

LLM-Assisted Programming: Trends, Evaluation Gaps, and Software Assurance

Jyoti Dabass^{1,*}, Bhupender Singh Dabass²

Abstract

Large language models (LLMs) are increasingly used for code generation, software testing, program repair, and security analysis. However, their evaluation still relies heavily on functional benchmarks that often focus on small, isolated programming tasks. Such benchmarks provide limited insight into security, test quality, repository-level dependencies, and maintainability. This study examines how the evaluation of LLM-assisted programming has developed across these application areas. The literature shows a gradual shift from function-level benchmarks toward repository-level tasks and agent-based development. At the same time, several recurring problems remain difficult to capture with conventional functional metrics. These include incorrect API use, weak test oracles, unreliable fault localization, test suite overfitting, insecure generated code, and difficulties in handling dependencies across files. To address these gaps, a four-stage assurance framework is proposed. It separates evaluation into functional correctness, test and oracle quality, security assessment, and maintainability. The framework is intended to complement existing benchmark measures by examining properties that are important when generated code is used in practical software development. It is a literature-derived framework and has not yet been empirically validated. Future work should evaluate it on real software repositories and establish suitable measures for each stage.

Keywords: Large language models (LLMs), LLM-assisted programming, code generation, automated program repair, unit test generation, software security, code review, evaluation benchmarks, benchmark contamination, software assurance, quantum software engineering, cyber reasoning systems

INTRODUCTION

Large language models (LLMs) are now being used in everyday software development. A developer can describe a task in natural language, review the generated code, make corrections, and continue the interaction until the result is suitable. Tools such as Cursor, Aider, and ChatGPT support this workflow for tasks ranging from code completion and boilerplate generation to repository-level changes and refactoring [1–3]. The use of machine-generated code, however, predates current LLMs. Earlier program synthesis approaches used formal specifications, constraint solving, and theorem proving to generate programs [4, 5]. The development of deep learning and Transformer-based models shifted this

work toward statistical learning from large collections of source code. Models such as CodeBERT introduced language modeling techniques for code understanding, while later systems including Codex, CodeT5, Code Llama, and DeepSeek-Coder extended code generation to tasks such as function synthesis, program repair, and documentation [6–10].

The rapid improvement in benchmark performance has not always translated into comparable results in practical software development. Benchmarks such as HumanEval and

*Author for Correspondence

Jyoti Dabass

E-mail: jyotidabas91@gmail.com

¹Global Professor of Practice, Golden Gate University, San Francisco, California, USA

²Researcher and Advocate, District Bar Association, Palwal, Haryana, India

Received Date: June 24, 2026

Accepted Date: August 25, 2026

Published Date: September 28, 2026

Citation: Jyoti Dabass, Bhupender Singh Dabass. LLM-Assisted Programming: Trends, Evaluation Gaps, and Software Assurance. Recent Trends in Programming Languages. 2026; 13(2): 28–42p.

Mostly Basic Python Problems (MBPP) mainly evaluate isolated programming tasks through measures such as pass@k [8]. Reported performance has increased substantially; however, these scores provide only limited information about how generated code behaves in larger software systems [8, 11]. They do not directly address issues such as security, maintainability, runtime efficiency, or dependencies between files and components [5, 12, 13]. Repository-level tasks present additional difficulties. For example, when SWE-bench was introduced, Claude 2 resolved only a small proportion of the GitHub issues included in the benchmark [14]. Other empirical studies have identified security vulnerabilities in generated code [15], incorrect or nonexistent API usage, and difficulties with complex type systems and cross-file dependencies [16], as well as cases where generated solutions overfit incomplete test suites [17]. These observations indicate that evaluating LLM-assisted programming requires a broader view than measuring whether generated code passes a predefined test set. Code generation systems can fail in different ways depending on the development task. A generated function may produce the expected output for a benchmark input but contains a security weakness. A generated test may pass while failing to detect important faults. A debugging system may modify code without correctly identifying the underlying defect. Similarly, generated code may work in isolation but introduce problems when integrated with other components of a software repository.

These differences make it difficult to compare results across studies that use different tasks, datasets, models, and evaluation procedures. This study examines four major areas of LLM-assisted programming: code generation and completion, software testing and unit test generation, debugging and automated program repair, and software security. Quantum software engineering (QSE) and cyber reasoning systems (CRS) are considered separately because they represent emerging application areas with different technical requirements and evaluation settings. The analysis examines how evaluation practices have developed, what forms of validation are being reported, and where important gaps remain. The review considers both benchmark-based and practical evaluations. Benchmarks are useful for measuring progress under controlled conditions, but their results do not necessarily reflect performance in real development environments. Repository-level evaluation, security analysis, execution-based testing, human review, and studies of software maintenance over time can provide additional insight into the practical reliability of generated code. Considering these forms of validation together gives a broader view of the strengths and limitations of current LLM-assisted programming systems. This study also makes several contributions. It traces changes in evaluation practices across the reviewed literature, compares programming tasks, evaluation measures, failure modes, and research gaps across the major application areas, and proposes a four-stage evaluation and assurance framework based on recurring failure patterns. The framework is intended to extend evaluation beyond functional correctness by considering reliability, security, testing quality, and integration with software systems. A central issue is that functional correctness alone is not sufficient to judge generated software. A pass@1 result indicates that a generated solution passed a particular set of tests, but it does not establish that the code is secure, that the tests provide a reliable basis for evaluation, that a review decision is sound, or that the code can be maintained when integrated into a larger repository.

Therefore, evaluation for practical software use must consider these additional properties. The remainder of this paper first describes the review scope, taxonomy, and trend analysis. It then examines the major application areas and compares their evaluation practices and limitations. The proposed evaluation and assurance framework is presented next, followed by a discussion of QSE and CRS. Finally, the paper discusses the implications and limitations of the analysis, identifies research directions toward 2030, and concludes the paper.

REVIEW SCOPE AND TAXONOMY

This section describes the scope of the review, the sources considered, the taxonomy used to organize models and adaptation methods, and the main changes observed in the reported results over the period covered by the study.

Scope and Sources

The review considers research published in software engineering and artificial intelligence venues, including ICSE, FSE, ISSTA, Association for Computing Machinery (ACM) TOSEM, Empirical Software Engineering, and Communications of the ACM. Several arXiv preprints are also included where peer review had not been completed at the time of writing. The selection process follows Preferred Reporting Items for Systematic Reviews and Meta-Analyses (PRISMA) principles adapted for systematic mapping studies. The initial corpus covers systematic reviews, surveys, taxonomies, and recent empirical studies on code generation, code review, program repair, testing, security, and quantum software. Collectively, these sources summarize a broad body of primary research, including large-scale reviews of the existing literature [13, 18].

The analysis records the task examined, method used, dataset, main reported result, and stated limitations for the selected studies. The reviewed work covers code generation, security and code review, general and agentic software engineering, testing, program repair, and quantum software. The analysis therefore brings together findings from existing studies rather than conducting a new systematic review of the complete primary literature. This approach should be considered when interpreting the scope and strength of the findings presented in the paper.

Model Architectures

The reviewed models are grouped according to their architecture, pre-training objective, and subsequent adaptation. Three broad families occur repeatedly in the literature.

- *Encoder-only models*: CodeBERT, GraphCodeBERT, and RoBERTa use bidirectional self-attention with masked language modeling to learn syntactic and semantic representations of code. Their main applications include code classification, vulnerability detection, and code search [2]. Because these models are not designed for autoregressive token generation, their use for direct code generation is limited [6].
- *Decoder-only models*: GPT-4, Code Llama, Llama 3, DeepSeek-V3, R1, and Qwen use causal language modeling to predict subsequent tokens from preceding context. This architecture has become prominent in code completion, multi-turn generation, and conversational program repair [1, 5]. Mixture-of-experts models have also become important; for example, DeepSeek-Coder-V2 reported performance comparable to GPT-4 on HumanEval and MBPP [11].
- *Encoder-decoder models*: CodeT5, CodeT5+, PLBART, and CodeGen2 use sequence-to-sequence architectures with objectives such as span denoising and masked sequence prediction. CodeT5 incorporates identifier-aware training and reported improvements on translation and refinement tasks [7]. Existing reviews also report their use in code translation, refinement, unit test generation, and code-review comment generation [1].

Adaptation Paradigms

The reviewed systems differ not only in architecture but also in how their underlying models are adapted to a particular programming task. Four approaches are particularly common.

- *Prompting*: Zero-shot and few-shot methods use the capabilities already present in the pre-trained model and guide generation through instructions, examples, and structured prompts. No model parameters are changed [19, 20].
- *Parameter-efficient fine-tuning (PEFT)*: PEFT keeps the main model parameters fixed while training a smaller set of additional parameters. Examples include LoRA, QLoRA, IA3, and continuous prefix methods such as SVEN [21–23]. This reduces training requirements while allowing specialization for tasks such as vulnerability detection and program repair. In secure code generation, prefix tuning increased the proportion of secure outputs to 92.3% while retaining functional correctness [23]. LoRA-based tuning on vulnerability-fixing commits improved secure-output ratios by up to 6.4%, whereas full fine-tuning was associated with catastrophic forgetting and fewer compilable snippets [21].

- *Retrieval-augmented generation (RAG)*: RAG supplies the model with relevant information retrieved from sources such as a local repository, documentation, or static analysis results. This approach can provide task-specific context and reduce the incorrect use of outdated APIs [24, 25]. RepoCoder extends this idea by performing retrieval repeatedly during generation, thereby improving repository-level completion [26]. However, excessive dependence on retrieved material can produce redundant or insufficient responses [1, 26].
- *Agentic orchestration*: Agent-based systems divide software tasks into multiple steps and use external tools during the process. Systems such as AutoCodeRover and SpecRover repeatedly interact with static analyzers, test runners, and editing tools, using the resulting feedback to guide subsequent actions [27, 28].

Trends in Reported Results, 2020–2026

The volume of research increased substantially over the period covered by the review, reflecting the rapid growth of interest in LLM-assisted software engineering [18]. As research activity expanded, the focus of evaluation also changed, moving beyond basic code-generation performance toward broader concerns such as testing, security, reliability, and practical software development.

Table 1. Representative results reported across the reviewed literature.

Period	Evaluation focus	Representative reported results	Limitations reported in the sources
2020–2021	Pre-trained encoders and encoders–decoders for code understanding; early large decoder-only code models; function-level benchmarks such as HumanEval and MBPP	Codex achieved 28.8% pass@1 on HumanEval [8]. CodeBERT and CodeT5 reported state-of-the-art results on code understanding and sequence-to-sequence tasks [6, 7]. MBPP established a benchmark for entry-level Python programming tasks [29].	Evaluation concentrated mainly on functional correctness, with limited assessment of security, maintainability, or readability [8]. Encoder-only architecture was not well suited to direct generation [6].
2022–2023	Instruction tuning, curated training data, security and code-review studies, agent-based generation, and repository-level benchmarks	HumanEval pass@1 reached 57.3% for WizardCoder 15B [30], 53.0% for Code Llama 70B [9], 50.6% for phi-1 [31]. and 96.3% for AgentCoder [32]. One study reported vulnerabilities in about 40% of Copilot-generated programs [15]. Prefix tuning increased secure outputs to 92.3% [23]. Claude 2 resolved 1.96% of SWE-bench issues [14].	Studies raised concerns about contamination in models trained with model-generated instructions [30], safety–utility trade-offs [9], and limited reasoning and context for repository-level tasks [14].
2024	Mixture-of-experts code models, library-oriented benchmarks, security-focused fine-tuning, and quantum program repair	DeepSeek-Coder-V2 reached 81.1% pass@1 on HumanEval [11]. while the best reported BigCodeBench result was 59.7% [33]. LoRA-based tuning improved secure-output ratios by up to 6.4% [21]. ChatGPT repaired 29 of 38 quantum bugs [34].	Reported concerns included contamination and output consistency in mixture-of-experts models [11]. instruction-following limitations in 7B models [33]. catastrophic forgetting during full fine-tuning [21]. and dependence on context and fault localization in quantum repair [34].
2025–2026	Cross-language repair, LLM-based code review, agentic software engineering, and security-oriented systems	Thirteen LLMs fixed 542 of 712 bugs, while access to failing-test information increased pass@1 by more than 40 points in some settings [19]. GPT-4o showed an 87.9% false-negative rate as a code reviewer under the fullest prompt condition [35]. Agentic configurations reported a 78.4% resolution rate on SWE-bench Verified [3]. A cyber reasoning pipeline identified 2.4 unique vulnerabilities per trial [36].	Automated fault localization reduced pass@1 by 8–16 points in one study [19]. Reviewer misjudgment increased with prompt complexity [35]. Differences in benchmark variants and agent scaffolds complicate longitudinal comparisons [3]. while soundness guarantees remain unresolved in cyber reasoning systems [36].

The results summarized in Table 1 indicate three broad changes. First, evaluation has gradually expanded from isolated functions toward repository-level issues and multi-step agentic workflows. Second, function-level benchmarks are showing increasingly high scores; AgentCoder, for example, reported 96.3% pass@1 on HumanEval [32]. Such results reduce the ability of these benchmarks to distinguish stronger systems and increase the importance of examining contamination and evaluation design [11, 30]. Third, recent studies increasingly investigate failure conditions rather than reporting only successful task completion. Examples include unreliable code-review decisions [35], the effect of realistic fault localization on repair [19], and the behavior of LLMs within security-oriented pipelines [36]. The numerical results should not be interpreted as a direct performance ranking across years. The studies use different models, datasets, benchmark versions, prompting strategies, and evaluation procedures. SWE-bench provides a clear example: the 1.96% result reported for Claude 2 in 2023 [14] and the 78.4% result reported for an agentic configuration in 2026 [3], concern different evaluation settings, with the latter using SWE-bench Verified. Changes in benchmark versions and surrounding scaffolds therefore limit direct longitudinal comparison. Overall, the literature shows a transition from measuring whether an LLM can generate a function that passes tests toward examining how it behaves within larger development processes. This change creates a need for evaluation methods that consider not only task completion but also security, testing reliability, repair behavior, review quality, and performance under realistic software development conditions.

COMPARATIVE ANALYSIS OF FOUR APPLICATION AREAS

The literature reviewed in this study covers four closely related uses of LLMs in software development: code generation, software testing, debugging and automated program repair, and software security and code review. Although these areas use overlapping models and techniques, their evaluation requirements differ. The following subsections summarize the main tasks, reported results, and recurring limitations in each area.

Code Generation

Code generation converts a natural language description or an existing code prefix into executable source code. Early benchmarks, including HumanEval and MBPP, primarily used self-contained programming problems [8, 29]. More recent benchmarks have introduced additional requirements, such as library calls, multi-file context, class-level design, and more realistic programming tasks. BigCodeBench, ClassEval, and NaturalCodeBench represent this broader evaluation setting [1, 33, 37]. The difference between isolated and more demanding tasks is reflected in the reported performance. DeepSeek-Coder-V2 achieved 81.1% pass@1 on HumanEval [11], whereas its instruct variant achieved 59.7% on BigCodeBench, where successful completion requires the use of diverse function and library calls [33]. Studies of both open and closed models generally find stronger performance on conventional Python programming tasks, with greater difficulty when solutions require multiple libraries, complex type constraints, or broader contextual understanding [16, 33]. Common failure patterns include incorrect API usage, unnecessary complexity in otherwise short solutions, and differences between the generated implementation and the intended requirement [5, 16]. Some of these errors can be addressed through iterative feedback. A training-free approach in which the model critiques its own generated code achieved a repair success rate of 29.2% after two iterations [16]. This finding suggests that iterative generation and review can improve some output without additional model training.

Software Testing and Unit Test Generation

LLMs are increasingly used to generate unit tests, test inputs, assertions, and related quality-assurance artifacts. The reviewed literature is concentrated largely on decoder-only models and Java and Python projects, with Defects4J and SF110 among the commonly used benchmarks [38, 39]. A review covering 38 studies reported time savings of up to 86% and test coverage of up to 93%, while also identifying limited industrial adoption, syntax errors, and context-window constraints as recurring problems [40]. Higher coverage has been reported when LLMs are combined with conventional

software testing techniques rather than used independently. Hybrid systems incorporate search-based generation, static analysis, or reinforcement learning to guide the model toward more useful test cases [38, 41]. CodaMosa, for example, uses an LLM to provide example tests when search-based generation reaches a difficult point and reported an average coverage improvement of 9–10% [41]. In contrast, unconstrained LLM prompting was reported to perform below traditional search-based approaches in some settings [41]. Language coverage remains uneven. Rust, C/C++, Go, and Kotlin have received considerably less attention, with ownership systems, pointer behavior, and strict type constraints creating additional difficulties for generated tests [38, 42]. The quality of tests generated is also variable. Reported problems include invalid syntax, assertions that do not reflect the intended behavior, and failure to exercise important edge cases [39, 40]. Thus, higher code coverage does not necessarily indicate that an automatically generated test suite provides equivalent fault-detection capability.

Debugging and Automated Program Repair

Automated program repair (APR) aims to identify and correct defects with limited or no manual intervention. LLM-based approaches have progressed from neural machine-translation formulations to zero-shot completion methods, where the model fills or modifies buggy code without requiring paired bug-fix examples [43, 44]. More recent systems incorporate multiple agents and external tools into the repair process [27, 28]. A recent evaluation involving 13 open and closed models tested Java, JavaScript, Python, and PHP programs using Defects4J, BugsJS, BugsInPy, and BugsPHP (PHP Bugs Benchmark) Across these benchmarks, the models fixed 542 of 712 bugs, although no individual model performed best across all four languages [19]. The availability of failing-test information had a particularly strong effect: for some model and language combinations, adding this information increased pass@1 by more than 40 percentage points. Localization hints produced smaller and less consistent improvements [19]. Earlier work similarly treated bug localization and patch generation as related but distinct stages, with localization influencing the quality of the resulting repair [45]. The distinction between known and automatically identified fault locations is important in realistic repair systems. When spectrum-based localization was used instead of the ground-truth location, pass@1 on Defects4J decreased by 8–16 percentage points [19]. This indicates that repair performance depends not only on the model's ability to generate a patch but also on whether the preceding localization step identifies the relevant code. Another concern is test suite overfitting. A generated patch may satisfy an incomplete test suite while leaving the underlying defect unresolved, concealing the failure, or removing required functionality. Empirical work suggests that the practical extent of this problem may be smaller than some earlier discussions imply, but it remains an important validity concern [17]. Benchmark leakage presents another challenge because datasets such as Defects4J may have appeared in model pre-training data, making it difficult to determine whether performance reflects general repair ability or prior exposure to benchmark examples [19, 43].

Software Security and Code Review

Security presents different requirements from ordinary functional code generation because an output can execute correctly while still introducing a vulnerability. An early evaluation of GitHub Copilot found that approximately 40% of programs generated for security-relevant scenarios contained vulnerabilities [15]. Subsequent work has shown that the problem can also be amplified through adversarial inputs. Prompt prefixes and suffixes have caused code models to generate functionally correct but vulnerable programs in up to 62.5% of tested cases [46]. Training data poisoning has similarly been shown to increase a completion model's confidence in insecure bait completions from a baseline of 0–20% to 30–100% [47]. Model adaptation can reduce some of these risks. Security-oriented fine-tuning and prefix-based methods have improved the proportion of secure outputs, although their effectiveness depends on the quality and coverage of vulnerability-related training data [21, 23]. Vulnerability detection has its own evaluation challenges. A systematic review of 263 studies identified noisy datasets, a strong concentration on C/C++ memory-safety vulnerabilities compared with web-logic vulnerabilities, and possible overlap between benchmark data and model pre-training corpora [13,

48]. Direct model evaluations also report difficulties in understanding the program semantics relevant to vulnerabilities. Models may produce explanations that appear plausible but do not correctly identify the security issue, and their decisions can change after relatively small textual modifications to the same code [49]. LLMs used for automated code review introduce another type of reliability problem. On HumanEval-X-Bugs and MBPP, general-purpose models frequently rejected implementations that were functionally correct because they inferred requirements that were not actually present.

Increasing prompt complexity was associated with higher misjudgment rates, and GPT-4o recorded a false-negative rate of 87.9% under the most extensive prompting condition reported in the study [35]. Earlier specialized review models provide evidence that task-specific training can improve review performance. CodeReviewer achieved an F1 score of 71.5% for quality estimation, although security-focused reviews received comparatively limited attention in that work [50]. More specialized systems have therefore incorporated security information directly into the adaptation process. SecureReviewer combines security-aware fine-tuning with a retrieval store containing relevant templates and introduces SecureBLEU as a task-specific evaluation measure [51]. The findings across these four areas show that LLM-assisted programming cannot be assessed using a single measure. Code generation depends on both functional correctness and adherence to the intended requirements. Test generation must show that the tests can identify actual faults. Program repair depends strongly on accurate fault localization and reliable tests. Security and review tasks require additional checks to ensure that apparently correct outputs do not contain vulnerabilities or lead to unsupported decisions. These differences form the basis for the cross-area comparison presented in the next section.

COMPARATIVE SYNTHESIS

The four application areas differ in their objectives and evaluation practices, but they share an important characteristic: the metric commonly used to measure performance does not always capture the property that matters most in practice. Tables 2 and 3 summarize these differences by comparing the main tasks, metrics, benefits, failure modes, validation requirements, and remaining research gaps.

The metrics in Table 2 serve different purposes and should not be treated as interchangeable. For example, `pass@k` measures whether the generated code satisfies a particular test condition [8], whereas the mutation score evaluates the ability of tests to detect deliberately introduced faults [38, 39]. Similarly, security-oriented measures such as precision, recall, F1, SecureBLEU, and proof-of-vulnerability (PoV) trigger rate capture properties that are not represented by ordinary functional correctness measures [13, 51].

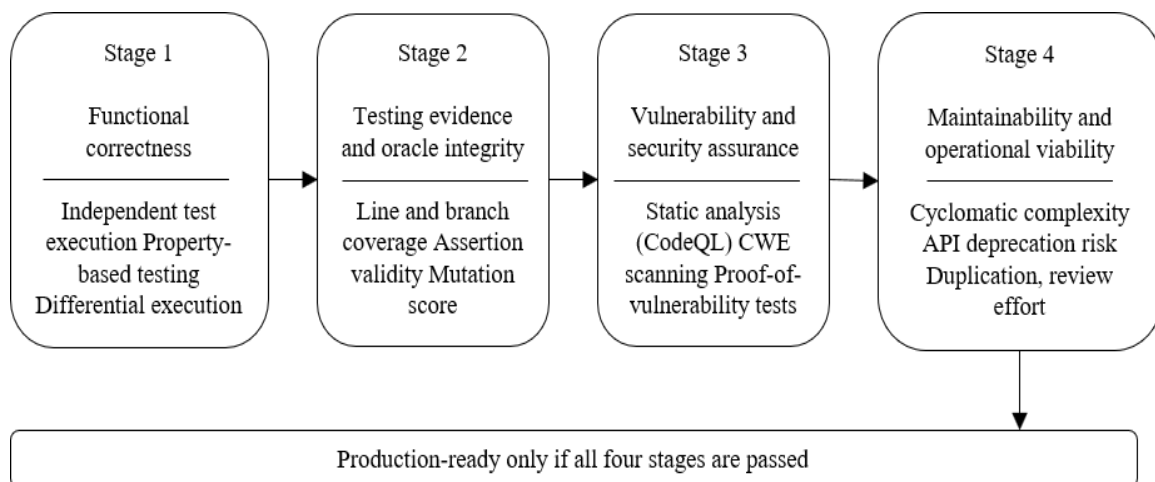
The comparison reveals a common evaluation problem across all four areas (Table 3). A system can achieve a strong score on its primary metric while failing to satisfy the underlying engineering requirement. A high `pass@k` does not guarantee that the generated code correctly captures the developer's intent [5, 16]. High coverage does not indicate that the automatically generated assertions are meaningful [39, 40]. A repair that passes the available tests may still suppress the visible symptom or introduce a regression [17]. Likewise, a vulnerability detector can report a strong F1 score when benchmark data overlap with information encountered during model pre-training [13, 48]. Security-oriented evaluations clearly show the need for broader validation. An early study reported vulnerabilities in approximately 40% of programs generated by GitHub Copilot for security-relevant scenarios [15]. Such findings cannot be adequately captured through conventional code-generation metrics alone. These observations suggest that LLM-assisted programming should be evaluated according to the specific property being assessed rather than through a single overall performance score. Functional correctness, test effectiveness, repair validity, and security assurance each require different validation methods. The next section builds on this comparison by presenting a four-stage evaluation and assurance framework that treats these requirements separately and defines the validation needed at each stage.

Table 2. Tasks, metrics, and benefits of LLM assistance across the four application areas.

Area	Typical tasks	Common metrics	Principal benefits
Code generation	Natural language to code generation; code completion; class-level synthesis	Pass@k; CodeBLEU; exact match	Faster development of routine code and integration with development environments
Software testing	Unit test synthesis; test-oracle construction; test evolution	Line and branch coverage; mutation score; compilability	Automated regression-test creation and broader test diversity
Program repair	Fault localization; candidate patch generation	Pass@k; plausible-patch yield; test-pass rate	Faster defect resolution and reduced developer effort during triage
Software security	Vulnerability detection; secure code generation; security review	Precision, recall, F1; SecureBLEU; proof-of-vulnerability (PoV) trigger rate	Earlier security detection and automated vulnerability triage

Table 3. Failure modes, validation requirements, and open research gaps.

Area	Major failure modes	Required validation	Open research gaps
Code generation	API misuse; unnecessarily complex implementations; mismatch with the intended requirement	Execution against independent tests; static (AST) Abstract Syntax Tree verification	Repository-level dependency context; reasoning across multiple libraries
Software testing	Hallucinated assertions; syntactically invalid tests; weak test oracles	Executed tests; mutation testing; dynamic trace analysis	Support for Rust, Go, and C++; project-level co-evolution of production code and tests
Program repair	Test suite overfitting; symptom suppression; regressions; inaccurate fault localization	Regression testing; semantic-equivalence analysis	Generalization beyond benchmark datasets; repair of complex multi-hunk defects
Software security	Insecure generated code; over-correction bias; false positives	Static analysis such as CodeQL; dynamic proof-of-vulnerability (PoV) testing	Datasets for low-resource languages; repository-scale, cross-file vulnerability context

**Figure 1.** Four-stage evaluation and assurance framework for LLM-generated code.

PROPOSED EVALUATION AND ASSURANCE FRAMEWORK

The comparison in the previous sections indicates that no single metric can adequately establish the quality of LLM-generated software. In particular, pass@k measures success under a defined test condition but does not by itself establish security, test quality, maintainability, or suitability for deployment. To address this limitation, we propose a four-stage evaluation and assurance framework in which generated code is assessed through four sequential gates (Figure 1). The stages are arranged from

relatively inexpensive checks to more resource-intensive forms of validation. Basic execution and testing are performed first, followed by security analysis and broader quality assessment. This ordering allows defective outputs to be rejected before additional analysis is applied.

Stage 1: Functional Correctness Under Independent Execution

The first stage establishes whether the generated artifact can compile, execute, and satisfy its intended functional requirements. Evaluation should use independent test cases that were not available to the model during generation. This reduces the possibility that a result reflects adaptation to a known test set rather than the required behavior. Property-based testing can be used to verify general program properties, such as idempotence or other task-specific invariants. Differential execution provides another check by comparing the generated implementation with a trusted implementation or expected behavior [5, 16]. Repeated sampling is also important because LLM outputs can vary across runs. Instead of reporting a result from a single generation, evaluations should generate multiple candidates for the same task and report the resulting variation. One recent study, for example, evaluated 15 candidates per task [19]. The need for repeated and systematic evaluation is also reflected in current testing tools, where mechanisms for repeated assessment of LLM-based software remain limited [52].

Stage 2: Testing and Oracle Integrity

The second stage evaluates whether the tests associated with the generated software provide meaningful support for judging correctness. This includes both automatically generated tests and tests created by the LLM during a self-testing process. Useful measures include line coverage, branch coverage, assertion validity, and mutation score. Mutation testing introduces controlled faults into the program and checks whether the test suite can detect them.

These measures capture different aspects of test quality: coverage shows which parts of the program are exercised, while mutation testing indicates whether the tests are capable of detecting changes in program behavior [38, 39]. This stage is particularly important because a test can execute successfully without providing a reliable correctness check. For example, an automatically generated assertion may be syntactically valid but fail to represent the intended program behavior. Test-oracle integrity therefore needs to be evaluated separately from the number of tests or the amount of code covered.

Stage 3: Vulnerability and Security Assurance

The third stage evaluates whether the generated artifact introduces known or exploitable security weaknesses. Static analysis tools, such as CodeQL, can identify potentially vulnerable code patterns, while mapping findings to the Common Weakness Enumeration (CWE) categories provides a standardized way of describing the identified weaknesses. Dynamic PoV testing can then determine whether the selected findings can be exercised in practice. Security validation should be performed in addition to functional testing. A program may produce the expected result and still contain an exploitable vulnerability. Studies of LLM-generated code and security-focused systems show that functional correctness alone is not sufficient to establish security [13, 15, 51].

Stage 4: Maintainability, Efficiency, and Operational Viability

The final stage considers whether the generated code remains suitable for continued use and maintenance. Relevant factors include cyclomatic complexity, duplicated code, dependence on potentially deprecated APIs, runtime or resource efficiency, and the amount of human effort required for review and subsequent modification [2, 16]. This stage extends the evaluation beyond the immediate task. Code that passes functional, testing, and security checks may still create additional technical debt if it is unnecessarily complex, duplicates existing functionality, relies on unstable dependencies, or requires substantial developer intervention. Operational viability therefore concerns not only whether the generated code works today, but also whether it can be incorporated into an existing development process without creating disproportionate maintenance costs.

Decision Rule

The framework treats production readiness as a conjunctive condition rather than a compensatory score. A generated artifact must satisfy all four requirements:

1. *Functional correctness*: the software executes correctly under independent evaluation.
2. *Test-oracle integrity*: the accompanying tests provide a reliable basis for assessing program behavior.
3. *Security assurance*: Identified security risks are assessed through static and, where appropriate, dynamic validation.
4. *Maintainability and operational viability*: the artifact is sufficiently maintainable, efficient, and practical for continued use.

Failure at one stage is not compensated by a strong result at another. For example, high functional accuracy does not offset confirmed security vulnerability, and high-test coverage does not establish maintainability. The framework therefore evaluates properties separately rather than treating a single benchmark score as a complete measure of LLM programming performance.

EMERGING AREAS: QUANTUM SOFTWARE ENGINEERING AND CYBER REASONING

QSE and CRS extend LLM-assisted programming to domains where conventional software development evaluation may not fully capture the relevant risks and requirements. Research in both areas remains relatively limited, but the studies reviewed identify distinct technical challenges and additional requirements for reliable validation.

Quantum Software Engineering

Advances toward fault-tolerant quantum computing, together with software platforms such as IBM Qiskit and hardware developments including Google's Willow chip, have increased research activity in QSE [53]. LLMs are being explored for quantum circuit generation, modeling, test-oracle construction, and program repair. Bugs4Q is one of the benchmarks developed for evaluating repair in this setting [34, 53]. In one evaluation, ChatGPT repaired 29 of 38 Bugs4Q defects. Its results approached those of manual repair when additional hints were provided, but performance depended strongly on the quality of the available context and the accuracy of bug localization [34]. Quantum programs introduce several difficulties that are not directly addressed by conventional software testing methods.

- *State collapse and the oracle problem*: Direct measurement of an intermediate quantum state changes that state by collapsing the superposition. As a result, observing whether a circuit behaves correctly is more difficult than checking an equivalent classical program, making reliable test oracles difficult to construct [53].
- *Non-local entanglement-related faults*: Errors involving qubit resets, ancilla measurements, or phase operations can affect a circuit through non-local relationships between qubits. Such faults may not be identified by conventional static analysis methods designed for classical programs [34].
- *Simulation-to-hardware gap*: Successful execution on a noiseless classical simulator does not necessarily indicate equivalent behavior on a physical quantum processing unit (QPU). Hardware noise, decoherence, and imperfect gate fidelity can alter the behavior of a circuit after deployment [53].

These characteristics suggest that QSE evaluation requires evidence from both simulation and physical execution, where possible. Functional correctness on a simulator alone is therefore insufficient for establishing the reliability of a quantum program.

Cyber Reasoning Systems and Autonomous Agents

CRS combine multiple security analysis techniques, including static analysis, automated input generation, dynamic fuzzing, and LLM-based repair agents. Systems such as AutoCodeRover and SpecRover illustrate the use of agentic workflows for iterative code analysis and modification [27, 28].

These components can also be combined into end-to-end vulnerability discovery and remediation pipelines [36]. Research on the reliability of autonomous programming systems remains limited. One study integrated GPT-4o with static analysis, input generation, and automated repair and evaluated the resulting pipeline on an Nginx benchmark from the AI Cyber Challenge (AIxCC) competition. The system was able to identify vulnerabilities and generate plausible patches, demonstrating the potential of combining LLMs with conventional security analysis and repair techniques [36].

The results also highlight an important assurance limitation. A system that combines several tools may discover vulnerabilities and generate patches without providing a formal guarantee that its findings are sound or that its proposed fixes are safe. Therefore, the study notes the possibility of false positives and emphasizes that statistical models cannot guarantee soundness [36]. For autonomous cyber reasoning, the challenge is consequently broader than vulnerability detection or patch generation in isolation. The complete pipeline must be evaluated for the reliability of its analysis, the validity of detected vulnerabilities, the correctness of generated repairs, and the behavior of the agents when individual components produce uncertain or incorrect results.

DISCUSSION AND LIMITATIONS

The synthesis highlights several practical considerations for using LLM-based programming tools and identifies areas where current research remains limited.

Implications

The findings have implications for both software teams and benchmark developers. Teams using LLM assistants should not treat a high pass@k result as proof that the generated code is ready to merge. Functional correctness is only one part of the evaluation. The generated code should also be tested independently, the quality of the generated tests should be examined, security checks should be performed, and the effort required for human review should be considered. These correspond to the four stages proposed in the evaluation framework. Benchmark developers also need to report these dimensions alongside functional correctness. Evaluation datasets should be checked for contamination between training and test data because leakage can produce artificially high results, particularly in vulnerability detection and code-generation tasks [12, 48]. The evaluation protocol should also specify the number of samples generated for each task. Because LLM outputs can differ across runs, the sampling procedure can affect the reported results [52].

Limitations

This review has several limitations. The synthesis draws on selected secondary reviews and empirical studies, so the findings may reflect differences in study selection, research methods, datasets, and reporting practices across the literature. The available research also covers different software engineering tasks and evaluation settings, which limit direct comparison across findings. The conclusions should therefore be interpreted as a synthesis of the available literature rather than as a complete representation of all research in the field. Second, LLM-based software engineering is changing rapidly. Some of the sources included in the review are recent preprints and therefore have not yet gone through full peer review. Their findings may change as further evaluations become available. Third, the results presented in Table 1 were obtained using different datasets, benchmarks, models, and evaluation protocols. Direct numerical comparison is therefore limited, and the reported values should be interpreted as indicative rather than as measurements under a common experimental setting. Fourth, the four-stage evaluation and assurance framework is a proposal derived from failure patterns identified across the reviewed literature. It has not yet been empirically validated. The appropriate thresholds, ordering, and relative importance of the four stages need to be tested using experiments on real-world repositories. Finally, many of the reviewed studies focus on Java and Python. The extent to which these findings apply to other programming languages remains uncertain, particularly where languages have different type systems, memory models, build processes, or tooling ecosystems.

RESEARCH DIRECTIONS TOWARD 2030

The reviewed literature points to several research directions that could improve the reliability, security, and practical use of LLM-assisted programming in the coming years.

Repository-Scale Context

Current evaluations often focus on individual functions or relatively small tasks. Future systems need to work with complete repositories and maintain an up-to-date view of dependencies such as files, APIs, and project structures that change [25]. This would support more reliable code generation and modification in real development environments.

Neuro-Symbolic Verification

Generative models can be integrated with formal verification, static analysis, and theorem proving techniques to provide stronger validation of generated programs. This combination can supplement probabilistic model outputs with explicit checks and, where supported by the verification method, guarantees concerning properties such as correctness and security [5].

Contamination-Free Benchmarking and Governance

Future benchmarks should be refreshed regularly and use project-level holdouts rather than relying only on individual held-out examples. Standardized data-audit procedures are also needed to identify and reduce memorization and training data contamination. Evaluation reports should consistently identify the benchmark variant and agent scaffold used, allowing results to remain comparable across studies and overtime [3, 12, 48].

Multi-Language and Multi-Paradigm Coverage

Current studies are largely concentrated on Java and Python. Further work should include languages with different memory and type-system characteristics, including Rust, Go, and C++, as well as quantum programming environments such as Qiskit and Cirq [42, 53]. Broader language coverage would help determine whether current findings generalize across programming paradigms.

Calibrated Human-AI Collaboration

As programming systems become more agentic, developers need sufficient information to judge when their outputs can be trusted and when additional verification is required. Transparent workflows, explanation traces, and uncertainty estimates can support better calibration between human review and AI-generated results [2, 52].

CONCLUSION

LLMs are increasingly used for code generation, unit test creation, program repair, and security analysis. The studies reviewed in this study show that a single benchmark score, such as pass@k, does not provide a complete measure of reliability. Function-level benchmark performance has improved rapidly, while studies of security, test quality, LLM-based code review, and repository-scale development continue to identify important failure cases. This study identified recurring failure modes across the main application areas and developed a four-stage evaluation and assurance framework. The framework considers functional execution, testing evidence, security, and maintainability as separate evaluation requirements rather than treating functional correctness as sufficient evidence for accepting generated code.

The proposed framework has not yet been validated through empirical experiments. Future work should apply it to real-world repositories, establish appropriate thresholds for each evaluation stage, and examine its applicability across low-resource programming languages and quantum software. These steps can help determine whether the framework provides a practical approach for evaluating LLM-assisted programming beyond conventional benchmark scores.

REFERENCES

1. Jiang J, Wang F, Shen J, Kim S, Kim S. A survey on large language models for code generation. *ACM Trans Softw Eng Methodol.* 2026;35(2):1-72. doi:10.1145/3747588.
2. Zhang Q, Fang C, Xie Y, Zhang Y, Yu S, Sun W, et al. A survey on large language models for software engineering. *Sci China Inf Sci.* 2026;69(4):141102. doi:10.1007/s11432-025-4670-0.
3. Bhati H. Agentic AI in the software development lifecycle: architecture, empirical evidence, and the reshaping of software engineering [Preprint]. 2026. arXiv:2604.26275.
4. Gulwani S. Dimensions in program synthesis. In: *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*. Hagenberg, Austria: Association for Computing Machinery; 2010. p. 13-24. doi:10.1145/1836089.1836091.
5. Gülmez B. Code generation with large language models: a survey from neural program synthesis to autonomous software development. *Appl Intell.* 2026;56(6):200. doi:10.1007/s10489-026-07230-0.
6. Feng Z, Guo D, Tang D, Duan N, Feng X, Gong M, et al. CodeBERT: a pre-trained model for programming and natural languages. In: *Findings of the Association for Computational Linguistics: EMNLP 2020*. Online: Association for Computational Linguistics; 2020. p. 1536-1547. doi:10.18653/v1/2020.findings-emnlp.139.
7. Wang Y, Wang W, Joty S, Hoi SCH. CodeT5: identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In: *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Online and Punta Cana, Dominican Republic: Association for Computational Linguistics; 2021. p. 8696-8708. doi:10.18653/v1/2021.emnlp-main.685.
8. Chen M, Tworek J, Jun H, Yuan Q, Pinto HP, Kaplan J, et al. Evaluating large language models trained on code [Preprint]. 2021. arXiv:2107.03374.
9. Roziere B, Gehring J, Gloeckle F, Sootla S, Gat I, Tan XE, et al. Code Llama: open foundation models for code [Preprint]. 2023. arXiv:2308.12950.
10. Guo D, Zhu Q, Yang D, Xie Z, Dong K, Zhang W, et al. DeepSeek-Coder: when the large language model meets programming—the rise of code intelligence [Preprint]. 2024. arXiv:2401.14196.
11. Zhu Q, Guo D, Shao Z, Yang D, Wang P, Xu R, et al. DeepSeek-Coder-V2: breaking the barrier of closed-source models in code intelligence [Preprint]. 2024. arXiv:2406.11931.
12. Jain N, Han K, Gu A, Li WD, Yan F, Zhang T, et al. LiveCodeBench: holistic and contamination free evaluation of large language models for code. In: *International Conference on Learning Representations*. Singapore: International Conference on Learning Representations; 2025. p. 25479-25491.
13. Kaniewski S, Schmidt F, Enzweiler M, Menth M, Heer T. A systematic literature review on detecting software vulnerabilities with large language models. *ACM Trans Softw Eng Methodol.* 2026. doi:10.1145/3815425.
14. Jimenez CE, Yang J, Wetteg A, Yao S, Pei K, Press O, et al. SWE-bench: can language models resolve real-world GitHub issues? In: *The Twelfth International Conference on Learning Representations*. Vienna, Austria: International Conference on Learning Representations; 2024.
15. Pearce H, Ahmad B, Tan B, Dolan-Gavitt B, Karri R. Asleep at the keyboard? Assessing the security of GitHub Copilot's code contributions. *Commun ACM.* 2025;68(2):96-105. doi:10.1145/3610721.
16. Dou S, Jia H, Wu S, Zheng H, Wu M, Tao Y, et al. What is wrong with your code generated by large language models? An extensive study. *Sci China Inf Sci.* 2026;69(1):112107. doi:10.1007/s11432-025-4632-8.
17. Petke J, Martinez M, Kechagia M, Aleti A, Sarro F. The patch overfitting problem in automated program repair: practical magnitude and a baseline for realistic benchmarking. In: *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*. Porto de Galinhas, Brazil: Association for Computing Machinery; 2024. p. 452-456. doi:10.1145/3663529.3663776.
18. Hou X, Zhao Y, Liu Y, Yang Z, Wang K, Li L, et al. Large language models for software engineering: a systematic literature review. *ACM Trans Softw Eng Methodol.* 2024;33(8):1-79. doi:10.1145/3695988.

19. Campos V, Shariffdeen R, Ulges A, Noller Y. Exploring generalizable automated program repair with large language models. *IEEE Access*. 2026;14:110543-110558. doi:10.1109/ACCESS.2026.3714292.
20. Wei J, Wang X, Schuurmans D, Bosma M, Ichter B, Xia F, et al. Chain-of-thought prompting elicits reasoning in large language models. In: *Advances in Neural Information Processing Systems* 35. New Orleans, LA, USA: Neural Information Processing Systems Foundation, Inc.; 2022. p. 24824-24837. doi:10.52202/068431-1800.
21. Li J, Rabbi F, Cheng C, Sangalay A, Tian Y, Yang J. An exploratory study on fine-tuning large language models for secure code generation. *Empir Softw Eng*. 2026;31(4):81. doi:10.1007/s10664-026-10803-9.
22. Hu EJ, Shen Y, Wallis P, Allen-Zhu Z, Li Y, Wang S, et al. LoRA: low-rank adaptation of large language models [Preprint]. 2021. arXiv:2106.09685.
23. He J, Vechev M. Large language models for code: security hardening and adversarial testing. In: *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. Copenhagen, Denmark: Association for Computing Machinery; 2023. p. 1865-1879. doi:10.1145/3576915.3623175.
24. Gao Y. Retrieval-augmented generation for large language models: a survey [Preprint]. 2023. arXiv:2312.10997.
25. Tao Y, Li Y, Qin Y, Liu Y. Retrieval-augmented code generation: a survey with focus on repository-level approaches [Preprint]. 2025. arXiv:2510.04905.
26. Zhang F, Chen B, Zhang Y, Keung J, Liu J, Zan D, et al. RepoCoder: repository-level code completion through iterative retrieval and generation. In: *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. Singapore: Association for Computational Linguistics; 2023. p. 2471-2484. doi:10.18653/v1/2023.emnlp-main.151.
27. Zhang Y, Ruan H, Fan Z, Roychoudhury A. AutoCodeRover: autonomous program improvement. In: *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. Vienna, Austria: Association for Computing Machinery; 2024. p. 1592-1604. doi:10.1145/3650212.3680384.
28. Ruan H, Zhang Y, Roychoudhury A. SpecRover: code intent extraction via LLMs. In: *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. Ottawa, ON, Canada: IEEE; 2025. p. 963-974. doi:10.1109/ICSE55347.2025.00080.
29. Austin J, Odena A, Nye M, Bosma M, Michalewski H, Dohan D, et al. Program synthesis with large language models [Preprint]. 2021. arXiv:2108.07732.
30. Luo Z, Xu C, Zhao P, Sun Q, Geng X, Hu W, et al. WizardCoder: empowering code large language models with Evol-Instruct. In: *Proceedings of the Twelfth International Conference on Learning Representations, ICLR 2024*. Vienna, Austria: International Conference on Learning Representations; 2024. p. 1-21.
31. Gunasekar S, Zhang Y, Aneja J, Mendes CC, Del Giorno A, Gopi S, et al. Textbooks are all you need [Preprint]. 2023. arXiv:2306.11644.
32. Huang D, Zhang JM, Luck M, Bu Q, Qing Y, Cui H. AgentCoder: multi-agent-based code generation with iterative testing and optimisation [Preprint]. 2023. arXiv:2312.13010.
33. Zhuo TY, Vu MC, Chim J, Hu H, Yu W, Widyasari R, et al. BigCodeBench: benchmarking code generation with diverse function calls and complex instructions. In: *International Conference on Learning Representations*. 2025.
34. Guo X, Zhao J, Zhao P. On repairing quantum programs using ChatGPT. In: *Proceedings of the 5th ACM/IEEE International Workshop on Quantum Software Engineering, Q-SE 2024*. Lisbon, Portugal: Association for Computing Machinery; 2024. p. 9-16. doi:10.1145/3643667.3648223.
35. Jin H, Chen H. Are LLMs reliable code reviewers? Systematic overcorrection in requirement conformance judgement. *Autom Softw Eng*. 2026;33:90. doi:10.1007/s10515-026-00638-5.
36. Wolff D, Mirchev M, Roychoudhury A. Large language models in software security analysis. *Commun ACM*. 2026;69:60-67. doi:10.1145/3770922.

37. Du X, Liu M, Wang K, Wang H, Liu J, Chen Y, et al. Evaluating large language models in class-level code generation. In: Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (ICSE 2024). Lisbon, Portugal: Association for Computing Machinery; 2024. p. 982-994. doi:10.1145/3597503.3639219.
38. Zhang J, Hu X, Gao C, Xia X, Li S. Enhancing automated unit test generation with large language models: a systematic literature review. *ACM Trans Softw Eng Methodol*. 2026;35:1-67. doi:10.1145/3802827.
39. Celik A, Mahmoud QH. A review of large language models for automated test case generation. *Mach Learn Knowl Extr*. 2025;7:97. doi:10.3390/make7030097.
40. Tasarsu M, Tokmak AV, Catal C. Test case generation using large language models: a systematic literature review. *Cluster Comput*. 2026;29:227. doi:10.1007/s10586-026-06021-z.
41. Lemieux C, Inala JP, Lahiri SK, Sen S. CodaMosa: escaping coverage plateaus in test generation with pre-trained large language models. In: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE). Melbourne, Australia: IEEE; 2023. p. 919-931. doi:10.1109/ICSE48619.2023.00085.
42. Chu B, Feng Y, Liu K, Guo Z, Zhang Y, Shi H, et al. Large language models for unit test generation: achievements, challenges, and opportunities [Preprint]. 2025. arXiv:2511.21382.
43. Xia CS, Zhang L. Less training, more repairing please: revisiting automated program repair via zero-shot learning. In: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. Singapore: Association for Computing Machinery; 2022. p. 959-971. doi:10.1145/3540250.3549101.
44. Xia CS, Wei Y, Zhang L. Automated program repair in the era of large pre-trained language models. In: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE). Melbourne, Australia: IEEE Computer Society; 2023. p. 1482-1494. doi:10.1109/ICSE48619.2023.00129.
45. Hossain SB, Jiang N, Zhou Q, Li X, Chiang WH, Lyu Y, et al. A deep dive into large language models for automated bug localization and repair. *Proc ACM Softw Eng*. 2024;1:1471-1493. doi:10.1145/3660773.
46. Wu F, Liu X, Deceptprompt XC. Exploiting LLM-driven code generation via adversarial natural language instructions [Preprint]. 2023. arXiv:2312.04730.
47. Schuster R, Song C, Tromer E, Shmatikov V. You autocomplete me: poisoning vulnerabilities in neural code completion. In: 30th USENIX Security Symposium (USENIX Security 21). Berkeley, CA, USA: USENIX Association; 2021. p. 1559-1575.
48. Croft R, Babar MA, Kholoosi MM. Data quality for software vulnerability datasets. In: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE). Melbourne, Australia: IEEE; 2023. p. 121-133. doi:10.1109/ICSE48619.2023.00022.
49. Steenhoek B, Rahman MM, Roy MK, Alam MS, Tong H, Das S, et al. To err is machine: vulnerability detection challenges LLM reasoning [Preprint]. 2024. arXiv:2403.17218.
50. Li Z, Lu S, Guo D, Duan N, Jannu S, Jenks G, et al. Automating code review activities by large-scale pre-training. In: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. New York, NY, USA: Association for Computing Machinery; 2022. p. 1035-1047. doi:10.1145/3540250.3549081.
51. Liu F, Liu S, Zhu Y, Lian X, Zhang L. SecureReviewer: enhancing large language models for secure code review through secure-aware fine-tuning. In: Proceedings of the 2026 IEEE/ACM 48th International Conference on Software Engineering. 2026. p. 310-322. doi:10.1145/3744916.3773191.
52. Dobslaw F, Feldt R, Yoon J, Yoo S. Challenges in testing large language model-based software: a faceted taxonomy. *ACM Trans Softw Eng Methodol*. 2026;35:1-24. doi:10.1145/3806396.
53. Ye J, Hu Q, Zhang F, Xia S, Guo X, Wu X, et al. Large language models for quantum software engineering: opportunities and challenges in modeling, generation, testing, and repair. *ACM Trans Softw Eng Methodol*. 2026;35:1-42. doi:10.1145/3828167.