Uncertainty and Assessment
Volume 1 · Number 1 · 2026

Testing Until the First Failure: Execution Order and Measured Candidate Work

Tyler Hayes[1] · Ethan Griffin[2] · Ryan King[3]

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

Original publication: October 9, 2026 | Article 640

Abstract

Finding a failure sooner can save work without discarding tests. On 26 QuixBugs tasks with 198 captured cases, ordering by prior reference-program execution cost used 116,723 candidate line events before the first failure, compared with an exact uniform-order expectation of 185,432. The equal-task geometric work ratio was 0.419, although ten tasks required more work under the guided ordering. Source-file order used slightly fewer total events than the guided policy, and their paired log-cost difference was inconclusive. These are costs reconstructed from isolated bounded runs; profiling expense and elapsed execution time are outside the comparison.

A replay analysis of independently initialized, bounded public-program execution traces. The full case suite remains eligible under every ordering policy.

Testing Until the First Failure: Execution Order and Measured Candidate Work

> A replay analysis of independently initialized, bounded public-program execution traces. The full case suite remains eligible under every ordering policy.

Abstract

Finding a failure sooner can save work without discarding tests. On 26 QuixBugs tasks with 198 captured cases, ordering by prior reference-program execution cost used 116,723 candidate line events before the first failure, compared with an exact uniform-order expectation of 185,432. The equal-task geometric work ratio was 0.419, although ten tasks required more work under the guided ordering. Source-file order used slightly fewer total events than the guided policy, and their paired log-cost difference was inconclusive. These are costs reconstructed from isolated bounded runs; profiling expense and elapsed execution time are outside the comparison.

1. A failure verdict and the route to obtaining it

An assistant evaluating a proposed code change often needs to know whether any available test fails. Once a failure has been observed, the candidate can be rejected under that decision rule without executing every remaining case. This creates a resource question even when the test suite itself is fixed: which available case should be evaluated first, and how much work is spent reaching the first negative verdict?

The question differs from retaining only a small subset of tests. Subset selection can lose failures that occur only in discarded cases. Ordering keeps all cases available and changes their sequence. If outcomes do not depend on execution order, stopping at the first failure preserves the complete suite's any-failure verdict. It does not preserve every individual result, identify every defect, or establish that a passing suite proves program correctness.

Tao and Kamata, 2026 study fine-tuning coding assistants from pairwise preferences about what to change. Such preference construction requires a defensible account of how candidates are judged. The present study does not fine-tune an assistant or supply human preferences. It examines an operational component of test-derived evidence: whether prior reference execution cost can prioritize a complete suite so that a candidate's first failing case is reached with less measured candidate work.

The research question is deliberately conditional on a recorded suite and execution protocol. We compare a reference-cost ordering, captured source-file order, and the exact expectation over uniformly random permutations. The first two have the same eventual failure verdict as the full suite in this sample. Their costs can nevertheless differ substantially, and a policy that lowers average work need not improve every task.

2. Prioritization, cost, and the evidence supplied by tests

Test prioritization seeks a useful order rather than automatically minimizing the number of available cases (Rothermel et al., 2001). The distinction between minimization, selection, and prioritization is developed by Yoo and Harman, 2012. Our decision concerns one captured candidate per task, not a growing regression history across software releases. We borrow the ordering question while keeping that narrower experimental object explicit.

Elbaum et al., 2001 incorporates varying test costs and fault severities into prioritization evaluation. Our endpoint is simpler: traced source-line events in the candidate up to and including the first nonpassing

case. We do not attach severity weights to defects, and we do not calculate an average percentage of faults detected. Cost-sensitive reasoning nevertheless matters because two cases that count equally as tests can require radically different work (Elkan, 2001).

The corrected reference provides an available ranking signal. Its observed line count is used to place cheaper reference cases earlier; ties follow the captured case index. The candidate's failure status and line count are withheld from this ranking. A cheap reference case may be expensive in the buggy candidate or may pass when a more costly case fails. Therefore, a lower reference cost is a hypothesis about useful ordering, not a guarantee about candidate stopping work.

Test outcomes also depend on the oracle. Barr et al., 2015 describes the broader problem of determining whether observed behavior is correct. Here we retain an earlier explicit compatibility convention for generated containers and numeric outputs. We do not silently switch to literal Python equality. The selected cohort comprises tasks whose designated corrected reference passed every captured case under this convention, so the ordering analysis starts from an already documented operational judgment.

The complexity of program control flow motivates measuring execution instead of using test count alone (McCabe, 1976). Python tracing supplies the recorded line-event unit (Python Software Foundation, 2026). This unit is neither machine instructions nor elapsed seconds. Different Python expressions can perform very different work within one traced line, and tracing itself adds overhead. The study's conclusions therefore concern the measured trace workload, with a separate unresolved question about wall-clock savings.

3. Captured tasks and reproducible ordering rules

The source is the public QuixBugs collection of algorithmic programs (Lin et al., 2017). The retained analysis includes 26 reference-compatible tasks and 198 JSON cases. Every one of these tasks contains at least one nonpassing buggy-program case. This is an intentionally failure-bearing cohort, so the observed universal detection of a failure cannot estimate how often unrelated generated candidates would be correct or incorrect.

The underlying cases were executed with independent initialization and a bounded source-line trace protocol. Runs exceeding the operational bound remain nonpassing outcomes with their recorded work; they are not removed to make ordering look cheaper. The present analysis reuses those measurements. It does not claim to have executed a new, state-sharing test process under every possible permutation, and it assumes that isolated case outcomes and costs remain fixed under the replayed order.

This assumption matters for transfer. Tests that mutate a shared database, depend on caches, or alter global state can change both cost and result when reordered. The replay calculations would then require additional measurements under the actual execution arrangement. For the recorded algorithmic cases, the isolated-run interpretation is transparent and reproducible: a test's measured candidate cost is attached to that case regardless of its position.

Reference-cost order sorts the full case list by increasing corrected-reference line count and then by source case index. Source order sorts only by that index. For either sequence, the measured work is the sum of candidate line counts for the prefix ending at its first nonpass. All remaining tests remain eligible in principle, but the stopping rule does not request their results once the task's negative verdict has been established.

The uniform comparator does not rely on Monte Carlo permutations. Suppose a task has f failing cases. Each failing case is executed exactly when it is the first failing case in the permutation, which occurs with

probability 1/f. Each passing case is executed when it precedes all f failures, with probability 1/(f+1). Summing its candidate cost times the appropriate probability gives exact expected stopping work. If there were no failures, every case would be executed; that branch is implemented although none of the 26 retained tasks requires it.

This derivation permits an exact reference expectation conditional on the measured suite. It does not make the costs known prospectively to the guided policy: failure indicators and candidate costs enter the retrospective comparator calculation only. The ranking rule itself reads corrected-reference costs alone. Maintaining that separation avoids using the answer being predicted to construct an apparently efficient ordering.

4. Regression summaries of paired work

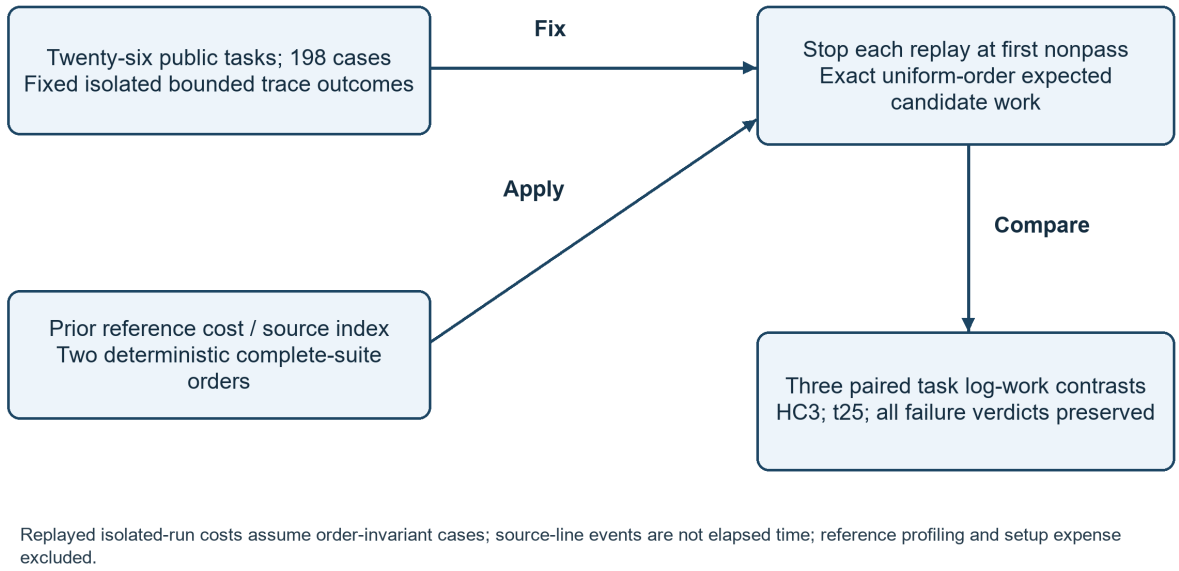

Figure 1. *Prior reference costs define an order; fixed isolated candidate traces determine first-failure prefix work. The complete case set and binary failure verdict remain fixed under the replay assumption.*

Each task contributes one row. Model 1 regresses the log ratio of reference-cost work to uniform expected work on an intercept. Model 2 uses the corresponding source-order log ratio. Model 3 directly compares reference-cost and source-order work within the same task. This last contrast is necessary because a smaller coefficient in one separate comparison does not itself establish a reliable difference between two policies.

All three models use HC3 covariance and t25 intervals with 26 task rows. The intercept estimates the mean task log ratio; exponentiation gives a geometric mean of task ratios. Total line events across tasks instead give an arithmetic workload comparison dominated by large tasks. We report both estimands and avoid treating them as interchangeable. Intercept-only R-squared supplies no substantive measure of predictive fit.

Heteroskedasticity and task dependence require care in empirical summaries (White, 1980; Cameron and Miller, 2015). HC3 does not account for an unobserved clustering of algorithm families, and these tasks are not a probability sample of software projects. Diagnostics and influential observations matter when costs are highly uneven (Cook, 1977). The intervals describe between-task dispersion under the working model, conditional on the captured costs, without uncertainty from a future profiling run.

Table 1. *Intercept-only paired log-work regressions. N=26 tasks in every model; HC3 standard errors and t25 confidence intervals. There are no covariate controls or fixed effects. Work counts candidate source-line events only, excluding reference profiling, trace setup, and all unexecuted suffix cases.*

Model	Term	Estimate (SE)	95% CI	N
M1	Mean outcome/contrast	-0.8699 (0.3812)	[-1.6550, -0.0848]	26
M1: fit	R-squared = 0.0000	-	-	26
M2	Mean outcome/contrast	-0.5790 (0.3391)	[-1.2773, 0.1194]	26
M2: fit	R-squared = 0.0000	-	-	26
M3	Mean outcome/contrast	-0.2909 (0.2013)	[-0.7055, 0.1236]	26
M3: fit	R-squared = 0.0000	-	-	26

Model labels: M1 = Reference-cost order versus uniform order, log candidate work; M2 = Source order versus uniform order, log candidate work; M3 = Reference-cost order versus source order, log candidate work.

5. Fewer events can require more tests

Uniform random ordering has total expected candidate work of 185,431.995 events across the tasks. Reference-cost order uses 116,723, while source order uses 116,347. Thus the guided rule reduces the aggregate relative to random ordering, but it does not produce the smallest aggregate among the two deterministic orders. The difference between those totals should be reported even when an equal-task summary favors the guided rule.

Model 1 estimates a mean log ratio of -0.869855, with standard error 0.381203 and interval [-1.654957, -0.084753]. Its geometric mean task ratio is 0.419012. Model 2 estimates -0.578954 with standard error 0.339063 and interval [-1.277267, 0.119359], corresponding to a ratio of 0.560484. These exponentiated means do not say that every task saves 58% or 44% of its work.

The direct paired estimate in Model 3 is -0.290901, with standard error 0.201283 and interval [-0.705451, 0.123649]. This interval includes no difference. It therefore does not support confidently ranking the guided policy ahead of source order on the equal-task log-work estimand. A favorable random-order comparison and an inconclusive source-order comparison can both hold without contradiction.

Relative to uniform expected work, the reference-cost order requires fewer events on fifteen tasks, more on ten, and the same on one. Source order improves fourteen, worsens eleven, and ties one. In the bitcount task, every case reaches the recorded bound behavior, and each ordering stops after one case with 100,001 events. This dominant cost remains in the totals; deleting it would redefine the experimental workload after seeing the outcome.

The number of requested tests tells a different story. Uniform ordering has an expected total of 43.641 cases before stopping. Reference-cost ordering requests 72, and source order requests 56. More low-cost cases can therefore precede a failure while requiring fewer traced events. A test-count measure would miss that distinction, and a claim that the guided policy always finds a failure in fewer tests would be false.

The find_first_in_sorted task illustrates the sensitivity to costly cases. Its uniform expected work is 66,687.5 events, compared with 74 under reference-cost order and 36 under source order. The guided

sequence requests five cases before failure; source order requests two. These are descriptive examples from the full retained cohort, not grounds for tuning a new ordering rule on the same tasks.

6. How this result should inform coding evaluation

Execution-based code evaluation requires explicit tests and reporting conventions, as illustrated by EvalPlus and LiveCodeBench (Liu et al., 2023; Jain et al., 2024). An agent benchmark such as SWE-agent adds interaction and repository context beyond these small algorithmic cases (Yang et al., 2024). The observed trace-cost ratios should not be transferred numerically to those systems. They instead identify a reporting choice that such evaluations can make explicit: whether cost means all tests, a fixed subset, or an ordered prefix until rejection.

Preference-learning methods introduce a further boundary. Direct Preference Optimization and preference-based language-model training use judgments in a learning procedure (Rafailov et al., 2023; Ouyang et al., 2022). Our negative test verdicts do not establish which patch a human would prefer or how training on an ordered evaluation would change an assistant. They provide evidence about a computational judgment process that might precede such learning, with the meaning of its labels kept narrow.

Measurement practice also limits performance claims. Uncontrolled measurement can produce misleading benchmarking conclusions (Mytkowicz et al., 2009), and rigorous performance comparisons require attention to experimental variation (Kalibera and Jones, 2013). Replaying trace counts removes timing noise from this particular comparison, but it also means we have not measured runtime. Reference profiling could consume more resources than ordering later saves, especially when a suite is used only once.

Empirical software-engineering guidance favors stating selection and analysis choices (Kitchenham et al., 2002). Misclassified defect labels can change software conclusions (Herzig et al., 2013), while leakage can enter when evaluation outcomes guide method construction (Kapoor and Narayanan, 2023). We therefore preserve the compatibility gate, fixed ordering rules, adverse task differences, and the distinction between ranking inputs and retrospective audit information. The validation concerns of Roberts et al., 2017 and Varma and Simon, 2006 also argue for a fresh task cohort before choosing among new ordering heuristics.

The unit of cost is part of the scientific claim (Stevens, 1946), and agreement of verdicts is separate from agreement of costs (Bland and Altman, 1986). The saved per-case prefixes permit both to be inspected. Distributed tracing for production-failure diagnosis, addressed by Yan, 2026, presents a related need to preserve execution context across observations; our standalone source-line prefixes supply much less information than a service trace. Larger pipelines may carry additional dependencies (Sculley et al., 2015). The documented case order and stopping rule (Gebru et al., 2021) therefore bound the conclusion, and the uncertainty should remain visible rather than becoming a favorable significance label (Wasserstein and Lazar, 2016).

7. A bounded operational conclusion

Within the recorded isolated-run protocol, reference-cost ordering reduces mean task log work relative to a uniform permutation while preserving all 26 full-suite failure verdicts. It increases work on ten tasks, requests more tests overall, and is not decisively better than the captured source order. The practical implication is to report the stopping rule, cost unit, and comparator together. The present evidence supports a conditional workload calculation; evaluating net execution savings or learned coding preferences requires additional experiments.

References

1. 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.
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. Yang, John; Jimenez, Carlos E.; Wettig, Alexander; Lieret, Kilian; Yao, Shunyu; Narasimhan, Karthik; Press, Ofir. (2024). SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. arXiv:2405.15793. https://arxiv.org/abs/2405.15793.
4. 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.
5. 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.
6. Python Software Foundation. (2026). Python 3.12 documentation: sys.settrace. Documentation snapshot captured 7 November 2026. https://docs.python.org/3.12/library/sys.html#sys.settrace.
7. 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.
8. Stevens, S. S.. (1946). On the Theory of Scales of Measurement. Science. https://doi.org/10.1126/science.103.2684.677.
9. 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.
10. Rothermel, G.; Untch, R.H.; Chengyun Chu; Harrold, M.J.. (2001). Prioritizing test cases for regression testing. IEEE Transactions on Software Engineering. https://doi.org/10.1109/32.962562.
11. 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.
12. 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.
13. 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.
14. 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.
15. Mytkowicz, Todd; Diwan, Amer; Hauswirth, Matthias; Sweeney, Peter F.. (2009). Producing wrong data without doing anything obviously wrong!. Proceedings of the 14th international conference on Architectural support for programming languages and operating systems. https://doi.org/10.1145/1508244.1508275.
16. Gebru, Timnit, et al. "Datasheets for datasets" *Communications of the ACM*, 2021. https://doi.org/10.1145/3458723.
17. Yoo, S.; Harman, M.. (2012). Regression testing minimization, selection and prioritization: a survey. Software Testing, Verification and Reliability. https://doi.org/10.1002/stvr.430.
18. Varma, Sudhir and Richard Simon. “Bias in error estimation when using cross-validation for model selection.” *BMC Bioinformatics*, vol. 7, no. 1, 2006. https://doi.org/10.1186/1471-2105-7-91.
19. Bland, J. Martin, and Douglas G. Altman. "Statistical Methods for Assessing Agreement between Two Methods of Clinical Measurement." *The Lancet*, vol. 327, no. 8476, 1986, pp. 307-310. https://doi.org/10.1016/S0140-6736(86)90837-8.
20. 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.

21. Kapoor, Sayash; Narayanan, Arvind. (2023). Leakage and the reproducibility crisis in machine-learning-based science. Patterns. https://doi.org/10.1016/j.patter.2023.100804.

22. 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

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

24. Kalibera, Tomas; Jones, Richard. (2013). Rigorous benchmarking in reasonable time. Proceedings of the 2013 international symposium on memory management. https://doi.org/10.1145/2464157.2464160.

25. 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.

26. 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.

27. 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.

28. 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.

29. Elkan, Charles. “The Foundations of Cost-Sensitive Learning.” *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence*, 2001, pp. 973-978. https://cseweb.ucsd.edu/~elkan/rescale.pdf.

30. Cook, R. Dennis. (1977). Detection of Influential Observation in Linear Regression. Technometrics. https://doi.org/10.1080/00401706.1977.10489493.