From Code Completion to Correct Programs: Human-Machine Differences and Hierarchical Debugging

Abstract

Ai-Assisted Programming poses a recurring problem of coordinating performance with evidence quality, resource limits, and transfer across settings. The review develops an evidence-centered account of treating code generation as an iterative reasoning and execution process rather than one-shot text prediction. The source set brings together 2 focal papers with 13 independently retrieved publications reviewed against traceable publication metadata. The analysis is organized around programmer behavior, error localization, execution feedback, repair hierarchy, and evaluation leakage. Instead of assuming that reported outcomes as directly interchangeable, the review compares study questions, technical premises, and validation scope. Across the literature, the central lesson is that advances in AI-assisted programming become credible when representation, objective, and evaluation protocol are evaluated together and when uncertainty about distribution shift is reported explicitly. The resulting account aligns method selection to implementation risk, making transfer failures visible, and proposes a research agenda centered on auditable baselines, controlled perturbations, and replicable records.

Keywords

Ai-Assisted Programming; Programmer Behavior; Error Localization; Execution Feedback; Repair Hierarchy; Evaluation Leakage

1. Introduction

The evidence base for AI-assisted programming connects scientific ambition and practical constraint. The objective includes not only stronger nominal performance but also explanations that locate performance within its operating envelope. This difficulty is evident in treating code generation as an iterative reasoning and execution process rather than one-shot text prediction. Evidence that appears persuasive within a specific substrate, benchmark, population, spatial unit, or workload can erode beyond the conditions that produced it. Consequently, synthesis must preserve the unit of analysis and the decision context. The review additionally distinguishes a mechanism from a correlation, a benchmark advantage from a deployment advantage, and an interpretable diagnostic from a causal explanation.

The analytical frame has five dimensions: programmer behavior, error localization, execution feedback, repair hierarchy, evaluation leakage. Their combination matters because they jointly cover design decisions, measurement practice, and the assumptions that support transfer. Its governing principle is representation, objective, and evaluation protocol. Under that principle, innovation must be accompanied by validation; the comparison also asks whether baselines are aligned, whether uncertainty is visible, and whether resource or safety constraints are represented. This contribution is a literature synthesis and adds no fabricated experiment, participant set, measurement, or model result.

2. Methodological Foundations

A useful conceptual decomposition begins with the input evidence, the transformation applied to it, the output used for evaluation, and the decision that follows. In AI-assisted programming, each element is often assessed independently even though errors propagate across them. An expressive model can magnify noise or bias already present in its evidence; an apparently accurate output can still be poorly calibrated for the final decision. The implication is that, ablation evidence should accompany aggregate performance. An auditable study identifies the constants, perturbations, and uncontrolled factors, then selects metrics that correspond to the intended use rather than to convenience alone.

The next requirement is control of scope. Programmer behavior defines the local design problem, while evaluation leakage determines whether the design can survive outside that boundary. Connecting the endpoints are error localization and execution feedback connect those ends by exposing trade-offs that a single score conceals. At this point, null findings and perturbation analysis reveals the interval in which an explanation can still be defended. For applied work, documentation of deployment constraints is therefore part of the scientific result, not an administrative afterthought.

3. Architecture and Learning Objectives

The target study GULU-01, Between lines of code: Unraveling the distinct patterns of machine and human programmers [1], addresses a concrete part of AI-assisted programming through comparative analysis of observable coding patterns produced by people and machines. It identifies a representation, mechanism, or evaluation boundary needed to compare claims. Against the focus on treating code generation as an iterative reasoning and execution process rather than one-shot text prediction, the paper is interpreted within its documented design envelope: transfer may depend on its sensing regime, preparation, workload, population, or benchmark. The synthesis protects the study's stated scope while auditing its assumptions comparatively.

The target study GULU-02, From code to correctness: Closing the last mile of code generation with hierarchical debugging [2], addresses a concrete part of AI-assisted programming through hierarchical debugging that closes the gap between generated code and executable correctness. It identifies a representation, mechanism, or evaluation boundary needed to compare claims. Against the focus on treating code generation as an iterative reasoning and execution process rather than one-shot text prediction, the paper is interpreted within its documented design envelope: transfer may depend on its sensing regime, preparation, workload, population, or benchmark. The synthesis protects the study's stated scope while auditing its assumptions comparatively.

Across the focal studies, programmer behavior should be interpreted jointly with error localization. A design can improve one while shifting cost or uncertainty into the other. The practical comparison is a matrix whose rows are operating conditions and whose columns are evidence quality, computation or process cost, robustness, and consequence of failure. This representation helps prevent an attractive mechanism from being detached from the circumstances that support it. It additionally indicates which ablations are informative: component removal is useful only when it tests an explicit role.

Execution feedback joins methodological claims to their use. At the least, assessment should distinguish nominal behavior from stress response and show dispersion alongside averages, and test plausible alternative explanations. Where distribution shift is central, calibration or sensitivity is as important as ranking. Where costs or hazards are asymmetric, utility-weighted assessment is more revealing than undifferentiated accuracy. These decisions need not homogenize studies; they make the reasons for their differences auditable.

4. Comparative Evaluation

For triangulation, the literature includes Approach to improving the quality of program code generation by large language models [3]; Code generation with large language models: a survey from neural program synthesis to autonomous softwar... [4]; Large Language Model-Based Interactive Code Generation for Developing a 3D Eye Movement Schematic [5]. These publications were retrieved and verified through DOI or publisher metadata, and their titles directly intersect the problem boundary of AI-assisted programming. Together they illuminate programmer behavior: some emphasize representations or mechanisms, whereas others foreground validation and application conditions. The useful comparison is not a leaderboard across incompatible settings, but a structured reading of what each study controls, leaves variable, and treats as a meaningful outcome. Divergence can then reveal a change in scale, data process, endpoint, or operating constraint.

The design space is broadened by Automating code generation for a new ecosystem: establishing baselines with large language model based c... [6]; A Model For Automated Code Debugging Using Small Language Models [7]; Large Language Model-Based Method for HVAC System Control Code Automatic Generation [8]. These publications were retrieved and verified through DOI or publisher metadata, and their titles directly intersect the problem boundary of AI-assisted programming. Together they illuminate error localization: some emphasize representations or mechanisms, whereas others foreground validation and application conditions. The useful comparison is not a leaderboard across incompatible settings, but a structured reading of what each study controls, leaves variable, and treats as a meaningful outcome. Divergence can then reveal a change in scale, data process, endpoint, or operating constraint.

A first comparison set connects Explainable automated debugging via large language model-driven scientific debugging [9]; Usage of Large Language Model for Code Generation Tasks: A Review [10]; A Method for Alleviating Illusions in Code Generation Based on a Large Language Model Generated by Retri... [11]. These publications were retrieved and verified through DOI or publisher metadata, and their titles directly intersect the problem boundary of AI-assisted programming. Together they illuminate execution feedback: some emphasize representations or mechanisms, whereas others foreground validation and application conditions. The useful comparison is not a leaderboard across incompatible settings, but a structured reading of what each study controls, leaves variable, and treats as a meaningful outcome. Divergence can then reveal a change in scale, data process, endpoint, or operating constraint.

The neighboring evidence base brings together Evolving code with a large language model [12]; PromptTone: A Dataset for Evaluating Large Language Model Code Generation Under Varying Prompt Politenes... [13]; TransferFuzz-Pro: Large Language Model Driven Code Debugging Technology for Verifying Propagated Vulnera... [14]. These publications were retrieved and verified through DOI or publisher metadata, and their titles directly intersect the problem boundary of AI-assisted programming. Together they illuminate repair hierarchy: some emphasize representations or mechanisms, whereas others foreground validation and application conditions. The useful comparison is not a leaderboard across incompatible settings, but a structured reading of what each study controls, leaves variable, and treats as a meaningful outcome. Divergence can then reveal a change in scale, data process, endpoint, or operating constraint.

A complementary strand is visible in A Systematic Review about Large Language Models (LLMs) applied to Code Generation [15]. These publications were retrieved and verified through DOI or publisher metadata, and their titles directly intersect the problem boundary of AI-assisted programming. Together they illuminate evaluation leakage: some emphasize representations or mechanisms, whereas others foreground validation and application conditions. The useful comparison is not a leaderboard

across incompatible settings, but a structured reading of what each study controls, leaves variable, and treats as a meaningful outcome. Divergence can then reveal a change in scale, data process, endpoint, or operating constraint.

5. Deployment and Governance

Implementation can follow a staged workflow. First, define the decision and its failure cost. Second, specify the evidence and operating envelope. Third, choose a method whose inductive assumptions match that evidence. Fourth, test repair hierarchy and evaluation leakage under controlled perturbations. Finally, retain provenance so that an unexpected outcome can be traced to data, configuration, material batch, environmental condition, or user interaction. This sequence keeps treating code generation as an iterative reasoning and execution process rather than one-shot text prediction connected to observable requirements.

The systems-level lesson is that responsible progress in AI-assisted programming depends on how components exchange evidence. Interfaces determine how uncertainty moves between measurement and inference, between algorithm and operator, or between microscopic design and system behavior. Teams spanning disciplines should predefine acceptance rules and triggers for revising conclusions. When those agreements are explicit, efficiency goals remain subordinate to explicit validity, safety, and interpretation constraints.

6. Limitations and Future Directions

The analysis cannot eliminate variation in definitions, study architecture, and disclosure granularity. Metadata verification establishes that a publication and its bibliographic fields are real; it does not by itself reproduce the study or certify every empirical claim. Publication status can change after metadata collection. No confirmed focal Scholar citation was normalized away, and anomalies were logged independently.

Subsequent studies need to preregister comparison protocols where feasible, publish machine-readable settings and test transfer under a substantively important shift in deployment constraints. Research should also document why failures occur in addition to how often. For AI-assisted programming, a valuable shared benchmark would combine ordinary performance, operational burden, uncertainty calibration, and resilience. Such a benchmark would reward designs that remain useful when evidence is incomplete or conditions change.

7. Conclusion

The evidence concerning AI-assisted programming is best understood as a connected evidence system rather than a collection of isolated scores. The focal studies show how treating code generation as an iterative reasoning and execution process rather than one-shot text prediction; the additional literature reveals the assumptions required to interpret those contributions. Across programmer behavior, error localization, execution feedback, repair hierarchy, and evaluation leakage, alignment remains the central requirement: the representation or mechanism must match the problem, metrics should fit the choice and real-world claims the evidence boundary. This alignment supports replication, bounded transfer, and interpretable failure.

References

1. Shi, Y., Zhang, H., Wan, C., & Gu, X. (2025). Between lines of code: Unraveling the distinct patterns of machine and human programmers. In 2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE) (pp. 1628-1639). IEEE.

2. Shi, Y., Wang, S., Wan, C., Wang, M., & Gu, X. (2024). From code to correctness: Closing the last mile of code generation with hierarchical debugging. arXiv preprint arXiv:2410.01215.

3. Timofeev, A. N., & Mikhaylova, S. S. (2024). Approach to improving the quality of program code generation by large language models. Neurocomputers. https://doi.org/10.18127/j19998554-202404-02

4. Gülmez, B. (2026). Code generation with large language models: a survey from neural program synthesis to autonomous software development. Applied Intelligence, 56(6). https://doi.org/10.1007/s10489-026-07230-0

5. Hamasaki, I., Kunimi, K., Shibata, K., & Toshiaki, G. (2026). Large Language Model-Based Interactive Code Generation for Developing a 3D Eye Movement Schematic. Cureus. https://doi.org/10.7759/cureus.107791

6. Aytekin, M. C., Yılmaz, F. G., & Demirezen, M. U. (2026). Automating code generation for a new ecosystem: establishing baselines with large language model based code generation for ArkTS and HarmonyOS. Automated Software Engineering, 33(2). https://doi.org/10.1007/s10515-026-00599-9

7. Kevin Gwindingwi,, & Monica Gondo, (2025). A Model For Automated Code Debugging Using Small Language Models. Journal of Scientific Research and Technology, 86-92. https://doi.org/10.61808/jsrt230

8. Jiang, R., Xia, K., Huang, J., & Lu, J. (2026). Large Language Model-Based Method for HVAC System Control Code Automatic Generation. Buildings, 16(9), 1722. https://doi.org/10.3390/buildings16091722

9. Kang, S., Chen, B., Yoo, S., & Lou, J. G. (2024). Explainable automated debugging via large language model-driven scientific debugging. Empirical Software Engineering, 30(2). https://doi.org/10.1007/s10664-024-10594-x

10. Bistarelli, S., Fiore, M., Mercanti, I., & Mongiello, M. (2025). Usage of Large Language Model for Code Generation Tasks: A Review. SN Computer Science, 6(6). https://doi.org/10.1007/s42979-025-04241-5

11. Wei, K. (2026). A Method for Alleviating Illusions in Code Generation Based on a Large Language Model Generated by Retrieval Enhancement. Advanced Electromagnetics, 15(3), 8519-8525. https://doi.org/10.7716/aem.v15i3.3976

12. Hemberg, E., Moskal, S., & O'Reilly, U. M. (2024). Evolving code with a large language model. Genetic Programming and Evolvable Machines, 25(2). https://doi.org/10.1007/s10710-024-09494-2

13. Andruccioli, M., Delnevo, G., Mirri, S., & Salomoni, P. (2026). PromptTone: A Dataset for Evaluating Large Language Model Code Generation Under Varying Prompt Politeness Levels. Data, 11(4), 88. https://doi.org/10.3390/data11040088

14. Li, S., Xie, K., Li, Y., Li, H., Ren, Y., Sun, L., et al. (2025). TransferFuzz-Pro: Large Language Model Driven Code Debugging Technology for Verifying Propagated Vulnerability. IEEE Transactions on Software Engineering, 51(8), 2396-2411. https://doi.org/10.1109/tse.2025.3584774

15. Alecsandro Bacin, F., Adriano de Mello, B., Dondoni Salton, G., & Da Silva Feitosa, S. (2025). A Systematic Review about Large Language Models (LLMs) applied to Code Generation. Revista Brasileira de Computação Aplicada, 17(3), 1-13. https://doi.org/10.5335/rbca.v17i3.16310