Evidence Alignment and Transfer Boundaries in Ai-Assisted Programming

Abstract

Progress in AI-assisted programming depends on more than accumulating favorable results. This critical synthesis connects comparative analysis of observable coding patterns produced by people and machines with hierarchical debugging that closes the gap between generated code and executable correctness and asks how measurement choices, boundary conditions, and decision costs shape the interpretation of both. A structured reading of two target studies and 12 verified companion references is conducted across five lenses: programmer behavior, error localization, execution feedback, repair hierarchy, evaluation leakage. Emphasis is placed on the provenance of evidence, the comparability of baselines, and the consequences of alternative explanations. The combined literature indicates that methodological gains become actionable only when programmer behavior and error localization are evaluated together and when limits associated with evaluation leakage are explicit. This shifts the emphasis from isolated scores toward traceable chains of evidence and decision relevance. The article concludes with a research agenda built around transparent comparators, targeted stress tests, and evidence records that can be reused without overstating causal or practical reach.

Keywords

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

1. Introduction

Research on AI-assisted programming connects scientific ambition and practical constraint. Researchers pursue not only stronger nominal performance but also explanations for when and why a method works. Its practical form appears in comparing comparative analysis of observable coding patterns produced by people and machines with hierarchical debugging that closes the gap between generated code and executable correctness through evidence alignment and transfer boundaries. An advantage observed in a bounded material, dataset, population, scale, or operating trace may be fragile outside its original boundary. The review process consequently has to preserve the unit of analysis and the decision context. The analysis also distinguishes a mechanism from a correlation, a benchmark advantage from a deployment advantage, and an interpretable diagnostic from a causal explanation.

This review uses five analytical lenses: programmer behavior, error localization, execution feedback, repair hierarchy, evaluation leakage. Their combination matters because they jointly cover method construction, evidence collection, and the conditions needed for generalization. The central organizing idea is representation, objective, and evaluation protocol. Under that principle, method novelty must be tested against operating limits; the comparison also tests whether comparisons, uncertainty, and operating limits have been specified. The work reported here is a critical review and contains no newly asserted sample, experiment, measurement, or benchmark outcome.

2. Methodological Foundations

The evidence pathway can be divided into the input evidence, the transformation applied to it, the output used for evaluation, and the decision that follows. In AI-assisted programming, research often disconnects these components 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. For that reason, ablation evidence should accompany aggregate performance. An auditable study identifies which factors remain invariant and which may change, then selects metrics that correspond to the intended use rather than to convenience alone.

A further foundation is explicit scope. Programmer behavior defines the local design problem, while evaluation leakage determines whether the design can survive outside that boundary. The link between them depends on error localization and execution feedback connect those ends by exposing trade-offs that a single score conceals. Here, adverse outcomes and sensitivity evidence maps the conditions under which the explanation remains viable. 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 comparing comparative analysis of observable coding patterns produced by people and machines with hierarchical debugging that closes the gap between generated code and executable correctness through evidence alignment and transfer boundaries, the paper is treated as a bounded design case: transfer may depend on its sensing regime, preparation, workload, population, or benchmark. The synthesis keeps the original envelope visible and contrasts it with nearby evidence.

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 comparing comparative analysis of observable coding patterns produced by people and machines with hierarchical debugging that closes the gap between generated code and executable correctness through evidence alignment and transfer boundaries, the paper is treated as a bounded design case: transfer may depend on its sensing regime, preparation, workload, population, or benchmark. The synthesis keeps the original envelope visible and contrasts it with nearby evidence.

Across the focal studies, programmer behavior should be interpreted jointly with error localization. A design can improve one while hiding a penalty in the other dimension. The analysis can be summarized in a compact matrix whose rows are operating conditions and whose columns are evidence quality, computation or process cost, robustness, and consequence of failure. That arrangement prevents an attractive mechanism from being detached from the circumstances that support it. It also clarifies which ablations are informative: each ablation should answer why a component matters.

Execution feedback joins methodological claims to their use. A minimal evaluation must 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 choices preserve study distinctions; they make the reasons for their

differences auditable.

4. Comparative Evaluation

A complementary strand is visible in 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.

For triangulation, the literature includes 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.

The design space is broadened by 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.

A first comparison set connects 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.

5. Deployment and Governance

A practical workflow has five stages. 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. The staged design keeps comparing comparative analysis of observable coding patterns produced by people and machines with hierarchical debugging that closes the gap between generated code and executable correctness through evidence alignment and transfer boundaries connected to observable requirements.

More generally, responsible progress in AI-assisted programming depends on interfaces as much as components. Interfaces determine how uncertainty moves between measurement and inference, between algorithm and operator, or between microscopic design and system behavior. A shared protocol should state acceptance conditions and what would overturn a claim. When those agreements are explicit, optimization can pursue efficiency without silently relaxing validity, safety, or interpretability.

6. Limitations and Future Directions

The analysis cannot eliminate inconsistent terminology, research designs, and reporting detail. Publication-level checks establish traceability, whereas claim-level replication remains a separate task. Rapid publication cycles can also alter venues and versions. Scholar-confirmed focal citations were protected and metadata conflicts were surfaced rather than hidden.

Priority should be given to studies that preregister comparison protocols where feasible, publish machine-readable settings and test transfer under a substantively important shift in deployment constraints. Research should also connect failure frequency to an explanatory taxonomy. For AI-assisted programming, a valuable shared benchmark would combine standard-condition utility, efficiency, confidence quality, and fault recovery. Such a benchmark would reward designs that remain useful when evidence is incomplete or conditions change.

7. Conclusion

Scholarship in AI-assisted programming is better interpreted through the relationships among studies rather than a collection of isolated scores. The focal studies show how comparing comparative analysis of observable coding patterns produced by people and machines with hierarchical debugging that closes the gap between generated code and executable correctness through evidence alignment and transfer boundaries; the additional literature reveals the assumptions required to interpret those contributions. Across programmer behavior, error localization, execution feedback, repair hierarchy, and evaluation leakage, credibility ultimately depends on alignment: the representation or mechanism must match the problem, the evaluation must match the decision, and the deployment claim must match the tested boundary. Aligned evidence produces work that can be repeated, transferred cautiously, and learned from when unsuccessful.

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