secure, one-size-fits-all AI models. The sole feasible approach is a proactive, defence-in-depth strategy that combines technology, processes, and people. This entails investing in a comprehensive security technology stack, forming robust internal AI governance committees with well-defined accountability, and crucially, cultivating a culture of security awareness and critical thinking among developers. In the end, as AI increasingly automates coding procedures, the uniquely human abilities to judge, understand context, and provide critical oversight become some of the most crucial and irreplaceable security assets an organisation holds.

8 Conclusion

The integration of LLMs into software development offers substantial productivity benefits but introduces systemic security risks. These vulnerabilities originate from a fundamental triad: the models' lack of semantic security awareness, the inadequacy of current benchmarks, and the cognitive biases inherent in human-AI interaction. Our review identifies risks ranging from insecure code generation to the propagation of outdated dependencies, highlighting that evaluation must move beyond functional correctness toward adversarial, context-specific security assessments. Furthermore, developer oversight remains a critical but fragile defense, requiring tools that support rather than mislead the user. To address these challenges, we propose a research roadmap focused on security-driven training, more stringent benchmarks, and robust collaboration frameworks. For practitioners, we advocate a defense-in-depth strategy rooted in a zero-trust approach to model-generated code. Ultimately, the secure adoption of this technology requires a coordinated effort between academia and industry to prioritize security-focused governance alongside innovation.

References

- [1] 2025. *What are the Risks Associated with Generative AI Code?* <https://www.accorian.com/what-are-the-risks-associated-with-generative-ai-code/>
- [2] Hojjat Aghakhani, Wei Dai, Andre Manoel, Xavier Fernandes, Anant Kharkar, Christopher Kruegel, Giovanni Vigna, David Evans, Ben Zorn, and Robert Sim. 2024. Trojanpuzzle: Covertly poisoning code-suggestion models. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1122–1140.
- [3] Kamel Alrashedy, Abdullah Aljasser, Pradyumna Tambwekar, and Matthew Gombolay. 2023. Can LLMs Patch Security Issues? *arXiv preprint arXiv:2312.00024* (2023).
- [4] Avinash Anand, Akshit Gupta, Nishchay Yadav, and Shaurya Bajaj. 2024. A Comprehensive Survey of AI-Driven Advancements and Techniques in Automated Program Repair and Code Generation. *arXiv preprint arXiv:2411.07586* (2024).
- [5] Owura Asare and Meiyappan Nagappan. 2023. Is github's copilot as bad as humans at introducing vulnerabilities in code? *Empirical Software Engineering* 28, 6 (2023), 129.
- [6] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732* (2021).
- [7] Yuntao Bai, Saurav Kadavath, Sandipan Kundu, Amanda Askell, Jackson Kernion, Andy Jones, Anna Chen, Anna Goldie, Azalia Mirhoseini, Cameron McKinnon, et al. 2022. Constitutional AI: Harmlessness from AI feedback. *arXiv preprint arXiv:2212.08073* (2022).
- [8] Bandit. [n. d.]. *Bandit*. <https://github.com/PyCQA/bandit>
- [9] Shraddha Barke, Michael B James, and Nadia Polikarpova. 2023. Grounded copilot: How programmers interact with code-generating models. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1 (2023), 85–111.
- [10] Enna Basic and Alberto Giarretta. 2024. Large Language Models and Code Security: A Systematic Literature Review. *arXiv preprint arXiv:2412.15004* (2024).
- [11] ENNA BASIC and ALBERTO GIARETTA. 2025. From Vulnerabilities to Remediation: A Systematic Literature Review of LLMs in Code Security. (2025).
- [12] Vladislav Belozarov, Peter J Barclay, and Ashkan Sami. 2025. Secure Coding with AI, From Creation to Inspection. *arXiv preprint arXiv:2504.20814* (2025).
- [13] Manish Bhatt, Sahana Chennabasappa, Yue Li, Cyrus Nikolaidis, Daniel Song, Shengye Wan, Faizan Ahmad, Cornelius Aschermann, Yaohui Chen, Dhaval Kapil, et al. 2024. Cyberseceval 2: A wide-ranging cybersecurity evaluation suite for large language models. *arXiv preprint arXiv:2404.13161* (2024).
- [14] Manish Bhatt, Sahana Chennabasappa, Cyrus Nikolaidis, Shengye Wan, Ivan Evtimov, Dominik Gabi, Daniel Song, Faizan Ahmad, Cornelius Aschermann, Lorenzo Fontana, et al. 2023. Purple llama cyberseceval: A secure coding benchmark for language models. *arXiv preprint arXiv:2312.04724* (2023).
- [15] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.
- [16] Angelica Chen, Jérémy Scheurer, Tomasz Korbak, Jon Ander Campos, Jun Shern Chan, Samuel R Bowman, Kyunghyun Cho, and Ethan Perez. 2023. Improving code generation by training with natural language feedback. *arXiv preprint arXiv:2303.16749* (2023).

- [17] Junkai Chen, Huihui Huang, Yunbo Lyu, Junwen An, Jieke Shi, Chengran Yang, Ting Zhang, Haoye Tian, Yikun Li, Zhenhao Li, et al. 2025. SecureAgentBench: Benchmarking Secure Code Generation under Realistic Vulnerability Scenarios. *arXiv preprint arXiv:2509.22097* (2025).
- [18] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).
- [19] Qirui Chen, Jingxian Shuai, Shuangwu Chen, Shenghao Ye, Zijian Wen, Xufei Su, Jie Jin, Jiangming Li, Jun Chen, Xiaobin Tan, et al. 2026. HardSecBench: Benchmarking the Security Awareness of LLMs for Hardware Code Generation. *arXiv preprint arXiv:2601.13864* (2026).
- [20] Wanyi Chen, Meng-Wen Su, and Mary L Cummings. 2025. Assessing LLM code generation quality through path planning tasks. *arXiv preprint arXiv:2504.21276* (2025).
- [21] Subramanian Chidambaram, Li Erran Li, Min Bai, Xiaopeng Li, Kaixiang Lin, Xiong Zhou, and Alex C Williams. 2024. Socratic human feedback (SoHF): Expert steering strategies for LLM code generation. In *Findings of the Association for Computational Linguistics: EMNLP 2024*. 15491–15502.
- [22] Chun Jie Chong, Zhihao Yao, and Iulian Neamtii. 2024. Artificial-Intelligence Generated Code Considered Harmful: A Road Map for Secure and High-Quality Code Generation. *arXiv preprint arXiv:2409.19182* (2024).
- [23] James CL Chow and Kay Li. 2025. Large language models in medical chatbots: opportunities, challenges, and the need to address AI risks. *Information* 16, 7 (2025), 549.
- [24] James CL Chow and Kay Li. 2026. From Dialogue Systems to Autonomous Agents: A Modeling Framework for Ethical Generative AI in Healthcare. *Information* 17, 4 (2026), 361.
- [25] CodeQL. 2025. *CodeQL*. <https://github.com/github/codeql>
- [26] Domenico Cotroneo, Cristina Improta, Pietro Liguori, and Roberto Natella. 2024. Vulnerabilities in ai code generators: Exploring targeted data poisoning attacks. In *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*. 280–292.
- [27] Marcos Cramer and Lucian McIntyre. 2025. Verifying LLM-Generated Code in the Context of Software Verification with Ada/SPARK. *arXiv preprint arXiv:2502.07728* (2025).
- [28] Dekun Dai, MingWei Liu, Anji Li, Jialun Cao, Yanlin Wang, Chong Wang, Xin Peng, and Zibin Zheng. 2025. Feedback-Eval: A Benchmark for Evaluating Large Language Models in Feedback-Driven Code Repair Tasks. *arXiv preprint arXiv:2504.06939* (2025).
- [29] Badhan Chandra Das, M Hadi Amini, and Yanzhao Wu. 2025. Security and privacy challenges of large language models: A survey. *Comput. Surveys* 57, 6 (2025), 1–39.
- [30] Connor Dilgren, Purva Chiniya, Luke Griffith, Yu Ding, and Yizheng Chen. 2025. SecRepoBench: Benchmarking LLMs for Secure Code Generation in Real-World Repositories. *arXiv preprint arXiv:2504.21205* (2025).
- [31] Yangruibo Ding, Zijian Wang, Wasi U Ahmad, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. 2024. CoCoMIC: Code Completion by Jointly Modeling In-file and Cross-file Context. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*. 3433–3445.
- [32] Greta Dolcetti, Vincenzo Arceri, Eleonora Iotti, Sergio Maffei, Agostino Cortesi, and Enea Zaffanella. 2024. Helping LLMs Improve Code Generation Using Feedback from Testing and Static Analysis. *arXiv preprint arXiv:2412.14841* (2024).
- [33] Shihan Dou, Haoxiang Jia, Shenxi Wu, Huiyuan Zheng, Weikang Zhou, Muling Wu, Mingxu Chai, Jessica Fan, Caishuang Huang, Yunbo Tao, et al. 2024. What's Wrong with Your Code Generated by Large Language Models? An Extensive Study. *arXiv preprint arXiv:2407.06153* (2024).
- [34] Shihan Dou, Yan Liu, Haoxiang Jia, Enyu Zhou, Limao Xiong, Junjie Shan, Caishuang Huang, Xiao Wang, Xiaoran Fan, Zhiheng Xi, et al. 2024. StepCoder: Improving Code Generation with Reinforcement Learning from Compiler Feedback. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 4571–4585.
- [35] Kohei Dozono, Tiago Espinha Gasiba, and Andrea Stocco. 2024. Large language models for secure code assessment: A multi-language empirical study. *arXiv preprint arXiv:2408.06428* (2024).
- [36] Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2023. Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation. *arXiv preprint arXiv:2308.01861* (2023).
- [37] Ranjie Duan, Yuefeng Chen, Dantong Niu, Yun Yang, A Kai Qin, and Yuan He. 2021. Advdrop: Adversarial attack to dns by dropping information. In *Proceedings of the IEEE/CVF international conference on computer vision*. 7506–7515.
- [38] Ran Elgedawy, John Sadik, Senjuti Dutta, Anuj Gautam, Konstantinos Georgiou, Farzin Gholamrezae, Fujiao Ji, Kyungchan Lim, Qian Liu, and Scott Ruoti. 2024. Occasionally secure: A comparative analysis of code generation assistants. *arXiv preprint arXiv:2402.00689* (2024).

- [39] Sarah Fakhoury, Aaditya Naik, Georgios Sakkas, Saikat Chakraborty, and Shuvendu K. Lahiri. 2024. LLM-Based Test-Driven Interactive Code Generation: User Study and Empirical Evaluation. *IEEE Transactions on Software Engineering* 50, 9 (2024), 2254–2268. doi:10.1109/TSE.2024.3428972
- [40] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M Zhang. 2023. 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, 31–53.
- [41] Ehsan Firouzi and Mohammad Ghafari. 2026. Persistent Human Feedback, LLMs, and Static Analyzers for Secure Code Generation and Vulnerability Detection. *arXiv preprint arXiv:2602.05868* (2026).
- [42] Michael Fu, Van Nguyen, Chakkrit Tantithamthavorn, Dinh Phung, and Trung Le. 2024. Vision transformer inspired automated vulnerability repair. *ACM Transactions on Software Engineering and Methodology* 33, 3 (2024), 1–29.
- [43] Yanjun Fu, Ethan Baker, Yu Ding, and Yizheng Chen. 2024. Constrained decoding for secure code generation. *arXiv preprint arXiv:2405.00218* (2024).
- [44] YUJIA FU, PENG LIANG, ZENGYANG LI, MOJTABA SHAHIN, JIAXIN YU, and JINFU CHEN. 2024. Security Weaknesses of Copilot Generated Code in GitHub. *ACM Trans. Softw. Eng. Methodol* 1, 1 (2024).
- [45] Stefan Goetz and Andreas Schaad. 2024. "You still have to study"—On the Security of LLM generated code. *arXiv preprint arXiv:2408.07106* (2024).
- [46] Jianian Gong, Nachuan Duan, Ziheng Tao, Zhaohui Gong, Yuan Yuan, and Minlie Huang. 2024. How Well Do Large Language Models Serve as End-to-End Secure Code Producers? *arXiv preprint arXiv:2408.10495* (2024).
- [47] Hossein Hajipour, Keno Hassler, Thorsten Holz, Lea Schönherr, and Mario Fritz. 2024. 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, 684–709.
- [48] Hossein Hajipour, Lea Schönherr, Thorsten Holz, and Mario Fritz. 2024. HexaCoder: Secure Code Generation via Oracle-Guided Synthetic Training Data. *arXiv preprint arXiv:2409.06446* (2024).
- [49] Sivana Hamer, Marcelo d'Amorim, and Laurie Williams. 2024. Just another copy and paste? Comparing the security vulnerabilities of ChatGPT generated code and StackOverflow answers. In *2024 IEEE Security and Privacy Workshops (SPW)*. IEEE, 87–94.
- [50] Jingxuan He and Martin Vechev. 2023. Large language models for code: Security hardening and adversarial testing. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 1865–1879.
- [51] Jingxuan He, Mark Vero, Gabriela Krasnopolska, and Martin Vechev. 2024. Instruction tuning for secure code generation. In *Proceedings of the 41st International Conference on Machine Learning*. 18043–18062.
- [52] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. 2021. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938* (2021).
- [53] Kai Huang, Zhengzi Xu, Su Yang, Hongyu Sun, Xuejun Li, Zheng Yan, and Yuqing Zhang. 2023. A survey on automated program repair techniques. *arXiv preprint arXiv:2303.18184* (2023).
- [54] Xue Jiang, Yihong Dong, Lecheng Wang, Zheng Fang, Qiwei Shang, Ge Li, Zhi Jin, and Wenpin Jiao. 2024. Self-planning code generation with large language models. *ACM Transactions on Software Engineering and Methodology* 33, 7 (2024), 1–30.
- [55] Gopinath Kathiresan. 2025. Human-in-the-Loop Testing for LLM-Integrated Software: A Quality Engineering Framework for Trust and Safety. *Authorea Preprints* (2025).
- [56] Mohammed Kharma, Soohyeon Choi, Mohammed AlKhanafseh, and David Mohaisen. 2025. Security and Quality in LLM-Generated Code: A Multi-Language, Multi-Model Analysis. *arXiv preprint arXiv:2502.01853* (2025).
- [57] Mohammed F Kharma, Soohyeon Choi, Mohammad Alkhanafseh, and David Mohaisen. 2026. Security and quality in llm-generated code: A multi-language, multi-model analysis. *IEEE Transactions on Dependable and Secure Computing* (2026).
- [58] Raphaël Khoury, Anderson R Avila, Jacob Brunelle, and Baba Mamadou Camara. 2023. How secure is code generated by chatgpt?. In *2023 IEEE international conference on systems, man, and cybernetics (SMC)*. IEEE, 2445–2451.
- [59] Vassilka D Kirova, Cyril S Ku, Joseph R Laracy, and Thomas J Marlowe. 2024. Software engineering education must adapt and evolve for an llm environment. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1*. 666–672.
- [60] Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2023. DS-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning*. PMLR, 18319–18345.
- [61] Junjie Li, Fazle Rabbi, Cheng Cheng, Aseem Sangalay, Yuan Tian, and Jinqiu Yang. 2024. An exploratory study on fine-tuning large language models for secure code generation. *arXiv preprint arXiv:2408.09078* (2024).
- [62] Xinfeng Li, Shenyu Dai, Kelong Zheng, Yue Xiao, Gelei Deng, Wei Dong, and Xiaofeng Wang. 2026. "Are You Sure?": An Empirical Study of Human Perception Vulnerability in LLM-Driven Agentic Systems. *arXiv preprint*

arXiv:2602.21127 (2026).

- [63] Xinghang Li, Jingzhe Ding, Chao Peng, Bing Zhao, Xiang Gao, Hongwan Gao, and Xinchun Gu. 2025. Safegenbench: A benchmark framework for security vulnerability detection in llm-generated code. *arXiv preprint arXiv:2506.05692* (2025).
- [64] Yue Li, Xiao Li, Hao Wu, Minghui Xu, Yue Zhang, Xiuzhen Cheng, Fengyuan Xu, and Sheng Zhong. 2025. Everything You Wanted to Know About LLM-based Vulnerability Detection But Were Afraid to Ask. *arXiv preprint arXiv:2504.13474* (2025).
- [65] Zongjie Li, Chaozheng Wang, Zhibo Liu, Haoxuan Wang, Dong Chen, Shuai Wang, and Cuiyun Gao. 2023. Cctest: Testing and repairing code completion systems. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1238–1250.
- [66] Sherlock A Licorish, Ansh Bajpai, Chetan Arora, Fanyu Wang, and Kla Tantithamthavorn. 2025. Comparing Human and LLM Generated Code: The Jury is Still Out! *arXiv preprint arXiv:2501.16857* (2025).
- [67] Fang Liu, Yang Liu, Lin Shi, Houkun Huang, Ruifeng Wang, Zhen Yang, Li Zhang, Zhongqi Li, and Yuchi Ma. 2024. Exploring and evaluating hallucinations in llm-powered code generation. *arXiv preprint arXiv:2404.00971* (2024).
- [68] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by ChatGPT really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems* 36 (2023), 21558–21572.
- [69] Zhijie Liu, Yutian Tang, Xiapu Luo, Yuming Zhou, and Liang Feng Zhang. 2024. No need to lift a finger anymore? assessing the quality of code generation by chatgpt. *IEEE Transactions on Software Engineering* (2024).
- [70] Zihan Liu, Ruinan Zeng, Dongxia Wang, Gengyun Peng, Jingyi Wang, Qiang Liu, Peiyu Liu, and Wenhai Wang. 2024. Agents4PLC: Automating Closed-loop PLC Code Generation and Verification in Industrial Control Systems using LLM-based Agents. *arXiv preprint arXiv:2410.14209* (2024).
- [71] Michael R Lyu, Baishakhi Ray, Abhik Roychoudhury, Shin Hwei Tan, and Patanamon Thongtanunam. 2024. Automatic programming: Large language models and beyond. *ACM Transactions on Software Engineering and Methodology* (2024).
- [72] Wanlun Ma, Yiliao Song, Minhui Xue, Sheng Wen, and Yang Xiang. 2024. The “code” of ethics: A holistic audit of AI code generators. *IEEE Transactions on Dependable and Secure Computing* 21, 5 (2024), 4997–5013.
- [73] Noble Saji Mathews and Meiyappan Nagappan. 2024. Test-Driven Development and LLM-based Code Generation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering* (Sacramento, CA, USA) (ASE '24). Association for Computing Machinery, New York, NY, USA, 1583–1594. doi:10.1145/3691620.3695527
- [74] Tanha Miah and Hong Zhu. 2024. User Centric Evaluation of Code Generation Tools (Invited Paper). In *2024 IEEE International Conference on Artificial Intelligence Testing (AITest)*. 109–119. doi:10.1109/AITest62860.2024.00022
- [75] MITRE. 2021. *Cwe top 25 most dangerous software weaknesses*. https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html
- [76] MITRE. 2024. *CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer*. <https://cwe.mitre.org/data/definitions/119.html>
- [77] MITRE. 2024. *CWE-120: Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')*. <https://cwe.mitre.org/data/definitions/120.html>
- [78] MITRE. 2024. *CWE-125: Out-of-bounds Read*. <https://cwe.mitre.org/data/definitions/125.html>
- [79] MITRE. 2024. *CWE-153: Improper Neutralization of Substitution Characters*. <https://cwe.mitre.org/data/definitions/153.html>
- [80] MITRE. 2024. *CWE-20: Improper Input Validation*. <https://cwe.mitre.org/data/definitions/20.html>
- [81] MITRE. 2024. *CWE-200: Exposure of Sensitive Information to an Unauthorized Actor*. <https://cwe.mitre.org/data/definitions/200.html>
- [82] MITRE. 2024. *CWE-22: Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')*. <https://cwe.mitre.org/data/definitions/22.html>
- [83] MITRE. 2024. *CWE-327: Use of a Broken or Risky Cryptographic Algorithm*. <https://cwe.mitre.org/data/definitions/327.html>
- [84] MITRE. 2024. *CWE-346: Origin Validation Error*. <https://cwe.mitre.org/data/definitions/346.html>
- [85] MITRE. 2024. *CWE-352: Cross-Site Request Forgery (CSRF)*. <https://cwe.mitre.org/data/definitions/352.html>
- [86] MITRE. 2024. *CWE-401: Missing Release of Memory after Effective Lifetime*. <https://cwe.mitre.org/data/definitions/401.html>
- [87] MITRE. 2024. *CWE-416: Use After Free*. <https://cwe.mitre.org/data/definitions/416.html>
- [88] MITRE. 2024. *CWE-434: Unrestricted Upload of File with Dangerous Type*. <https://cwe.mitre.org/data/definitions/434.html>
- [89] MITRE. 2024. *CWE-502: Deserialization of Untrusted Data*. <https://cwe.mitre.org/data/definitions/502.html>
- [90] MITRE. 2024. *CWE-502: Deserialization of Untrusted Data*. <https://cwe.mitre.org/data/definitions/502.html>

- [91] MITRE. 2024. *CWE-732: Incorrect Permission Assignment for Critical Resource*. <https://cwe.mitre.org/data/definitions/732.html>
- [92] MITRE. 2024. *CWE-759: Use of a One-Way Hash without a Salt*. <https://cwe.mitre.org/data/definitions/759.html>
- [93] MITRE. 2024. *CWE-77: Improper Neutralization of Special Elements used in a Command ('Command Injection')*. <https://cwe.mitre.org/data/definitions/77.html>
- [94] MITRE. 2024. *CWE-78: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')*. <https://cwe.mitre.org/data/definitions/78.html>
- [95] MITRE. 2024. *CWE-787: Out-of-bounds Write*. <https://cwe.mitre.org/data/definitions/787.html>
- [96] MITRE. 2024. *CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')*. <https://cwe.mitre.org/data/definitions/79.html>
- [97] MITRE. 2024. *CWE-835: Loop with Unreachable Exit Condition ('Infinite Loop')*. <https://cwe.mitre.org/data/definitions/835.html>
- [98] MITRE. 2024. *CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')*. <https://cwe.mitre.org/data/definitions/89.html>
- [99] MITRE. 2024. *CWE-94: Improper Control of Generation of Code ('Code Injection')*. <https://cwe.mitre.org/data/definitions/94.html>
- [100] Ahmad Mohsin, Helge Janicke, Adrian Wood, Iqbal H Sarker, Leandros Maglaras, and Naeem Janjua. 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* (2024).
- [101] Fangwen Mu, Lin Shi, Song Wang, Zhuohao Yu, Binqun Zhang, ChenXue Wang, Shichao Liu, and Qing Wang. 2024. Clarifygpt: A framework for enhancing llm-based code generation via requirements clarification. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 2332–2354.
- [102] National Institute of Standards and Technology. 2023. *AI Risk Management Framework (AI RMF 1.0)*. <https://www.nist.gov/itl/ai-risk-management-framework>. Accessed: 2026-07-06.
- [103] National Institute of Standards and Technology. 2024. *Artificial Intelligence Risk Management Framework: Generative Artificial Intelligence Profile*. Technical Report NIST AI 600-1. National Institute of Standards and Technology. <https://nvlpubs.nist.gov/nistpubs/ai/NIST.AI.600-1.pdf> Accessed: 2026-07-06.
- [104] Mahmoud Nazzal, Issa Khalil, Abdallah Khreishah, and NhatHai Phan. 2024. PromSec: Prompt Optimization for Secure Generation of Functional Source Code with Large Language Models (LLMs). In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. 2266–2280.
- [105] Yu Nong, Haoran Yang, Long Cheng, Hongxin Hu, and Haipeng Cai. 2024. Automated software vulnerability patching using large language models. *arXiv preprint arXiv:2408.13597* (2024).
- [106] Ana Nunez, Nafis Tanveer Islam, Sumit Kumar Jha, and Peyman Najafirad. 2024. Autosafecoder: A multi-agent framework for securing llm code generation through static analysis and fuzz testing. *arXiv preprint arXiv:2409.10737* (2024).
- [107] Sanghak Oh, Kiho Lee, Seonhye Park, Doowon Kim, and Hyoungshick Kim. 2024. Poisoned chatgpt finds work for idle hands: Exploring developers' coding practices with insecure suggestions from poisoned ai models. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1141–1159.
- [108] Theo X Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. [n.d.]. Is Self-Repair a Silver Bullet for Code Generation?. In *The Twelfth International Conference on Learning Representations*.
- [109] Theo X Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. 2023. Is self-repair a silver bullet for code generation? *arXiv preprint arXiv:2306.09896* (2023).
- [110] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *Advances in neural information processing systems* 35 (2022), 27730–27744.
- [111] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2022. Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 754–768.
- [112] Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. 2023. Examining zero-shot vulnerability repair with large language models. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2339–2356.
- [113] Jinjun Peng, Leyi Cui, Kele Huang, Junfeng Yang, and Baishakhi Ray. 2025. CWEval: Outcome-driven Evaluation on Functionality and Security of LLM Code Generation. *arXiv preprint arXiv:2501.08200* (2025).
- [114] Neil Perry, Megha Srivastava, Deepak Kumar, and Dan Boneh. 2023. Do users write more insecure code with AI assistants?. In *Proceedings of the 2023 ACM SIGSAC conference on computer and communications security*. 2785–2799.
- [115] Ruizhong Qiu, Weiliang Will Zeng, James Ezick, Christopher Lott, and Hanghang Tong. 2024. How efficient is llm-generated code? a rigorous & high-standard benchmark. *arXiv preprint arXiv:2406.06647* (2024).

- [116] Md Fazle Rabbi, Arifa Islam Champa, Minhaz F Zibran, and Md Rakibul Islam. 2024. AI writes, we analyze: The ChatGPT python code saga. In *Proceedings of the 21st International Conference on Mining Software Repositories*. 177–181.
- [117] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297* (2020).
- [118] Gustavo Sandoval, Hammond Pearce, Teo Nys, Ramesh Karri, Siddharth Garg, and Brendan Dolan-Gavitt. 2023. Lost at c: A user study on the security implications of large language model code assistants. In *32nd USENIX Security Symposium (USENIX Security 23)*. 2205–2222.
- [119] Roei Schuster, Congzheng Song, Eran Tromer, and Vitaly Shmatikov. 2021. You autocompile me: Poisoning vulnerabilities in neural code completion. In *30th USENIX Security Symposium (USENIX Security 21)*. 1559–1575.
- [120] Erfan Shayegani, Md Abdullah Al Mamun, Yu Fu, Pedram Zaree, Yue Dong, and Nael Abu-Ghazaleh. 2023. Survey of vulnerabilities in large language models revealed by adversarial attacks. *arXiv preprint arXiv:2310.10844* (2023).
- [121] Ze Sheng, Zhicheng Chen, Shuning Gu, Heqing Huang, Guofei Gu, and Jeff Huang. 2025. Large Language Models in Software Security: A Survey of Vulnerability Detection Techniques and Insights. *arXiv preprint arXiv:2502.07049* (2025).
- [122] Ze Sheng, Zhicheng Chen, Shuning Gu, Heqing Huang, Guofei Gu, and Jeff Huang. 2025. LLMs in Software Security: A Survey of Vulnerability Detection Techniques and Insights. *arXiv e-prints* (2025), arXiv–2502.
- [123] Mohammed Latif Siddiq, Joanna Cecilia da Silva Santos, Sajith Devareddy, and Anna Muller. 2024. Sallm: Security assessment of generated code. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering Workshops*. 54–65.
- [124] Mohammed Latif Siddiq and Joanna CS Santos. 2022. SecurityEval dataset: mining vulnerability examples to evaluate machine learning-based code generation techniques. In *Proceedings of the 1st International Workshop on Mining Software Repositories Applications for Privacy and Security*. 29–33.
- [125] Ihor Taldenko. 2026. Human-in-the-Loop Governance for LLM-Generated Code: An Engineering Control, Accountability, and Verification Model in AI-Augmented Software Development. *EJSMT* 2, 3 (2026), 140–149.
- [126] Melissa Tessa, Iyiola E Olatunji, Aicha War, Jacques Klein, and Tegawendé F Bissyandé. 2026. How Secure is Secure Code Generation? Adversarial Prompts Put LLM Defenses to the Test. *arXiv preprint arXiv:2601.07084* (2026).
- [127] Haoye Tian, Weiqi Lu, Tsz On Li, Xunzhu Tang, Shing-Chi Cheung, Jacques Klein, and Tegawendé F Bissyandé. 2023. Is ChatGPT the ultimate programming assistant—how far is it? *arXiv preprint arXiv:2304.11938* (2023).
- [128] Zhao Tian, Junjie Chen, and Xiangyu Zhang. 2025. Fixing Large Language Models' Specification Misunderstanding for Better Code Generation. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 645–645.
- [129] Jiessie Tie, Bingsheng Yao, Tianshi Li, Syed Ishtiaque Ahmed, Dakuo Wang, and Shurui Zhou. 2024. LLMs are Imperfect, Then What? An Empirical Study on LLM Failures in Software Engineering. *arXiv preprint arXiv:2411.09916* (2024).
- [130] Norbert Tihanyi, Tamas Bisztray, Mohamed Amine Ferrag, Ridhi Jain, and Lucas C Cordeiro. 2024. Do neutral prompts produce insecure code? formai-v2 dataset: Labelling vulnerabilities in code generated by large language models. *arXiv preprint arXiv:2404.18353* (2024).
- [131] Norbert Tihanyi, Tamas Bisztray, Mohamed Amine Ferrag, Ridhi Jain, and Lucas C Cordeiro. 2025. How secure is AI-generated code: a large-scale comparison of large language models. *Empirical Software Engineering* 30, 2 (2025), 1–42.
- [132] Norbert Tihanyi, Tamas Bisztray, Ridhi Jain, Mohamed Amine Ferrag, Lucas C Cordeiro, and Vasileios Mavroeidis. 2023. The formai dataset: Generative ai in software security through the lens of formal verification. In *Proceedings of the 19th International Conference on Predictive Models and Data Analytics in Software Engineering*. 33–43.
- [133] Weixi Tong and Tianyi Zhang. 2024. CodeJudge: Evaluating Code Generation with Large Language Models. *arXiv preprint arXiv:2410.02184* (2024).
- [134] Catherine Tony, Markus Mutas, Nicolás E Díaz Ferreyra, and Riccardo Scandariato. 2023. Llmseceval: A dataset of natural language prompts for security evaluations. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. IEEE, 588–592.
- [135] Rebeka Tóth, Tamas Bisztray, and László Erdődi. 2024. LLMs in Web Development: Evaluating LLM-Generated PHP Code Unveiling Vulnerabilities and Limitations. In *International Conference on Computer Safety, Reliability, and Security*. Springer, 425–437.
- [136] Christoph Treude. 2026. Accountable Agents in Software Engineering: An Analysis of Terms of Service and a Research Roadmap. *arXiv preprint arXiv:2605.04532* (2026).
- [137] Mark Vero, Niels Mündler, Victor Chibotaru, Veselin Raychev, Maximilian Baader, Nikola Jovanović, Jingxuan He, and Martin Vechev. [n. d.]. BaxBench: Can LLMs Generate Correct and Secure Backends?. In *ICLR 2025 Third Workshop on*

Deep Learning for Code.

- [138] Jiexin Wang, Xitong Luo, Liuwen Cao, Hongkui He, Hailin Huang, Jiayuan Xie, Adam Jatowt, and Yi Cai. 2024. Is your ai-generated code really safe? evaluating large language models on secure code generation with codeseeval. *arXiv preprint arXiv:2407.02395* (2024).
- [139] Shuhe Wang, Shengyu Zhang, Jie Zhang, Runyi Hu, Xiaoya Li, Tianwei Zhang, Jiwei Li, Fei Wu, Guoyin Wang, and Eduard Hovy. 2024. Reinforcement learning enhanced LLMs: A survey. *arXiv preprint arXiv:2412.10400* (2024).
- [140] Yuxiang Wei, Chunqiu Steven Xia, and Lingming Zhang. 2023. Copiloting the copilots: Fusing large language models with completion engines for automated program repair. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 172–184.
- [141] Man Fai Wong and Chee Wei Tan. 2024. Aligning crowd-sourced human feedback for code generation with bayesian inference. In *2024 IEEE Conference on Artificial Intelligence (CAI)*. IEEE, 158–163.
- [142] Chunqiu Steven Xia and Lingming Zhang. 2023. Conversational automated program repair. *arXiv preprint arXiv:2301.13246* (2023).
- [143] Cihang Xie, Mingxing Tan, Boqing Gong, Jiang Wang, Alan L Yuille, and Quoc V Le. 2020. Adversarial examples improve image recognition. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 819–828.
- [144] Shenaoyan, Shen Wang, Yue Duan, Hanbin Hong, Kiho Lee, Doowon Kim, and Yuan Hong. 2024. An {LLM-Assisted} {Easy-to-Trigger} Backdoor Attack on Code Completion Models: Injecting Disguised Vulnerabilities against Strong Detection. In *33rd USENIX Security Symposium (USENIX Security 24)*. 1795–1812.
- [145] Boyang Yang, Haoye Tian, Weiguo Pian, Haoran Yu, Haitao Wang, Jacques Klein, Tegawendé F Bissyandé, and Shunfu Jin. 2024. Cref: An llm-based conversational software repair framework for programming tutors. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 882–894.
- [146] Yu Yang, Yuzhou Nie, Zhun Wang, Yuheng Tang, Wenbo Guo, Bo Li, and Dawn Song. 2024. SecCodePLT: A Unified Platform for Evaluating the Security of Code GenAI. *arXiv preprint arXiv:2410.11096* (2024).
- [147] Yifan Yao, Jinhao Duan, Kaidi Xu, Yuanfang Cai, Zhibo Sun, and Yue Zhang. 2024. A survey on large language model (llm) security and privacy: The good, the bad, and the ugly. *High-Confidence Computing* (2024), 100211.
- [148] Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Qianxiang Wang, and Tao Xie. 2024. Codereval: A benchmark of pragmatic code generation with generative pre-trained models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–12.
- [149] Binqi Zeng, Quan Zhang, Chijin Zhou, Gwihwan Go, Yu Jiang, and Heyuan Shi. 2025. Inducing Vulnerable Code Generation in LLM Coding Assistants. *arXiv preprint arXiv:2504.15867* (2025).
- [150] Quanjun Zhang, Chunrong Fang, Yang Xie, YuXiang Ma, Weisong Sun, Yun Yang, and Zhenyu Chen. 2024. A systematic literature review on large language models for automated program repair. *arXiv preprint arXiv:2405.01466* (2024).
- [151] Dewu Zheng, Yanlin Wang, Ensheng Shi, Ruikai Zhang, Yuchi Ma, Hongyu Zhang, and Zibin Zheng. 2025. Humanevo: An evolution-aware benchmark for more realistic evaluation of repository-level code generation. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 764–764.
- [152] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, et al. 2023. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 5673–5684.
- [153] Tianyu Zheng, Ge Zhang, Tianhao Shen, Xueling Liu, Bill Yuchen Lin, Jie Fu, Wenhui Chen, and Xiang Yue. 2024. OpenCodeInterpreter: Integrating Code Generation with Execution and Refinement. In *Findings of the Association for Computational Linguistics ACL 2024*. 12834–12859.
- [154] Li Zhong and Zilong Wang. 2024. Can llm replace stack overflow? a study on robustness and reliability of large language model code generation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 38. 21841–21849.
- [155] Wangchunshu Zhou and Ke Xu. 2020. Learning to compare for better training and evaluation of open domain natural language generation models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34. 9717–9724.
- [156] Xin Zhou, Sicong Cao, Xiaobing Sun, and David Lo. 2024. Large language model for vulnerability detection and repair: Literature review and the road ahead. *ACM Transactions on Software Engineering and Methodology* (2024).
- [157] Xin Zhou, Kisub Kim, Bowen Xu, DongGyun Han, and David Lo. 2024. Out of sight, out of mind: Better automatic vulnerability repair by broadening input ranges and sources. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.
- [158] Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widayarsi, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, et al. 2024. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877* (2024).