



International Journal of Computer Science Engineering and Research

Journal homepage: <https://trpub.org/journal.php?id=EihcudPEIP>

TRPub

IJCSE

International Journal of
Computer Science
Engineering and
Research

Vol. 1 Issue 1 2026

Methodology Article

Open Access

A Reproducible Benchmarking Framework for Evaluating Code-Generating Language Models on Maintenance Tasks

Ananya Espinoza^{1*}, Pawel Chandrasekar², Camila Eriksson³

¹Department of Computer Science, Limpopo Graduate Institute of Computer Science, Barbados

²Department of Computer Science, Nejd Research Institute of Computer Science, Sri Lanka

³Department of Computer Science, Balkan Center for Computer Science, United Arab Emirates

Article Info

Volume 1, Issue 1, 2026

[doi: 10.62715/ijcser.2026.v1.i1.pp19-24](https://doi.org/10.62715/ijcser.2026.v1.i1.pp19-24)

Received: 30 March 2026

Accepted: 31 May 2026

Published: 17 July 2026

Abstract

Benchmarks for code-generating language models grew up around the task of writing a new function from a description. Most professional programming is something else: changing code that already exists, to fix a defect, to keep pace with a dependency or to improve structure without altering behaviour. Published scores on function-synthesis benchmarks say little about that work, and they are hard to reproduce because models change behind fixed names, sampling is random and test suites are incomplete or unstable. This article proposes a framework for benchmarking models on maintenance tasks in a way that others can repeat. It defines four task categories taken from the software maintenance literature, a procedure for deriving tasks from the history of open-source repositories, containerised execution with pinned dependencies, test-based oracles supplemented by checks for unintended change, controls against training-data contamination, a fixed protocol for invoking models, and rules for statistical reporting and artefact release. Each element is justified from prior work on software engineering benchmarks and on reproducibility in machine learning. The framework is a design. It has not been instantiated or used to evaluate any model here, and the article identifies the assumptions that a first implementation would need to test.

Keywords: Code generation, Large language models, Software maintenance, Benchmarking, Reproducibility, Program repair

© 2026 Ananya Espinoza et al. This is an open access article under the CC BY license (<https://creativecommons.org/licenses/by/4.0/>), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

1. Introduction

The capability of language models to write code is usually reported as a score on a benchmark, and the best known benchmarks ask for a self-contained function. HumanEval consists of 164 hand-written Python problems, each with a signature, a docstring and hidden unit tests (Chen et al., 2021), and MBPP follows the same pattern with crowd-sourced problems (Austin et al., 2021). They are cheap to run and easy to interpret. They also describe a kind of programming that occupies a small part of a developer's time.

The larger part is maintenance. Lientz and Swanson (1980) distinguished corrective, adaptive and

* Corresponding author: Ananya Espinoza, Department of Computer Science, Limpopo Graduate Institute of Computer Science, Barbados.
E-mail: ananya.espinoza@limpopo.edu

© 2026 Ananya Espinoza et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

perfective maintenance, and the international standard adds a preventive category (ISO/IEC/IEEE, 2022). In each, the input is an existing system and a reason to change it, and success means that the change does what was asked and nothing else. A model that writes a correct new function may still be poor at locating the right place in a large codebase, at respecting its conventions, or at leaving unrelated behaviour intact.

This article proposes a benchmarking framework aimed at maintenance and designed from the start for reproducibility. It is a proposal: the framework is specified, with reasons, but has not been built or applied here, and no model is ranked.

2. Background

The design responds to two strands of earlier work, one on benchmarks drawn from real repositories and one on what makes computational results repeatable.

2.1. Benchmarks built from real software

Software engineering research has long used curated collections of real faults. Defects4J packages bugs from open-source Java projects with the tests that expose them and a harness for checking out faulty and fixed versions (Just et al., 2014). CodeXGLUE gathered a range of code understanding and generation tasks, among them code repair and translation, and scored most of them by textual similarity to a reference (Lu et al., 2021). SWE-bench moved to whole repositories: each task pairs a reported issue with the codebase at that commit, and a submission passes if tests associated with the resolving pull request succeed while previously passing tests still do (Jimenez et al., 2024). The framework proposed here adopts that structure and extends it beyond issue resolution.

2.2. Threats to reproducibility

Four threats recur in the literature. Similarity metrics such as BLEU (Papineni et al., 2002) and its code-aware variant CodeBLEU (Ren et al., 2020) reward resemblance to one reference, although many different patches can be correct. Test suites can be too weak: when the tests of an existing benchmark were extended, solutions previously counted as correct were found to be wrong (Liu et al., 2023). Tests can also be flaky, passing or failing without any change to the code (Luo et al., 2014). And the models themselves are moving targets: hosted models are updated or withdrawn, outputs are sampled, and benchmark material may already be in the training data (Sallou et al., 2024). Experience elsewhere in computing suggests that without deliberate packaging, even published code often cannot be rebuilt by others (Collberg and Proebsting, 2016).

3. The Proposed Framework

The framework has seven elements. Table 1 lists them with the threat each is meant to address, and the subsections that follow take them in order.

3.1. Task taxonomy

Tasks fall into four categories. Corrective tasks repair a defect described in an issue. Adaptive tasks respond to a change in the environment, typically a new major version of a library or language whose interfaces have moved. Perfective tasks add or alter a small feature in existing code. Preventive tasks restructure code without changing what it does, in the sense of refactoring given by Fowler (2018). The categories differ in how success is judged, so results are always reported by category and never only as one aggregate.

3.2. Deriving tasks from repository history

Candidate tasks are taken from merged changes in open-source repositories that have a permissive licence, an executable test suite and an active history. A change qualifies if it touches source and tests, if the tests it adds or alters fail before it and pass after it, and if it can be assigned to one category from its description and content. The task statement given to the model is the issue text or, where none exists, a description written from the commit message with any hint of the solution removed. Refactoring tasks

have no failing test by definition; their statement names the restructuring wanted, and the oracle described below changes accordingly. Each task records the repository, the two commits, the category and the person who classified it, and a sample is classified twice so that agreement can be reported.

Table 1: Elements of the proposed framework and the threat to validity or reproducibility that each addresses		
Element	What is specified	Threat addressed
Task taxonomy	Corrective, adaptive, perfective and preventive tasks, reported separately	Scores that hide uneven ability
Task derivation	Rules for mining commits and writing the task statement	Selection bias; leaked solutions
Execution environment	One container image per task with pinned dependencies	Builds that cannot be repeated
Oracles	Target tests, regression tests and a check on the size of the change	Weak tests; unintended edits
Contamination control	Cut-off dates, a held-back split and periodic refresh	Memorised benchmark content
Invocation protocol	Fixed prompt, context budget, sampling settings and model identifier	Unreported tuning; model drift
Reporting and release	Intervals, paired comparisons, failure categories, full logs	Unsupported rankings

3.3. Execution environment

Every task is distributed with a container image in which the repository builds and its tests run at the parent commit. Containers are the established means of fixing an execution environment (Merkel, 2014). Dependency versions are pinned by hash, network access is disabled during evaluation, and the image digest is part of the task identifier. Before a task is admitted, its tests are run repeatedly at both commits. Any test whose outcome varies is excluded from the oracle, and a task left without a stable discriminating test is dropped.

3.4. Oracles and measures

The primary measure is the proportion of tasks resolved. A corrective, adaptive or perfective task is resolved when the target tests pass and all previously passing tests still pass. A preventive task is resolved when the whole suite still passes and a static check confirms the requested structural property, for example that a named function has been split or a deprecated call no longer appears. Two secondary measures guard against unintended change: the number of files and lines altered relative to the reference patch, and whether the patch applies and builds at all. Textual similarity to the reference is not used as a measure of correctness.

3.5. Contamination control

Three controls limit the chance that a model has seen the answer. Tasks are drawn only from changes made after a stated date, and every report gives that date next to the documented training cut-off of each model. Part of the task set is held back and released only as aggregate scores, so that a gap between public and held-back results can reveal memorisation. The public set is refreshed on a fixed schedule with newer changes, and old versions remain available under their own identifiers so that earlier results stay interpretable.

3.6. Invocation protocol

The prompt template, the method of selecting repository context, the context budget, the sampling

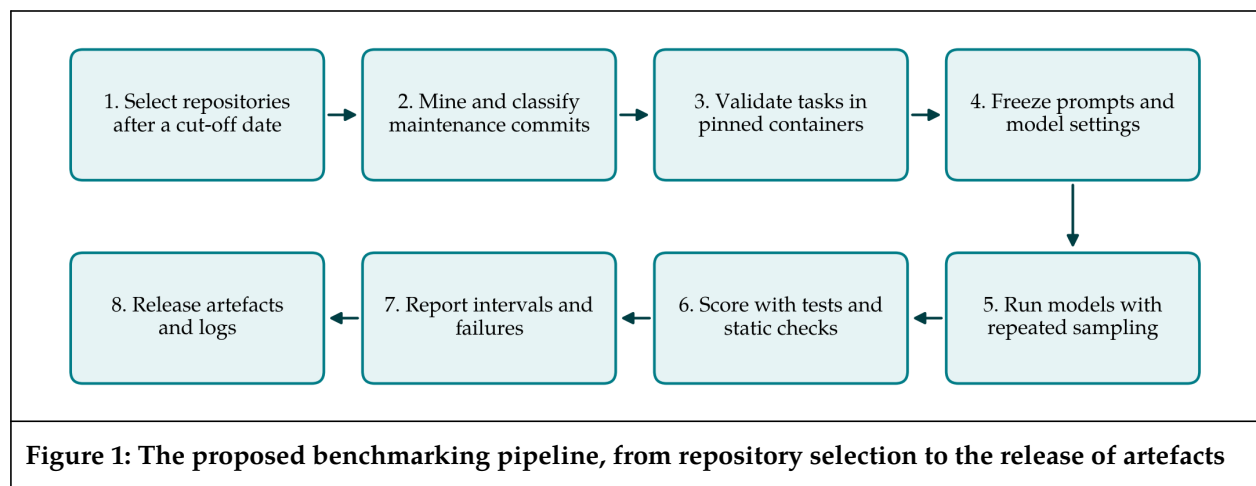
temperature, the number of samples per task and any tool access are fixed in a configuration file published with the results. Hosted models are identified by the provider's dated version string together with the dates of the run. For open-weight models the checksum of the weights is recorded. Raw requests and responses are logged in full. Development of prompts is confined to a small tuning split that plays no part in reported scores. Figure 1 shows the pipeline from repository selection to release.

3.7. Statistical reporting and release

Because model output is sampled, each model is run several times per task and the resolved proportion is reported with a bootstrap confidence interval (Efron and Tibshirani, 1993). Comparisons between two models use a paired test on per-task outcomes and state an effect size, in line with the guidance of Arcuri and Briand (2011) for randomised algorithms in software engineering. Unresolved tasks are sorted into failure categories: no patch produced, patch does not apply, build fails, target tests fail, regression introduced. The task set, images, configuration, logs and analysis scripts are released together, following the practice encouraged by reproducibility checklists in machine learning (Pineau et al., 2021).

4. Fitting the Framework to Practice

The framework is demanding, and a group adopting it will have to make economies. The costly steps are building a container for every task and running each model many times. Both scale linearly, so a first instantiation can start with a modest number of tasks in one programming language and still report honest intervals; wide intervals are informative in themselves. Classification of tasks needs people who can read the code concerned, and is the step least suited to automation.



An organisation that wants to evaluate models on its own code can apply the same elements to private repositories. Contamination is then less of a concern, since the code has not been published, while confidentiality becomes the constraint on which models may be sent the material at all. The reporting rules apply unchanged.

5. Limitations and What Remains to Be Shown

Nothing in this article demonstrates that the framework works. Several assumptions need testing by an implementation. One is that enough qualifying changes exist in each category. Adaptive and preventive changes are often mixed into larger commits and may prove hard to isolate. Another is that the static checks for refactoring tasks can be written without encoding one particular solution. A third is that test-based oracles are adequate. Passing the project's tests does not make a patch maintainable or acceptable to its reviewers, and the framework measures neither.

Contamination controls reduce the problem without removing it. Cut-off dates published by model providers cannot be verified from outside, and newer changes may resemble older ones the model has seen. Nor can any protocol make a withdrawn hosted model available again. What the framework secures

in that case is the record: the exact inputs, outputs and environment, from which the scoring at least can be repeated.

6. Conclusion

Evaluating code-generating models on maintenance calls for tasks drawn from real change history, environments that rebuild identically, oracles that detect unintended edits, and reporting that respects randomness. The framework proposed here assembles those requirements into seven specified elements. It is offered as a design to be implemented, criticised and revised, and its usefulness will be shown only when independent groups run it and reach the same numbers.

Conflict of interest

To be declared by the author(s) before publication.

Funding

To be declared by the author(s) before publication.

References

- Arcuri, A. and Briand, L. (2011). [A practical guide for using statistical tests to assess randomized algorithms in software engineering. *Proceedings of the 33rd International Conference on Software Engineering*](https://doi.org/10.1145/1985793.1985795), 1-10. <https://doi.org/10.1145/1985793.1985795>
- Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q. and Sutton, C. (2021). [Program synthesis with large language models](https://arxiv.org/abs/2108.07732). arXiv preprint arXiv:2108.07732.
- Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H.P.O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G. et al. (2021). [Evaluating large language models trained on code](https://arxiv.org/abs/2107.03374). arXiv preprint arXiv:2107.03374.
- Collberg, C. and Proebsting, T.A. (2016). [Repeatability in computer systems research. *Communications of the ACM*](https://doi.org/10.1145/2812803), 59(3): 62-69. <https://doi.org/10.1145/2812803>
- Efron, B. and Tibshirani, R.J. (1993). [An Introduction to the Bootstrap](https://www.chapmanandhall.com/book/9780022778043). Chapman & Hall, New York.
- Fowler, M. (2018). [Refactoring: Improving the Design of Existing Code, 2nd edition](https://www.addison-wesley.com/books/9780130263658). Addison-Wesley, Boston.
- ISO/IEC/IEEE (2022). [ISO/IEC/IEEE 14764:2022, Software engineering - Software life cycle processes - Maintenance](https://www.iso.org/standard/75461.html). International Organization for Standardization, Geneva.
- Jimenez, C.E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O. and Narasimhan, K. (2024). [SWE-bench: Can language models resolve real-world GitHub issues? *Proceedings of the Twelfth International Conference on Learning Representations \(ICLR 2024\)*](https://arxiv.org/abs/2402.02701).
- Just, R., Jalali, D. and Ernst, M.D. (2014). [Defects4J: A database of existing faults to enable controlled testing studies for Java programs. *Proceedings of the 2014 International Symposium on Software Testing and Analysis*](https://doi.org/10.1145/2610384.2628055), 437-440. <https://doi.org/10.1145/2610384.2628055>
- Lientz, B.P. and Swanson, E.B. (1980). [Software Maintenance Management](https://www.addison-wesley.com/books/9780130263658). Addison-Wesley, Reading, MA.
- Liu, J., Xia, C.S., Wang, Y. and Zhang, L. (2023). [Is your code generated by ChatGPT really correct? Rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems 36 \(NeurIPS 2023\)*](https://arxiv.org/abs/2305.18455).
- Lu, S., Guo, D., Ren, S., Huang, J., Svyatkovskiy, A., Blanco, A., Clement, C., Drain, D., Jiang, D., Tang, D. et al. (2021). [CodeXGLUE: A machine learning benchmark dataset for code understanding and generation](https://arxiv.org/abs/2102.04664). arXiv preprint arXiv:2102.04664.
- Luo, Q., Hariri, F., Eloussi, L. and Marinov, D. (2014). [An empirical analysis of flaky tests. *Proceedings of*](https://arxiv.org/abs/1406.0041)

- the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 643-653.
<https://doi.org/10.1145/2635868.2635920>
- Merkel, D. (2014). *Docker: Lightweight Linux containers for consistent development and deployment. Linux Journal*, 2014(239): 2.
- Papineni, K., Roukos, S., Ward, T. and Zhu, W.-J. (2002). *BLEU: A method for automatic evaluation of machine translation. Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, 311-318. <https://doi.org/10.3115/1073083.1073135>
- Pineau, J., Vincent-Lamarre, P., Sinha, K., Larivière, V., Beygelzimer, A., d'Alché-Buc, F., Fox, E. and Larochelle, H. (2021). *Improving reproducibility in machine learning research (a report from the NeurIPS 2019 reproducibility program). Journal of Machine Learning Research*, 22(164): 1-20.
- Ren, S., Guo, D., Lu, S., Zhou, L., Liu, S., Tang, D., Sundaresan, N., Zhou, M., Blanco, A. and Ma, S. (2020). *CodeBLEU: A method for automatic evaluation of code synthesis*. arXiv preprint arXiv:2009.10297.
- Sallou, J., Durieux, T. and Panichella, A. (2024). *Breaking the silence: The threats of using LLMs in software engineering. Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*.