Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

PatchBudget: A Controlled Study of Budgeted Candidate Verification and Abstention

Krishna Varun Kasamneni

kasamnenivarun@gmail.com

Independent researcher

Article

Keywords: software testing, candidate verification, test prioritization, abstention, empirical software engineering

Posted Date: November 18th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-11263567/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: No competing interests reported.

PatchBudget: A Controlled Study of Budgeted Candidate Verification and Abstention

Krishna Varun Kasamneni[1*]

[1*]Independent researcher, Bengaluru, India.

Corresponding author(s). E-mail(s): kasamnenivarun@gmail.com;

Abstract

Budgeted candidate verification couples where tests are allocated with how much evidence is required before accepting an implementation. We examine this interaction in a controlled, reference-oracle-assisted study of 16 selected QuixBugs algorithms and 241 candidate implementations, including deterministic single-edit mutants and a planted reference. All candidate outcomes are obtained by executing code. Four transparent scheduling policies observe only charged verification outcomes, and a distinct input pool provides final evaluation. With a one-test developer gate, 32 additional candidate-test queries, and eight required passing probes, candidate-focused allocation yields 93.13% useful coverage and 3.75% false acceptance, compared with 62.08% and 1.46% for round robin. Useful coverage counts accepted candidates that match every held-out case; it is not proof of semantic correctness. The coverage advantage must be interpreted alongside increased observed false acceptance and a counting argument showing how distributed testing can miss the acceptance threshold. Stronger developer gates, budget and threshold ablations, and failure-history baselines are retained. The contribution is an auditable protocol illustrating why allocation, abstention, initial suite strength, and oracle access must be reported together. The study introduces no new scheduling algorithm and establishes no performance claim for real repository patches or autonomous repair systems.

Keywords: software testing, candidate verification, test prioritization, abstention, empirical software engineering

1 Introduction

When several candidate implementations are available, running every possible test on every candidate may be unnecessary or too expensive. A verifier can spread tests across candidates, concentrate on a promising candidate, or reuse the failures observed on earlier candidates to change subsequent test order. However, apparent performance also depends on the acceptance rule. A policy that distributes a small budget widely may leave every candidate below a minimum evidence threshold, while a concentrated policy can qualify one candidate for acceptance.

We investigate this interaction in a deliberately small setting where every outcome can be retained and audited. The research question is: under the same cap on additional candidate-test executions, how do candidate allocation and online failure history affect accepted coverage and held-out agreement? The intended artifact is a controlled empirical diagnostic, not a new program-repair system. A reference implementation is available to construct expected outputs and is included among the candidate variants. This makes screening tractable while removing the oracle uncertainty and candidate-generation challenges of real repair.

The contribution is a controlled evaluation protocol and an auditable account of its coverage–risk trade-off: data construction, charged verification, and final evaluation are separated; allocation and ordering policies are paired; and selections, source hashes, outcome matrices, and protocol corrections are retained. A simple counting argument already predicts an allocation advantage when acceptance requires a minimum number of passes. The empirical question is how that mechanism appears alongside candidate rejections and stronger developer gates, rather than whether focused allocation is universally superior. The limits are integral to the interpretation: short algorithm programs, artificial mutations, a known reference, and a convenience sample cannot establish the effectiveness of an autonomous coding agent.

2 Related work

Fault-recorded testing prioritization (FRTP) already updates test priorities from failures observed during automated repair (Qi et al. 2013). Modification-point-aware prioritization and sampling further specializes this idea (Yazhini et al. 2020). Learning-based test scheduling in continuous integration also has an established literature, including bandit prioritization (Prado Lima and Vergilio 2020). Thus online scheduling, failure history, and budgeted test selection are not novel ideas introduced here.

Test-based assessment of candidate correctness has likewise been studied (Xiong et al. 2018; Ye et al. 2021b). Additional tests can distinguish candidates that pass an original suite, but finite test agreement is not a semantic proof. SWT-Bench evaluates issue-reproduction tests at repository scale and studies their use as filters for candidate fixes (Mündler et al. 2024). Preprint versions also discuss pre-resolution test generation (Ahmed et al., 2024, https://arxiv.org/abs/2412.02883), assertion inversion (Khatib et al., 2025, https://arxiv.org/abs/2507.17542), and multilingual reproduction benchmarks (Kusama et al., 2026, https://arxiv.org/abs/2609.34752).

These preprint versions are identified here as contextual material, not as reproduced baselines or confirmed journal publications.

QuixBugs comprises small algorithmic challenges with defective and corrected implementations (Lin et al. 2017); repair studies have examined it previously (Ye et al. 2021a). We use a selected Python subset to measure a transparent screening protocol. Our candidate variants are generated with AST mutations, not a language model. The unresolved practical question of whether these effects transfer to real LLM patches remains outside the evidence provided here. This targeted review does not establish a novel research gap.

3 Study design

3.1 Programs, candidates, and developer gate

We pin QuixBugs to commit `4257f44b0ff1181dedaedee6a447e133219fcebf`. Sixteen programs were selected for self-contained, bounded integer, list, or string adapters. Table A1 lists the final corpus. The selection was made before policy results; it does not represent a random sample of real software defects.

For each program, the candidate pool contains its corrected implementation, its original buggy implementation, and up to 40 deterministic single-edit AST mutations of the corrected source. The grammar changes integer constants by one, flips selected comparisons, replaces selected arithmetic or bitwise operators, and swaps Boolean conjunction/disjunction. Candidates are deduplicated by source hash, then capped by sorted hash rather than observed behavior. Equivalent mutants remain. A random candidate ordering hides the reference's position from the scheduler. Including a reference guarantees that a valid positive control exists; no inference about the ability to generate repairs follows.

All policies share a developer gate with either the first distributed test, the first three tests, or all resource-eligible distributed tests. Candidates failing the selected gate are removed before additional verification. The main setting deliberately uses a weak one-test gate. Stronger gates are reported as sensitivity analyses. One official knapsack input has capacity 6,404,180 and 24 items, requiring an impractically large reference dynamic program; we exclude official inputs with capacity times item count exceeding one million. This removes exactly one case and is recorded with its input hash. "All" therefore means all eligible cases, not the unmodified upstream suite.

3.2 Verification and held-out evaluation

Each program receives 64 distinct generated verification inputs and 256 distinct held-out inputs. Independent deterministic random streams sample the same declared bounded domain, excluding all original developer inputs and exact overlap between the two generated pools. Every generated expected output is cross-checked between the reference implementation and an independently written specification implementation (standard library, dynamic programming, or brute force as appropriate). Distributed

developer assertions are separately checked against the reference. This detects inconsistencies, but does not make the two implementations an external authority on all possible program behavior.

Generated input distributions and source code are retained. As examples, bitcount receives integers in $[0, 65535]$; list algorithms receive short signed-integer lists; sieve receives integers in $[0, 600]$. QuixBugs `lcs_length` denotes a longest common *substring*, which the independent oracle respects. Preconditions are specific to these adapters. We claim agreement only on the generated evaluation distribution.

The harness executes every candidate on the developer, verification, and held-out pools, saving Boolean outcome matrices. Exceptions and a deterministic Python line-event cap count as failures. Generated tests use 20,000 events; developer cases use 100,000. The reference passes every retained case. The cap prevents runaway benchmark mutants but is not a security sandbox for arbitrary untrusted code.

Policy evaluation replays these actual outcomes through an interface that exposes one verification cell per charged query. It receives no held-out labels, reference identity, or uncharged failure statistics. Held-out outcomes are consulted only after candidate selection. Full matrix construction is an offline experimental expense: replay query counts must not be mistaken for the physical total work performed to collect the dataset.

4 Scheduling and acceptance

Let C denote gate-passing candidates and T verification tests. A query (c, t) costs one unit, reveals whether candidate c agrees with the oracle on input t, and immediately rejects c on failure. Repeated queries of the same pair are prohibited. Every policy uses the same candidate ranking and initial test permutation for a paired seed.

Round robin.

Query a least-tested surviving candidate, breaking ties by the shared candidate ranking, and select its next untried test in the fixed permutation.

Candidate focused.

Query the highest-ranked surviving candidate until it is rejected or all verification tests have run, then proceed to the next candidate. Tests follow the fixed permutation.

Failure count.

Use candidate-focused allocation, choosing the candidate's untried test with the largest count of observed failures on earlier candidates; ties follow the fixed permutation. All counts start at zero and reset for each run. This is FRTP-inspired, rather than an exact reproduction: original FRTP seeds an original failing test, randomizes priority ties, and requires complete-suite success (Qi et al. 2013).

Smoothed failure rate.

Use the same allocation and tie rule, but rank test t by $(f_t+1)/(o_t+2)$, where f_t and o_t are its charged failures and observations. The smoothing rule is fixed in advance; it is a heuristic, not a calibrated correctness probability.

At budget exhaustion, select the highest-ranked surviving candidate with at least k distinct passing verification tests, or abstain if none qualifies. The policy may use less than its cap only after it exhausts available queries. It does not stop on a hidden label or when a reference candidate happens to be selected. We evaluate $B \in \{8, 16, 32, 64, 128\}$ and $k \in \{1, 4, 8, 16\}$. An existing-only baseline selects the first gate passer with no additional verification and is explicitly outside the k-probe requirement.

The primary comparison is candidate focused versus round robin at $B = 32$, $k = 8$, and the one-test gate. There are 30 paired ranking/test-order seeds per program. Threshold sensitivity is essential because the acceptance criterion interacts mechanically with allocation.

To make this mechanism explicit, suppose $m \geq 1$ candidates survive the gate, every verification outcome passes, and each candidate has at least $k \geq 1$ available distinct tests. After $m(k-1)$ round-robin queries, every candidate has exactly $k-1$ passing probes. The next query is therefore the first to qualify any candidate. The first-qualification query counts are

$$q_{\mathrm{RR}} = m(k-1)+1, \qquad q_{\mathrm{focused}} = k. \tag{1}$$

Thus a cap $B < m(k-1)+1$ makes round robin abstain under these assumptions, while focused allocation has a qualifying candidate whenever $B \geq k$. The focused implementation still continues after qualification until the cap or all available queries are exhausted; qualification is not its stopping rule. This counting argument is not a new scheduling result. The executed study examines how observed failures, developer-gate strength, and ordering interact with that mechanism across the declared corpus and parameter grid. Any coverage difference must be interpreted accordingly.

5 Metrics, uncertainty, and execution

For a program/ranking pair, let A indicate acceptance and H indicate that the selected candidate passes all 256 held-out tests. We report acceptance coverage $\mathbb{E}[A]$, useful coverage $\mathbb{E}[AH]$, and false acceptance $\mathbb{E}[A(1-H)]$. The conditional agreement rate is $\mathbb{E}[AH]/\mathbb{E}[A]$ when acceptance is nonzero; it is not the mean of per-program conditional ratios. False acceptance denotes failure on this finite held-out set, not an exhaustive proof of faulty behavior outside it. Conversely, useful coverage is test-supported evidence only.

We first average rankings within each algorithm, then weight algorithms equally. Paired percentile intervals resample the 16 algorithms 5,000 times, using the same resampled indices for both policies. These intervals describe variability across this selected corpus, not a representative population of software projects. Ranking seeds are not counted as independent bugs. Probe generation is fixed per algorithm; uncertainty across alternative generated suites is not estimated.

The full experiment produces 116,640 selection records. Outcome construction took 24.4 seconds, including 0.9 seconds of oracle/preflight work on the recorded arm64 macOS environment. Execution used Python 3.14.7; numerical analysis used Python 3.12.13, NumPy 2.5.3, and Matplotlib 3.11.2. Each manifest records exact versions, source hashes, and runtime. No paid model or external compute was used. Scheduling time in the CSV measures replay computation over stored Booleans; it is not end-to-end verifier latency. Developer-gate query counts model short-circuit evaluation, whereas matrix construction physically evaluates all eligible developer cases.

The protocol was written locally before final evaluation, not publicly preregistered. A separate two-program pilot measured feasibility. Initial matrix construction aborted on the large knapsack input; independent preflight also detected that the initial sieve domain exceeded the reference's line-event cap. A second construction was interrupted during a costly, growing-integer mutant. Preflight established that all eligible reference developer cases use at most 43,631 events, so the developer cap was reduced from two million to 100,000 before a fresh full run. Partial matrices and the deviation log are retained. No parameters were changed to improve a policy's reported results.

AI assistance.

AI tools materially assisted with literature review, study design, code, experiments, analysis, manuscript drafting, and preparation of this journal-format manuscript. This assistance extends beyond language editing. AI tools are not listed as authors. The sole human author reviewed and approved the final manuscript and takes responsibility for the accuracy and integrity of the work.

6 Results and ablations

Table 1 reports the prespecified budget and threshold across developer gates. With the one-test gate, the existing-only baseline attains 32.50% useful coverage and 67.50% false acceptance. Candidate-focused allocation yields 93.13% useful coverage, while round robin yields 62.08%. Their paired difference is 31.04 percentage points, with a corpus bootstrap 95% interval of [12.29, 51.25]. The corresponding false-acceptance difference is 2.29 points [-2.29, 10.00]. These are differences between complete allocation-plus-acceptance procedures, not proof that concentrating tests universally produces better tests.

Failure-count ordering reaches 93.33% useful coverage and 3.75% false acceptance in the primary setting. This baseline isolates one established use of failure history while keeping allocation and acceptance fixed. The complete results retain ties and unfavorable outcomes; no superiority over published full repair systems is implied. Figure 1 also makes developer-suite strength visible. When the gate already eliminates distinguishable variants, additional scheduling has less opportunity to improve held-out agreement.

The threshold ablation in Figure 2 prevents interpreting abstention as a correctness improvement. A policy can lower false acceptance by selecting nothing; useful coverage and total coverage must be considered alongside conditional agreement. Similarly, the existing-only baseline spends fewer calls but does not satisfy the verification evidence

Gate	Policy	Accept (%)	Useful (%)	False (%)	Calls
One	Existing tests only	100.0	32.5	67.5	0.0
One	Round robin	63.5	62.1	1.5	32.0
One	Candidate focused	96.9	93.1	3.8	32.0
One	Failure count	97.1	93.3	3.8	32.0
One	Smoothed failure rate	96.5	92.5	4.0	32.0
Three	Existing tests only	100.0	80.4	19.6	0.0
Three	Round robin	84.4	82.7	1.7	32.0
Three	Candidate focused	98.1	95.0	3.1	32.0
Three	Failure count	98.1	95.0	3.1	32.0
Three	Smoothed failure rate	98.1	95.0	3.1	32.0
All	Existing tests only	100.0	91.3	8.8	0.0
All	Round robin	87.3	85.6	1.7	32.0
All	Candidate focused	98.5	95.6	2.9	32.0
All	Failure count	98.3	95.4	2.9	32.0
All	Smoothed failure rate	98.3	95.6	2.7	32.0

Table 1 Primary setting: budget 32 and minimum eight passing verification tests; existing-only has no extra probes. "Useful" and "False" are fractions of all program/ranking cases and sum to acceptance. Calls are actual logical additional queries, excluding the shared developer gate. All gates use resource-eligible cases.

threshold. Its point is to quantify the gate's initial screening strength, not provide an equal-information comparison.

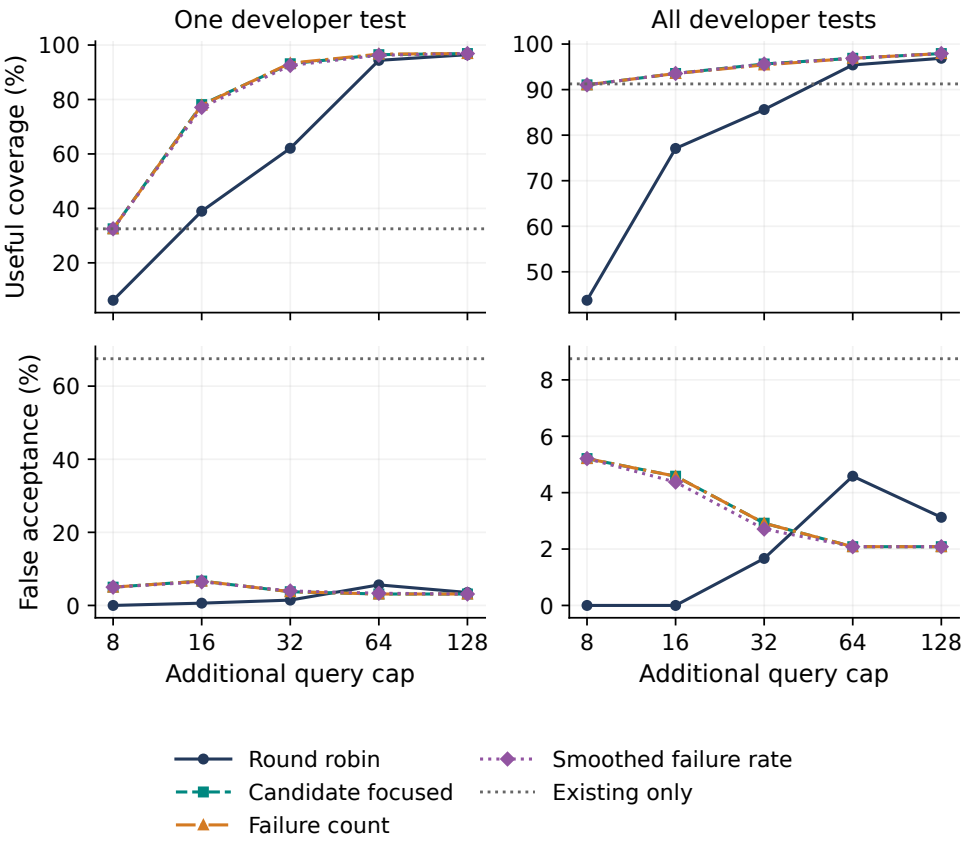


Fig. 1 Budget sensitivity with minimum eight passing probes. The dotted reference uses only developer tests. Curves average 30 rankings per algorithm and then the 16 selected algorithms. The full machine-readable grid includes the three-test gate and all thresholds.

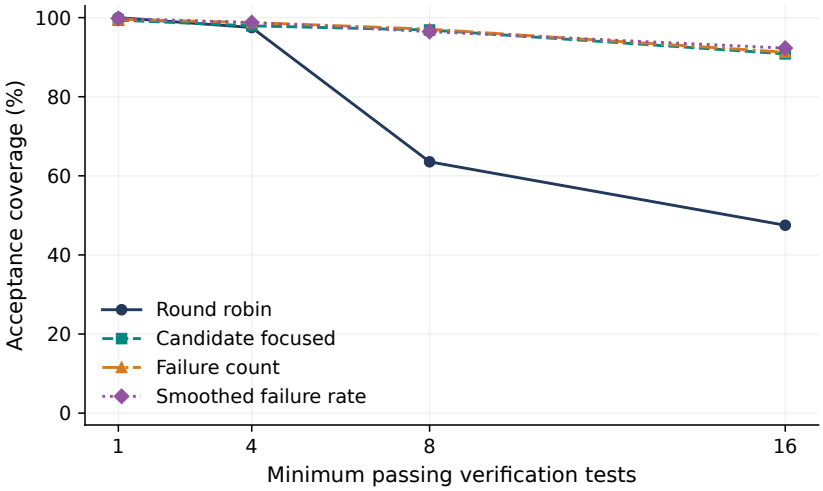


Fig. 2 Acceptance-rule ablation. At a common query cap, raising the minimum evidence threshold can make broadly distributed testing abstain. This is an intended diagnostic of allocation and acceptance coupling.

7 Discussion and limitations

The observations support a narrow methodological conclusion: fixed-budget verifier evaluations must specify candidate allocation, acceptance thresholds, developer-suite strength, and oracle privileges together. Reported reliability without coverage can reward trivial abstention, and a query budget alone does not control runtime. A concentrated policy's coverage advantage can arise directly from the acceptance threshold even without better test ordering.

The fault model is artificial. Mutations of a known corrected program need not resemble plausible LLM repairs, and the planted reference ensures a solution is available. The inputs cover bounded, documented domains; finite agreement cannot identify semantic equivalence. Inputs are exact-disjoint but identically distributed, so the held-out pool does not evaluate distribution shift. Shared reference-oracle access excludes oracle hallucination, weak assertions, and specification ambiguity that real generated tests face. Independent specification implementations reduce a source of implementation error without proving the specification complete.

The algorithms are small and deterministic. We evaluate neither repositories nor flaky tests, integration setup, expensive model calls, parallel execution, or adversarial candidate code. Every query has unit logical cost despite large differences in execution time. Thirty ranking seeds explore order sensitivity but do not enlarge the corpus to 480 independent defects. Published FRTP, MPTPS, and CI systems solve related but different end-to-end tasks; only a transparent FRTP-inspired ordering is implemented here. There is no claim of state-of-the-art performance, established novelty, or publication readiness.

A stronger submission would require real fixed candidate-patch pools, independent test oracles unavailable to the solver, repository-level evaluation, meaningful cost models, and faithful relevant baselines. The current manuscript is a complete report of the scoped controlled experiment and a starting artifact for that work.

8 Conclusion

PatchBudget provides an auditable execution-based study of budgeted candidate screening. The protocol separates actual outcome collection from charged scheduling and discloses the effects of abstention, suite strength, and oracle access. The reported comparisons characterize this selected mutation corpus; extending them to autonomous program repair requires additional evidence.

Statements and Declarations

Funding. This research received no funding.

Competing interests. The author declares no competing interests.

Author contributions. K.V.K. is the sole human author and was responsible for conceptualization, methodology, software development, investigation, data curation, formal analysis, validation, visualization, and manuscript preparation, review,

and editing. AI tools assisted with study design, implementation, experiments, analysis, and manuscript drafting, as disclosed in the manuscript. K.V.K. reviewed and approved the final manuscript and takes responsibility for the accuracy and integrity of the work.

Data and code availability. The source code, experimental configurations, candidate inventory, recorded test outcomes, selection records, and analysis results supporting this study are provided in the supplementary research artifact. The artifact includes instructions and verification scripts for reproducing and checking the reported analyses.

AI assistance. The scope of substantive AI assistance is described in the execution and analysis methods in Section 5.

Appendix A Corpus inventory

Program	Candidates	Gate 1	Gate all	Holdout pass
bitcount	10	1	1	1
bucketsort	9	6	1	1
find_first_in_sorted	24	10	5	2
gcd	6	3	1	1
get_factors	16	13	2	4
is_valid_parenthesization	17	3	1	1
knapsack	23	8	5	4
kth	10	2	2	2
lcs_length	14	4	1	1
lis	24	22	3	2
max_sublist_sum	9	6	2	2
mergesort	23	23	2	2
next_permutation	25	8	5	2
quicksort	11	4	2	2
sieve	11	9	1	1
to_base	9	2	1	1

Table A1 Candidate counts and gate survivors. Holdout pass counts agreement on all evaluation inputs before gate filtering. Equivalent mutants are intentionally retained.

Appendix B Reproduction and evidence

Run the execution experiment with Python 3.14 as documented in the artifact; the recorded version is 3.14.7. From the artifact's repository root, install the documented dependencies and run:

```
python scripts/prepare.py
python -m unittest discover -s tests -v
python scripts/run.py --output reproduction/full
python scripts/verify.py --reproduction reproduction/full
python scripts/analyze.py
```

The analysis script regenerates the reported tables and figures from `results/full`; the verifier compares a fresh reproduction with that retained evidence. The journal-format source supplied separately with this manuscript can be compiled with `tectonic main.tex`, or with a standard LaTeX/BibTeX workflow. No experiment was rerun to prepare this journal-format version.

References

Lin D, Koppel J, Chen A, et al (2017) QuixBugs: A multi-lingual program repair benchmark set based on the quixey challenge. In: Proceedings of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity, pp 55–56, https://doi.org/10.1145/3135932.3135941

Mündler N, Müller MN, He J, et al (2024) SWT-Bench: Testing and validating real-world bug-fixes with code agents. In: Advances in Neural Information Processing Systems, https://doi.org/10.52202/079017-2601

Prado Lima JA, Vergilio SR (2020) A multi-armed bandit approach for test case prioritization in continuous integration environments. IEEE Transactions on Software Engineering https://doi.org/10.1109/TSE.2020.2992428, online-first publication; accepted May 4, 2020

Qi Y, Mao X, Lei Y (2013) Efficient automated program repair through fault-recorded testing prioritization. In: 2013 IEEE International Conference on Software Maintenance, pp 180–189, https://doi.org/10.1109/ICSM.2013.29

Xiong Y, Liu X, Zeng M, et al (2018) Identifying patch correctness in test-based program repair. In: Proceedings of the 40th International Conference on Software Engineering, https://doi.org/10.1145/3180155.3180182

Yazhini V, Phung QN, Lee E (2020) Modification point aware test prioritization and sampling to improve patch validation in automatic program repair. Applied Sciences 10(5):1593. https://doi.org/10.3390/app10051593

Ye H, Martinez M, Durieux T, et al (2021a) A comprehensive study of automatic program repair on the QuixBugs benchmark. Journal of Systems and Software 171:110825. https://doi.org/10.1016/j.jss.2020.110825

Ye H, Martinez M, Monperrus M (2021b) Automated patch assessment for program repair at scale. Empirical Software Engineering 26. https://doi.org/10.1007/s10664-020-09920-w

Supplementary Files

This is a list of supplementary files associated with this preprint. Click to download.

- patchbudgetreviewartifact.zip