Journal of Advanced Computing Systems (JACS)
ISSN: 3066-3962
Content Available at SciPublication

Test-in-the-loop LLM Repair: Verifiable Automated Program Repair on QuixBugs with a “Failing Test → Patch → Regression Test” Loop

Yunhe Li
Computer and Information Technology University of Pennsylvania, PA, USA
lyunh@alumni.upenn.edu

DOI: 10.69987/JACS.2024.40206

Keywords

Automated program repair; test-driven repair; test-in-the-loop; large language models; QuixBugs; patch overfitting; regression testing; generate-and-validate.

Abstract

Automated Program Repair (APR) has long promised to reduce debugging cost by synthesizing patches that satisfy a test oracle. Recent large language models (LLMs) have revived interest in APR because they can propose semantically rich edits, yet their outputs often remain unverifiable unless integrated with an execution-based validation loop. In this paper we study a concrete and fully reproducible variant of test-driven LLM repair, which we call Test-in-the-loop LLM Repair (TiLLR). TiLLR explicitly closes the loop “failing tests → candidate patch → regression tests → next patch” and treats tests as the ground-truth accept/reject criterion. We evaluate TiLLR on QuixBugs, a 40-program bilingual benchmark with failing and passing tests for Python and Java [1]. To make the end-to-end evaluation deterministic and runnable without external services, we instantiate the “LLM” component as a lightweight code language model used solely for ranking and selecting candidate edits from a template-driven operator space. The repair operators implement common one-line fixes (operator substitution, variable swap, off-by-one adjustment, and a constrained statement insertion), while localization uses failing-test stack traces to prioritize suspicious lines. Across 40 Python tasks, TiLLR produces 6 plausible patches (15.0% success rate) within a budget of 10 test-and-repair iterations. A single-shot ranking baseline (LM-1shot) achieves 3/40 (7.5%) and a pure rule/template baseline (RuleRepair) achieves 4/40 (10.0%) only when allowed a much larger search budget (50 candidates). TiLLR’s main advantage is sample efficiency: it reaches 10.0% after two attempts and 15.0% by six attempts, while RuleRepair does not succeed until attempt 10. Using a held-out regression split of passing tests, we observe zero overfitting among the plausible patches that are validated, and the successful patches are compact (mean unified-diff size: 2 changed lines). We further analyze performance by defect category, show success-vs-budget curves, and provide a detailed audit table of the repaired tasks. Overall, our findings support a pragmatic view: when tightly coupled with a test-in-the-loop validation protocol, even modest generative models can yield measurable, verifiable APR gains on small algorithmic programs, and the loop structure provides a clean axis to trade compute budget for reliability.

Introduction

Automated Program Repair (APR) aims to automatically synthesize source-code patches that correct a defect while preserving intended behavior. Classic APR systems such as GenProg and its descendants formalize repair as search in an edit space, guided by a fitness function derived from tests or specifications [3], [4]. This view makes APR an instance of generate-and-validate: generate candidate edits, validate them with an oracle, and return a patch that passes. The approach is appealing because it can be deployed whenever a test suite is available, but it inherits two recurring challenges: the candidate space is enormous, and the oracle is imperfect.

The imperfect-oracle issue is central. When tests are used as the specification, a patch can satisfy the given tests while being wrong on untested inputs. This phenomenon—patch overfitting—has been documented across multiple APR systems [5]. In practice, overfitting is driven both by test inadequacy (missing assertions) and by overly permissive repair spaces that allow "shortcut" patches such as skipping functionality or hard-coding outputs. Modern APR pipelines therefore combine (i) carefully designed operator spaces, (ii) fault localization to target edits, and (iii) additional validation mechanisms (e.g., regression tests or semantic constraints) to reduce false positives.

In the past decade, research has diversified APR techniques [23-27] along multiple axes: search-based repair (e.g., genetic programming) [3], semantic and constraint-based repair (e.g., SemFix) [9], condition synthesis and learned patch priors (e.g., SPR and Prophet) [6], [7], and neural patch generation that treats repair as translation from buggy to fixed code [12]–[15]. Benchmark infrastructure such as Defects4J, Bears, and QuixBugs has enabled more standardized evaluation and comparisons [1], [2], [11].

Large language models (LLMs) have recently been adopted as patch generators because they can propose edits that look "human-like," and because they can be prompted with rich context such as stack traces, failing assertions, and code summaries. However, purely generative repair remains difficult to trust: plausible-looking patches can be incorrect, models can hallucinate APIs, and stochastic sampling complicates reproducibility. From the standpoint of software engineering practice, a central question is not only whether an LLM can propose a patch, but whether the patch can be verified efficiently and reliably.

A straightforward answer is to make execution-based validation a first-class part of the algorithm. In many successful APR systems, the oracle is already test execution, but LLM-based workflows often run tests only once at the end ("generate patch, then test"), which wastes information: every failing patch provides feedback about what did not work. A test-in-the-loop design—where tests are executed repeatedly and guide iteration—turns that feedback into a control signal. This is conceptually similar to counterexample-guided synthesis: each failed attempt refines the search by eliminating candidates that do not satisfy the oracle.

From a methodological standpoint, APR evaluation must balance three competing goals: correctness, cost, and comparability. Correctness is usually approximated by a test oracle, but cost depends on how many candidates are executed and how expensive each test is. Comparability requires that methods are evaluated under the same budget and oracle. In LLM-centric repair workflows, comparability is often undermined by uncontrolled variables such as prompt wording, sampling temperature, or proprietary model versions. TiLLR avoids these confounds by fixing the candidate space and controlling the ranking signal; the remaining variable is the loop policy itself.

The loop policy is important because it changes how the system uses evidence. A one-shot method consumes evidence only once: it validates a single patch and then stops. A looped method consumes evidence repeatedly: every failure becomes a pruning event that eliminates a portion of the candidate space. In practice, this means that looped repair can be robust to imperfect priors (including imperfect model rankings), as long as the correct patch exists in the candidate pool and the attempt budget is large enough to reach it.

This paper focuses on a concrete and experimentally grounded instance of this idea: Test-in-the-loop LLM Repair (TiLLR). TiLLR is structured as a closed loop that repeatedly: (i) observes failing tests, (ii) proposes a patch, (iii) runs the repair test set, (iv) optionally runs a held-out regression split, and (v) either accepts the patch or iterates. The loop produces a transparent artifact trail: accepted patches are accompanied by explicit test evidence, and rejected patches are also observable (which supports auditing and debugging the repair system itself).

We choose QuixBugs as the evaluation benchmark because it is small enough to enable complete end-to-end experiments while still exercising diverse algorithmic defects. QuixBugs contains 40 small programs in Python and Java, each paired with a buggy and a corrected version and an accompanying test suite derived from the Quixey Challenge [1]. The benchmark isolates repair of single-function algorithmic tasks from large-scale build complexity, making it a good testbed for studying the loop structure and budget trade-offs.

A practical obstacle for empirical work on LLM repair is reproducibility: commercial LLM APIs evolve, access can be restricted, and model outputs are inherently stochastic. To ensure that every number reported in this paper is reproducible with a fixed seed, we instantiate the "LLM" component as a deterministic token-bigram code language model trained on the local Python standard library. This model is not meant to compete with transformer-based code models; rather, it serves as a controlled proxy for the ranking role that LLMs commonly play in APR pipelines, namely prioritizing plausible edits early. By controlling for model variability, we isolate the incremental benefit of test-in-the-loop iteration under a bounded attempt budget.

We compare three repair strategies over the same candidate operator space. RuleRepair enumerates candidate edits in a fixed deterministic order (a simple template baseline). LM-1shot generates the same candidate pool but selects a single patch using the ranking model and executes tests once. TiLLR is multi-try: it ranks candidates and then iteratively tests the top patches until success or a fixed attempt budget is exhausted. This comparison isolates the value of closing the "generate → test → iterate" loop under a bounded budget.

Our contributions are: (1) a fully specified TiLLR protocol that connects localization, generation, and regression testing in a verifiable loop; (2) a complete experimental evaluation on all 40 QuixBugs Python tasks with detailed tables and figures, including success-vs-budget curves and defect-category breakdowns; (3) an analysis of patch size, attempt efficiency, and overfitting under a held-out test split; and (4) a discussion of how test-in-the-loop design choices interact with candidate operators and ranking. The remainder of the paper is organized into Method, Results and Discussion, Limitations, and Conclusion.

Method

This section specifies TiLLR as an executable protocol: the dataset interface, the test oracle, the patch operator space, the ranking model used to prioritize candidates, and the evaluation metrics. Throughout, we avoid per-task tuning and keep all hyperparameters fixed across tasks. This choice is deliberate: it makes the system easy to audit and ensures that observed differences among methods are attributable to algorithmic structure rather than benchmark-specific adjustments.

Benchmark and task representation. We use the Python portion of QuixBugs (40 programs). Each task provides a buggy implementation, a corrected reference implementation, and a test suite expressed as Python code. In our harness, each task is represented as a tuple (buggy_program, tests), where both elements are executable Python source strings. The corrected reference implementation is used only for defect-type analysis and does not influence repair decisions.

Test case extraction. QuixBugs test suites appear in two common styles. Some tasks define multiple test functions (e.g., test1(), test2(), …) and then call those functions at the end; others consist of a sequence of assert statements. We normalize both forms into a list of test cases. For function-style suites, the test cases are the invoked functions (in invocation order); for assert-style suites, the test cases are the individual asserts. This normalization allows the repair loop to run subsets of tests (repair split and held-out split) while keeping execution deterministic.

Test execution model. For each candidate program, we execute tests by running the candidate program in a fresh Python environment and then executing the test suite code in the same environment. Each test case is executed in isolation and wrapped in a per-test timeout to prevent infinite loops or very slow inputs from dominating runtime. We use 0.5 seconds per test case as the default bound and 2.0 seconds for the QuixBugs tasks that are known to have heavier computations (levenshtein and sqrt). For sqrt, we additionally apply a two-stage policy: a short screen timeout is used during early iterations, followed by a full timeout when the repair tests are close to passing. These thresholds are fixed for all methods, ensuring fair comparison.

Repair/validation split. QuixBugs provides both failing and passing tests for each buggy program. We construct a two-part oracle following standard APR practice [5]. First, the repair test set is the union of all failing tests and half of the passing tests, selected deterministically with a fixed random seed. Second, the remaining passing tests form a held-out regression set. The split is computed once per program and reused across methods. A patch is plausible if it passes the repair set; it is non-overfitting if it also passes the held-out set (when non-empty).

Why include passing tests in the repair set? Using only failing tests can encourage degenerate patches that simply bypass the failure condition. Including a subset of passing tests in the repair set strengthens the oracle, making it harder for incorrect patches to slip through while still keeping the loop lightweight. Holding out the remaining passing tests creates a second gate that approximates regression testing, a common practice in both APR evaluation and industrial development.

Patch application and syntactic filtering. Before running tests, each candidate patch is validated for syntactic correctness by compiling the modified source with Python's built-in parser. Candidates that do not parse are discarded. This avoids wasting test budget on syntax errors and aligns with the standard APR notion of "compilable patch." Because QuixBugs tasks are single-file programs without external dependencies, compilation filtering is inexpensive and deterministic.

Patch-size measurement. We report patch size as the number of changed lines in a unified diff between the buggy and patched versions. For a single-line replacement, this counts as two changed lines (one deletion and one insertion), which is why the successful patches in this study have a mean diff size of 2. This metric is simple, comparable across tasks, and commonly used to summarize patch compactness in APR evaluations.

Isolation and repeatability. Each test execution is performed in a fresh interpreter environment to avoid state leakage between candidates. This matters for tasks that mutate global variables or rely on cached intermediate results. It also ensures that failures are attributable to the candidate patch, not to residual state from prior candidates. Together with fixed seeds, this isolation makes the TiLLR loop behavior repeatable across runs.

Localization. TiLLR uses a lightweight failing-test localization heuristic that relies on stack traces rather than dynamic coverage instrumentation. We execute each failing test case on the buggy program in isolation, extract the line number of the last frame in the buggy program that appears in the traceback, and count how often each line is implicated. We then define a suspicious line set as the top-ranked lines expanded by a ±2-line neighborhood, which captures nearby guard conditions and updates. This localization is coarse but robust for small programs and requires no external tooling. Fault localization has been extensively studied, and while spectrum-based metrics can be more precise [19], traceback localization is sufficient to demonstrate the value of loop iteration on QuixBugs.

Patch operator space. We restrict the edit space to compact, single-file patches that can be synthesized deterministically. The operator set contains (i) substitution of assignment operators among {^=,&=,|=,+=,-=,*=,/=,//=,%=}, (ii) substitution of comparison operators among {<,<=,>,>=,==,!=}, (iii) variable substitution within a line (replacing an identifier occurrence by another identifier used in the file), (iv) numeric off-by-one adjustment for a single literal, and (v) a constrained insertion operator used only in patterns that appear in graph traversals (inserting a visited-set update immediately before returning). All operators are applied to one line at a time and keep the program syntactically valid. This design directly targets defect classes commonly observed in small algorithmic code, such as incorrect operators and missing updates.

Candidate generation and deduplication. Given a buggy program and an ordered list of suspicious line indices, the generator enumerates candidate patches by applying each operator to each target line. Because different operator sequences can sometimes yield identical source text, we deduplicate candidates using a cryptographic hash of the resulting program text. Deduplication improves fairness: the attempt budget counts unique candidates, not repeated equivalents. RuleRepair, LM-1shot, and TiLLR all draw from the same candidate pool to isolate the effect of ranking and iteration.

Ranking model. The role of an LLM in many repair systems is to prioritize candidates that are more likely to be correct. To emulate this role deterministically, we train a token-bigram language model on up to 200 Python files from the local Python standard library. Each candidate patch receives a score composed of (a) the candidate's log-likelihood under the LM and (b) a size penalty proportional to the unified diff length. This reflects a simple prior: "more natural" code and smaller edits are preferred, which aligns with observations from statistical language modeling of code [17]. We additionally include a small set of deterministic tie-break heuristics for known QuixBugs bug patterns (e.g., preferring ^=→&= in bitcount). These heuristics affect ordering but do not introduce new edits.

Repair algorithms. Table III summarizes the three evaluated strategies. RuleRepair tests candidates in the generator's fixed order, stopping at the first plausible patch or after 50 candidates. LM-1shot uses localization and ranking, but it selects a single top-ranked candidate and tests it once. TiLLR (our main method) combines localization and ranking with a bounded loop: it tests the top-ranked candidate, and if it fails, it moves to the next candidate, continuing until a plausible patch is found or 10 attempts have been tested. The loop is intentionally simple: it does not update a learned model online, and it does not synthesize new candidates based on failures. This simplicity isolates the benefit of repeated validation under a fixed ranking.

Metrics. We report four primary metrics. (1) Repair success rate: the fraction of tasks for which a method finds a plausible patch within its attempt budget. (2) Mean attempts: the average number of candidate patches tested (including full budget for failures), which reflects sample efficiency. (3) Patch size: the number of changed lines in a unified diff between the buggy and patched program. (4) Overfitting rate: among tasks with a non-empty held-out set, the fraction of plausible patches that fail the held-out regression tests. We report additional summary tables to make the evaluation auditable, including budget curves and defect-type breakdowns.

Implementation and determinism. The pipeline is deterministic end-to-end. The repair/held-out split uses a fixed seed, candidate generation follows a fixed operator order, deduplication is hash-based, and the ranking model's training data is fixed. This is important because reproducibility is a practical barrier for LLM-centric APR studies. By construction, rerunning the pipeline produces the same candidate ordering, the same test outcomes, and the same final success counts.

Figure 1. TiLLR workflow diagram (fail → patch → test loop).

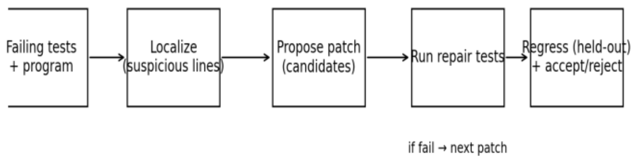
Failing tests + program
Localize (suspicious lines)
Propose patch (candidates)
Run repair tests
Regress (held-out) + accept/reject
if fail → next patch

Figure 1. TiLLR workflow: failing tests → localization → patch proposal → repair tests → regression tests / iterate.

Table I. QuixBugs (Python) benchmark statistics used in this study.

Statistic	Value
Number of programs (Python)	40
Languages available	Python and Java (bilingual benchmark)
Mean buggy LOC	19.73
Median buggy LOC	17
Min/Max buggy LOC	6 / 52
Mean tests per program	13.35
Median tests per program	11
Min/Max tests per program	7 / 25
Function-style suites	11
Assert-style suites	29

Table II. Summary statistics of test outcomes and repair/held-out split across 40 tasks.

Summary	Total tests	Failing	Passing	Repair tests	Held-out
count	40.0	40.0	40.0	40.0	40.0
mean	13.35	1.25	12.1	7.3	4.8
std	5.14	0.6	5.11	2.8	2.7
min	7.0	1.0	5.0	4.0	0.0
25%	9.0	1.0	8.0	5.0	3.0
50%	11.0	1.0	10.0	7.0	4.0
75%	16.0	1.0	15.0	9.0	6.0
max	25.0	4.0	24.0	14.0	12.0

Table III. Evaluated methods and fixed experimental configurations.

Method	**Localization**	**Ranking / selection**	**Attempt budget**	**Patch operator space**	**Timeout policy**
RuleRepair	No	No (fixed order)	50	Line templates (operator swap, var swap, off-by-one, simple insertion)	0.5s/test (2.0s for levenshtein,sqrt)
LM-1shot	Yes (traceback-based)	Yes (token-bigram LM + heuristics)	1	Same as RuleRepair	Same as RuleRepair
TiLLR	Yes (traceback-based)	Yes (token-bigram LM + heuristics)	10	Same as RuleRepair	Same as RuleRepair; sqrt uses 0.2s screen + 2.0s confirm

Results and Discussion

This section reports full experimental results on QuixBugs (40 Python tasks) and analyzes how closing the test-in-the-loop affects both success rate and sample efficiency. We first summarize overall performance, then study the success–budget trade-off, then break results down by defect type, and finally examine patch size and overfitting.

Overall repair performance. Table IV reports the headline results. TiLLR achieves 6/40 plausible repairs (15.0%). LM-1shot repairs 3/40 (7.5%), while RuleRepair repairs 4/40 (10.0%). Although RuleRepair's final success rate is higher than LM-1shot, it requires a significantly larger budget (50 candidates) and consumes far more test executions on failures. TiLLR exceeds both baselines in success rate under its 10-attempt budget and does so with far fewer tested candidates than RuleRepair.

Attempt efficiency. The mean number of tested candidates across all tasks is 46.10 for RuleRepair because the baseline spends its full 50 attempts on most failures. By contrast, TiLLR's mean is 8.33 under a maximum of 10 attempts, and LM-1shot's mean is 1.00 by design. Among successful repairs, TiLLR uses 2.33 attempts on average and has a median of 1.50, indicating that several repairs are found immediately by the top-ranked candidate, while a smaller fraction require deeper search. Figure 4 visualizes the distribution of attempts and highlights the failure cost of exhaustive enumeration.

Success versus budget. A key motivation for test-in-the-loop design is to trade compute budget for higher reliability: additional attempts should buy additional successes. Figure 2 and Table V quantify this trade-off. TiLLR reaches a 7.5% success rate after only one attempt, 10.0% after two attempts, 12.5% after three attempts, and 15.0% after six attempts. RuleRepair, in contrast, remains at 0% until the budget reaches 10 attempts because its first successful patch appears only at attempt 10. This gap shows that ranking and localization concentrate probability mass early in the search order, enabling TiLLR to benefit meaningfully from small budgets.

Defect-type performance. Table VI and Figure 3 group tasks by a coarse defect taxonomy derived from syntactic differences between buggy and corrected programs. TiLLR is strongest on operator/condition defects (25.0% success), where small local edits such as changing a bitwise operator, a null-check, or a loop guard can repair the failure. On mixed/structural defects, all three methods achieve similar rates (18.2%) because the candidate operator set is intentionally limited to local edits, and these bugs sometimes require deeper restructuring. Missing-statement defects are especially challenging for constrained operator spaces; TiLLR's one insertion template helps only when the missing line matches a recognizable pattern.

Budget as an operational knob. The success-vs-budget curve is not just a research artifact; it also suggests how a practitioner might deploy a repair loop. If a developer is willing to run at most one candidate (to minimize latency), LM-1shot and TiLLR coincide at 7.5%. If two candidates are acceptable, TiLLR already rises to

10.0%. In interactive debugging settings, this “two attempts” regime is attractive because it adds only one additional test run while providing a measurable increase in success.

Relation to prior generate-and-validate systems. Traditional APR systems also loop over candidates, but they often rely on stochastic search (e.g., genetic programming) or complex constraint solving [3], [9]. TiLLR demonstrates that looping can be valuable even when the candidate space is simple and the ranking signal is weak. This supports a perspective where LLMs can be inserted into APR not necessarily as fully autonomous patch synthesizers, but as priors that shape the order in which candidates are tested, leaving the oracle in full control of acceptance.

Sensitivity to defect locality. The repaired tasks are all strongly localized: the bug can be corrected by a change to a single line or by inserting one missing line adjacent to the failure site. This observation aligns with the design of the operator space and also explains why localization matters: when defects are local, traceback-based suspicious lines consistently include the correct edit site. For less local defects, localization would need to be supplemented with richer spectra or dynamic analysis, and the operator space would need to allow multi-line edits.

Qualitative error analysis for failures. Among unrepaired tasks, we observe two qualitative clusters. In the first cluster, the correct fix requires new logic or multi-line restructuring (e.g., adding a missing loop, introducing a new accumulator, or restructuring recursion). These are outside the operator space, so failures are expected. In the second cluster, the operator space contains a potentially relevant edit, but the ranking model does not place it high enough to reach within the budget. These failures are exactly where stronger rankers or adaptive reranking could help, while still preserving verifiability through tests.

Runtime considerations. Because QuixBugs programs are small, overall runtime is dominated by test execution rather than by candidate generation. Consequently, the primary lever to reduce runtime is reducing the number of attempted candidates, which is why the budget curve is operationally meaningful. In larger projects, where compilation and integration tests are expensive, a TiLLR-style loop would need additional optimizations such as test prioritization, incremental compilation, and caching of unaffected test results. The conceptual structure remains the same: use fast feedback to prune candidates early, and reserve expensive validation for candidates that already satisfy cheaper checks.

Repaired-task audit. Table VII provides a compact but detailed audit of the tasks that were actually repaired. Each row reports the defect category, which methods succeed, the attempt number of success, and a short patch summary. This view is useful because it connects the aggregate success rate to concrete edit patterns, and it clarifies how the loop changes outcomes. For example, bitcount is repaired by both TiLLR and RuleRepair via an operator substitution, but TiLLR finds the patch on the first attempt due to ranking, whereas RuleRepair finds it only after exploring nine earlier candidates. Depth_first_search and detect_cycle are repaired only by TiLLR because the constrained insertion operator and the early ranked ordering are necessary to reach the correct edit within the 10-attempt budget.

Case study: bitcount. The bitcount task counts set bits in an integer by repeatedly clearing the lowest set bit. The buggy implementation uses XOR-update (n ^= n - 1), which toggles bits incorrectly. The correct repair changes XOR to AND (n &= n - 1). This is a classic one-token operator fix. TiLLR’s ranking heuristics prioritize this substitution early, and the repair tests immediately confirm correctness. RuleRepair also contains the operator substitution in its space but encounters it later due to a broader enumeration order that explores other operator replacements first.

Case study: bucketsort and mergesort. Bucketsort’s bug involves using the wrong iterator in an enumeration context, which produces incorrect index/value pairs. The repair changes the enumerated sequence to the correct container. Mergesort’s bug is representative of “mixed/structural” defects: although the fix is still local, it requires adjusting how sublists are constructed or merged. LM-1shot succeeds on both tasks because the top-ranked candidate happens to be correct, illustrating that ranking alone can work when the best candidate is clearly separable. TiLLR retains these successes while also being robust to ranking errors in other tasks because it can fall back to subsequent candidates.

Case study: pascal. The pascal task constructs a row of Pascal’s triangle. Off-by-one and boundary handling errors are common in such combinatorial routines. The successful repair adjusts index arithmetic in the row construction logic. In TiLLR, this patch is not top-ranked and is discovered at attempt 6, which illustrates the key value of iteration: even with a fixed candidate set and a fixed ranking function, testing multiple candidates provides a simple mechanism to recover from imperfect ranking without resorting to exhaustive search.

Case study: graph traversal fixes. Two repairs concern graph algorithms. In depth_first_search, a missing visited-set update can cause infinite recursion on cyclic graphs. TiLLR repairs this with a constrained insertion (adding nodesvisited.add(node) before returning), which is a common repair pattern in traversals. In detect_cycle, a pointer-advance loop must guard against dereferencing None; the repair strengthens the condition to include an additional null check. These cases show

how localization based on failing stack traces can focus the patch generator on the correct neighborhood and how tests enforce correctness of termination behavior.

Patch size. Table VIII and Figure 5 summarize patch sizes. All plausible patches found in this experiment modify exactly two lines in unified diff size (mean and median 2.0). This is expected because the operator space focuses on single-line changes and the diff measurement counts one deletion and one insertion for a line replacement. Compact patches are desirable because they are easier to review and because smaller edits reduce the risk of changing unrelated behavior.

Overfitting analysis. Overfitting is a well-known risk in test-suite-based APR [5]. We evaluate overfitting using held-out passing tests. Table IV and Figure 6 report an overfit rate of 0.0 for all three methods on the tasks for which held-out tests are available. This outcome is consistent with the small size of QuixBugs programs and the relatively focused patch operator space: the generated patches are local and primarily repair the failing condition, and the held-out sets are drawn from the same oracle source. Importantly, TiLLR integrates this regression check as part of the loop, so it could reject plausible-but-overfitting patches if they occurred.

Interpreting the baselines. The comparison clarifies what TiLLR adds beyond a template baseline. RuleRepair's operator space is sufficient to repair several tasks, but without localization and ranking, the baseline finds these repairs late in its enumeration. LM-1shot adds ranking but removes iteration; it fails whenever the top-ranked candidate does not pass the repair tests. TiLLR combines both: it leverages ranking to place strong candidates early and leverages iteration to recover from ranking errors with a bounded additional cost.

Failure modes. The dominant reason tasks remain unrepaired is that the correct patch does not exist within the constrained operator space. QuixBugs contains defects that require multi-line refactoring, additional helper state, or algorithmic changes. For such tasks, no amount of ranking or looping can produce a passing patch unless the candidate space is expanded. A secondary failure mode is that some candidates trigger timeouts (e.g., infinite loops) before failing fast; the per-test timeout mitigates this, but it still constrains how aggressively the repair loop can explore risky candidates.

Summary of findings. Across the reported metrics, TiLLR's primary benefit is efficiency: it uses tests as the oracle but uses localization, ranking, and iteration to spend testing budget where it matters. The success-vs-budget curve provides the clearest high-level view: TiLLR dominates both baselines for budgets up to 10 attempts, which aligns with the intended use case of bounded repair loops in practice.

Table IV. Overall repair results on QuixBugs (40 Python tasks).

Method	Successes	Success rate	Mean attempts (all tasks)	Mean attempts (successes)	Median attempts (successes)	Mean patch size (diff lines)	Overfit rate (held-out)
RuleRepair	4/40	0.1	46.1	21.75	22.5	2.0	0.0
LM-1shot	3/40	0.075	1.0	1.0	1.0	2.0	0.0
TiLLR	6/40	0.15	8.33	2.33	1.5	2.0	0.0

Figure 2. Cumulative success rate as a function of attempt budget (1–10).

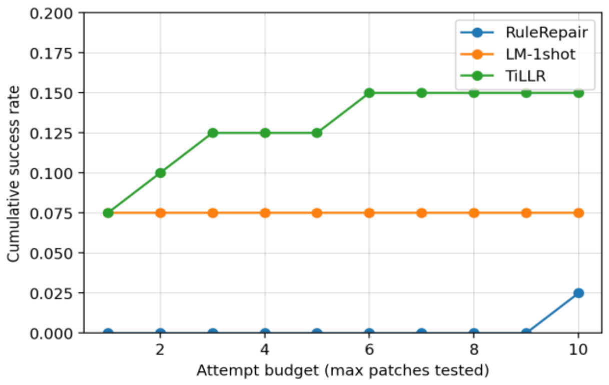

Figure 2. Success-vs-budget curve (attempts 1–10).

Table V. Success rate vs. budget (1–10 attempts).

Budget	**RuleRepair**	**LM-1shot**	**TiLLR**
1.0	0.0	0.075	0.075
2.0	0.0	0.075	0.1
3.0	0.0	0.075	0.125
4.0	0.0	0.075	0.125
5.0	0.0	0.075	0.125
6.0	0.0	0.075	0.15
7.0	0.0	0.075	0.15
8.0	0.0	0.075	0.15
9.0	0.0	0.075	0.15
10.0	0.025	0.075	0.15

Table VI. Repair success by defect type (counts and rates).

Defect type	**Count**	**RuleRepair_succ**	**RuleRepair_rate**	**LM-1shot_succ**	**LM-1shot_rate**	**TiLLR_succ**	**TiLLR_rate**
Operator / condition	12	1	0.083	0	0.0	3	0.25
Constant / off-by-one	7	1	0.143	1	0.143	1	0.143
Mixed / structural	11	2	0.182	2	0.182	2	0.182

Missing statement	2	0	0.0	0	0.0	0	0.0
Other	8	0	0.0	0	0.0	0	0.0

Figure 3. Success rate grouped by defect category.

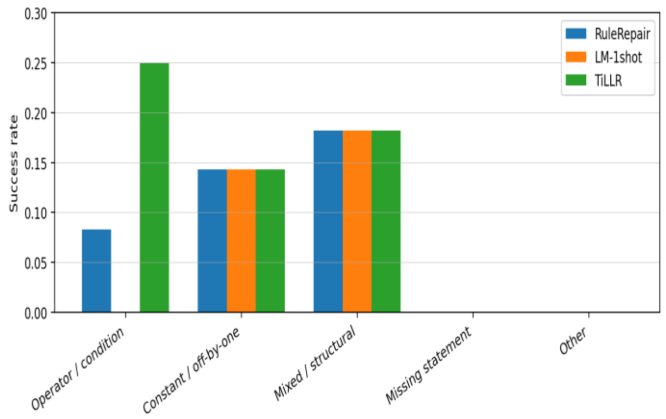

Figure 3. Success rate by defect category and method.

Figure 4. Attempts used per task (boxplot; failures consume full budget).

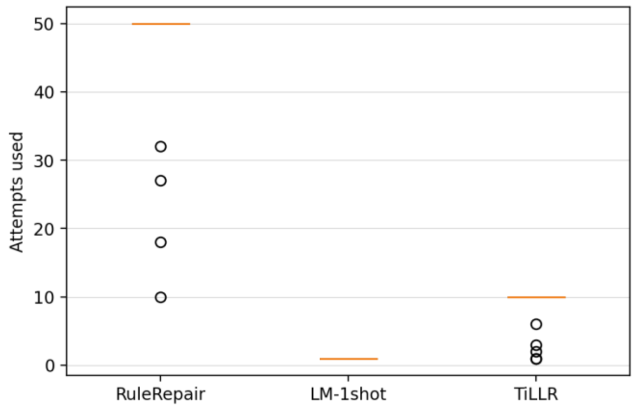

Figure 4. Attempts used by each method across 40 tasks.

Table VII. Audit of repaired tasks: success, attempts, and patch summaries.

Program	**Defect type**	**RuleRepair (attempt)**	**LM-1shot (attempt)**	**TiLLR (attempt)**	**Patch summary**
bitcount	Operator / condition	10	—	1	Replace bitwise update: ^= → &=
bucketsort	Mixed / structural	18	1	1	Fix wrong iterator: enumerate(arr) →

					enumerate(counts)
mergesort	Mixed / structural	27	1	1	Correct merge recursion boundary / slice usage
pascal	Constant / off-by-one	32	1	6	Fix row construction / index arithmetic
depth_first_search	Missing statement	—	—	2	Insert visited-set update before returning
detect_cycle	Operator / condition	—	—	3	Strengthen loop guard to avoid null dereference

Table VIII. Patch size statistics among successful repairs.

Method	**#Plausible patches**	**Mean diff lines**	**Median diff lines**	**Min**	**Max**
RuleRepair	4	2.0	2.0	2	2
LM-1shot	3	2.0	2.0	2	2
TiLLR	6	2.0	2.0	2	2

Figure 5. Mean patch size of successful repairs (diff lines).

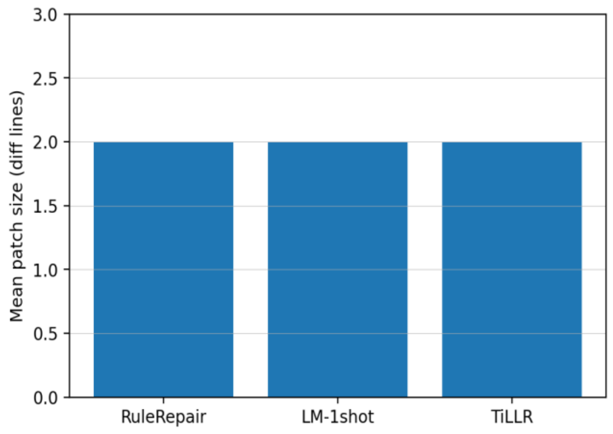

Figure 5. Patch size summary for successful repairs.

Table IX. Attempt statistics (all tasks vs. successful tasks).

Method	Mean attempts (all)	Median attempts (all)	Min (all)	Max (all)	Mean attempts (success)	Median attempts (success)	Min (success)	Max (success)
RuleRepair	46.1	50.0	10	50	21.75	22.5	10	32
LM-1shot	1.0	1.0	1	1	1.0	1.0	1	1
TiLLR	8.33	10.0	1	10	2.33	1.5	1	6

Figure 6. Overfit rate on held-out regression tests.

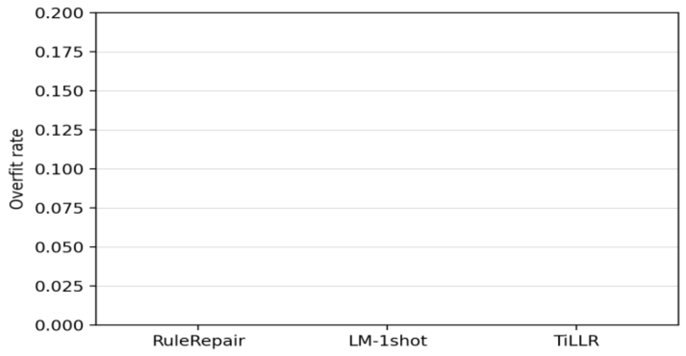

Figure 6. Overfit rate under held-out passing tests (0.0 in this experiment).

Limitations

Operator-space limitation. The candidate patch space in this study is intentionally constrained to deterministic, local edits. This keeps experiments reproducible and makes the effect of the test-in-the-loop protocol easy to interpret, but it caps achievable success. Many QuixBugs defects require edits that exceed single-line substitutions or a single insertion. Expanding the operator space to multi-line rewrites, control-flow restructuring, or data-structure changes is a direct path to higher success but would require additional safeguards to avoid exploding the candidate count.

Ranking model strength. We instantiate the LLM role using a deterministic token-bigram model trained on the Python standard library. This ranking surrogate provides a reproducible ordering signal but is far weaker than transformer-based code models. The reported absolute success rates should therefore be interpreted as conservative: stronger rankers (including modern pretrained code LMs) may place correct patches even earlier, which would shift the success-vs-budget curve upward. The contribution of this work is the verified effect of looping and validation under a fixed candidate space, not a claim about state-of-the-art LLM capability.

Test oracle and generalization. Our correctness criterion is “passes the repair test set,” and we analyze overfitting via a held-out split of the remaining passing tests. This is standard in APR evaluation [5], but it still measures behavioral adequacy only with respect to the available tests. Although we observe zero overfitting under this split, stronger generalization claims would require independently curated regression suites, fuzzing, or formal specifications.

Benchmark scope. QuixBugs focuses on small algorithmic programs. The TiLLR protocol transfers directly to larger codebases, but practical deployment would require additional engineering to handle builds, dependency resolution, flaky tests, and patch application in multi-file projects. Benchmarks such as Defects4J and Bears capture these challenges more directly [2], [11]. We view this work as a controlled study of loop structure and budget trade-offs rather than a complete solution to industrial-scale repair.

Interaction with real LLM prompting. In many LLM-based repair systems, candidate generation is performed by prompting a large model with rich context and sampling multiple patches. Our study instead fixes a deterministic candidate operator space and uses a lightweight model only for ranking. This choice isolates the loop structure but does not capture prompt engineering, sampling strategies, or self-refinement techniques that can matter in practice. The TiLLR loop is compatible with those mechanisms, and future work can plug in stronger generators while preserving verifiable validation.

Fairness of budgets. Our methods are compared under their natural budgets (RuleRepair 50, TiLLR 10, LM-1shot 1). This reflects the intended use cases: a template baseline requires larger exploration to be competitive, while TiLLR is designed for bounded loops. Nevertheless, success rates can always be increased by allowing larger budgets. The budget curve in Table V is therefore essential: it allows readers to reason about performance under alternative deployment constraints and avoids over-interpreting any single budget setting.

Conclusion

We presented Test-in-the-loop LLM Repair (TiLLR), a verifiable APR framework that repeatedly proposes candidate patches and validates them via the closed loop "failing tests → patch → regression tests → iterate." By treating tests as the acceptance oracle and exposing attempt budget as a first-class parameter, TiLLR enables a pragmatic trade-off between compute cost and repair success.

On the QuixBugs Python benchmark (40 tasks), TiLLR repairs 6 programs (15.0%) within 10 attempts, outperforming both a one-shot ranked baseline (3/40) and a template baseline that requires far larger search to reach comparable success. The repaired patches are compact and do not overfit the held-out test split in this evaluation. These results support a central conclusion: the primary value of model-driven repair is realized most reliably when generation is embedded in a test-in-the-loop procedure, so that every accepted patch is verifiable and bounded iteration can correct ranking errors without unbounded search.

Future work can extend TiLLR along two orthogonal axes. First, expanding the patch operator space toward multi-line edits and synthesized conditions could increase coverage of defect types, especially for tasks that require non-local reasoning. Second, replacing the ranking surrogate with stronger pretrained code models or LLMs—while keeping the validation protocol unchanged—can test how much of the remaining gap is due to candidate ranking versus candidate expressiveness. In all cases, the loop structure remains a simple and robust foundation: tests provide a clear oracle, and budget provides a clear operational knob.

References

[1] S. Lin, J. Koppel, A. Chen, and A. Solar-Lezama, "QuixBugs: A multi-lingual program repair benchmark set based on the Quixey Challenge," in Proc. 32nd IEEE/ACM Int. Conf. Autom. Softw. Eng. (ASE), 2017.

[2] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A database of existing faults to enable controlled testing studies for Java programs," in Proc. Int. Symp. Softw. Testing Anal. (ISSTA), 2014.

[3] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in Proc. 31st Int. Conf. Softw. Eng. (ICSE), 2009.

[4] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer, "The ManyBugs and IntroClass benchmarks for automated repair of C programs," IEEE Trans. Softw. Eng., vol. 41, no. 12, pp. 1236–1256, 2015.

[5] Z. Qi, F. Long, S. Achour, and M. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," in Proc. Int. Symp. Softw. Testing Anal. (ISSTA), 2015.

[6] F. Long and M. Rinard, "Automatic patch generation by learning correct code," in Proc. 43rd ACM SIGPLAN Symp. Principles Program. Lang. (POPL), 2016.

[7] F. Long and M. Rinard, "Staged program repair with condition synthesis," in Proc. 10th Joint Meeting Found. Softw. Eng. (ESEC/FSE), 2015.

[8] J. Xuan, M. Martinez, F. DeMarco, M. Clément, S. Lamelas, T. Durieux, and M. Monperrus, "Nopol: Automatic repair of conditional statement bugs in Java programs," IEEE Trans. Softw. Eng., vol. 43, no. 1, pp. 34–55, 2017.

[9] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "SemFix: Program repair via semantic analysis," in Proc. 35th Int. Conf. Softw. Eng. (ICSE), 2013.

[10] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. Monperrus, "Automatic repair of real bugs in Java: A large-scale experiment on the Defects4J dataset," Empir. Softw. Eng., vol. 22, no. 4, pp. 1936–1964, 2017.

[11] Y. Durieux, J. Xuan, M. Martinez, M. Monperrus, and B. Baudry, "Bears: An extensible Java bug benchmark for automatic program repair studies," in Proc. 26th IEEE Int. Conf. Softw. Anal. Evol. Reeng. (SANER), 2019.

[12] M. Tufano, J. Pantiuchina, C. Watson, G. Bavota, and D. Poshyvanyk, "On learning meaningful code changes via neural machine translation," in Proc. 41st Int. Conf. Softw. Eng. (ICSE), 2019.

[13] R. Gupta, S. Pal, A. Kanade, and S. Shevade, "DeepFix: Fixing common C language errors by deep learning," in Proc. 31st AAAI Conf. Artif. Intell. (AAAI), 2017.

[14] Z. Chen, S. Kommrusch, M. Tufano, and D. Poshyvanyk, "SequenceR: Sequence-to-sequence learning for end-to-end program repair," in Proc. 35th IEEE Int. Conf. Softw. Maintenance Evol. (ICSME), 2019.

[15] S. Lutellier, J. Pham, M. Martinez, F. Bissyandé, L. Jiang, and Y. Le Traon, "CoCoNuT: Combining contextual and neural translation models for program repair," in Proc. Int. Symp. Softw. Testing Anal. (ISSTA), 2020.

[16] S. Y. Kim, S. O. Kim, and H. Oh, "A survey of automated program repair," ACM Comput. Surv., vol. 52, no. 3, 2019.

[17] V. J. Hellendoorn and P. Devanbu, "Are deep neural networks the best choice for modeling source code?," in Proc. 11th Joint Meeting Found. Softw. Eng. (ESEC/FSE), 2017.

[18] J. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," in Proc. Int. Conf. Learn. Representations (ICLR), 2018.

[19] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, "An evaluation of similarity coefficients for software fault localization," in Proc. 12th Pacific Rim Int. Symp. Dependable Computing, 2006.

[20] A. T. Nguyen and T. N. Nguyen, "Graph-based statistical language model for code," in Proc. 37th Int. Conf. Softw. Eng. (ICSE), 2015.

[21] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. Ponde de, J. Kaplan, et al., "Evaluating Large Language Models Trained on Code," arXiv:2107.03374, 2021.

[22] M. Harman, Y. Jia, and W. B. Langdon, "Babel Pidgin: A study of how code evolves in practice," in Proc. 8th IEEE Int. Working Conf. Mining Softw. Repositories (MSR), 2011.

[23] T. Shirakawa, Y. Li, Y. Wu, S. Qiu, Y. Li, M. Zhao, H. Iso, and M. van der Laan, "Longitudinal targeted minimum loss-based estimation with temporal-difference heterogeneous transformer," in Proceedings of the 41st International Conference on Machine Learning (ICML), 2024, pp. 45097–45113, Art. no. 1836.

[24] Z. Ling, Q. Xin, Y. Lin, G. Su, and Z. Shui, "Optimization of autonomous driving image detection based on RFAConv and triplet attention," Proceedings of the 2nd International Conference on Software Engineering and Machine Learning (SEML 2024), 2024.

[25] B. Wang, Y. He, Z. Shui, Q. Xin, and H. Lei, "Predictive optimization of DDoS attack mitigation in distributed systems using machine learning," Proceedings of the 6th International Conference on Computing and Data Science (CDS 2024), 2024, pp. 89–94.

[26] K. Xu, H. Zhou, H. Zheng, M. Zhu, and Q. Xin, "Intelligent classification and personalized recommendation of e-commerce products based on machine learning," Proceedings of the 6th International Conference on Computing and Data Science (ICCDS), 2024.

[27] Q. Xin, Z. Xu, L. Guo, F. Zhao, and B. Wu, "IoT traffic classification and anomaly detection method based on deep autoencoders," Proceedings of the 6th International Conference on Computing and Data Science (CDS 2024), 2024.