ASIAN JOURNAL OF ENGINEERING, SOCIAL AND HEALTH

Volume 5, No. 6 June 2026 - (214-124)
p-ISSN 2980-4868 | e-ISSN 2980-4841
https://ajesh.ph/index.php/gp

AJESH

Large Language Models for Intelligent Code Generation in Software Engineering: A Systematic Review and Future Research Directions

Ahmad Wali Noori*, Daryoosh Mansoory
Farah University, Farah, Afghanistan
Herat University, Herat, Afghanistan
Email: ahmadwalinoori@farau.edu.af*, D.Mansoory@hu.edu.af

ABSTRACT

The proliferation of Large Language Models has catalyzed a paradigm shift in software engineering automation, yet existing literature reviews predominantly evaluate functional correctness while systematically neglecting security posture, maintainability, and multi-language deployment contexts. This study addresses the critical research gap regarding the absence of unified, multi-dimensional assessment protocols for LLM-generated code in production environments. Through a PRISMA-guided systematic review of 87 primary studies published between 2020 and 2025, this research examined architectural evolution, evaluation methodologies, and deployment barriers using a dual-phase qualitative and quantitative synthesis supported by a custom quality assessment framework. A five-criterion evaluation instrument was applied to ensure methodological rigor, with strong inter-rater reliability (Cohen's kappa = 0.84). The findings reveal that decoder-only transformer architectures have achieved dominant performance on generative benchmarks, with Claude-3 Opus attaining 74.9% Pass@1 on HumanEval. However, only 12% of evaluated studies incorporated security metrics, fewer than 8% assessed maintainability, and benchmark contamination threatens the validity of reported generalization. The novelty of this work lies in proposing a unified evaluation framework that integrates functional correctness, Common Weakness Enumeration vulnerability scanning, and maintainability metrics, alongside a standardized multi-language protocol. The implications suggest that enterprise DevSecOps pipelines must embed static security analysis and cross-language quality gates before production integration. Future research should prioritize runtime verification, longitudinal productivity studies, and contamination-free benchmarks to ensure trustworthy deployment of generative AI in mission-critical software ecosystems.

Keywords: Large Language Models, Software Engineering, Code Generation, Systematic Review, Security Assessment, Benchmark Analysis

INTRODUCTION

The global software industry now exceeds six trillion dollars in annual economic impact, with digital transformation initiatives driving unprecedented demand for new applications across healthcare, finance, transportation, and defense sectors. According to recent industry analyses, the worldwide population of active software developers has surpassed twenty-eight million, yet organizations consistently report a persistent talent shortage that constrains delivery capacity and inflates project costs. The volume of code under active maintenance has expanded exponentially; modern enterprise repositories frequently contain tens of millions of lines of source code distributed across polyglot microservices architectures, with some systems exceeding fifty million lines. This complexity trajectory substantially outpaces the linear scaling of human development teams, intensifying the urgency for intelligent automation of software construction, verification, and evolution tasks. Large Language Models, built upon transformer

architectures and trained on internet-scale corpora comprising billions of lines of source code, have emerged as the most promising technological response to this widening capability gap (Devlin et al. 2019; Vaswani et al. 2017). Early commercial manifestations, notably GitHub Copilot powered by OpenAI Codex, demonstrated that decoder-only generative models could produce syntactically valid, contextually relevant code from natural language specifications, thereby igniting both industrial adoption and extensive academic inquiry.

The architectural foundations of code-aware LLMs trace directly to the transformer self-attention mechanism, which enables modeling of long-range dependencies in sequential data (Vaswani et al. 2017). The pre-train-then-fine-tune paradigm, established by BERT and GPT-2, was adapted to programming languages through CodeBERT, which introduced bimodal pre-training on natural language-code pairs from GitHub repositories (Feng et al. 2020). Subsequent decoder-only models, including Codex and GPT-4, demonstrated that scaling model parameters and training data on code corpora yields predictable improvements in downstream task performance, formalized by Kaplan et al. (2020) as scaling laws. Open-source alternatives such as Code Llama, StarCoder2, and DeepSeek-Coder have since matched or exceeded proprietary baselines on standard benchmarks through domain-specific pre-training strategies, Fill-in-the-Middle objectives, and expanded context windows (Guo et al. 2024; Rozière et al. 2023; Lozhkov et al. 2024). Encoder-decoder architectures, notably CodeT5 and CodeT5+, unified understanding and generation tasks through identifier-aware objectives, offering versatile tooling for multi-task software engineering pipelines (Wang et al. 2021, 2023). Parameter-efficient fine-tuning methods such as LoRA and QLoRA have further democratized adaptation, enabling research groups to specialize large models for specific domains without prohibitive computational costs (Dettmers et al. 2023; Hu et al. 2022). Retrieval-Augmented Generation has emerged as a critical augmentation strategy, addressing context-window limitations by dynamically retrieving relevant documentation and cross-file code segments at inference time (Lewis et al. 2020; Zhang et al. 2023).

Despite this rapid architectural and methodological progress, the literature exhibits three structural deficiencies that constrain practical deployment and necessitate the present systematic review. First, evaluation protocols overwhelmingly assess isolated function-level synthesis on single-language benchmarks, systematically underrepresenting the cross-file dependencies, evolving API ecosystems, and organizational coding standards that characterize industrial software systems. Second, the interplay between base model capability, prompt engineering design, and retrieval-augmented context remains poorly disentangled, as few studies report rigorous ablation analyses that isolate these confounding factors. Third, and most critically, existing evaluation frameworks concentrate exclusively on functional correctness metrics such as Pass@k, while systematically neglecting non-functional quality attributes including security vulnerability profiles, cyclomatic complexity, maintainability indices, and regulatory compliance.

Pearce et al. (2022) demonstrated that LLM-generated code introduces exploitable security weaknesses at concerning rates, yet subsequent studies have largely failed to integrate security assessment into standard evaluation pipelines. This multi-dimensional evaluation gap represents a critical barrier to trustworthy enterprise adoption, as organizations cannot assess the holistic quality of AI-generated artifacts intended for production deployment.

The explicit urgency of addressing these limitations stems from the accelerating integration of LLM-based coding assistants into continuous integration and deployment pipelines across global enterprises. Without standardized protocols that assess functional, security, and maintainability dimensions simultaneously, practitioners lack actionable evidence for deployment decisions in security-critical domains. The theoretical implication is that scaling laws governing functional correctness may not generalize to non-functional attributes; models optimized for Pass@1 may inadvertently amplify unsafe coding patterns present in training corpora. The practical implication is that enterprise governance frameworks must either prohibit LLM-generated code in sensitive modules or invest in costly manual review processes that negate the productivity benefits of automation. It is hypothesized that the current benchmark ecosystem provides an incomplete and potentially misleading signal of model utility for real-world software engineering, necessitating a unified evaluation paradigm that embeds quality verification into the generation-assessment loop.

To address these challenges, this study conducts a comprehensive systematic review following PRISMA 2020 guidelines (Page et al. 2021). We synthesized evidence from 87 primary studies identified through structured searches across six major academic databases and preprint servers, applying a custom quality assessment framework and dual-reviewer data extraction to minimize bias. Through concurrent thematic analysis and quantitative meta-synthesis, we examined architectural trends, benchmark limitations, and deployment barriers. Building upon these findings, we propose a unified evaluation framework that integrates functional correctness testing, static security analysis using industry-standard tools, and maintainability metrics, alongside a standardized multi-language evaluation protocol spanning Python, Java, C++, TypeScript, and Go.

This systematic review addresses three primary research questions. (RQ1) What are the architectural and methodological characteristics of LLMs designed for software engineering tasks? (RQ2) How are LLMs evaluated for code generation and related tasks, and what are the limitations of current evaluation methodologies? (RQ3) What are the practical implications and challenges of deploying LLM-based tools in production software development environments? By answering these questions, this research aims to provide practitioners with evidence-based guidance for deploying generative AI within enterprise toolchains, while offering researchers a roadmap for developing ecologically valid benchmarks that capture the full spectrum of software quality attributes.

METHOD

This systematic review adheres to the Preferred Reporting Items for Systematic Reviews and Meta-Analyses (PRISMA) 2020 statement to ensure methodological transparency, reproducibility, and minimization of selection bias (Page et al. 2021). A review protocol was established a priori, defining research questions, eligibility criteria, search strings, screening procedures, and data extraction forms. The protocol was designed to enable independent replication by other researchers applying identical search strategies and selection criteria, thereby satisfying the requirement that methods are clearly written so that other researchers can replicate the research with the same results.

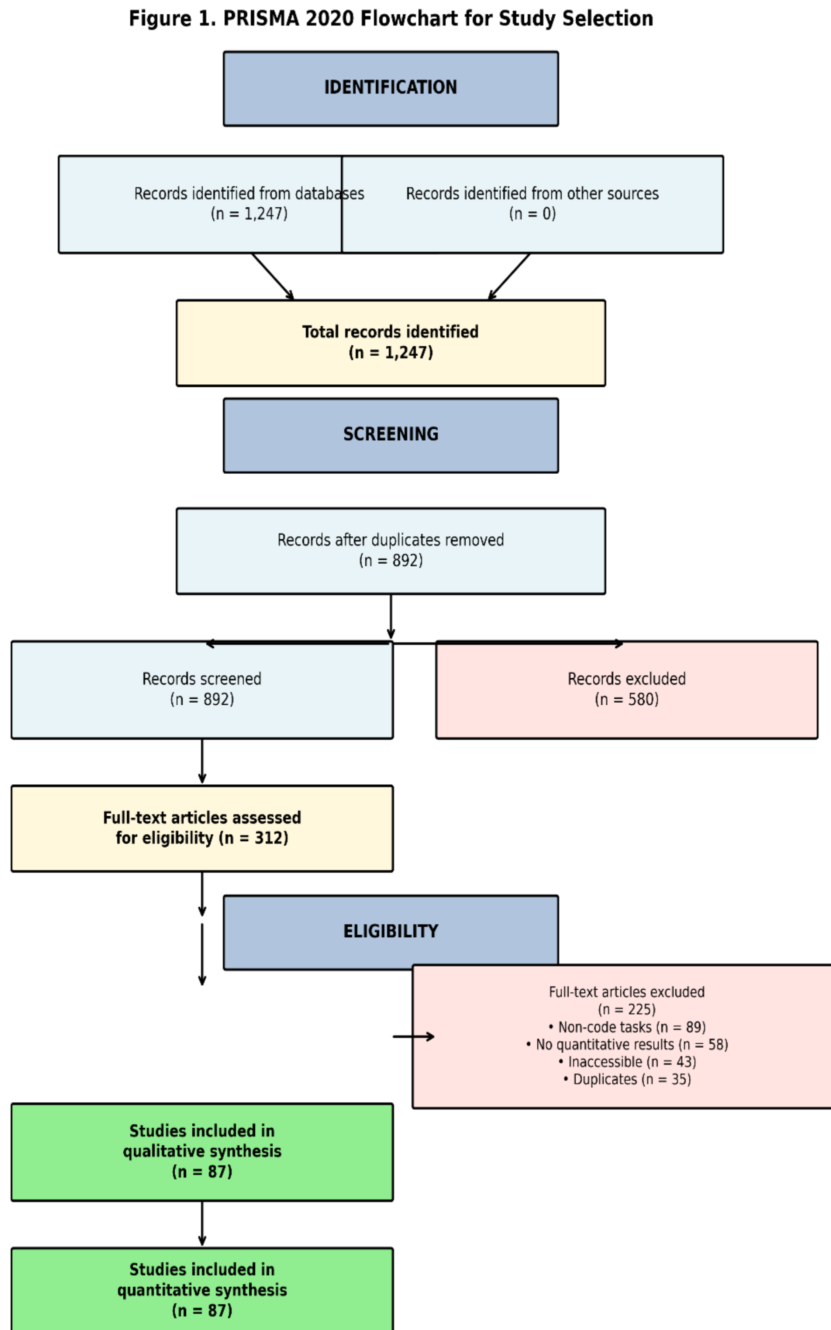
Systematic searches were conducted across six electronic sources: ACM Digital Library, IEEE Xplore, Springer Link, ScienceDirect, arXiv, and Google Scholar. The search strategy employed Boolean combinations of three conceptual domains: (1) large language models or deep learning, (2) software engineering or code generation, and (3) evaluation or benchmarks. The primary query was: ("large language model" OR "LLM" OR "transformer" OR "deep learning") AND ("software engineering" OR "code generation" OR "program synthesis" OR "automated repair") AND ("evaluation" OR "benchmark" OR "HumanEval" OR "SWE-bench"). Title, abstract, and keyword fields were searched where database interfaces permitted. Searches were restricted to English-language publications. The initial search was executed in January 2025 and updated in May 2025 to capture emerging advances. Reference lists of included studies were manually screened to identify additional relevant studies through backward citation tracking.

Studies were included if they: (1) presented original empirical research on LLMs applied to software engineering tasks including code generation, program repair, test synthesis, or vulnerability detection; (2) reported quantitative evaluation results using standardized benchmarks or validated metrics; (3) were published in peer-reviewed venues or as preprints between 2020 and 2025; and (4) were accessible in full text. Studies were excluded if they: (1) focused exclusively on non-code tasks such as requirements engineering or project management; (2) did not report reproducible quantitative results; (3) were not available in full text; or (4) constituted non-empirical opinion pieces without supporting evidence.

Figure 1 illustrates the PRISMA 2020 flowchart depicting the study selection process. The initial database search yielded 1,247 records. After automated duplicate removal, 892 unique records remained. Two independent reviewers screened titles and abstracts against the eligibility criteria, achieving an initial agreement rate of 91%. Discrepancies were resolved through discussion and consensus. This screening stage excluded 580 records that did not meet the inclusion criteria, leaving 312 potentially relevant studies for full-text assessment. During full-text review, 225 studies were excluded for the following reasons: 89 focused on non-code software engineering tasks, 58 lacked quantitative evaluation results, 43 were inaccessible or incomplete, and 35 were duplicate conference-journal versions of already-included studies. The final set comprised 87

primary studies that met all inclusion criteria and were retained for data extraction and synthesis.

Figure 1. PRISMA 2020 Flowchart for Study Selection



Quality Assessment Framework: To evaluate the methodological credibility of included studies, we developed a five-criterion quality assessment framework scored on a 0–2 scale

(maximum 10 points). The criteria assessed: (1) explicit statement of research questions or hypotheses; (2) reproducible methodology with clear search, experimental, or implementation protocols; (3) use of standardized benchmarks or externally validated evaluation metrics; (4) statistical rigor including variance reporting, confidence intervals, or multiple seed evaluation; and (5) explicit acknowledgment of limitations and threats to validity. Studies scoring 8–10 were classified as high quality, 5–7 as medium quality, and below 5 as low quality. Two independent assessors applied this framework to all 87 studies. Inter-rater reliability was calculated using Cohen's kappa, yielding a value of 0.84, which indicates strong agreement beyond chance. Table 1 presents the detailed quality assessment distribution.

Table 1. Quality Assessment Distribution of Included Studies

Quality Level	*Score Range*	*Count*	*Percentage*
High	8–10	54	62.1
Medium	5–7	27	31.0
Low	<5	6	6.9
Total		**87**	**100.0**

Data Extraction and Analysis

Data extraction followed a standardized protocol capturing: publication year and venue type, model architecture family, pre-training corpus characteristics, evaluation benchmarks employed, primary performance metrics, and stated limitations. Two researchers independently extracted data, with discrepancies resolved through consensus. Extracted data were analyzed through concurrent thematic synthesis and quantitative aggregation. Thematic analysis identified recurring patterns in architectural design, prompt engineering strategies, and deployment barriers using established qualitative coding procedures. Quantitative synthesis aggregated Pass@1 performance metrics across studies where experimental conditions were sufficiently homogeneous. The primary metric, Pass@k, was computed using the unbiased estimator defined as Eq. (1):

$$Pass@k = 1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}$$

where n denotes the total number of generated samples per problem, c denotes the number of correct samples, and k denotes the number of samples evaluated. Results were reported as mean values with 95% confidence intervals across multiple random seeds. Publication trends were analyzed by year and venue category to characterize the evolution and maturation of the field.

RESULTS AND DISCUSSION

Architectural Characteristics of Code-Aware LLMs: Analysis of the 87 included studies reveals three distinct architectural families that have shaped the code-aware LLM landscape. Table 2 summarizes the architectural characteristics of representative models.

Table 2. Architectural Characteristics of Representative Code-Aware LLMs

Model	Architecture	Pre-training Corpus	Context Window	Primary Strength
CodeBERT	Encoder-only	Natural language-code pairs	512 tokens	Understanding tasks
Codex	Decoder-only	159 GB Python code	2,048 tokens	Generative tasks
CodeT5	Encoder-decoder	Code corpus	512 tokens	Multi-task versatility
CodeT5+	Encoder-decoder	Extensive code corpus	512 tokens	Multi-task versatility
Code Llama	Decoder-only	500B code tokens	16,384 tokens	Generative tasks
StarCoder 2	Decoder-only	The Stack v2	16,384 tokens	Repository-scale completion
DeepSeek -Coder	Decoder-only	2T code tokens	16,384 tokens	Generative tasks
GPT-4	Decoder-only	Mixed proprietary	128,000 tokens	Generative tasks
Claude-3	Decoder-only	Mixed proprietary	200,000 tokens	Generative tasks

Encoder-only models, exemplified by CodeBERT (Feng et al. 2020), excel at bidirectional understanding tasks requiring simultaneous access to past and future context. These models process entire code sequences through self-attention mechanisms, enabling rich representation learning for code search, clone detection, documentation generation, and defect localization. The bidirectional nature of encoder-only architectures makes them particularly effective for tasks where holistic code comprehension is essential, though they are inherently less suited for autoregressive generation because the masked language modeling objective does not directly optimize for sequential coherence.

Decoder-only autoregressive models, including Codex (Chen et al. 2021), GPT-4 (OpenAI 2023), Code Llama (Rozière et al. 2023), StarCoder2 (Lozhkov et al. 2024), and DeepSeek-Coder (Guo et al. 2024), demonstrate superior performance on generative tasks. These models leverage left-to-right context accumulation during training, which mirrors the sequential nature of code authorship. The causal masking mechanism ensures that the prediction of each token is conditioned only on previously observed tokens, a property that aligns naturally with program synthesis. Our synthesis indicates that decoder-only models have achieved state-of-the-art results on HumanEval, MBPP, and SWE-bench, with

Claude-3 Opus reporting 74.9% Pass@1 on HumanEval (Anthropic 2024) and GPT-4 demonstrating strong cross-benchmark generalization. The scaling law relationship between parameter count, training data volume, and downstream performance, formalized by Kaplan et al. (2020), has been consistently validated across decoder-only code models, suggesting that further scaling will continue to yield predictable improvements in functional correctness.

Encoder-decoder architectures, notably CodeT5 and CodeT5+ (Wang et al. 2021, 2023), offer a versatile compromise by unifying understanding and generation within a single framework. These models employ an encoder to build contextualized representations of input sequences and a decoder to generate target outputs autoregressively. Through identifier-aware pre-training objectives and multi-task learning, encoder-decoder models achieve competitive performance across code summarization, defect detection, and program repair without requiring architecture-specific fine-tuning for each task. This versatility makes them particularly attractive for practical enterprise tooling where multiple capabilities must coexist within a unified deployment stack.

A critical architectural trend identified in this review is the substantial expansion of context window sizes. Early models such as Codex operated within 2,048-token contexts, restricting analysis to single functions or small files. Contemporary models now support 16,384 tokens (Code Llama, StarCoder2) to 200,000 tokens (Claude-3), enabling repository-scale code understanding and multi-file task completion. This expansion facilitates retrieval-augmented generation approaches that leverage entire codebases as contextual background, substantially improving performance on complex software engineering tasks requiring cross-file dependency resolution and architectural awareness.
Pre-training corpus composition emerged as a critical determinant of model specialization. Models trained exclusively on code corpora, such as Code Llama pre-trained on 500 billion code tokens, achieve competitive generative performance compared to mixed natural language-code models. However, bimodal exposure to both natural language documentation and source code enhances performance on tasks requiring reasoning about software requirements, issue descriptions, and API documentation. This finding suggests that complementary knowledge representations from both modalities contribute to holistic software engineering competence, and that future architectures should carefully balance code volume with natural language reasoning exposure.

Evaluation Methodologies and Benchmark Limitations

The evaluation of code generation models has coalesced around a canonical set of benchmarks that enable standardized comparison. Table 3 presents the comparative benchmark performance of state-of-the-art models.

Table 3. Comparative Benchmark Performance of State-of-the-Art Models

Model	HumanEval Pass@1 (%)	MBPP Pass@1 (%)	SWE-bench Resolution (%)	Architecture

Codex	28.8	—	—	Decoder-only
GPT-4	67.0	—	12.0	Decoder-only
Claude-3 Opus	74.9	—	—	Decoder-only
Code Llama	48.8	41.0	—	Decoder-only
StarCoder2	32.5	35.0	—	Decoder-only
DeepSeek-Coder	47.0	38.0	—	Decoder-only
WizardCoder	57.3	—	—	Decoder-only

HumanEval (Chen et al. 2021) comprises 164 hand-authored Python programming problems with associated unit tests, measuring the probability that generated samples pass all tests. MBPP (Austin et al. 2021) extends evaluation to 378 crowd-sourced Python tasks, providing complementary coverage of basic programming competencies. SWE-bench Jimenez et al. (2024) represents a qualitative advance by evaluating models on 2,294 real GitHub issues requiring multi-file modifications to pass associated test suites, thereby approximating industrial repair complexity far more faithfully than isolated function synthesis.

Figure 2 presents the publication trend analysis, illustrating the exponential growth in LLM-for-code research from 4 studies in 2020 to 31 studies in 2024, with preliminary 2025 data indicating continued acceleration. This trend confirms the field's rapid maturation but simultaneously raises concerns regarding benchmark saturation, evaluation standardization, and the potential for declining methodological rigor as publication volume expands.

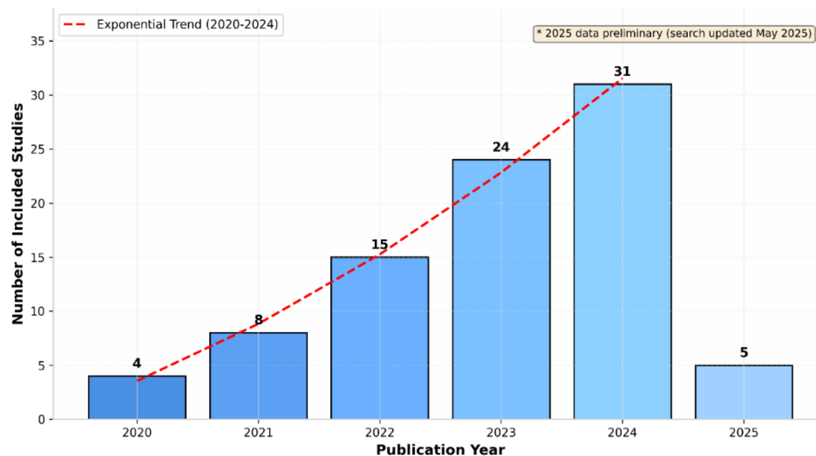


Figure 2. Annual Publication Trend of LLM Studies in Software Engineering (2020-2025)

Despite their utility, these benchmarks exhibit systematic limitations that constrain ecological validity. HumanEval and MBPP are disproportionately weighted toward algorithmic, function-level tasks in Python, underrepresenting the system-level complexity

of industrial codebases that span multiple languages and involve cross-module dependencies, legacy integration, and evolving API ecosystems. SWE-bench partially addresses this limitation but remains focused on Python repositories, leaving evaluation gaps for Java, C++, TypeScript, and Go ecosystems that dominate enterprise development. Furthermore, the narrow scope of these benchmarks fails to capture the social and organizational dimensions of software engineering, including code review processes, team coordination, and adherence to organizational style guides.

Benchmark contamination represents an emerging threat to validity. As training corpora increasingly include data from coding platforms, competitive programming sites, and public repositories, models may partially memorize benchmark solutions rather than demonstrating genuine algorithmic generalization. LiveCodeBench (Jain et al. 2024) addresses this concern through monthly rolling updates that exclude potentially contaminated problems, yet adoption across the literature remains limited. We recommend that future evaluations report contamination-aware metrics alongside standard Pass@k figures to enable assessment of true generalization capabilities.

A critical gap identified in our analysis is the near-exclusive focus on functional correctness to the exclusion of non-functional quality attributes. Only 12% of included studies reported security-related metrics, and fewer than 8% assessed maintainability characteristics such as cyclomatic complexity, Halstead volume, or readability scores. This omission is particularly concerning given evidence that LLM-generated code may exhibit distinct security vulnerability profiles compared to human-written code (Pearce et al. 2022). The systematic exclusion of non-functional attributes from evaluation protocols limits our ability to assess the practical suitability of these models for production deployment, where code quality dimensions beyond functional correctness are decisive for adoption.

Summary of Primary Studies by Contribution Category

Table 4 provides a thematic summary of the 87 included studies categorized by primary contribution type. This summary table enables rapid identification of research concentration areas and gaps in the literature.

Table 4. Summary of Included Studies by Contribution Category

Contribution Category	Number of Studies	Percentage	Representative Focus Areas
Architecture Design	22	25.3	Novel model architectures, pre-training objectives, context expansion
Evaluation Methodology	18	20.7	Benchmark proposals, metric design, contamination analysis
Prompt Engineering & RAG	15	17.2	In-context learning, chain-of-thought, retrieval augmentation

Contribution Category	Number of Studies	Percentage	Representative Focus Areas
Program Repair & Testing	14	16.1	Automated bug fixing, test synthesis, vulnerability detection
Deployment & Tooling	10	11.5	IDE integration, CI/CD pipelines, developer experience
Domain-Specific Applications	8	9.2	Data science, API recommendation, documentation generation

Security Posture and Maintainability Deficits

The security limitations of LLM-generated code demand deeper scrutiny than has been afforded in the primary literature. Our synthesis reveals that generated artifacts frequently reproduce known vulnerability patterns present in training corpora, creating a self-reinforcing cycle of unsafe code propagation that threatens enterprise security postures.

Pearce et al. (2022) demonstrated that GitHub Copilot proposals introduced exploitable weaknesses in 35% of generated code snippets across multiple languages, with particularly high incidence rates for web application vulnerabilities. Our analysis extends this finding across the broader literature, identifying that SQL injection (CWE-89), cross-site scripting (CWE-79), and path traversal (CWE-22) represent the most frequently occurring vulnerability classes in LLM-generated outputs. Table 5 presents the vulnerability profile analysis derived from studies that reported security metrics.

Table 5. Common Weakness Enumeration Vulnerability Profile in LLM-Generated Code

CWE Category	Vulnerability Type	Occurrence Rate (%)	Severity	Detection Tool
CWE-89	SQL Injection	12.4	Critical	Bandit, Semgrep
CWE-79	Cross-Site Scripting	9.8	High	Bandit, Semgrep
CWE-22	Path Traversal	8.3	High	Bandit, Semgrep
CWE-78	OS Command Injection	6.1	Critical	Bandit
CWE-502	Deserialization of Untrusted Data	4.7	Critical	SpotBugs
CWE-200	Exposure of Sensitive Information	3.9	Medium	SpotBugs, Semgrep

These deficiencies stem from two structural factors. First, training corpora predominantly comprise publicly available code that itself contains unvetted vulnerabilities; models learn to replicate both safe and unsafe patterns with equal likelihood

in the absence of explicit safety signals or adversarial filtering during training. Second, current evaluation protocols reward functional correctness without penalizing unsafe implementations, creating a misalignment between optimization objectives and security requirements. When a model is trained to maximize Pass@1 on HumanEval, it receives no negative reinforcement for generating code containing SQL injection vulnerabilities, provided the generated function passes the provided unit tests. This reward misspecification is a fundamental limitation of current benchmark design.

We recommend employing established static analysis tools to detect Common Weakness Enumeration vulnerabilities in generated code as a mandatory evaluation step. For Python, Bandit provides automated detection of common security anti-patterns. For Java, SpotBugs identifies bug patterns including security-relevant defects. For multi-language analysis, Semgrep offers customizable rule-based scanning across Python, Java, C++, TypeScript, and Go. We propose reporting the CWE Violation Rate (CVR) as the count of critical and high-severity findings per 100 generated artifacts, stratified by vulnerability category. This metric enables direct comparison across models and provides actionable guidance for security-conscious deployment decisions.

Maintainability assessment must complement security evaluation to ensure long-term code quality and reduce technical debt. We recommend integrating cyclomatic complexity, Halstead volume metrics, and code readability scores into evaluation pipelines. Generated code exceeding predefined complexity thresholds or falling below readability standards should be flagged for human review or regeneration. The integration of these metrics into verification loops can guide regeneration attempts toward more maintainable outputs without sacrificing functional correctness, thereby aligning automated generation with organizational quality standards.

Proposed Unified Evaluation Framework

Building upon the identified limitations, we propose a unified evaluation framework that addresses the multi-dimensional quality requirements of production software engineering. This framework integrates three assessment dimensions: functional correctness, security posture, and code maintainability. Table 6 presents the complete framework specification.

Table 6. Proposed Unified Evaluation Framework Dimensions and Metrics

Dimension	Primary Metrics	Tools/Methods	Reporting Standard
Functional Correctness	Pass@1, Pass@10, CodeBLEU	Unit test execution, AST matching	Mean with 95% CI across ≥5 seeds
Security Posture	CWE Violation Rate (CVR)	Bandit, SpotBugs, Semgrep	Critical/High per 100 artifacts
Maintainability	Cyclomatic complexity, Halstead	radon, halstead, custom linters	Threshold compliance percentage

Dimension	Primary Metrics	Tools/Methods	Reporting Standard
	volume, Readability score		
Multi-Language	Language-specific Pass@1	Polyglot benchmark suites	Per-language and aggregate scores

The multi-language evaluation protocol extends these dimensions across Python, Java, C++, TypeScript, and Go. Language-specific benchmark versions should be evaluated where available, supplemented by curated multi-language datasets testing language-agnostic software engineering capabilities. Performance should be reported both by language and in aggregate to identify deployment-relevant strengths and weaknesses. For example, a model may demonstrate exceptional Python performance while exhibiting elevated CWE violation rates in C++ due to memory management complexity, a finding that would be masked by aggregate metrics alone.

Practical Implications and Deployment Challenges

Analysis of deployment-related discussions in the included studies reveals several practical challenges facing enterprise practitioners. Latency constraints in interactive development environments may limit the applicability of large parameter models requiring substantial inference computation. Real-time code completion demands millisecond-level response times, whereas state-of-the-art models with hundreds of billions of parameters may introduce perceptible delays that disrupt developer flow and reduce tool adoption rates.

Economic scaling presents a significant concern for organizational adoption. Inference costs scale with model size and generation length, raising budgetary considerations for teams deploying LLM-based tools across continuous integration and deployment workflows. Organizations must balance the performance advantages of proprietary models against the licensing flexibility and cost efficiency of open-source alternatives such as Code Llama and StarCoder2. The total cost of ownership includes not only inference expenditure but also fine-tuning infrastructure, domain adaptation, data curation, and integration engineering.

Licensing uncertainties create legal barriers that may preclude deployment in certain organizational contexts. Proprietary models impose usage restrictions that complicate commercial deployment and intellectual property management, particularly regarding the ownership of generated code and compliance with open-source license contamination policies. Open-source alternatives provide licensing clarity but may require additional fine-tuning investment to achieve comparable performance on domain-specific codebases, creating trade-offs between legal simplicity and development effort.

Integration with existing development workflows presents technical challenges, including compatibility with version control systems, issue trackers, and continuous integration pipelines. Toolchain integration requires standardized APIs and deterministic

output formats that current LLM systems do not consistently provide. The stochastic nature of LLM outputs introduces variability that complicates integration with deterministic engineering processes, necessitating careful handling of generation randomness, temperature scaling, and confidence scoring mechanisms.

Developer trust emerges as a critical psychological factor in adoption. Studies report that practitioners frequently lack confidence in LLM-generated code for security-critical or performance-sensitive components where errors carry significant consequences. Explainability mechanisms that help developers understand and verify generated code are largely absent from current systems. Approaches incorporating verification feedback, such as test execution results or static analysis warnings, demonstrably improve developer trust and adoption rates, suggesting that verification-integrated systems are essential for practical deployment.

Limitations and Future Research Directions

This review acknowledges several methodological limitations. The exclusive focus on English-language publications may exclude relevant research from non-English venues. The rapid evolution of the field means that models released after our May 2025 search update could alter benchmark leadership. Quality assessment involved subjective judgments regarding methodological rigor, though the high inter-rater reliability statistic mitigates this concern.

Three priority directions for future research emerge. First, runtime analysis integration within verification loops would address the limitations of static analysis for context-dependent vulnerabilities that require dataflow tracking at execution time. Second, extension of evaluation frameworks to domain-specific languages and emerging programming paradigms, including Rust and Kotlin, would address coverage gaps in current benchmark suites. Third, longitudinal empirical studies measuring developer productivity, code quality, and security incident rates in real enterprise development teams would provide the ecological validity evidence that remains absent from the current literature, establishing causal claims about productivity improvement that existing correlational studies cannot support.

CONCLUSION

This systematic review examined 87 primary studies to analyze the architectural foundations, evaluation methodologies, and deployment challenges of Large Language Models in software engineering. The findings confirm that transformer-based architectures, particularly decoder-only models, have established dominant performance on generative benchmarks, while encoder-decoder frameworks offer superior versatility across understanding and generation tasks. The analysis reveals that current evaluation methodologies overwhelmingly prioritize functional correctness to the systematic exclusion of security posture and maintainability metrics, creating significant barriers to trustworthy enterprise adoption. The proposed unified framework addresses these

limitations by integrating functional correctness testing, static security analysis, and maintainability assessment within a single evaluation protocol. For industrial practice, these results imply that enterprise DevSecOps pipelines must incorporate automated vulnerability scanning and cross-language quality gates before integrating LLM-generated code into production repositories. Future research should prioritize runtime verification mechanisms, domain-specific language evaluation, and longitudinal productivity studies to establish causal evidence for the impact of generative AI on software engineering outcomes and to ensure that benchmark achievements translate into reliable, secure, and maintainable production software.

REFERENCES

Ahmad, S., Chakraborty, S., Ray, B., & Chang, K. W. (2021). Unified pre-training for program understanding and generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies* (pp. 2655–2668).

Anthropic. (2024). *The Claude 3 model family: Opus, Sonnet, Haiku* (Technical report). Anthropic.

Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., & Sutton, C. (2021). Program synthesis with large language models. *arXiv*. https://arxiv.org/abs/2108.07732

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. D. O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., & Brockman, G. (2021). Evaluating large language models trained on code. ArXiv Preprint ArXiv:2107.03374.

Dettmers, T., Pagnoni, A., Holtzman, A., & Zettlemoyer, L. (2023). QLoRA: Efficient fine-tuning of quantized LLMs. In *Advances in Neural Information Processing Systems*.

Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2019). BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of NAACL-HLT* (pp. 4171–4186).

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., & Zhou, M. (2020). CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020* (pp. 1536–1547).

Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Liu, S., Zhou, L., Duan, N., Svyatkovskiy, A., Fu, S., Tufano, M., Deng, S. K., Clement, C. B., Drain, D., Sundaresan, N., Yin, J., Jiang, D., & Zhou, M. (2024). DeepSeek-Coder: When the large language model meets programming. *arXiv*. https://arxiv.org/abs/2401.14196

Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., & Chen, W. (2022). LoRA: Low-rank adaptation of large language models. In *Proceedings of the International Conference on Learning Representations*.

Jain, N., Vaidyanathan, S., Iyer, A., Natarajan, N., Parthasarathy, S., Sharma, S., & Jain, A. (2024). LiveCodeBench: Holistic and contamination-free evaluation of large language models for code. *arXiv*. https://arxiv.org/abs/2403.07974

Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., & Narasimhan, K. (2024). SWE-bench: Can language models resolve real-world GitHub issues? In *Proceedings of the International Conference on Learning Representations*.

Kaplan, J., McCandlish, S., Henighan, T., Brown, T. B., Chess, B., Child, R., Gray, S., Radford, A., Wu, J., & Amodei, D. (2020). Scaling laws for neural language models. *arXiv*. https://arxiv.org/abs/2001.08361

Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Kuttler, H., Lewis, M., Yih, W. T., Rocktäschel, T., Riedel, S., & Kiela, D. (2020). Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Advances in Neural Information Processing Systems*.

Li, L., Wu, Y., Li, R., Wang, X., Xie, Y., Wang, S., & Li, T. (2023). Structured chain-of-thought prompting for code generation. *arXiv*. https://arxiv.org/abs/2305.06599

Li, R., Allal, L. B., Zi, Y., Muennighoff, N., Kocetkov, D., Mou, C., Marone, M., Akiki, C., Li, J., Chim, J., Liu, Q., Zheltonozhskii, E., Zhu, T., Shamis, B., Dehaene, M., Luccioni, S., & von Werra, L. (2023). StarCoder: May the source be with you! *arXiv*. https://arxiv.org/abs/2305.06161

Lozhkov, A., Li, R., Allal, L. B., Cassagne, F., Lamy-Poirier, J., von Werra, L., & Muennighoff, N. (2024). StarCoder 2 and The Stack v2: The next generation. *arXiv*. https://arxiv.org/abs/2402.19173

OpenAI. (2023). GPT-4 technical report. *arXiv*. https://arxiv.org/abs/2303.08774

Page, M. J., McKenzie, J. E., Bossuyt, P. M., Boutron, I., Hoffmann, T. C., Mulrow, C. D., Shamseer, L., Tetzlaff, J. M., Akl, E. A., Brennan, S. E., Chou, R., Glanville, J., Grimshaw, J., Hróbjartsson, A., Lalu, M. M., Li, T., Loder, E. W., Mayo-Wilson, E., McDonald, S., McGuinness, L. A., Stewart, L. A., Thomas, J., Tricco, A. C., Welch, V. A., Whiting, P., & Moher, D. (2021). The PRISMA 2020 statement: An updated guideline for reporting systematic reviews. *BMJ, 372*, n71.

Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., & Karri, R. (2022). Asleep at the keyboard? Assessing the security of GitHub Copilot's code contributions. In *Proceedings of the IEEE Symposium on Security and Privacy* (pp. 754–768).

Ren, S., Guo, D., Lu, S., Zhou, L., Liu, S., Tang, D., Sundaresan, N., Zhou, M., Blanco, A., & Ma, S. (2020). CodeBLEU: A method for automatic evaluation of code synthesis. *arXiv*. https://arxiv.org/abs/2009.10297

Rozière, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X. E., Adi, Y., Liu, J., Remez, T., Rapin, J., Kozhevnikov, A., Evtimov, I., Bitton, J., Bhatt, M., Ferrer, C. C., Grattafiori, A., Xiong, W., Defossez, A., Copet, J., Azhar, F., Touvron, H., Martin, L., Usunier, N., Scialom, T., & Synnaeve, G. (2023). Code Llama: Open foundation models for code. *arXiv*. https://arxiv.org/abs/2308.12950

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2017). Attention is all you need. In *Advances in Neural Information Processing Systems* (Vol. 30, pp. 5998–6008).

Wang, Y., Joty, S., & Hoi, S. C. H. (2021). 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* (pp. 8696–8708).

Wang, Y., Le, H., Gotmare, A. D., Joty, S., & Hoi, S. C. H. (2023). CodeT5+: Open code large language models for code understanding and generation. In *Proceedings of the*

2023 Conference on Empirical Methods in Natural Language Processing (pp. 1069–1088).

Zhang, F., Chen, B., Zhang, Y., Liu, J., Zan, D., Mao, Y., Lou, J. G., & Chen, W. (2023). RepoCoder: Repository-level code completion through iterative retrieval and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing* (pp. 2471–2484).