HEJ-Robust: A Robustness Benchmark for LLM-based Automated Program Repair

Anonymous Author(s)

Abstract

Recent Large Language Models (LLMs) have shown strong performance on automated program repair across standard benchmarks. However, these benchmarks evaluate models on a single canonical form of buggy code and do not reflect the syntactic variations commonly observed in real-world software, leaving robustness largely unexamined. In this work, we construct HEJ-Robust, a robustness benchmark built from HumanEval-Java-Bug using eight semantic-preserving code transformations, resulting in 1,450 transformed instances. We evaluate five fine-tuned LLMs on this benchmark and show that model performance drops by over 50% under several transformations, indicating that current LLM-based repair models lack robustness to minor syntactic variations.

Keywords

Large Language Models, Automated Program Repair, Benchmark, Robustness Testing

ACM Reference Format:

Anonymous Author(s). 2026. HEJ-Robust: A Robustness Benchmark for LLM-based Automated Program Repair. In . ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Automated program repair (APR) aims to automatically generate patches that fix buggy programs. Early APR approaches primarily followed the generate-and-validate paradigm, where candidate patches are synthesized using predefined or learned repair operators and validated against test suites. Representative systems include GenProg [12], PAR [11], and systematic mutation-based repair techniques [17]. While these approaches have demonstrated effectiveness on specific bug classes, they often suffer from scalability limitations and test-suite overfitting.

Recent advances in deep learning have significantly reshaped APR research by formulating program repair as a code translation problem, where buggy code is translated into its fixed version [9, 29]. Pre-trained LLMs, such as PLBART [1], and CodeT5 [24], have shown strong repair capability when fine-tuned on bug-fix data [4, 8, 27]. More recent studies further explore instruction-tuned and agent-based LLMs for automated program repair [2, 7, 28]. To evaluate these approaches, existing benchmarks commonly rely on Defects4J [10] or HumanEval-Java-Bug [8], which assume a fixed syntactic representation of buggy programs.

Existing APR benchmarks evaluate repair accuracy on a single canonical buggy program, ignoring syntactic diversity among semantically equivalent code. Prior studies show neural code models are sensitive to semantics-preserving transformations [18, 23]. While robustness testing via transformations, fuzzing, and adversarial examples has been studied in other SE tasks [16, 26], robustness evaluation for LLM-based APR on function-level, human-crafted benchmarks remains largely unexplored. A related effort, Defects4J-TRANS [13], applies transformations to project-level real bugs in Defects4J; our benchmark is complementary, focusing on function-level, human-crafted bugs from HumanEval-Java-Bug.

We address this gap by introducing a transformation-based robustness benchmark for automated program repair. Constructed by applying eight semantic-preserving transformations to HumanEval-Java-Bug [8], our benchmark enables controlled evaluation of repair consistency. We use it to assess the robustness of five fine-tuned LLM repair models against code perturbations.

The contributions of this paper are as follows:

(1) We introduce a transformation-based robustness benchmark built on HumanEval-Java-Bug, covering eight semantic-preserving transformations and providing function-level, test-executable robustness evaluation for APR.
(2) We provide a systematic evaluation of LLM-based repair models under semantics-preserving transformations.
(3) We release the benchmark to facilitate future research on robust and reliable automated program repair.

Our Code, dataset, and Artifacts are publicly available [1]

2 Related Work

Automated program repair (APR) has been extensively studied over the past two decades. Early work primarily follows the generate-and-validate paradigm, where candidate patches are generated and validated against test suites [12, 17]. While effective on curated benchmarks such as Defects4J [10], these approaches suffer from overfitting and scalability issues [25].

More recently, deep learning-based APR approaches reformulate bug fixing as a neural machine translation problem, translating buggy code into fixed code [9, 14, 22, 29]. Pre-trained LLMs further improve repair performance by leveraging large-scale code corpora before fine-tuning on repair data [1, 4, 5, 24, 27]. Most of these approaches evaluate on bug-fix pairs (BFPs) [5, 22], which largely consist of abstract or canonicalized code. More recent benchmarks derived from HumanEval [6] enable functional validation using test cases [8]. Complementary studies explore LLM-based repair in competitive programming and agent-based settings [2, 7, 28].

Parallel to APR research, robustness testing of neural models for code has gained attention. Prior work demonstrates that neural code

[1] https://github.com/anonprox/hej-robust

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Conference'17, July 2017, Washington, DC, USA

© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Table 1: Fine-tuned models against different semantic-preserving code transformations on HumanEval-Java-Bug.

(a) Local Variable Renaming (100 bugs)

	Pass@10			CodeBLEU	
	orig.	trans.	change	orig.	trans.
plbart_base	14.53	6.54	54.99↓	82.11	81.91
plbart_large	21.88	9.91	54.71↓	82.75	82.15
codet5_small	19.35	8.26	57.31↓	82.17	81.43
codet5_base	24.81	12.28	50.5↓	82.02	81.58
codet5_large	23.66	11.5	51.39↓	80.74	80.92

(b) Method Renaming (149 bugs)

	Pass@10			CodeBLEU	
	orig.	trans.	change	orig.	trans.
plbart_base	19.46	18.13	6.83↓	82.96	82.78
plbart_large	23.98	22.8	4.92↓	83.34	83.04
codet5_small	21.16	19.46	8.03↓	83.1	82.96
codet5_base	25.87	24.37	5.8↓	83.06	82.71
codet5_large	24.75	23.98	3.11↓	81.73	81.69

(c) Parameter Renaming (162 bugs)

	Pass@10			CodeBLEU	
	orig.	trans.	change	orig.	trans.
plbart_base	17.86	18.69	4.65↑	83.63	83.84
plbart_large	22.6	23.33	3.23↑	84.11	83.86
codet5_small	20.3	19.1	5.91↓	83.92	84.02
codet5_base	24.77	24.41	1.45↓	83.86	83.94
codet5_large	24.41	23.7	2.91↓	82.67	82.55

(d) Boolean Exchange (7 bugs)

	Pass@10			CodeBLEU	
	orig.	trans.	change	orig.	trans.
plbart_base	12.5	22.22	77.76↑	86.03	85.8
plbart_large	22.22	30.0	35.01↑	86.14	85.69
codet5_small	22.22	22.22	0%	83.68	83.3
codet5_base	22.22	12.5	43.74↓	86.02	85.47
codet5_large	12.5	12.5	0%	85.26	84.67

(e) Loop Exchange (142 bugs)

	Pass@10			CodeBLEU	
	orig.	trans.	change	orig.	trans.
plbart_base	19.32	18.39	4.81↓	84.66	84.76
plbart_large	25.26	23.66	6.33↓	84.77	85.5
codet5_small	21.55	17.92	16.84↓	84.08	84.61
codet5_base	26.04	23.66	9.14↓	84.58	85.12
codet5_large	28.28	26.8	5.23↓	83.29	84.36

(f) Reorder Condition (603 bugs)

	Pass@10			CodeBLEU	
	orig.	trans.	change	orig.	trans.
plbart_base	16.88	15.69	7.05↓	83.83	85.64
plbart_large	21.41	18.48	13.69↓	84.17	86.1
codet5_small	19.7	17.92	9.04↓	84.01	85.87
codet5_base	23.25	20.99	9.72↓	83.94	85.75
codet5_large	23.45	21.62	7.8↓	82.56	85.04

(g) Insert Log Statement (173 bugs)

	Pass@10			CodeBLEU	
	orig.	trans.	change	orig.	trans.
plbart_base	17.22	16.43	4.59↓	83.64	83.6
plbart_large	22.07	22.42	1.59↑	84.1	83.85
codet5_small	19.53	18.4	5.79↓	83.86	83.69
codet5_base	24.45	22.07	9.73↓	83.8	83.57
codet5_large	24.78	24.45	1.33↓	82.61	82.91

(h) Insert Try catch (114 bugs)

	Pass@10			CodeBLEU	
	orig.	trans.	change	orig.	trans.
plbart_base	16.91	13.74	18.75↓	83.86	83.65
plbart_large	21.53	19.29	10.4↓	84.17	83.94
codet5_small	19.29	11.02	42.87↓	84.52	84.49
codet5_base	25.17	18.12	28.01↓	84.29	84.3
codet5_large	26.14	18.12	30.68↓	83.16	83.52

models are vulnerable to small, semantics-preserving transformations [19]. Transformation-based testing, fuzzing, and adversarial example generation have been applied to code models [16, 19], with later work emphasizing natural and context-aware transformations [26]. ReCode [23] evaluates robustness of code generation models under semantic-preserving perturbations. More recently, Defects4J-TRANS [13] evaluates the generalizability of LLMs in APR by applying transformations to project-level bugs in Defects4J. Defects4J-TRANS applies five transformations: variable renaming, loop transformation, switch transformation, dead code injection, and boolean transformation. Our benchmark shares three of these (variable renaming, loop exchange, and boolean exchange), and adds five transformations not present in Defects4J-TRANS: method renaming, parameter renaming, condition reordering, log statement insertion, and try–catch insertion. Switch transformation was not applicable to HumanEval-Java-Bug as the dataset contains no switch statements. The two benchmarks are therefore complementary: Defects4J-TRANS targets project-level real bugs, while HEJ-Robust targets function-level, human-crafted bugs. While robustness has been studied for tasks such as code summarization and

code representation learning, function-level robustness evaluation for automated program repair on human-crafted benchmarks remains largely unexplored. In particular, existing APR benchmarks do not systematically evaluate the robustness of repair models under semantics-preserving code transformations.

In this work, we bridge this gap by focusing on robustness benchmarking for LLM-based program repair rather than proposing a new repair technique.

3 Robustness Benchmark Design

3.1 Base Dataset

We adopt the HumanEval-Java-Bug dataset introduced by Jiang et al. [8], which is derived from HumanEval [6]. The dataset contains 164 Java programs with manually injected bugs and annotated buggy-line locations. Each instance is accompanied by executable test cases and human-written patches. We select this dataset because it is manually curated, recent, and less likely to suffer from data leakage issues common in earlier APR benchmarks.

3.2 Semantic-Preserving Code Transformations

We apply eight semantic-preserving code transformations that reflect common syntactic variations observed in real-world software. Prior work has shown that such transformations are effective for evaluating the robustness of neural code models. Our goal is not to improve code quality, but to assess robustness under benign syntactic changes.

The eight transformations are: (1) local variable renaming, (2) method renaming, (3) parameter renaming, (4) log statement insertion, (5) try–catch insertion, (6) boolean exchange, (7) loop exchange, and (8) condition reordering. All transformations are semantics-preserving by construction: renaming transforms replace all occurrences of an identifier consistently using tree-sitter parsing, and structural transforms (loop exchange, condition reordering, boolean exchange) apply well-known program equivalences. The tools used have been validated for semantics preservation in prior robustness research [20, 26].

Renaming transformations (1–3). For identifier renaming, we adopt the naturalness-aware substitution strategy proposed by Yang et al. [26]. Unlike prior approaches that use random strings or fixed patterns [16, 19], this method generates context-aware and developer-natural identifiers, ensuring that performance degradation reflects robustness issues rather than unnatural code artifacts.

We use masked language prediction with CodeBERT and GraphCodeBERT to generate candidate identifiers and select substitutions based on cosine similarity in embedding space. Java code is parsed using tree-sitter [3] to ensure consistent replacement across all occurrences. To control transformation strength, only one identifier is renamed per program.

Structural and syntactic transformations (4–8). The remaining transformations are implemented using JavaTransformer [20], which applies AST-based modifications via JavaParser. Logging statements are inserted at method entry points, and try–catch blocks are added at syntactically valid locations. Boolean exchange inverts boolean initializations while preserving semantics. Loop exchange converts `for` loops to equivalent `while` loops and vice versa. Condition reordering swaps operands in equality and inequality expressions. Transformations are applied only when syntactically valid.

This yields 1,450 transformed instances in total.

Threats to validity. The eight transformations are drawn from two established tools — Yang et al. [26] for renaming and JavaTransformer [20] for structural changes — both of which have been validated in prior robustness studies. These transformations correspond to common coding practices such as refactoring, logging, and loop restructuring. However, they do not cover all possible syntactic variations, and their prevalence in real-world Java code has not been empirically measured. This limits the external validity of the benchmark. Additionally, not all transformations can be applied to every program; applicability depends on the structure of the code, which explains the varying instance counts per transformation.

3.3 Benchmark Construction and Task Formulation

After applying transformations, the locations of buggy lines may change. We manually re-annotate the buggy-line locations for all transformed programs by inspecting each transformed instance and mapping the original buggy statement to its updated position. Instances where a transformation directly modifies the buggy line itself are removed from the dataset to avoid ambiguity. To reduce annotation errors, a second author independently verified more than 10% of the re-annotated instances, and disagreements were resolved by discussion. Combined with the original human-written patches and test cases, this yields a fully executable benchmark suitable for robustness evaluation. Model outputs are evaluated using both code-similarity metrics, such as CodeBLEU [15, 21], and functional correctness via test-based metrics (e.g., pass@10) provided by HumanEval-Java-Bug [8].

4 Experimental Setup

To evaluate the proposed benchmark, we consider five LLMs: two PLBART variants (base and large) and three CodeT5 variants (small, base, and large), all fine-tuned and released by Jiang et al. [8]. We select these models because they are the only publicly available fine-tuned APR models evaluated on HumanEval-Java-Bug, which allows us to study robustness of APR-specific models under our transformations without introducing confounds from retraining. We directly evaluate these models without any modification to their original training or decoding configuration. Following the evaluation protocol of Jiang et al. [8], we generate 10 candidate patches per bug and evaluate using Pass@10, where a bug is considered fixed if at least one generated patch passes all developer-written test cases. We do not modify seed settings, and we use the same generation procedure as provided in the original released models. We use CodeBLEU as a similarity-based metric, while Pass@10 serves as the primary indicator of functional correctness. Results are compared to quantify robustness degradation under semantic-preserving transformations, as reported in Table 1. More results with pre-trained models are available in our shared repository.

5 Results

The evaluation results are summarized in Table 1, which reports the performance of five fine-tuned models across eight transformed

datasets. Each transformation is presented in a separate subtable, showing Pass@10 and CodeBLEU scores for both the original and transformed datasets. We also report the relative change from the original to the transformed dataset, indicated by ↑ for improvements and ↓ for degradations.

Across all eight transformations, we observe drops in Pass@10 for most models, with the largest degradation occurring under the Local Variable Renaming transformation, where performance decreases by 50.5% to 57.31%. This is likely because fine-tuned models rely heavily on identifier patterns learned during training; renaming local variables introduces distribution shifts in token sequences that disrupt the model's ability to identify the buggy location and generate a correct patch. Similar behavior has been reported in code generation robustness studies [23]. In contrast, transformations such as Parameter Renaming and Insert Log Statement cause smaller drops, suggesting that method-level context or appended logging code is less disruptive to the repair process. Structural transformations such as Loop Exchange cause moderate drops, as they change control-flow structure while preserving variable names. Notably, robustness does not correlate with model size: larger models often degrade more than their smaller counterparts (e.g., `CodeT5_large` vs. `CodeT5_base`, and `PLBART_large` vs. `PLBART_base`) across multiple transformations. Results for Boolean Exchange are unstable due to the small number of affected samples (7 instances).

In contrast, CodeBLEU scores remain largely stable across transformations, with only minor increases or decreases. This discrepancy suggests that CodeBLEU may not fully capture functional robustness, which we leave for future investigation.

6 Conclusion

We present **HEJ-Robust**, a robustness benchmark of 1,450 bug instances constructed from HumanEval-Java-Bug using eight semantic-preserving transformations. Evaluating five fine-tuned LLMs, we show that even minor syntactic variations cause consistent drops in Pass@10, revealing substantial robustness gaps in current repair models. Future work will study richer transformations, larger contexts, and improved robustness-aware training and evaluation metrics.

References

[1] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang. Unified pre-training for program understanding and generation. *arXiv preprint arXiv:2103.06333*, 2021.

[2] I. Bouzenia, P. Devanbu, and M. Pradel. Repairagent: An autonomous, llm-based agent for program repair. *arXiv preprint arXiv:2403.17134*, 2024.

[3] M. Brunsfeld and contributors. tree-sitter. https://github.com/tree-sitter/tree-sitter, 2024. Accessed: 2024-05-23.

[4] S. Chakraborty, T. Ahmed, Y. Ding, P. Devanbu, and B. Ray. Natgen: Generative pre-training by" naturalizing" source code. *arXiv preprint arXiv:2206.07585*, 2022.

[5] S. Chakraborty and B. Ray. On multi-modal learning of editing source code. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 443–455. IEEE, 2021.

[6] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[7] Z. Fan, X. Gao, M. Mirchev, A. Roychoudhury, and S. H. Tan. Automated repair of programs from large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1469–1481. IEEE, 2023.

[8] N. Jiang, K. Liu, T. Lutellier, and L. Tan. Impact of code language models on automated program repair. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1430–1442. IEEE, 2023.

[9] N. Jiang, T. Lutellier, and L. Tan. Cure: Code-aware neural machine translation for automatic program repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1161–1173. IEEE, 2021.

[10] R. Just, D. Jalali, and M. D. Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 437–440, 2014.

[11] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 802–811. IEEE, 2013.

[12] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38(1):54–72, 2012.

[13] F. Li, J. Jiang, J. Sun, and H. Zhang. Evaluating the generalizability of llms in automated program repair. In *2025 IEEE/ACM 47th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, pages 91–95. IEEE, 2025.

[14] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan. Coconut: combining context-aware neural translation models using ensemble for program repair. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*, pages 101–114, 2020.

[15] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318, 2002.

[16] M. V. Pour, Z. Li, L. Ma, and H. Hemmati. A search-based testing framework for deep neural networks of source code embedding. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 36–46. IEEE, 2021.

[17] Z. Qi, F. Long, S. Achour, and M. Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 24–36, 2015.

[18] F. Rabbi, Z. Ding, and J. Yang. A multi-language perspective on the robustness of llm code generation. 2025.

[19] M. R. I. Rabin, N. D. Bui, K. Wang, Y. Yu, L. Jiang, and M. A. Alipour. On the generalizability of neural program models with respect to semantic-preserving program transformations. *Information and Software Technology*, 135:106552, 2021.

[20] M. R. I. Rabin, K. Wang, and M. A. Alipour. Testing neural program analyzers. *arXiv preprint arXiv:1908.10711*, 2019.

[21] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*, 2020.

[22] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 28(4):1–29, 2019.

[23] S. Wang, Z. Li, H. Qian, C. Yang, Z. Wang, M. Shang, V. Kumar, S. Tan, B. Ray, P. Bhatia, et al. Recode: Robustness evaluation of code generation models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 13818–13843, 2023.

[24] Y. Wang, W. Wang, S. Joty, and S. C. Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*, 2021.

[25] B. Yang and J. Yang. Exploring the differences between plausible and correct patches at fine-grained level. In *2020 IEEE 2nd International Workshop on Intelligent Bug Fixing (IBF)*, pages 1–8. IEEE, 2020.

[26] Z. Yang, J. Shi, J. He, and D. Lo. Natural attack for pre-trained models of code. *arXiv preprint arXiv:2201.08698*, 2022.

[27] J. Zhang, S. Panthaplackel, P. Nie, J. J. Li, and M. Gligoric. Coditt5: Pretraining for source code and natural language editing. *arXiv preprint arXiv:2208.05446*, 2022.

[28] Q. Zhang, T. Zhang, J. Zhai, C. Fang, B. Yu, W. Sun, and Z. Chen. A critical review of large language model on software engineering: An example from chatgpt and automated program repair. *arXiv preprint arXiv:2310.08879*, 2023.

[29] Q. Zhu, Z. Sun, Y.-a. Xiao, W. Zhang, K. Yuan, Y. Xiong, and L. Zhang. A syntax-guided edit decoder for neural program repair. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 341–353, 2021.