BEHAVIORAL TEST DIVERSITY: QUANTIFYING TEST SUITE DIVERSITY THROUGH AGENTIC CODE REPAIR

Anonymous authors
Paper under double-blind review

ABSTRACT

As coding agents make it easier to generate code at scale, ensuring that the resulting code aligns with human intent becomes a central challenge. Testing is widely used to check whether code behaves as intended, with tests serving as executable specifications. The increasing effort required to construct tests has motivated growing use of LLM-based test generation, which requires effective objectives to guide test-suite construction. Existing objectives do not directly quantify the diversity of behavioral constraints specified by a test suite. High performance under these metrics can therefore coexist with superfluous redundancy of some behaviors and omissions of others, leaving gaps in the testing surface. We introduce Behavioral Test Diversity (BTD), an agent-conditioned metric that measures this diversity and provides an objective for test-suite optimization. BTD quantifies behavioral overlap by observing whether code repairs guided by one test also satisfy others, without needing predefined behavior categories. The resulting score serves as an objective for preserving behavioral diversity during test suite pruning and expansion. Our experiments show that BTD yields stable scores and preserves relative test-suite rankings. Across our evaluations on ProgramBench, vLLM, and ComputeEval, BTD-guided pruning reduces the number of tests supplied as additional executable specifications by 61–90% while retaining sufficient behavioral guidance for full-suite recovery. We also show that BTD-guided expansion covers previously unchecked behavior which baseline methods misses. Together, these studies support BTD's validity and practical utility.

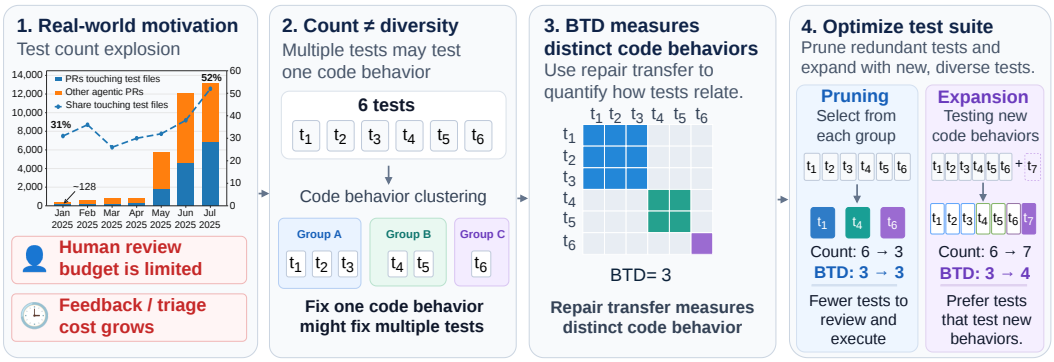

Figure 1: **Overview of the paper.** Test-related PR counts (left), estimated from Haque et al. (2026, Table 1), illustrate growing testing activity. BTD measures code behavioral diversity through test-guided code repairs, providing an objective for pruning that preserves diversity with fewer tests and expansion that adds new behavioral constraints.

1 INTRODUCTION

As coding agents make code generation cheaper, the bottleneck shifts to verifying that code aligns with human intent. Testing is widely used for this purpose. Tests serve as partial executable specifi-

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

cations (Le Goues et al., 2012; Smith et al., 2015) that encode behavioral requirements that can be checked automatically and used to guide coding agents.

While developers turn to agents to generate code, they are also using them to generate tests (Altmayer Pizzorno & Berger, 2025; Moradi Dakhel et al., 2023). Automated generation makes it easy to add tests, but more tests do not necessarily constrain more behaviors. In many cases, a growing suite may repeatedly check set of identical behavior while completely missing coverage for others. For example, ProgramBench uses coverage-guided agent generation to construct its test suites (Yang et al., 2026). In its `sd` task, 12 tests check the same requirement that previewing replacements must leave the input file unchanged (Appendix E.4). Yet multiple implementations with different error-precedence behavior can pass all 810 tests: one reports the invalid search pattern, while another prioritizes the ambiguous replacement.

This mismatch exposes a gap in test-suite optimization. Existing metrics such as code coverage, mutation score, and input distance reward structural reach, mutant discrimination, or separation in a chosen representation. They do not directly reward distinct behavioral constraints, making redundant tests difficult to distinguish from useful additions. Our work therefore asks: *how can we measure the diversity of code behaviors specified by a test suite and use it to improve that suite?*

We introduce Behavioral Test Diversity (BTD), an agent-conditioned metric of code behavior diversity. BTD uses code repair as a behavioral probe, connecting diversity measurement to how tests guide agent-based code generation. Tests provide executable constraints, and repairs reveal how an agent translates them into code changes. If repairs guided by either of two tests also satisfy the other, the tests provide overlapping behavioral guidance under that agent's repair process. BTD aggregates these relationships into a diversity score. Since BTD is able to discover diversity without predefined behavioral categories, it applies to existing and newly generated test suites.

BTD provides an objective for pruning test suites while preserving behavioral diversity. Because each test must be reviewed to ensure that it reflects human intent and executed to provide feedback, removing redundant tests can reduce review effort and shorten feedback cycles. The same objective can also guide test-suite expansion toward new tests that constrain previously uncovered behaviors, adding distinct behavioral guidance to the suite.

Overall, we make three contributions in this paper (Figure 1):

- **Behavioral diversity metric.** We introduce Behavioral Test Diversity (BTD), an agent-conditioned metric that quantifies the diversity of code behaviors specified by tests through code repair. We establish its mathematical properties and explain how it captures behavioral overlap without needing predefined behavior categories.
- **Validation and generalization.** We empirically validate BTD's stability and its ability to reveal behavioral overlap in test suites. We further demonstrate its applicability across software domains.
- **Test suite improvement.** We use BTD to guide test-suite pruning and expansion. These applications reduce redundant tests and add previously missing behavioral constraints, making test suites more compact and informative.

2 Background and Motivation

2.1 The Costs of Test-Suite Growth

Agent-based generation makes it inexpensive to propose additional tests, allowing suites to grow rapidly. The test-related pull-request activity in Figure 1 illustrates the growing scale of agent-driven testing (Haque et al., 2026). Test count explosion creates two practical bottlenecks.

Human review becomes a bottleneck. Each generated test must be checked to ensure that its expected outcome and assertions reflect human intent. Rapid AI-assisted code generation can outpace human review capacity (Anthropic, 2026), while redundant tests consume limited attention without adding new behavioral constraints.

Execution and diagnosis become more costly. Tests are rerun as code is developed and repaired. Larger suites increase execution time (Yoo & Harman, 2012), while multiple reports of the same

failure cound add diagnostic effort without necessarily providing new information. This bloat can delay useful feedback and prolong the repair cycle.

2.2 Limitations of Existing Test-Suite Metrics

Existing test-suite optimization objectives rely on proxies that do not directly quantify the diversity of code behaviors specified by a suite. Common examples include structural coverage, mutation score, and input or syntactic distance.

Structural coverage. Line, branch, and path coverage count executed program elements (Zhu et al., 1997). They locate unexercised code and provide an effective search signal (Cadar et al., 2008; Fraser & Arcuri, 2011), but structural reach is orthogonal to behavioral identity. After controlling for suite size, coverage has only low-to-moderate correlation with fault-detection effectiveness, and stronger criteria do not necessarily yield stronger suites (Inozemtseva & Holmes, 2014).

Mutation score. Mutation score measures a test suite's ability to detect injected code changes (Jia & Harman, 2011). The score depends on the chosen implementation and mutation operators. Multiple mutants may perturb the same behavior, while other behavioral distinctions may not be represented. A higher mutation score therefore does not necessarily imply that the suite specifies more distinct behavioral constraints.

Input and syntactic distance. Adaptive random testing selects inputs far from previously executed tests (Chen et al., 2010). More generally, token, tree, embedding, and domain-specific distances reward separation in a chosen representation. Syntactically different tests may constrain the same behavior, while nearby inputs may trigger different boundary conventions.

LLM-based test generators use these signals to guide optimization. CoverUp feeds uncovered lines and branches back to an LLM (Altmayer Pizzorno & Berger, 2025); MuTAP and MUTGEN use surviving mutants (Moradi Dakhel et al., 2023; Wang et al., 2026). Recent work also extracts explicit behavior catalogues from documentation and source to estimate behavioral gaps (Paul & Holmes, 2026). Such catalogues are useful, but their counts depend on which behaviors are extracted and how they are partitioned. These approaches leave a need to generalize across software domains and codebases without relying on fixed categories.

3 BTD: Behavioral Test Diversity

To address these limitations, BTD uses code repair to measure the distinct constraints tests provide. Starting from a fixed implementation, we repair against each failing test and measure how often those repairs satisfy the others. BTD then aggregates these cross-test transfer rates into an effective number of distinct constraints represented by the suite. (Figure 2).

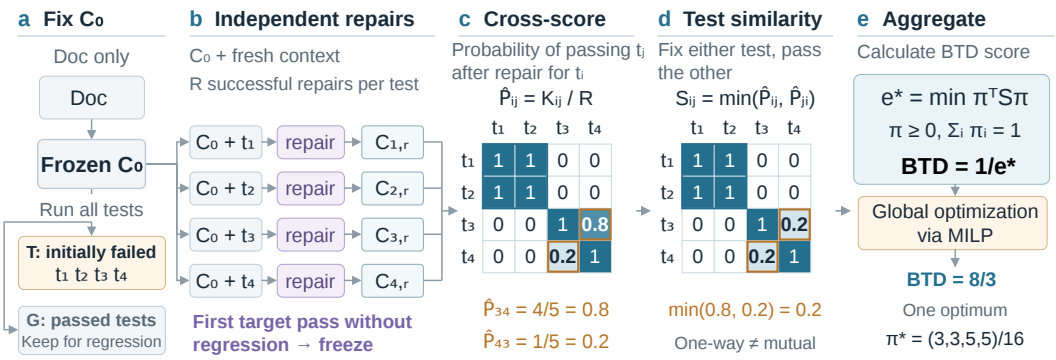

Figure 2: BTD uses code repair to measure the diversity of code behaviors specified by a test suite. Cross-test pass rates yield pairwise similarities, which BTD aggregates into an effective number of distinct behavioral constraints.

3.1 Definition and Measurement Protocol

BTD starts from an initial implementation C_0. A natural choice is to generate C_0 from documentation alone, reflecting the agent's interpretation of the documented specification before receiving test feedback. We hold C_0 fixed to make repairs guided by different tests comparable. For the available suite T_{all}, define

$$T = \{t \in T_{\text{all}} : C_0 \text{ fails } t\} = \{t_1, \ldots, t_n\}. \tag{1}$$

The failing tests in T specify behavioral requirements not yet satisfied by C_0. We use these tests as additional executable specifications for the agent, and BTD measures the diversity of the behavioral guidance they provide. The initially passing tests $G = T_{\text{all}} \setminus T$ are retained as regression constraints to preserve already satisfied requirements.

For each t_i, initialize R independent repair trajectories from a fresh copy of C_0 and a fresh context. Each trajectory continues until a candidate $C_{i,r}$ passes the target t_i without regression. Once accepted, the candidate is frozen and evaluated on all of T.

Let P_{ij} denote the cross-test pass probability: the probability that code repaired to pass t_i also passes t_j.

$$P_{ij} = \Pr[C_{i,r} \text{ passes } t_j \mid C_0, d, t_i, \text{protocol}], \qquad \widehat{P}_{ij} = \frac{1}{R}\sum_{r=1}^{R} \mathbf{1}\{C_{i,r} \text{ passes } t_j\}. \tag{2}$$

We define pairwise test similarity as the smaller of the two cross-test pass rates:

$$S_{ii} = 1, \qquad S_{ij} = \min(\widehat{P}_{ij}, \widehat{P}_{ji}) \quad (i \neq j). \tag{3}$$

We choose the minimum because a high pass probability in one direction alone can overstate behavioral overlap. For example, if t_j specifies only part of what t_i requires, repairs for t_i may pass both tests, while repairs for t_j may still fail t_i. The minimum thus provides a conservative measure of the behavioral guidance shared by the two tests.

To aggregate these pairwise similarities into a diversity score, we assign weights π to the tests, with $\pi \in \Delta_n = \{\pi \geq 0 : \sum_i \pi_i = 1\}$. We minimize the weighted average pairwise similarity $\pi^\mathsf{T} S \pi$ and take its reciprocal:

$$e^* = \min_{\pi \in \Delta_n} \pi^\mathsf{T} S \pi, \qquad \widehat{\text{BTD}}(T \mid C_0, d, \text{protocol}, R) = \frac{1}{e^*}. \tag{4}$$

Section 3.2 explains how this aggregation yields an effective number of distinct behavioral constraints.

Algorithm 1 presents the main loop for measurements. Appendix B offers full algorithm.

Algorithm 1 BTD measurement: main loop

Require: Documentation d; frozen initial implementation C_0; full suite T_{all}; repeats R
Require: Targeted repair procedure $\mathcal{A}$ with a fresh context for every run

$T \leftarrow \{t \in T_{\text{all}} : C_0 \text{ fails } t\}$ ▷ initially failed tests
$G \leftarrow T_{\text{all}} \setminus T$ ▷ initially passing regression tests
$n \leftarrow |T|$; $K \leftarrow 0_{n \times n}$
for $i = 1, \ldots, n$ **do**
  for $r = 1, \ldots, R$ **do**
    $C_{i,r} \leftarrow \mathcal{A}(\text{copy}(C_0), d, \text{target} = t_i, \text{regressions} = G)$
    $y_{i,r} \leftarrow \left(\mathbf{1}\{C_{i,r} \text{ passes } t_j\}\right)_{j=1}^{n}$
    $K_{i,:} \leftarrow K_{i,:} + y_{i,r}$
  end for
end for
$\widehat{P} \leftarrow K/R$ ▷ cross-test pass rates
$S \leftarrow \min(\widehat{P}, \widehat{P}^\mathsf{T})$; $\text{diag}(S) \leftarrow 1$ ▷ pairwise test similarity
$e^* \leftarrow \min_{\pi \in \Delta_n} \pi^\mathsf{T} S \pi$
return $\widehat{\text{BTD}} = 1/e^*$

3.2 PROPERTIES OF BTD

Agent-conditioned behavioral diversity. The same specification may be interpreted differently by different implementers, whether human developers or coding agents. These interpretations affect how they translate test feedback into code changes and which other tests those changes satisfy. BTD therefore conditions on the implementer to capture the distinct behavioral guidance a suite provides to that implementer.

Diversity properties. BTD summarizes pairwise test similarity as an effective number of distinct behavioral constraints. For weights $\pi \in \Delta_n$, draw two test indices I and J independently from π. Their expected similarity is

$$\mathbb{E}_{I,J\sim\pi}[S_{IJ}] = \pi^{\mathsf{T}} S \pi = \sum_{i,j} \pi_i \pi_j S_{ij}. \tag{5}$$

When tests form m groups with complete mutual overlap within groups and no overlap across groups, this expectation is the probability that two sampled tests provide the same behavioral guidance. Let w_g be the total weight assigned to group g. Then

$$e^* = \min_{\pi\in\Delta_n} \pi^{\mathsf{T}} S \pi = \min_{w\in\Delta_m} \sum_{g=1}^{m} w_g^2 = \frac{1}{m}, \qquad \mathrm{BTD} = m. \tag{6}$$

Minimizing this probability assigns equal total weight $1/m$ to each group. Its reciprocal therefore counts distinct groups of behavioral guidance, regardless of how many tests each group contains. Thus, ten tests in one group and one in another yield BTD 2, just as one test from each group would. More generally, adding a test that fully overlaps with an existing test and has the same mutual overlap with every other test leaves BTD unchanged.

For partial overlap, the same optimization can produce fractional diversity scores without partitioning tests into predefined behavior groups. This aggregation follows established maximum similarity-sensitive diversity (Leinster & Meckes, 2016). Appendix A gives the derivations.

3.3 ALIGNMENT EFFICIENCY PERSPECTIVE

In practical software development, documentation and tests often serve as complementary specifications. With documentation held fixed, the number of tests needed to achieve a given level of alignment with human intent provides a measure of agent alignment efficiency. Under comparable generation and repair budgets, an agent that requires fewer tests uses the supplied specification more efficiently. BTD provides a proxy for the additional guidance an agent needs from tests as executable specifications to align its code with human intent. Given the same documentation and test suite, a lower BTD indicates that fewer distinct behavioral requirements remain unresolved in the agent's documentation-only implementation. This suggests less need for additional specification and thus higher alignment efficiency.

4 VALIDATING BEHAVIORAL TEST DIVERSITY

4.1 EXPERIMENTAL SETUP

We study the setting in which a coding agent implements software from documentation and then uses tests as executable specifications to refine its behavior. ProgramBench (Yang et al., 2026) provides real-world program-reconstruction tasks with documentation, executable references, and test suites.

We adapt these tasks to the measurement protocol in Section 3. A GLM-5.2 agent in a Codex harness (OpenAI, 2026) first generates a documentation-only implementation C_0, without access to the reference executable or benchmark tests. We then use the tests to guide repairs and measure how repairing code for one test transfers to others. This measures the diversity of behavioral guidance the suite supplies beyond the agent's initial interpretation of the documentation. We compute BTD on initially failing tests and retain initially passing tests as regression constraints.

4.2 Stability and Repeatability

BTD is measured through stochastic code generation and repair. We therefore examine score sensitivity to repair sampling and to the documentation-only initialization C_0, as well as whether suite rankings persist across initializations. To span the difficulty range, we select five tasks from the published difficulty-score groups at evenly spaced percentiles: yj (P10), lz4 (P30), bedtools2 (P50), trdsql (P70), and jsonschema (P90).

For repair-sample sensitivity, we fix one C_0 per task and construct a 100-test subset from its official suite. We use this common size to make repeated repair and cross-test evaluation tractable. Each subset samples 30 initial failures and 70 initially passing regression tests, keeping the numbers of repair targets and regression constraints constant across tasks.

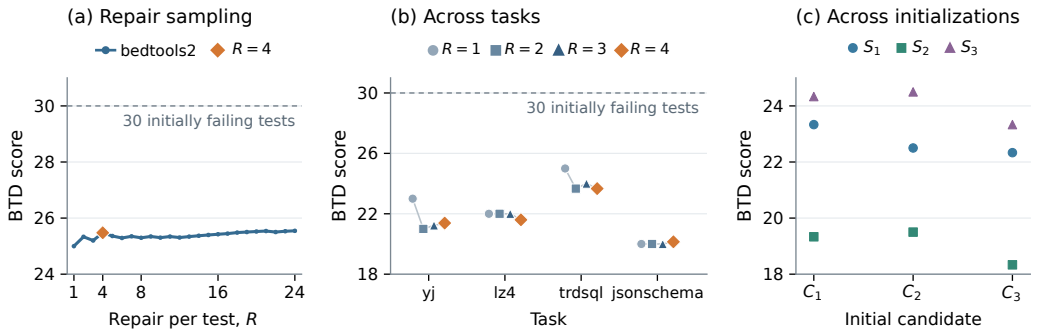

Figure 3: BTD stability across repair budgets (a–b) and initial implementations (c, $R = 2$).

BTD scores are stable and repeatable. With the initial implementation and test suite fixed, three independent batches of eight repairs per target yield BTD scores of 25.30, 25.60, and 25.89 on bedtools2. Their close agreement supports a repeatable diversity estimate despite the stochasticity of individual code repairs. Separately, pooling the bedtools2 batches and increasing R from 1 to 24 produces a score range spanning only 2.14% of the final estimate (Figure 3(a)). The four additional tasks show a similar pattern: although yj and trdsql exhibit larger changes from $R = 1$, increasing R from two to four changes every task's score by at most 1.86% of its $R = 4$ estimate (Figure 3(b)). The agreement between smaller- and larger-budget estimates shows that stable BTD measurements can be obtained with few repairs per target.

BTD-based test-suite rankings are stable across initial implementations. We evaluate three 100-test bedtools2 suites across three independently generated initial implementations at $R = 2$. Each suite's scores span at most 1.17 BTD units, while all three implementations yield the same ordering, $S_3 > S_1 > S_2$ (Figure 3(c)). The unchanged ordering shows that these initialization differences do not alter which suite BTD favors, supporting consistent comparisons of behavioral guidance. Appendix D provides the exact scores and protocol details.

4.3 Behavioral Redundancy in Test Suites

BTD reveals uneven behavioral redundancy in existing test suites. We reuse the sampled suites from Section 4.2 and compare them at a common repair budget of $R = 4$. With no measured overlap among the 30 initially failing tests, BTD would be 30. Figure 3(a–b) shows that all five tasks score below this 30-test reference at $R = 4$, indicating overlapping behavioral guidance. This overlap is uneven: groups of mutually overlapping tests coexist with tests having zero observed similarity to the others. These results show why test count is an incomplete proxy for behavioral diversity: it treats repeated checks and distinct behavioral guidance alike. Appendix D provides a concrete example of the tests and their observed overlap.

This uneven representation can make benchmark scores misleading indicators of correctness. When each test contributes equally, behaviors covered by many overlapping checks receive disproportionate weight. An implementation can therefore score highly by satisfying densely tested behaviors

while still violating sparsely tested, but still potentially important, requirements. BTD quantifies this imbalance. We next use it to guide suite construction toward distinct behavioral constraints.

5 Applications of BTD

5.1 Whole-Suite Optimization on ProgramBench

We use ProgramBench's `sd` as a case study because it exhibits both behavioral redundancy and gaps in the existing specification, as illustrated in Section 1. These two issues motivate complementary uses of BTD: pruning tests while preserving behavioral diversity, and expanding the suite to cover previously unchecked behaviors.

5.1.1 Test-Suite Pruning

Pruning uses BTD as a retention criterion to reduce the number of tests while preserving measured behavioral diversity. For a retention fraction $0 < \alpha < 1$, we seek a small subset $T' \subseteq T$ satisfying

$$\mathrm{BTD}(T'; S[T', T']) > \alpha \, \mathrm{BTD}(T; S). \tag{7}$$

BTD-guided pruning preserves behavioral diversity with fewer tests. The documentation-only implementation of `sd` initially fails 186 tests, revealing unmet behavioral requirements. With $\alpha = 0.95$, BTD-guided pruning yields 72 tests as additional executable specifications, reducing failing-test feedback by 61.29% while retaining 95.55% of the original empirical BTD. Using the documentation and the pruned test suite as specifications, the agent produces a candidate that passes all 810 official tests. This result shows how BTD can help construct a compact executable specification while retaining the guidance needed for full-suite recovery. Appendix D details the protocol.

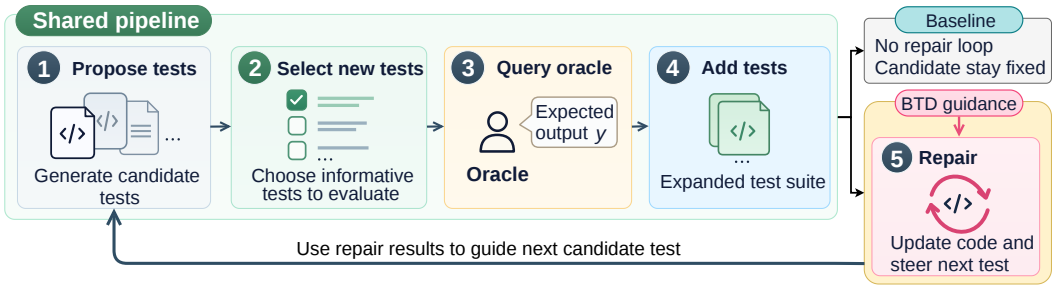

Figure 4: BTD-guided expansion and the no-repair-loop baseline.

5.1.2 Test Suite Expansion

Expansion aims to add tests that manifest previously unchecked behaviors. BTD motivates a repair-based construction principle: after an implementation satisfies the accumulated suite, seek a new test that still requires a behavioral correction. For `sd`, we start from an implementation passing all 810 official tests. Each accepted addition triggers a repair that preserves the accumulated suite, and subsequent proposals target failures in the updated implementation (Figure 4). This process directs expansion toward requirements that preceding repairs have not yet satisfied.

We first compare three proposal strategies within the BTD-guided repair-loop framework: mutation-guided generation with Reach/Kill scheduling, Flat, and Coverage. Flat proposes inputs from test/probe history, while Coverage searches for branch-coverage gains. We then compare the best-performing Reach/Kill configuration with a no-repair-loop batch baseline that keeps the implementation fixed. We use test-generation attempts as a proxy for human review burden, since proposed tests require checking that their expected behavior reflects human intent. Reach/Kill, Flat, and the no-repair-loop baseline each receive a budget of 24 attempts. Across three runs per condition, we evaluate the resulting suites through BTD gains measured after generation against a separate, fixed documentation-only implementation at $R = 4$.

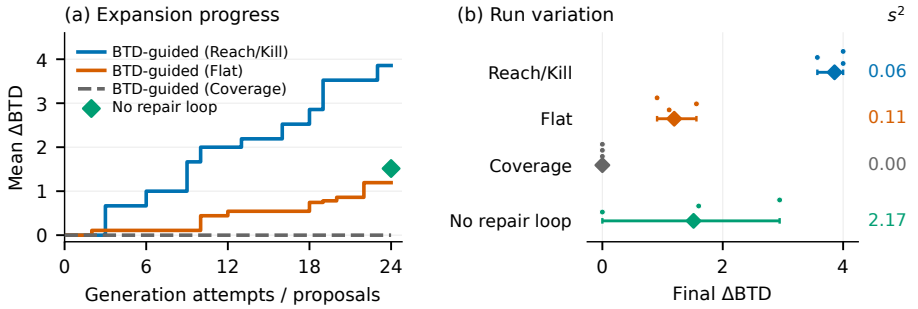

Figure 5: BTD gains from expanding the `sd` suite. (a) Mean progress. (b) Endpoints of three runs per method (circles), means (diamonds), min–max ranges, and sample variances s^2.

BTD-guided expansion manifests previously unchecked behaviors and increases behavioral diversity. Within the repair-loop framework, mutation-guided Reach/Kill achieves the highest mean BTD gain, yielding over three times Flat's mean gain; Coverage adds no tests in these runs (Figure 5). Under the same budget of 24 test-generation attempts, the Reach/Kill repair-loop method adds more distinct behavioral constraints than the no-repair-loop batch baseline. Repair-loop expansion also addresses the problem of unconstrained behavior highlighted in Section 1, adding explicit behavioral requirements absent from both the documentation and the original suite. The behavioral discoveries and BTD gains together show that the additions fill gaps in the existing specification and supply more diverse guidance for code repair. Appendix D.4 provides a concrete example and experimental details.

5.2 Test Suite Pruning in a Large Software Project: vLLM

Table 1: vLLM repair results.

	BTD	Coverage
Selected tests	7	7
Branch edges	78	84
BTD	7	4.3
Native (/81)	**81, 81, 81**	75, 75, 74
Regressions	0, 0, 0	0, 0, 0

We next evaluate whether BTD can guide test-suite pruning in a large, real-world software repository. Using vLLM (Kwon et al., 2023), we examine whether a compact suite preserves the behavioral guidance needed for full-suite recovery of its KV-cache `BlockPool`. GLM-5.2 + Codex reconstructs `block_pool.py` from documentation, interface skeletons, and surrounding production code, withholding the original implementation and evaluation tests. The resulting C_0 passes 53 of 81 native tests; we profile the remaining 28 with $R = 2$ singleton repairs.

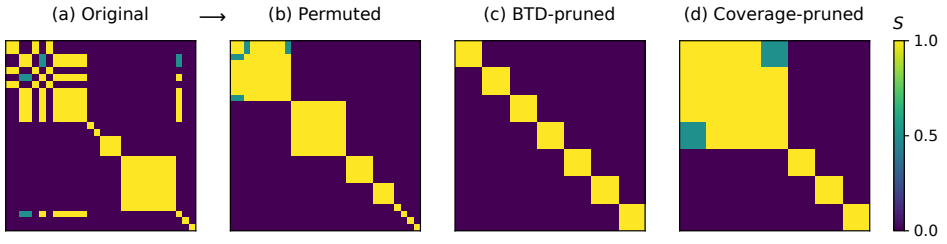

Figure 6: Test similarity S in vLLM ($R = 2$). (a–b) The same 28-test matrix before and after reordering. (c–d) Seven-test subsets selected by BTD and coverage. All panels share the color scale.

BTD-guided pruning remains effective in a large real-world software repository. We progressively increase the test-selection budget, seeking a compact suite that supports full-suite recovery across fresh repairs. At a budget of seven tests, the BTD-guided suite achieves full-suite recovery in all three fresh repairs, while the paired coverage-optimal suite leaves failures unresolved despite covering more branch edges (Table 1). Figure 6 shows the difference in feedback: BTD selects tests with zero observed pairwise similarity, whereas coverage includes a high-similarity group.

Coverage selects four tests from the same high-similarity group but omits `test_prefill_plp`, which BTD retains. This test requires freed uncached blocks to precede cached blocks in the free queue. The coverage-selected suite does not enforce this ordering, and all three resulting candidates violate it. These results extend the evidence from `sd` to a large real-world repository: on vLLM's KV-cache component, BTD-guided pruning retains behavioral requirements missed by the coverage-selected suite and provides compact feedback that supports full-suite recovery. Appendix C details the repair setup, matrix visualization, and omitted-test outcomes.

5.3 Applying BTD Across Domains: ComputeEval

We test BTD's applicability across software domains on GPU kernels from ComputeEval (NVIDIA, 2026). We select CUDA/104, CUDA/197, and LayerNorm because their documentation-only candidates fail some official tests, allowing tests to supply additional executable specifications. Across three candidates, we profile these failures with $R = 2$ repairs per target. Throughout repair, the documentation is fixed and initially passing tests remain regression constraints.

BTD generalizes across software domains. Table 2 shows that the selected sets retain the full measured BTD and support full-suite recovery in fresh repairs. Successful recovery of the unprovided tests indicates that the retained feedback carries behavioral guidance shared with the omitted checks. These results show that BTD supports test-suite reduction across domains while preserving full-suite recovery, without needing domain-specific behavior categories. Appendix F provides the controls and experimental details.

Table 2: BTD can be applied to different software domains such as gpu kernel generation.

Task	BTD	Selected test count	BTD retained	Full-suite recovery
CUDA/104	1	1	100%	30/30
CUDA/197	1	1	100%	3/3
LayerNorm	1.3	2	100%	3/3

Table 3: Using BTD to characterize agents alignment efficiency.

Agent configuration	C_0 passes out of 100	Initial failures n	BTD $R = 2$
GLM-5.2 + Codex	70	30	25.3
GPT-5.5 (high) + Codex	55	45	28.3
Opus 4.6 (high) + Claude	64	36	26.0

5.4 Characterizing Agent-Specific Alignment Efficiency

As a complementary application, we use BTD to characterize agent alignment efficiency through the behavioral requirements left unresolved by an agent's documentation-only implementation (Section 3.3). On the same bedtools2 documentation and suite, each agent generates its own C_0 and supplies $R = 2$ fresh-context repairs per failing target, preserving its own regression set G. Test satisfaction serves as an operational proxy for alignment with human intent.

BTD provides a measure of agent alignment efficiency. Although test satisfaction and BTD give the same ordering of agent configurations in this comparison, the relative gaps differ. GPT-5.5 (high) + Codex has 50% more initial failures than GLM-5.2 + Codex, but only about 12% higher BTD (Table 3). More failing tests therefore need not imply proportionally more distinct unmet behavioral requirements. These results show how BTD complements pass rates as a proxy for agent alignment efficiency by quantifying how much additional guidance each agent needs to satisfy a shared specification. Appendix D.5 details the configurations and measurement protocol.

6 Conclusion

We introduced Behavioral Test Diversity (BTD), an agent-conditioned metric that uses code repair to measure the diversity of code behaviors specified by a test suite. We demonstrated the stability of BTD scores and the consistency of BTD-based test-suite rankings. By making this diversity an explicit optimization objective, BTD guides pruning to preserve behavioral coverage with fewer tests and expansion to manifest previously unchecked behaviors. Our experiments support these uses and BTD's applicability across software domains without needing predefined behavior categories. BTD thus helps construct test suites that provide effective behavioral guidance for aligning generated code with human intent.

AI USE STATEMENT

In this work, generative AI tools were used for text editing, figure refinement, and implementing methods designed by the authors. The authors developed the conceptual framework, methodology, and experimental design, and interpreted the results. AI-assisted writing and implementation were reviewed by the authors. The authors take full responsibility for the paper's content, claims, and accompanying artifacts.

REPRODUCIBILITY STATEMENT

The code we use will be released upon publication of the paper. The main text and appendices describe the benchmarks, test-suite construction, agent configurations, repair and evaluation protocols, sampling budgets, and BTD optimization procedure. The release will additionally provide prompts, configuration files, recorded repair outcomes, and scripts for recomputing scores and regenerating the figures and tables. These materials will support independent verification of the reported measurements and reproduction of the experimental workflow and experiment results.

REFERENCES

Juan Altmayer Pizzorno and Emery D. Berger. CoverUp: Effective high coverage test generation for python. *Proceedings of the ACM on Software Engineering*, 2(FSE):2897–2919, 2025. doi: 10.1145/3729398. URL https://arxiv.org/abs/2403.16218.

Anthropic. Bringing Code Review to Claude Code. Anthropic blog, March 2026. URL https://claude.com/blog/code-review/. March 9. Accessed 2026-09-26.

Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, pp. 209–224, 2008. URL https://www.usenix.org/legacy/events/osdi08/tech/full_papers/cadar/cadar.pdf.

Tsong Yueh Chen, Fei-Ching Kuo, Robert G. Merkel, and T. H. Tse. Adaptive random testing: The ART of test case diversity. *Journal of Systems and Software*, 83(1):60–66, 2010. doi: 10.1016/j.jss.2009.02.022.

Gordon Fraser and Andrea Arcuri. EvoSuite: Automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, pp. 416–419, 2011. doi: 10.1145/2025113.2025179.

Sabrina Haque, Sarvesh Ingale, and Christoph Csallner. An empirical study of tests in agentic pull requests. In *Proceedings of the 23rd International Conference on Mining Software Repositories*, pp. 929–933, 2026. doi: 10.1145/3793302.3793604. URL https://c.csallner.org/papers/Haque26Empirical.pdf.

Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering*, pp. 435–445, 2014. doi: 10.1145/2568225.2568271.

Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2011. doi: 10.1109/TSE.2010.62.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pp. 611–626, 2023. doi: 10.1145/3600006.3613165. URL https://arxiv.org/abs/2309.06180.

Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. GenProg: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38(1):54–72, 2012. doi: 10.1109/TSE.2011.104.

Tom Leinster and Mark Meckes. Maximizing diversity in biology and beyond. *Entropy*, 18(3):88, March 2016. ISSN 1099-4300. doi: 10.3390/e18030088. URL http://dx.doi.org/10.3390/e18030088.

Arghavan Moradi Dakhel, Amin Nikanjam, Vahid Majdinasab, Foutse Khomh, and Michel C. Desmarais. Effective test generation using pre-trained large language models and mutation testing, 2023. URL https://arxiv.org/abs/2308.16557.

NVIDIA. ComputeEval. GitHub repository, 2026. URL https://github.com/NVIDIA/compute-eval. Accessed 2026-08-26.

OpenAI. Codex CLI. GitHub repository, 2026. URL https://github.com/openai/codex. Accessed 2026-09-25.

Partha Protim Paul and Reid Holmes. Beyond coverage and kill scores: Empirically measuring test suite behavioural gaps, 2026. URL https://arxiv.org/abs/2606.10417.

Edward K. Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pp. 532–543, 2015. doi: 10.1145/2786805.2786825.

Guancheng Wang, Qinghua Xu, Lionel C. Briand, and Kui Liu. Mutation-guided unit test generation with a large language model. *IEEE Transactions on Software Engineering*, 52(5):1657–1671, 2026. doi: 10.1109/TSE.2026.3682975. URL https://arxiv.org/abs/2506.02954.

John Yang, Kilian Lieret, Jeffrey Ma, Parth Thakkar, Dmitrii Pedchenko, Sten Sootla, Emily McMilin, Pengcheng Yin, Rui Hou, Gabriel Synnaeve, Diyi Yang, and Ofir Press. ProgramBench: Can language models rebuild programs from scratch?, 2026. URL https://arxiv.org/abs/2605.03546.

Shin Yoo and Mark Harman. Regression testing minimization, selection and prioritization: A survey. *Software Testing, Verification and Reliability*, 22(2):67–120, 2012. doi: 10.1002/stvr.430.

Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, 1997. doi: 10.1145/267580.267590.

A Fixed-Matrix Properties of BTD

We derive the properties used in Section 3.2 for a fixed, nonempty similarity matrix S with entries in $[0, 1]$ and unit diagonal.

Since S is nonnegative and has unit diagonal,

$$\pi^{\mathsf{T}} S \pi \geq \sum_i \pi_i^2 \geq 1/n.$$

A simplex vertex has objective one, hence $1/n \leq e^* \leq 1$. For $S = \mathbf{1}\mathbf{1}^{\mathsf{T}}$, every feasible objective equals one. For $S = I$, uniform weights attain $1/n$.

For k ideal all-ones blocks, let m_a be the total weight in block a. Then $\pi^{\mathsf{T}} S \pi = \sum_{a=1}^{k} m_a^2$, whose minimum is $1/k$. Exact duplicate coordinates can be merged or split without changing the objective. When a matrix is extended while preserving its original principal submatrix, extending an old optimizer with zero weights is feasible and keeps its objective. Therefore the new minimum cannot increase and BTD cannot decrease.

B Complete Measurement Protocol and Solver

In Algorithm 2, $\mathcal{V}$ fixes the repeated-execution and acceptance rules; evaluations return complete binary pass indicators. All collection calls are bounded by the shared deadline D and return $\perp$ if unresolved; an incomplete measurement yields no BTD.

Algorithm 2 BTD measurement and solver certification

Require: Frozen d, C_0, T_{all}; $R \in \mathbb{N}_{>0}$ preassigned independent trajectories per target
Require: Repair protocol $\mathcal{A}$, validation policy $\mathcal{V}$, collection deadline D
Require: Solver budget B_s and certification tolerance ϵ
$o_0 \leftarrow \text{Evaluate}(C_0, T_{\text{all}}; \mathcal{V}, D)$
If $o_0 = \perp$, **return** $\perp$
$T \leftarrow \{t : o_0(t) = 0\}$; $G \leftarrow \{t : o_0(t) = 1\}$; $n \leftarrow |T|$
If $n = 0$, **return** BTD undefined
$K \leftarrow 0_{n \times n}$
for $(i, r) \in \{1, \ldots, n\} \times \{1, \ldots, R\}$ **do**
    Initialize session h once from $\text{copy}(C_0)$ with fresh context
    $C \leftarrow \mathcal{A}.\text{RepairUntilAccepted}(h, d, t_i, G; \mathcal{V}, D)$
    Resume h; expose only t_i, G as test feedback; accept only when all pass under $\mathcal{V}$.
    If $C = \perp$, **return** $\perp$
    Freeze C; $y \leftarrow \text{BlindEvaluate}(C, T; \mathcal{V}, D)$
    Retries retain C and evaluation history; no outcomes return to $\mathcal{A}$.
    If $y = \perp$ or $y_i \neq 1$, **return** $\perp$
    $K_{i,:} \leftarrow K_{i,:} + y$
end for
$\widehat{P} \leftarrow K/R$; $S \leftarrow \min(\widehat{P}, \widehat{P}^{\mathsf{T}})$ elementwise
$(L, U, \pi, s_s) \leftarrow \text{SolveBTD}(S; B_s, \epsilon)$
If no valid bounds $0 < L \leq U$ are available, **return** $\perp$
$I_{\text{BTD}} \leftarrow [1/U, 1/L]$
If s_s certifies tolerance ϵ, **return** $\widehat{\text{BTD}} = 1/U$, I_{BTD}, and π
return I_{BTD} and solver gap

MILP and solver reporting. The quadratic minimum in Equation 4 admits the following mixed-integer linear formulation:

$$\min_{\pi, z, e} \quad e \tag{8}$$

$$\text{subject to} \quad \pi_i \geq 0, \quad z_i \in \{0, 1\}, \quad \sum_i \pi_i = 1, \quad 1/n \leq e \leq 1, \tag{9}$$

$$\pi_i \leq z_i, \qquad 0 \leq (S\pi)_i - e \leq 1 - z_i \quad \text{for all } i. \tag{10}$$

Every feasible solution has $(S\pi)_i = e$ on the positive support of π, hence $\pi^\mathsf{T} S\pi = e$. Conversely, a global simplex minimizer satisfies $(S\pi)_i = e^*$ on its positive support and $(S\pi)_i \geq e^*$ elsewhere. It is therefore representable by these constraints; the upper bound is valid because entries of S lie in $[0, 1]$. Minimizing over the encoded stationary points recovers the global objective without assuming positive semidefiniteness.

Return a certified score only with the stated solver tolerance. If valid lower and upper objective bounds are $0 < L \leq e^* \leq U$, report the BTD interval $[1/U, 1/L]$ and the solver gap. A time-limited incumbent is not an exact score.

C vLLM Repair and Evaluation Details

C.1 Native Suite and Repair Setup

Test counts refer to collected pytest nodes, including parameterized instances. The study uses the complete 81-node native prefix-caching suite at the pinned revision. Only `block_pool.py` is reconstructed; production dependencies and native assertions remain fixed. Selected-test projections preserve the required fixtures and helpers and are checked against native execution on the reference and C_0. Reference code is available to the trusted coverage measurement, but is withheld from repair agents along with unselected tests and Git history. The model is OpenRouter `z-ai/glm-5.2`, restricted to Z.AI FP8 with no provider fallback. Repairs continue in the same session until the selected tests pass.

C.2 Fresh Replication

The seven-test subsets in Table 1 are frozen before three independent repairs per subset. All replications preserve the 53 initially passing tests.

C.3 Permuting the Test Similarity Matrix

Figure 6 reorders the matrix for visualization. Tests are connected when $S_{ij} > 0$; components are ordered by decreasing size, with ties broken by their smallest original index, and indices within each component remain ascending. Applying the same permutation to both axes preserves all entries, selected-set membership, and BTD.

The component sizes are $(9, 8, 4, 3, 1, 1, 1, 1)$. BTD selects from seven distinct components, whereas four coverage-selected tests lie in the nine-test component. Each non-singleton component is a clique in this $R = 2$ matrix.

C.4 Omitted-Test Analysis

The common omitted set is $H = F \setminus (A_{\mathrm{BTD}} \cup A_{\mathrm{coverage}})$, where F contains the 28 initial failures. The selected sets overlap on two tests, leaving 16 nodes in H. BTD repairs pass 16/16 of these nodes in every replication; coverage repairs pass 12/16, 12/16, and 11/16. These outcome vectors are verified against both existing terminal evaluation receipts for each repair.

D Additional Experimental Details

This appendix records the measurement settings and downstream evaluation protocols behind Sections 4, 5.1.1, 5.1.2, and 5.4. The settings and measurement records below support the reported results.

D.1 Repeated Calibration on ProgramBench

The calibration fixes ProgramBench instance `arq5x__bedtools2.dd57059`, a documentation-only C_0, and one 100-test suite. Thirty tests fail initially; the other 70 are fixed regression constraints. Each of three independent batches contains eight fresh-context trajectories per target.

Smaller R values reuse the first R trajectories within their batch. All 720 trajectories reached target-plus-regression acceptance and completed two blind replays on the 30 targets.

Repair uses GLM-5.2 through OpenRouter with only Z.AI FP8 enabled. Interruptions resume the same trajectory. All 12 batch-prefix calibration scores reached the declared MILP optimality tolerance of 10^{-8} relative gap.

Across batches, BTD means and sample standard deviations are 25.6667 ± 0.5774 at $R = 1$, 25.6508 ± 0.4321 at $R = 4$, and 25.5961 ± 0.2968 at $R = 8$. Figure 3(a) instead concatenates trials in the frozen batch-major order and recomputes every cumulative prefix through $R = 24$. Its $R = 8, 16, 24$ values are 25.2974, 25.4236, and 25.5459. Pooling precedes symmetrization and optimization: the last value is computed from all 720 repairs. Only seven of the 435 distinct off-diagonal pairs have positive pairwise test similarity. The pooled score exceeds the matrix's 25 connected components because fractional similarities allow partially distinct feedback within a component.

Figure 7 reports all three triplets selected by source review before repair outcomes were observed. Rows identify repaired targets and columns identify evaluated tests. The frozen data snapshot records manifests, input hashes, anonymized test indices, outcome vectors, and the source commit.

A concrete example of behavioral overlap. Tests `t0006` and `t0007` both call `bed12tobed6 -i FILE` to split a genomic interval into its component blocks. The first input contains two blocks and the second three. Both require one BED6 output row per block, with the correct coordinates and the original chromosome, name, score, and strand. The inputs and expected outputs are shown below, with spaces representing tabs. Each test compares stdout after stripping surrounding whitespace and requires exit status 0.

```
t0006 input (two_blocks.bed):
chr1 0 50 two_blocks_match 0 + 0 0 0 2 10,10, 0,40,
Expected output:
chr1 0 10 two_blocks_match 0 +
chr1 40 50 two_blocks_match 0 +

t0007 input (three_blocks.bed):
chr1 0 50 three_blocks_match 0 + 0 0 0 3 10,10,10, 0,20,40,
Expected output:
chr1 0 10 three_blocks_match 0 +
chr1 20 30 three_blocks_match 0 +
chr1 40 50 three_blocks_match 0 +
```

In the first batch's $R = 4$ prefix used in Section 4.3, all four accepted repairs targeting either test also pass the other. Thus, the cross-test pass rates are $\widehat{P}_{6,7} = \widehat{P}_{7,6} = 1$, giving pairwise test similarity $S_{6,7} = 1$. These distinct inputs provide fully overlapping guidance in the observed repairs.

D.2 Cross-Task and Ranking Validation

The completed four-task extension measures yj, lz4, trdsql, and jsonschema. Each fixes one documentation-only candidate, 30 failing targets, 70 regression tests, and one $R = 4$ bank. All 480 trajectories and their native acceptance and blind-evaluation records are complete and audited. Each suite includes nine source-reviewed anchors with related local checks, 21 further stratified failures, and 70 stratified passes. Figure 3(b) uses original replicate indices $0, \ldots, R-1$ for $R = 1, 2, 3, 4$, pooling P before constructing S and scoring each prefix. All 16 scores meet the declared optimization tolerance. The frozen scores, matrices, source hashes, and figure checks are released with the plotting data.

The $R = 4$ points in Figure 3(a–b) compare these four scores with the first bedtools2 batch's $R = 4$ prefix, whose BTD is 25.4762; all fall below the dashed reference at 30 initially failing tests. In task order (bedtools2, yj, lz4, trdsql, jsonschema), the numbers of distinct test pairs with positive pairwise test similarity are 7, 19, 42, 12, and 46, respectively. These pairs involve 8, 15, 16, 10, and 12 of each suite's 30 targets; all remaining targets have zero observed similarity to other targets. Thus, each matrix combines localized overlap with isolated targets.

The ranking comparison crosses the earlier bedtools2 candidate with two later documentation-only candidates and three 100-test suites. Every candidate has 28, 30, and 32 failures in the three respec-

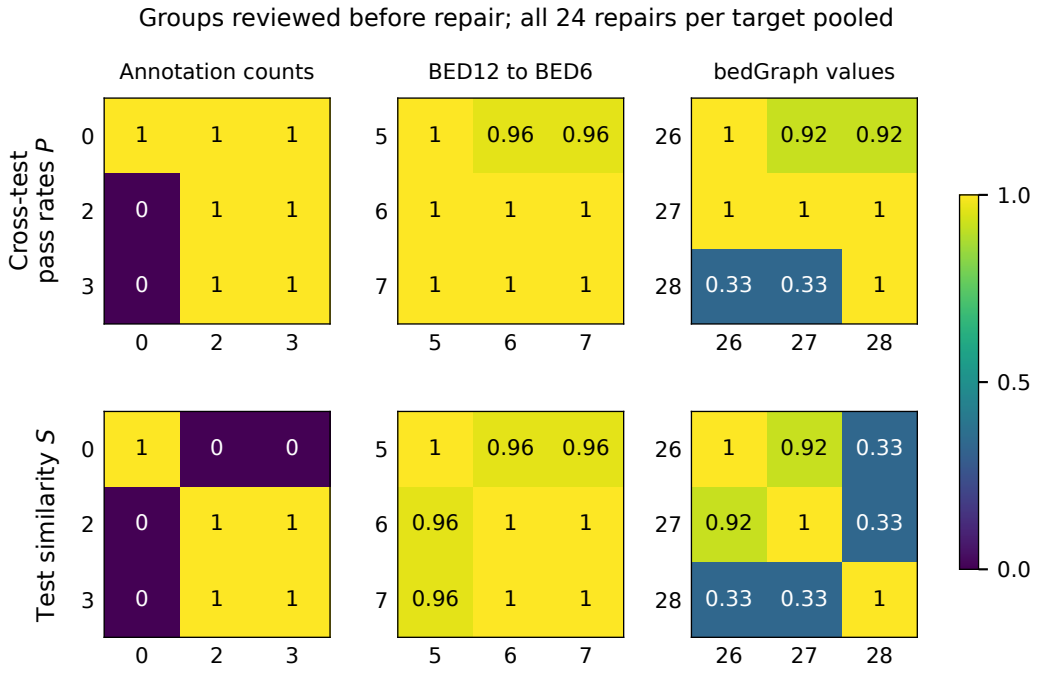

Figure 7: Cross-test pass rates P and test similarities S within all three local-similarity triplets selected before repair. Values pool 24 observations per target. Annotation counts show one-way repair transfer, BED12-to-BED6 has pairwise similarities close to 1, and bedGraph-value tests have asymmetric cross-test pass rates. Indices refer to the original 30-target order, retained in the figure data.

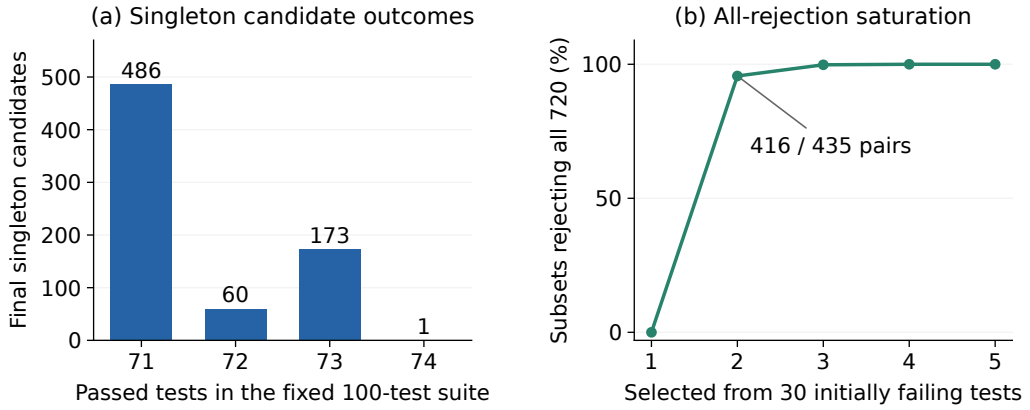

Figure 8: Descriptive audit of the existing singleton candidate pool. (a) Final candidates pass 71–74 of the fixed 100 tests. (b) Exact proportions of size-k subsets of the 30 initially failing tests that reject all 720 candidates; 416/435 pairs already suffice. The 70 regression tests remain fixed and pass for every candidate.

tive suites. The 40-target union requires 240 unique prefixes across three candidates and $R = 2$, supplying 540 suite-specific sample positions. Final acceptance checks the target and regression tests.

Table 4: BTD scores and test-suite rankings across three initial implementations of bedtools2. Each suite has 100 tests; $R = 2$.

	BTD score			
Initial implementation	Test suite 1 (S_1)	Test suite 2 (S_2)	Test suite 3 (S_3)	Test suite ranking
Candidate 1	23.33	19.33	24.33	$S_3 > S_1 > S_2$
Candidate 2	22.50	19.50	24.50	$S_3 > S_1 > S_2$
Candidate 3	22.33	18.33	23.33	$S_3 > S_1 > S_2$

Initial failures are 28, 30, and 32 for S_1, S_2, and S_3, respectively, for every candidate. All 540 suite-specific result slots are complete and audited.

Figure 3(c) and Table 4 report the completed snapshot. Candidate 1, 2, and 3 correspond to the earlier C_0 and later C_1 and C_2, respectively. The snapshot contains all 540 suite-specific result slots. All nine scores were reconstructed from native acceptance and blind-evaluation records and certified in two index orders. Every candidate has $S_3 > S_1 > S_2$, with no ties or reversals and pairwise Kendall's $\tau_b = 1$.

D.3 Pruning and Full-Suite Validation

Figure 8 enumerates subsets using calibration outputs as the candidate bank and describes their rejection rates.

The full-task pruning study uses the complete historical `sd` matrix with a 95% BTD-retention threshold. The final selected failing-test feedback contains 72 tests with BTD 68, compared with 71.1658 for all 186 failures, retaining 95.55% of the score. Joint repair retains all 624 initially passing tests as regression constraints.

The final 72-test set reduces failing-test feedback from 186 by 61.29%. Cumulative feedback contains 696 checks, and the final candidate passes all 810 official tests, including all 114 unprovided tests. Independent optimization of the 63 blocks with exactly zero between-block similarity certifies BTD 68.

D.4 Suite Expansion and Oracle Evaluation

Figure 9 illustrates the Reach/Kill variant's proposal and repair loop.

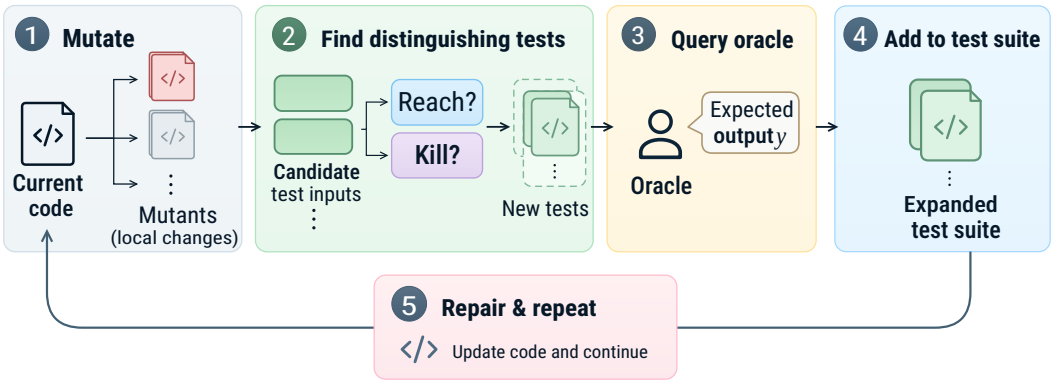

Figure 9: Reach/Kill-guided test-suite expansion. Mutants of the current implementation guide the search for distinguishing inputs; the oracle supplies expected outputs, and accepted tests trigger repair while preserving the accumulated suite. The repaired implementation seeds the next iteration.

The `sd` expansion comparison reports four conditions: BTD-guided (Reach/Kill), BTD-guided (Flat), a no-repair-loop batch control, and BTD-guided (Coverage). The three BTD-guided variants share the repair-based framework. All start from the same C_{seed} passing all 810 official tests,

with three generation runs per condition. The Reach/Kill variant uses Reach/Kill scheduling; the Flat variant exposes raw test/probe history to its proposal generator. Both methods accept oracle-divergent inputs after repair and replay of accumulated constraints.

Table 5: Four conditions for expanding the 810-test `sd` suite. New tests and BTD gains are run means. Baseline BTD is 71.1658.

Method	Runs	New tests	ΔBTD
BTD-guided (Reach/Kill)	3	4.00	3.857
No repair loop	3	24.00	1.515
BTD-guided (Flat)	3	2.67	1.194
BTD-guided (Coverage)	3	0.00	0.000

Reach/Kill, Flat, and Coverage are BTD-guided variants sharing the repair loop. Reach/Kill and Flat use 24 charged generation attempts. The no-loop control labels all 24 batch proposals; its three BTD gains are 2.945, 1.600, and 0. All methods use the same measurement C_0 and $R = 4$ scoring setup.

Specification-gap example. The multi-file preview example is the first accepted test in Reach/Kill run 2. The frozen documentation describes preview output without prescribing the path spelling in file headers. Before this addition, the candidate passes all 810 official tests but displays absolute paths for the supplied relative filenames; the reference preserves those filenames. The generated test supplies this missing output constraint. Its accepted repair passes the new test and accumulated feedback, with 810/810 on the original suite. This demonstrates resolution of the preview-path omission.

D.5 Cross-Agent Measurements

The completed comparison fixes the same bedtools2 instance, documentation, and native tests as the calibration study. Each agent generates its own documentation-only C_0 using public build/deployment feedback, freezes the first deployable implementation before native testing, and repairs its own naturally arising failure set. Table 3 reports the resulting pass/fail counts.

GLM-5.2 reuses the calibration study's first-batch $R = 2$ prefix through Codex and OpenRouter/Z.AI FP8. GPT-5.5 uses Codex with high reasoning effort and a ChatGPT subscription; Opus 4.6 uses Claude Code with high effort and a Claude subscription. Generation and repair use the same configuration within each agent. The new clients expose only a tool bound to an isolated, network-disabled candidate container; tests, gold artifacts, host state, and credentials are withheld except for the target feedback and fixed G allowed during repair.

Paired $R = 1$ sensitivity. After $R = 2$ collection, we extract the original replicate-0 trajectory for every target, without choosing by completion time or outcome. This gives primary $R = 1$ scores of 25, 30, and 26 for GLM, GPT, and Opus, respectively. A separate replicate-1 diagnostic gives BTD 26, 28, and 26 for GLM, GPT, and Opus. The pooled $R = 2$ scores are computed by combining the cross-test pass-rate matrices before symmetrization and optimization. The GLM replicate-0 matrix also matches its historical $R = 1$ result. All six binary $R = 1$ scores are cross-checked with an independent maximum-independent-set formulation; the $R = 2$ matrices are reconstructed from native receipts and match the original published matrices. Every score meets the declared optimality tolerance.

E Full Task Test Suite Study

E.1 Scope and Collection Protocol

As a full-task complement to the curated 100-test study, we examine the complete official suite for ProgramBench's `sd` task (instance `chmln__sd.87d1ba5`). A frozen doc-only C_0 passes 624 of 810 tests and fails 186. We target *all* 186 failures, without selecting a smaller subsuite, and obtain four successful terminal programs for each target: 744 programs and complete 186×186 matrices.

E.2 Complete Matrices and Clustered Structure

Each terminal program passes its own target twice before being frozen and replayed twice on all 186 targets. Repair receives only its designated target's feedback, not the blind cross-test outcomes. All 1,488 cross-test vectors were verified against the saved program and evaluation artifacts; the two replays agree for every program. We compute P_{ij} as the fraction of the four accepted programs for target i that also pass test j, and $S_{ij} = \min(P_{ij}, P_{ji})$. Neither matrix has missing entries, and diagonal ones follow from observed target passes. An independent reconstruction agrees exactly with the saved matrices and CSV.

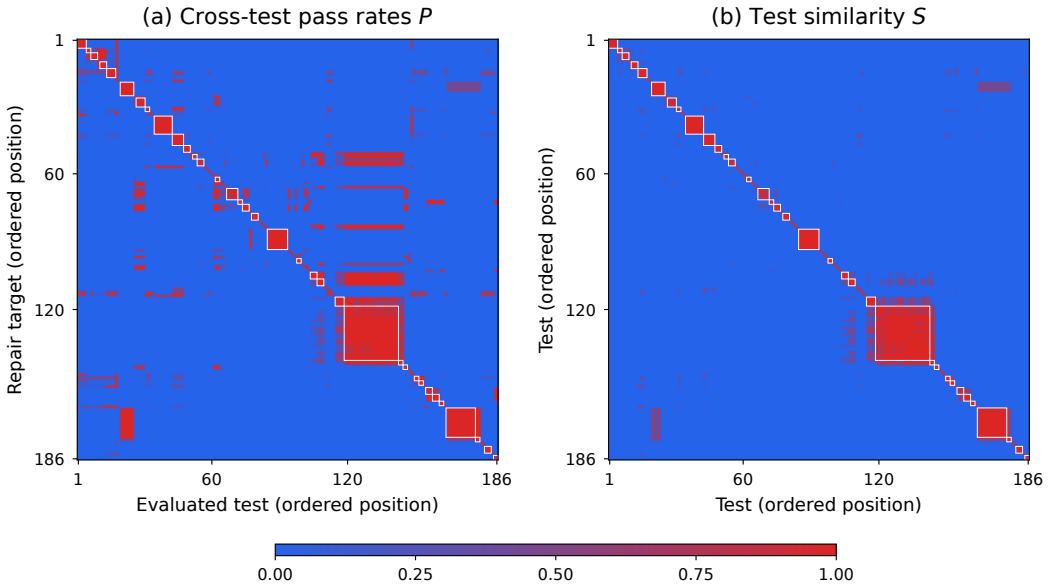

Figure 10: Full-task study of `sd`: all 186 initially failing tests from an 810-test official suite, with four accepted programs per target. (a) Cross-test pass rates P. (b) Test similarities $S = \min(P, P^{\mathsf{T}})$. Both panels use the same complete-linkage leaf order and blue-to-red scale from 0 to 1. White boxes mark multi-test groups whose every pair has observed similarity at least 0.5. Axis values are reordered positions; the original anonymous test IDs are retained in the supplied mapping. Values are unchanged by reordering.

Figure 10 applies complete-linkage hierarchical clustering to the precomputed dissimilarity $1 - S$, followed by optimal leaf ordering. The same permutation is used on both axes and in both panels; every measured value is retained. Complete linkage gives an explicit pairwise meaning to a cut: within each displayed group, every pair satisfies $S_{ij} \geq \tau$. We use $\tau = 0.5$ for the display and report the group counts at each cutoff in Table 6.

The hierarchy contains both large groups of similar tests and many singleton groups (Table 6). The largest displayed group has minimum pairwise similarity 0.75 and mean 0.9547; the next three largest groups have pairwise similarity of 1 in these observations. The pass-rate panel also contains positive entries between groups where the similarity panel has zeros. Thus, the full failure pool exhibits both high-similarity blocks and one-way repair effects.

E.3 Pruning Results and Data

The subsequent pruning study reuses this matrix and retains all 624 regressions during joint repair. Its final candidate, using 72 selected failing tests as feedback, reaches 810/810 (Appendix D). Matrix mappings, clustering thresholds, and audit hashes are retained with the plotting data.

Table 6: Descriptive threshold sensitivity for the same full-task hierarchy. The threshold τ bounds every within-group pair's observed S.

τ	All groups	Multi-test groups	Singletons	Largest group
0.25	65	32	33	26
0.50	76	35	41	24
0.75	84	32	52	24
1.00	92	30	62	13

E.4 Concrete Tests Behind the Introductory Example

Repeated preview requirements. The 12 tests below all assert successful execution, the expected replacement output, and unchanged input-file contents in preview mode. They vary the input text, pattern, replacement, or options, while repeatedly checking the shared requirement that previewing a replacement must not modify the file. These are 12 concrete examples from the official suite. Test IDs match the full-task matrix; each identifier includes its benchmark branch and original test node. In the descriptions, `FILE` denotes the temporary input file, `\n` denotes a newline, and quoted strings preserve trailing spaces. Every listed test checks that the file still contains its original input after the command.

t003. `08a92821049f/tests.test_basic_operations.test_preview_mode_no_modification`.
Command: `sd -p hello goodbye FILE`. Input `"hello world\n"`; expected stdout `"goodbye world\n"`.

t047. `08a92821049f/tests.test_harvest.test_limit_replacements_file_preview`.
Command: `sd --preview -n 1 foo bar FILE`. Input `"foo\nfoo\nfoo"`; expected stdout `"bar\nfoo\nfoo"`.

t049. `08a92821049f/tests.test_harvest.test_replace_into_stdout`.
Command: `sd -p 'abc\d+' '' FILE`. Input `"abc123def"`; expected stdout `"def"`.

t053. `08a92821049f/tests.test_io_modes.test_preview_single_file_no_modification`.
Command: `sd -p hello goodbye FILE`.
Input `"hello world\nfoo bar\nhello again\n"`; expected stdout `"goodbye world\nfoo bar\ngoodbye again\n"`, read from the test's golden-output fixture.

t094. `3bf67c4734d7/eval.tests.test_file_operations.test_file_preview_mode`.
Command: `sd -p foo baz FILE`. Input `"foo bar"`; expected stdout `"baz bar"`.

t124. `55ccb3ccb510/eval.tests.test_preview.test_preview_empty_replacement`.
Command: `sd -p world '' FILE`. Input `"hello world"`; expected stdout `"hello "`.

t125. `55ccb3ccb510/eval.tests.test_preview.test_preview_long_flag`.
Command: `sd --preview world universe FILE`. Input `"hello world"`; expected stdout `"hello universe"`.

t128. `55ccb3ccb510/eval.tests.test_preview.test_preview_short_flag`.
Command: `sd -p world universe FILE`. Input `"hello world"`; expected stdout `"hello universe"`.

t130. `55ccb3ccb510/eval.tests.test_preview.test_preview_with_regex`.
Command: `sd -p '\d+' X FILE`. Input `"abc123def456"`; expected stdout `"abcXdefX"`.

t154. `de0de3e16642/eval.tests.test_sd_io.test_file_preview_does_not_modify_and_prints_to_stdout`.
Command: `sd -p foo bar FILE`. Input `"foo\n"`; expected stdout `"bar\n"`. This test also requires empty stderr.

t164. `e7da309aed82/eval.tests.test_cli.test_replace_into_stdout`.
Command: `sd -p 'abc\d+' '' FILE`. Input `"abc123def"`; expected stdout `"def"`.

t167. ec79d4d365e9/eval.tests.test_sd_behavior.test_preview_prints_result_and_does_not_modify_file.

Command: `sd -p foo baz FILE`. Input `"foo\nbar\nfoo\n"`; expected stdout `"baz\nbar\nbaz\n"`.

Observed repair overlap. These 12 tests belong to the same 13-test group in the full-task matrix. The remaining member, `t162`, is `e7da309aed82/eval.tests.test_cli.test_limit_replacements_file_preview`: it uses the same command, input, and stdout expectation as `t047`, but does not assert that the file remains unchanged, so it is excluded from the count of 12. All 52 accepted programs obtained by targeting these 13 tests (four per target) pass every test in the group, giving $P_{ij} = S_{ij} = 1$ within the group. The source-inspected test functions, assertions, input/output values, archive hashes, and matrix mapping are frozen in `figure/data/sd_preview_overlap.json`. The test archives are from ProgramBench-Tests revision `de0ddfb637590c7ecb54fa0b5301f6dc7dfbcee5`.

Unchecked error precedence. The contrasting example in the introduction uses a separate candidate generated from the provided documentation and tests, which passes all 810 official tests. On `sd '(' '$1abc'`, the search pattern is invalid and the replacement string contains an ambiguous capture reference. The candidate reports the invalid search pattern; the reference reports the ambiguous replacement. Both exit with status 1. The supplied documentation does not specify which error should take precedence, and reversing the candidate's checking order also leaves all 810 tests passing. Thus, repeated checks of preview behavior coexist with an unconstrained choice in error reporting. This historical diagnostic is separate from the doc-only candidate used for the repair matrix.

F ComputeEval Protocol and Repair Evaluation

The ComputeEval study tests whether compact feedback remains useful when moving from command-line programs to GPU kernels. This appendix gives the native execution settings and repair evaluation for the reductions in Section 5.3.

F.1 Benchmark and Execution Settings

We use ComputeEval 2026-2, with 591 released tasks, at dataset revision `60301f090273ee64967105ebb2eb5bad0aed3595`. Test units are complete author-provided CUDA cases or cuTile test functions, including their generators, assertions, and fixtures. A selector exposes these existing units without introducing new tests, splitting assertions, or changing tolerances. The original full harness and selected-case execution were checked against the reference; cross-case random-number consumption and initialization were preserved where applicable.

Generation receives the official public documentation and interface/build context, without functional-test feedback or hidden reference code. Repairs use GLM-5.2 through OpenRouter with only Z.AI FP8 enabled. Each independent trajectory starts from the same frozen C_0 and fresh context, receives the selected test constraints and all fixed G, and continues until the selected set passes twice. For CUDA cases, feedback includes the selected invocation's launch metadata. Reference/checker source and omitted-test outcomes are withheld. The accepted source is frozen before full-suite evaluation; omitted failures do not trigger further repair. The measurements retain the definitions of cross-test pass rates, pairwise test similarity, and BTD.

CUDA kernels run on H100 with CUDA 12.8. cuTile uses cuda-tile 1.6, TileIR 13.3, and CUDA compatibility libraries 13.3 on H100/SM90, with PyTorch 2.7.1+cu128. Native reference qualification precedes candidate evaluation.

F.2 Selection and Replication

For every failing target, $R = 2$ independent accepted repairs yield a complete row of cross-test pass rates. Using the corresponding test similarity matrix, we enumerate subsets in increasing cardinality,

retain those with at least 95% of full failing-set BTD, and break remaining ties by higher BTD and then test index. All initially passing tests stay in G and in the feedback count.

CUDA/104 has $P = S = \mathbf{1}$. Its original implementation assumes that the host launch spans the full concatenated output width; the official harness launches over one input width, a constraint absent from the public documentation. Selected-case feedback exposes this invocation contract. Every singleton choice is evaluated in three independently initialized repairs, and all recover the full suite (Table 2). The choices were frozen before their joint outcomes were observed. These trials also provide an independent $R = 3$ matrix, again all ones, without adding samples or expenditure to the recovery evaluation.

The CUDA/197 matrix is also all ones. LayerNorm's fractional BTD comes from $P_{0,4} = P_{0,5} = 1/2$, the only entries below one. BTD selects failing targets $\{0, 4\}$ plus $G = \{6, 7\}$, preserving the full empirical score. Table 7 reports full-suite recovery from three independently initialized repairs for each BTD-selected set. Initially passing tests exhibit no recorded regressions in these runs.

Table 7: Full-suite recovery with BTD-selected feedback on ComputeEval. Each entry counts trials passing every original test out of three independently initialized repairs. All initially passing tests are retained as regression constraints.

Task	Full-suite recovery
CUDA/104	$3/3$
CUDA/197	$3/3$
LayerNorm	$3/3$