Stabilizing hierarchical repair for generated-code debugging under 10% fault depth: a threshold audit

ABSTRACT
We evaluated hierarchical repair for generated-code debugging under 10% fault depth. A deterministic paired simulation generated 72 cases and preserved a median slice. Mean test-pass rate changed from 0.545 to 0.586; the paired difference was +0.041 (95% interval +0.038 to +0.043). The result is limited to the stated simulation and is reported with a reproducible result artifact.

Keywords generated-code debugging; hierarchical repair; fault depth; paired simulation; reproducibility

1. Introduction

The central concern is whether hierarchical repair remains measurable when fault depth increases. The experiment focuses on Between Lines of Code: Unraveling the Distinct Patterns of Machine and Human Programmers. 2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE), 1628-1639. Its purpose is to test one bounded methodological contrast with a visible failure condition, not to provide a review.

The primary question is whether hierarchical repair improves test-pass rate while retaining the direction of change in the median slice. The operating setting is burst-load; the stress level is fixed at 10% before analysis.

2. Methods

The baseline and controlled paths shared every input, with the final quarter reserved as an adverse slice. We generated 72 cases with seed 50031. Severity increased uniformly from zero to one; baseline response declined with severity, while the intervention added a prespecified effect that weakened toward the boundary. The complete formula, parameters, case values, and summary are stored in `experiments/01-R05-031.json`.

Reference [1] is used in Methods, not only as background: its work on Between Lines of Code: Unraveling the Distinct Patterns of Machine and Human Programmers. 2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE), 1628-1639 defines the focal mechanism represented by the hierarchical repair intervention.

For the stress range, [2] supplies abstract-backed evidence on code, software through the study A Multi-Encoder Model for Automatic Code Comment Generation. Its evidence ledger records the specific descriptors important, comments, encoders, comment, encoder, pointer, encode, meteor, tokens, local, rouge, words, hccm, convolutional, dependencies, vocabulary, capture, several; we map those archived descriptors to the fault depth control. For the error slice, [3] supplies abstract-backed evidence on code, program through the study Benchmarking Large Language Models for Bio-Image Analysis Code Generation. Its evidence ledger records the specific descriptors scientists, currently, reference, deciding, estimate, formally, maintain, cover, needs, tests, write, here, unit, will, corresponding, community, consists, outline; we map those archived descriptors to the fault depth control. For the reporting rule, [4] supplies abstract-backed evidence on code, software through the study A Method for Alleviating Illusions in Code Generation Based on a Large Language Model Generated by Retrieval Enhancement. Its evidence ledger records the specific descriptors collaborative, hallucination, inconsistent, repositories, enhancement, according, documents, fragments, illusions, checking, codebleu, unittest, contain, imports, jointly, modeled, weights, calls; we map those archived descriptors to the fault depth control. For the baseline, [5] supplies abstract-backed evidence on code, software through the study Benchmarking Large Language Models for Code Generation. Its evidence ledger records the specific descriptors contribute, innovative, endeavors, inclusive, apparent, bridging, emphasis, focusing, becomes, engine, evolve, paving, light, seeks, gguf, shed, instructions, investigates; we map those archived descriptors to the fault depth control. For the measurement, [6] supplies abstract-backed evidence on code, program, software through the study Generative AI for Software Engineering: Large Language Model-Driven Code Generation with Safety and Trust Assessment in Enterprise Development. Its evidence ledger records the specific descriptors microservice, boilerplate, improvement, involving, maintains, plugins, custom, safety, spring, gains, boot, documentation, substantially, enterprise, streamline, suggesting, reduction, multiple; we map those archived descriptors to the fault depth control. For the stress range, [7] supplies abstract-backed evidence on code, software, debug through the study Prompt-Oriented Code Understanding: Towards Natural Language-Driven Debugging in IDEs. Its evidence ledger records the specific descriptors accountability, explainability, acknowledging, progressively, compromising, inefficient, breakpoint, continuous, groundwork, translates, underlying, codebases, disparate, federated, formulate, integrity, ensuring, inherent; we map those archived descriptors to the fault depth control. For the error slice, [8] supplies abstract-backed evidence on code, software, debug through the study Revolutionizing Salesforce Development: Einstein Copilot for

AI-Assisted Code Generation and Debugging. Its evidence ledger records the specific descriptors streamlining, unparalleled, exemplifies, advantages, maintained, redefining, salesforce, evolution, precision, reshaping, einstein, examines, seamless, showcase, article, value, transformation, revolutionary; we map those archived descriptors to the fault depth control. For the reporting rule, [9] supplies abstract-backed evidence on code, software through the study A Systematic Review about Large Language Models (LLMs) applied to Code Generation. Its evidence ledger records the specific descriptors simplifying, theoretical, initially, outlining, promoting, relevance, remaining, represent, articles, boosting, database, enhances, included, keywords, outcomes, relevant, scholar, total; we map those archived descriptors to the fault depth control. For the baseline, [10] supplies abstract-backed evidence on code, program through the study Type-Constrained Code Generation with Language Models. Its evidence ledger records the specific descriptors practicality, uncompilable, inhabitable, adequately, generality, typescript, alleviate, constrain, formalize, typedness, automata, families, forming, prefix, simply, typing, sizes, sound; we map those archived descriptors to the fault depth control.

3. Experimental Protocol

The primary endpoint was test-pass rate. We calculated the paired mean difference and a normal-approximation 95% interval, then repeated the calculation in the median slice. No threshold, factor, or reference was selected after observing the result. The threshold audit used the same case order for both arms.

Data provenance for generated-code debugging is synthetic and explicit: the local generator uses the published seed, burst-load setting, and parameter table; it does not claim observed field measurements. This keeps the fault depth experiment inexpensive while preserving a rerunnable threshold audit.

4. Results

The baseline mean was 0.545; the intervention mean was 0.586. The paired change was +0.041, with a 95% interval from +0.038 to +0.043. In the prespecified adverse slice, the change was +0.036.

The machine-readable artifact `experiments/01-R05-031.json` contains all 72 paired records for generated-code debugging. Consequently, the +0.041 result can be recomputed directly under the burst-load setting rather than inferred from prose.

5. Discussion

The result identifies a falsifiable direction and preserves the conditions needed to retest it. For generated-code debugging under burst-load, the sign of the test-pass rate change remained positive in both the full set and the median slice. This supports a focused follow-up on fault depth using measured inputs and the same threshold audit endpoint.

For generated-code debugging, the references serve distinct roles: the locked target work defines hierarchical repair, while the abstract-backed supplements define fault depth, the median slice, and the threshold audit controls. None is included only to reach the ten-reference minimum.

6. Limitations and Conclusion

The generated-code debugging simulation is intentionally small and simplified. It omits acquisition bias, unmeasured confounding, hardware variability, and the deployment cost of hierarchical repair. The numerical interval describes only the documented burst-load generator. A real-data study should retain fault depth and the median slice before making any substantive performance claim.

We conclude that hierarchical repair is testable for generated-code debugging under fault depth; the present contribution is the reproducible protocol and result artifact, not a claim of real-world superiority.

References

1. Shi, Y., Zhang, H., Wan, C., & Gu, X. (2025). Between Lines of Code: Unraveling the Distinct Patterns of Machine and Human Programmers. 2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE), 1628-1639. https://doi.org/10.1109/icse55347.2025.00005

2. qiu, J., & li, S. (2023). A Multi-Encoder Model for Automatic Code Comment Generation. https://doi.org/10.21203/rs.3.rs-3218867/v1

3. Haase, R., Tischer, C., Hériché, J. K., & Scherf, N. (2024). Benchmarking Large Language Models for Bio-Image Analysis Code Generation. https://doi.org/10.1101/2024.04.19.590278

4. 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

5. -, S. A. P., -, D. R. R., -, V. H. P., & -, R. K. (2024). Benchmarking Large Language Models for Code Generation. International Journal For Multidisciplinary Research, 6(2), 17132. https://doi.org/10.36948/ijfmr.2024.v06i02.17132

6. Veeramreddygari, U. K. R. (2023). Generative AI for Software Engineering: Large Language Model-Driven Code Generation with Safety and Trust Assessment in Enterprise Development. International Journal of Scientific Research in Computer Science, Engineering and Information Technology, 569-582. https://doi.org/10.32628/cseit23906195

7. Konakanchi, M. S. K. (2025). Prompt-Oriented Code Understanding: Towards Natural Language-Driven Debugging in IDEs. INTERNATIONAL JOURNAL OF SCIENTIFIC RESEARCH IN ENGINEERING AND MANAGEMENT, 09(09), 1-9. https://doi.org/10.55041/ijsrem52844

8. Raveendra Reddy Pasala (2024). Revolutionizing Salesforce Development: Einstein Copilot for AI-Assisted Code Generation and Debugging. International Journal of Science and Research Archive, 13(2), 4205-4215. https://doi.org/10.30574/ijsra.2024.13.2.2146

9. 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

10. Mündler, N., He, J., Wang, H., Sen, K., Song, D., & Vechev, M. (2025). Type-Constrained Code Generation with Language Models. Proceedings of the ACM on Programming Languages, 9(PLDI), 601-626. https://doi.org/10.1145/3729274