Decision Thresholds and Policy
Volume 1 · Number 1 · 2026

A Passing Mutant Is an Unresolved Label: Zero-One Changes in HumanEval References

Connor Palmer[1] · George Hayes[2] · Evan Anderson[3]

[1] Belgium · [2] Czechia · [3] Poland

Original publication: November 5, 2026 | Article 590

Abstract

Changing a single zero or one does not always make a program visibly fail its captured tests. Among 370 such changes to 115 eligible HumanEval reference tasks, 37 completed the original checks. Average survival was 8.57% when tasks received equal weight and 10% when mutations were pooled. A within-task comparison found no clear detection difference between mutation directions. Survival did not establish correctness: an additional singleton-list probe exposed one surviving index change. Other survivors could include equivalent or unvisited changes. Test-derived labels therefore require an explicit account of what was checked and what remains unresolved.

Bounded execution of canonical programs and single-site numeric mutations. Primary rates use the original check functions; one later boundary probe illustrates a confirmed failure without relabelling the whole survivor set.

A Passing Mutant Is an Unresolved Label: Zero-One Changes in HumanEval References

> Bounded execution of canonical programs and single-site numeric mutations. Primary rates use the original check functions; one later boundary probe illustrates a confirmed failure without relabelling the whole survivor set.

Abstract

Changing a single zero or one does not always make a program visibly fail its captured tests. Among 370 such changes to 115 eligible HumanEval reference tasks, 37 completed the original checks. Average survival was 8.57% when tasks received equal weight and 10% when mutations were pooled. A within-task comparison found no clear detection difference between mutation directions. Survival did not establish correctness: an additional singleton-list probe exposed one surviving index change. Other survivors could include equivalent or unvisited changes. Test-derived labels therefore require an explicit account of what was checked and what remains unresolved.

A finite check is evidence about a program, not its entire meaning

When two code versions pass the same examples, it is tempting to give them the same correctness label. That label can then enter an evaluation set or a preference-learning pipeline as though the examples had settled their behavior over the intended domain. A small source change provides a useful way to examine what the captured checks actually distinguish, without generating new code with a language model.

We investigate how often single numeric zero/one changes survive the original HumanEval checks. A second question asks whether detection differs between zero-to-one and one-to-zero changes within tasks that contain both kinds of eligible site. The changes are deliberately local, but they are not assumed to be semantically invalid merely because their source syntax differs from the canonical program.

Tao and Kamata, 2026 study fine-tuning coding assistants from pairwise preferences about what to change. We do not train that system or infer its results from mutation survival. The connection concerns the evidence behind labels assigned to code alternatives. If a label is based on finite tests, passing and semantic correctness must remain distinguishable when interpreting what a preference or evaluation comparison has established.

Mutation-testing research supplies the background for evaluating test sensitivity through program changes (DeMillo et al., 1978; Jia and Harman, 2011). The equivalent-mutant problem is particularly relevant: a changed program can preserve the intended behavior, so every survivor is not automatically an undiscovered fault. Our primary quantity is therefore an observed surviving-mutant fraction, not an equivalence-adjusted adequacy score or a count of confirmed bugs.

Source programs and the execution gate

The captured HumanEval source contains 164 tasks, each with a prompt, canonical solution, entry-point name and original check function (Chen et al., 2021). We assemble the prompt and canonical solution into a Python module and parse the check separately. A task's original canonical program must complete its captured check under our execution convention before any associated mutant contributes to the primary analysis.

A static screen permits only the reviewed standard-library import roots used in this source and excludes designated dynamic-code or file-input builtin calls. HumanEval/160 contains an eval call and is excluded

before execution. This is a stated execution-eligibility decision, not a claim that the task or its source implementation is malicious or incorrectly specified.

The remaining 163 originals are run with a limit of 200,000 Python line events counted across candidate and check code. Of these, 160 complete the check and three reach the limit: HumanEval/36, HumanEval/75 and HumanEval/147. Reaching the bound does not demonstrate an error in a canonical program. It means that the original cannot establish the required comparison under this declared resource convention.

Forty-five of the passing originals contain no eligible zero/one body literal. The final mutation cohort is therefore 115 tasks. The original collection has 119 tasks with such literals, but four are removed by the static or bounded-reference gate. We retain the full screening record so that those exclusions cannot disappear behind a statement that the experiment covers every HumanEval task.

The original check function runs as supplied. We do not extract only literal assertions, replace its comparison logic or add a stronger benchmark suite to the primary calculation. EvalPlus motivates the broader question of test adequacy (Liu et al., 2023), but its expanded tests are not used here. A surviving mutant consequently survives this captured check protocol, not every public or hidden test associated with the task.

Check functions can contain loops, generated inputs and computed expectations, so counting visible assertion statements would not necessarily count executed examples. We avoid equating a completed check with a common amount of testing effort across tasks. The line-event record describes work under the current interpreter, but no runtime-normalized adequacy comparison is fitted. Each survivor label retains the particular source check that produced it.

Reference gating can also select a different task mix from the source collection. A slow original that reaches the bound may have a more demanding suite or more expensive implementation, rather than poorer test quality. Excluding its mutants protects the local comparison but narrows the population. Dividing the survivor count by all 164 original tasks, or assuming the omitted tasks have the same survivor distribution, would add an unsupported extrapolation.

Enumerating a narrow class of source changes

We traverse the entry function's body and locate numeric constants whose values are zero or one. Integer and floating-point constants are allowed, with their value type preserved during replacement. Boolean constants are excluded even though Python can compare them numerically with zero and one. Function-signature defaults and code outside the target body are not mutation sites.

For every eligible occurrence, we make a fresh copy of the original abstract syntax tree and replace only that occurrence. A zero becomes one, and a one becomes zero. Nested bodies within the entry function remain within the traversal. The original syntax tree is restored for the next mutation, so the experiment does not accumulate edits or examine higher-order combinations of altered sites.

The AST representation makes the changed source location inspectable (Python Software Foundation, 2026a). Occurrences are counted separately even if two changes eventually compile to behaviorally equivalent programs. We do not deduplicate mutants by guessed semantics or select only visually suspicious edits. The exhaustive eligible set after the reference gate contains 370 one-site mutants.

Each mutant receives a fresh execution namespace and the same pseudorandom seed, 20261108, before its original check is invoked. Standard-library imports remain available under the static screen. Standard output and error are discarded for the scoring call, while the outcome category and counted line events are

saved. The tracing mechanism provides the declared resource bound (Python Software Foundation, 2026b); it is not a wall-clock runtime measure.

A mutant survives when the full check completes. Assertion failure, another exception or reaching the line-event limit counts as detected under this protocol. A line-limit outcome is not proof of mathematical nontermination, and an exception is not independently adjudicated against the full natural-language contract. These categories describe what the bounded check observes and should remain visible alongside the binary survivor label.

Two estimands with different observation units

First, we calculate each task's fraction of surviving mutants and average those 115 fractions. This gives each task equal importance, regardless of how many zero/one occurrences it contains. Model 1 is an intercept regression on those task summaries, with HC3 covariance and t114 intervals. There are no extra controls in that model.

Second, we restrict the mutant-level table to tasks that contain both mutation directions. Fifty-seven tasks contribute 272 mutants. Model 2 is a linear probability association between detection and a zero-to-one indicator, with task fixed effects. Covariance clusters by those fifty-seven tasks with a finite correction, and intervals use t56. The table displays the direction coefficient rather than the collection of task intercepts.

Robust and clustered covariance address different working error structures (White, 1980; Cameron and Miller, 2015). The task fixed effects absorb between-task levels, but numeric literal roles are not randomized within a task. A zero used as an initializer can serve a different purpose from a one used as a loop bound. The direction coefficient is therefore a conditional source-location association, not a causal effect of changing a zero rather than a one.

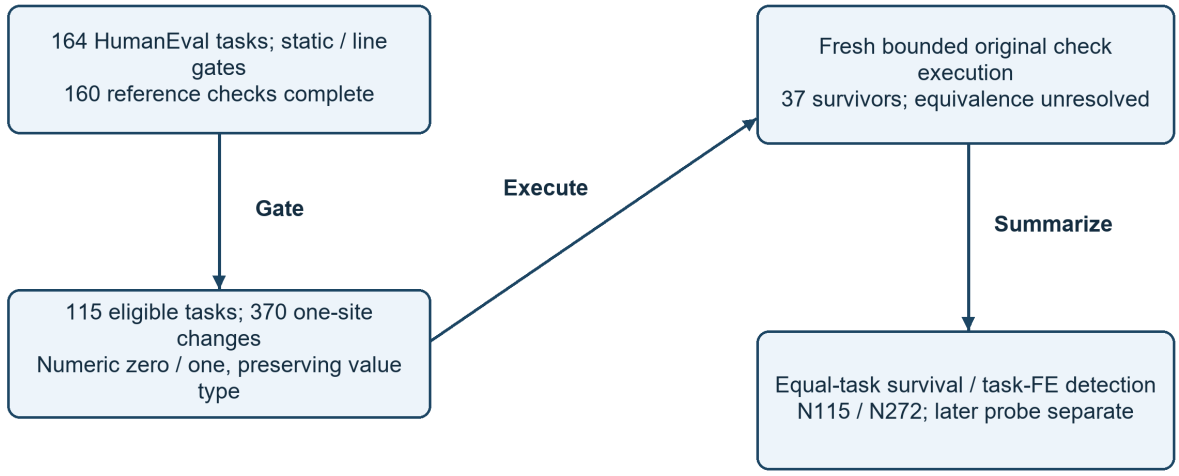

Figure 1. *Static and bounded-reference gates precede one-site mutation execution. Task-average survival and within-task detection association use different observation units; the later boundary probe is outside the primary rate calculation.*

Table 1. *M1: equal-task surviving fraction, N=115, intercept only, HC3 and t114. M2: zero-to-one detection coefficient, N=272 mutants from 57 tasks, task fixed effects, task-clustered covariance and t56. M1 R-squared is zero; M2 reports its full-model fit.*

Model	Term	Estimate (SE)	95% CI	N
M1	Mean outcome/contrast	0.085697 (0.020656)	[0.044779, 0.126616]	115
M1: fit	R-squared = 0.0000	-	-	115
M2	Zero-to-one versus one-to-zero mutation	-0.015426 (0.041162)	[-0.097882, 0.067031]	272
M2: fit	R-squared = 0.3822	-	-	272

Model labels: M1 = Equal-task surviving-mutant fraction; M2 = Zero-to-one versus one-to-zero detection, within-task association.

The different weighting is intentional. Pooling all mutants gives more influence to programs with many eligible literals. That can be appropriate for a source-site question, but it does not estimate the same task-average quantity as Model 1. The descriptive pooled fraction is saved separately so that a denominator choice does not silently alter the result being communicated.

Most changes are detected, with a nonempty unresolved set

Thirty-seven of the 370 mutants survive, giving pooled survival of 0.100000. Twenty-two of the 115 tasks have at least one survivor. The equal-task mean survival fraction is 0.085697, with standard error 0.020656 and interval [0.044779, 0.126616]. This is a conditional summary of the enumerated changes, not an estimated error rate for arbitrary code edits produced by a coding assistant.

Of the detected mutants, 285 produce AssertionError, sixteen reach the line limit, fifteen produce IndexError, eight ValueError, six RecursionError, two ZeroDivisionError and one OverflowError. These categories are retained rather than all being presented as ordinary failed output comparisons. A different resource limit could change which bounded outcomes are observed, even if no source edit changes.

Across the full mutant table, the detection fractions are approximately 0.897590 for zero-to-one changes and 0.901961 for one-to-zero changes. Within the both-direction task cohort, the adjusted direction coefficient is -0.015426, standard error 0.041162 and interval [-0.097882, 0.067031]. Full-model R-squared is approximately 0.382178.

That interval does not establish equality or an equivalence margin between directions. It indicates that this conditional comparison is imprecise relative to the observed effect size. Statistical reporting guidance supports keeping that uncertainty visible rather than treating a nonsignificant coefficient as proof that literal direction never matters (Wasserstein and Lazar, 2016). The task effects and selected site composition also remain part of the result's scope.

The linear probability model is an association summary and can produce fitted levels outside the unit interval. We do not use it as a calibrated detector or assign predicted correctness probabilities to new programs. Its purpose is to compare detection within the observed tasks under a transparent adjustment, not to replace the actual execution outcomes with model-generated labels.

One survivor with a concrete counterexample

Inspection of the survivor table identified a change in HumanEval/35, which returns the maximum element of a list. The mutation changes the initial index from zero to one. It passes the two supplied checks, both of which provide lists containing multiple elements. That survival alone does not reveal how the edited program handles a singleton list.

We subsequently ran a separate boundary probe using the list [7] and expected result 7. The canonical program completes it, while the surviving mutant raises IndexError. The probe is selected after inspecting the survivor and is recorded as a post hoc illustration. It is not a blind held evaluation set, and it does not alter the primary count of thirty-seven survivors under the original checks.

This case confirms a missed distinction for one particular mutant on a straightforward nonempty list. It does not adjudicate the other thirty-six survivors. Some changes can add redundant computation, alter an unvisited branch or preserve behavior on the intended domain. Oracle research explains why evidence about outputs and evidence about specification compliance require care (Barr et al., 2015).

There are consequently two errors to avoid: declaring every survivor correct because the check passed, or declaring every survivor incorrect because a literal changed. The experiment leaves an unresolved category and supplies an executed counterexample for one member. That is a more precise basis for a test-derived pair label than either universal interpretation.

Relevance to coding evaluation and preference data

Other code resources organize different tasks and test protocols. MBPP, QuixBugs and SWE-bench provide distinct evaluation settings (Austin et al., 2021; Lin et al., 2017; Jimenez et al., 2023), while LiveCodeBench addresses another benchmark construction and evaluation boundary (Jain et al., 2024). We do not merge their tasks into the mutation denominator or infer that the observed survivor fraction applies across those resources.

Learning methods also rely on different supervision signals. Direct preference optimization, human-feedback learning and CodeRL illustrate preference- or execution-informed routes (Rafailov et al., 2023; Ouyang et al., 2022; Le et al., 2022). The current test results are not human preferences. A preference for one coding change can involve maintainability, scope or intent even when both alternatives satisfy the same examples.

Code models and representations such as DeepSeek-Coder, GraphCodeBERT and InCoder address generation or understanding with richer learned information (Guo et al., 2024; Guo et al., 2020; Fried et al., 2022). No such model proposes or ranks our mutants. The edits are enumerated mechanically, preserving a simple source of variation whose relationship to real assistant-generated errors would need a separate empirical study.

Static structure and testing cost provide further dimensions. Cyclomatic complexity characterizes control structure (McCabe, 1976), and cost-aware prioritization studies how tests can be ordered or valued (Elbaum et al., 2001). Defect classifications and empirical software-engineering guidance encourage matching the measured changes to the claimed fault population (Herzig et al., 2013; Kitchenham et al., 2002). One zero/one mutation class cannot represent all software defects or all useful code revisions.

What the saved evidence permits next

The artifact table retains task identifiers, literal locations, mutation direction, execution category and line counts. Influence analysis could guide deeper review of tasks contributing many survivors (Cook, 1977). A later validation study should define its intended edit population and independent test boundary explicitly (Roberts et al., 2017), including how semantic equivalence and resource-limited originals are handled.

A passing check and an adjudicated correct program remain different labels, and the dataset record should preserve that distinction (Gebru et al., 2021). Production-failure detection using distributed tracing is the setting named by Yan, 2026; it illustrates why an observed status must be tied to the evidence that produced it. Here that evidence is a bounded test execution, not a service-level trace or a root-cause diagnosis. Hidden dependencies in reused labels can create technical debt (Sculley et al., 2015). The observed passing set therefore stays unresolved except where an additional boundary test supplies a concrete counterexample.

References

1. Sculley, D., et al. "Hidden Technical Debt in Machine Learning Systems." *Advances in Neural Information Processing Systems*, vol. 28, 2015. https://papers.nips.cc/paper/2015/hash/86df7dcfd896fcaf2674f757a2463eba-Abstract.html
2. Liu, Jiawei; Xia, Chunqiu Steven; Wang, Yuyao; Zhang, Lingming. (2023). Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation. arXiv:2305.01210. https://arxiv.org/abs/2305.01210.
3. DeMillo, R.A.; Lipton, R.J.; Sayward, F.G.. (1978). Hints on Test Data Selection: Help for the Practicing Programmer. Computer. https://doi.org/10.1109/c-m.1978.218136.
4. Herzig, Kim; Just, Sascha; Zeller, Andreas. (2013). It's not a bug, it's a feature: How misclassification impacts bug prediction. https://doi.org/10.1109/ICSE.2013.6606585.
5. Jain, Naman; Han, King; Gu, Alex; Li, Wen-Ding; Yan, Fanjia; Zhang, Tianjun; Wang, Sida; Solar-Lezama, Armando; Sen, Koushik; Stoica, Ion. (2024). LiveCodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code. arXiv:2403.07974. https://arxiv.org/abs/2403.07974.
6. Cameron, A. Colin, and Douglas L. Miller. "A Practitioner's Guide to Cluster-Robust Inference." *Journal of Human Resources*, vol. 50, no. 2, 2015, pp. 317-372. https://doi.org/10.3368/jhr.50.2.317.
7. Ouyang, Long; Wu, Jeff; Jiang, Xu; Almeida, Diogo; Wainwright, Carroll L.; Mishkin, Pamela; Zhang, Chong; Agarwal, Sandhini; Slama, Katarina; Ray, Alex; Schulman, John; Hilton, Jacob; Kelton, Fraser; Miller, Luke; Simens, Maddie; Askell, Amanda; Welinder, Peter; Christiano, Paul; Leike, Jan; Lowe, Ryan. (2022). Training language models to follow instructions with human feedback. arXiv:2203.02155. https://arxiv.org/abs/2203.02155.
8. Fried, Daniel; Aghajanyan, Armen; Lin, Jessy; Wang, Sida; Wallace, Eric; Shi, Freda; Zhong, Ruiqi; Yih, Wen-tau; Zettlemoyer, Luke; Lewis, Mike. (2022). InCoder: A Generative Model for Code Infilling and Synthesis. arXiv:2204.05999. https://arxiv.org/abs/2204.05999.
9. Guo, Daya; Zhu, Qihao; Yang, Dejian; Xie, Zhenda; Dong, Kai; Zhang, Wentao; Chen, Guanting; Bi, Xiao; Wu, Y.; Li, Y. K.; Luo, Fuli; Xiong, Yingfei; Liang, Wenfeng. (2024). DeepSeek-Coder: When the Large Language Model Meets Programming -- The Rise of Code Intelligence. arXiv:2401.14196. https://arxiv.org/abs/2401.14196.
10. Cook, R. Dennis. (1977). Detection of Influential Observation in Linear Regression. Technometrics. https://doi.org/10.1080/00401706.1977.10489493.
11. Gebru, Timnit, et al. "Datasheets for datasets" *Communications of the ACM*, 2021. https://doi.org/10.1145/3458723.

12. Chen, Mark; Tworek, Jerry; Jun, Heewoo; Yuan, Qiming; Pinto, Henrique Ponde de Oliveira; Kaplan, Jared; Edwards, Harri; Burda, Yuri; Joseph, Nicholas; Brockman, Greg; Ray, Alex; Puri, Raul; Krueger, Gretchen; Petrov, Michael; Khlaaf, Heidy; Sastry, Girish; Mishkin, Pamela; Chan, Brooke; Gray, Scott; Ryder, Nick; Pavlov, Mikhail; Power, Alethea; Kaiser, Lukasz; Bavarian, Mohammad; Winter, Clemens; Tillet, Philippe; Such, Felipe Petroski; Cummings, Dave; Plappert, Matthias; Chantzis, Fotios; Barnes, Elizabeth; Herbert-Voss, Ariel; Guss, William Hebgen; Nichol, Alex; Paino, Alex; Tezak, Nikolas; Tang, Jie; Babuschkin, Igor; Balaji, Suchir; Jain, Shantanu; Saunders, William; Hesse, Christopher; Carr, Andrew N.; Leike, Jan; Achiam, Josh; Misra, Vedant; Morikawa, Evan; Radford, Alec; Knight, Matthew; Brundage, Miles; Murati, Mira; Mayer, Katie; Welinder, Peter; McGrew, Bob; Amodei, Dario; McCandlish, Sam; Sutskever, Ilya; Zaremba, Wojciech. (2021). Evaluating Large Language Models Trained on Code. arXiv:2107.03374. https://arxiv.org/abs/2107.03374.

13. McCabe, T.J.. (1976). A Complexity Measure. IEEE Transactions on Software Engineering. https://doi.org/10.1109/tse.1976.233837.

14. Barr, Earl T.; Harman, Mark; McMinn, Phil; Shahbaz, Muzammil; Yoo, Shin. (2015). The Oracle Problem in Software Testing: A Survey. IEEE Transactions on Software Engineering. https://doi.org/10.1109/tse.2014.2372785.

15. Kitchenham, Barbara A.; Pfleeger, Shari Lawrence; Pickard, Lesley M.; Jones, Peter W.; Hoaglin, David C.; El Emam, Khaled; Rosenberg, Jarrett. (2002). Preliminary guidelines for empirical research in software engineering. https://doi.org/10.1109/TSE.2002.1027796.

16. Yan, Biyun. (2026). From Observability to Causality: Detecting Production Failures in Microservices with Distributed Tracing. 2026 9th International Conference on Computer Information Science and Application Technology (CISAT), 1522-1525. https://doi.org/10.1109/CISAT71030.2026.11682113.

17. Python Software Foundation. (2026a). ast: Abstract Syntax Trees. Python3.12 documentation, captured7November2026; capture year is not the language release year. https://docs.python.org/3.12/library/ast.html.

18. Guo, Daya; Ren, Shuo; Lu, Shuai; Feng, Zhangyin; Tang, Duyu; Liu, Shujie; Zhou, Long; Duan, Nan; Svyatkovskiy, Alexey; Fu, Shengyu; Tufano, Michele; Deng, Shao Kun; Clement, Colin; Drain, Dawn; Sundaresan, Neel; Yin, Jian; Jiang, Daxin; Zhou, Ming. (2020). GraphCodeBERT: Pre-training Code Representations with Data Flow. arXiv:2009.08366. https://arxiv.org/abs/2009.08366.

19. Le, Hung; Wang, Yue; Gotmare, Akhilesh Deepak; Savarese, Silvio; Hoi, Steven C. H.. (2022). CodeRL: Mastering Code Generation through Pretrained Models and Deep Reinforcement Learning. arXiv:2207.01780. https://arxiv.org/abs/2207.01780.

20. Roberts, David R., et al. “Cross-validation strategies for data with temporal, spatial, hierarchical, or phylogenetic structure.” *Ecography*, vol. 40, no. 8, 2017, pp. 913-929. https://doi.org/10.1111/ecog.02881.

21. Austin, Jacob; Odena, Augustus; Nye, Maxwell; Bosma, Maarten; Michalewski, Henryk; Dohan, David; Jiang, Ellen; Cai, Carrie; Terry, Michael; Le, Quoc; Sutton, Charles. (2021). Program Synthesis with Large Language Models. arXiv:2108.07732. https://arxiv.org/abs/2108.07732.

22. Rafailov, Rafael; Sharma, Archit; Mitchell, Eric; Ermon, Stefano; Manning, Christopher D.; Finn, Chelsea. (2023). Direct Preference Optimization: Your Language Model is Secretly a Reward Model. arXiv:2305.18290. https://arxiv.org/abs/2305.18290.

23. Jia, Yue; Harman, Mark. (2011). An Analysis and Survey of the Development of Mutation Testing. IEEE Transactions on Software Engineering37(5):649-678. https://doi.org/10.1109/TSE.2010.62.

24. Elbaum, S.; Malishevsky, A.; Rothermel, G.. (2001). Incorporating varying test costs and fault severities into test case prioritization. Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001. https://doi.org/10.1109/icse.2001.919106.

25. Python Software Foundation. (2026b). Python 3.12 documentation: sys.settrace. Documentation snapshot captured 7 November 2026. https://docs.python.org/3.12/library/sys.html#sys.settrace.

26. Lin, Derrick; Koppel, James; Chen, Angela; Solar-Lezama, Armando. (2017). QuixBugs: a multi-lingual program repair benchmark set based on the quixey challenge. Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity. https://doi.org/10.1145/3135932.3135941.

27. Wasserstein, Ronald L.; Lazar, Nicole A.. (2016). The ASA Statement on p -Values: Context, Process, and Purpose. The American Statistician. https://doi.org/10.1080/00031305.2016.1154108.

28. Jimenez, Carlos E.; Yang, John; Wettig, Alexander; Yao, Shunyu; Pei, Kexin; Press, Ofir; Narasimhan, Karthik. (2023). SWE-bench: Can Language Models Resolve Real-World GitHub Issues?. arXiv:2310.06770. https://arxiv.org/abs/2310.06770.

29. Tao, Hongmo; Kamata, Hiroyuki. (2026). Teaching Coding Assistants What to Change: Fine-Tuning from Pairwise Preferences. Proceedings of the Second Workshop for REsearch on Agent Language Models (REALM 2026), 1363-1374. https://doi.org/10.18653/2026.realm-1.95.

30. White, Halbert. “A Heteroskedasticity-Consistent Covariance Matrix Estimator and a Direct Test for Heteroskedasticity.” *Econometrica*, vol. 48, no. 4, 1980, pp. 817-838. https://doi.org/10.2307/1912934.