

Debug or Regenerate? Test-Guided Selection for LLM Code Repair

Atharv Kulkarni¹ Archiki Prasad¹ Vivek Srikumar²
Elias Stengel-Eskin³ Hyunji Lee¹ Mohit Bansal¹

¹UNC Chapel Hill ²University of Utah ³The University of Texas at Austin

{athkulk, archiki, hyunjil, mbansal}@cs.unc.edu
svivek@cs.utah.edu esteng@cs.utexas.edu

 PATSSSEL Code

Abstract

Large language models (LLMs) are increasingly used as coding assistants, but the programs they generate often fail. Such failures can be addressed by debugging the existing implementation or regenerating a new solution from the problem specification, with existing repair workflows typically defaulting to debugging. We find that the effectiveness of debugging and regeneration depends strongly on initial program quality, suggesting that no single repair strategy is universally optimal. Motivated by this observation, we present PATSSEL (Program-Adaptive Test-guided Repair Strategy Selection), a test-guided repair selection framework that dynamically chooses between debugging and regeneration. Rather than committing to a single repair strategy, PATSSEL generates debugged and regenerated candidates for each buggy program, evaluates them using generated verifier tests, and selects the stronger candidate, leveraging the complementary strengths of debugging and regeneration. Experiments on LeetCodeDataset, LiveCodeBench, and HE+Fix show that PATSSEL achieves average absolute improvements of 2.7% with Qwen3-8B and 3.2% with Llama3-8B over the best single-strategy baselines under the same inference compute budget and routing using gold unit tests. The gains persist across repair models, routing models, and stronger debugger models. Further analysis shows that PATSSEL benefits from the complementary strengths of debugging and regeneration, highlighting the importance of adaptive strategy selection for reliable LLM-based code repair.

1 Introduction

Large language models (LLMs) are increasingly deployed as coding assistants, yet ensuring the correctness of generated programs remains a fundamental challenge (Sonwane et al., 2025; Le et al., 2022; Zeng et al., 2025). When a generated program fails, there are two natural ways to obtain a

correct solution. One can *debug* the existing implementation, refining the current draft while preserving its existing structure and reasoning, or *regenerate* a new solution directly from the problem specification. These alternatives allocate repair effort differently: debugging conditions on the previous attempt, whereas regeneration reasons independently from scratch. While existing repair pipelines typically default to debugging (Chen et al., 2023; Fan et al., 2023), it is unclear whether preserving the existing implementation is always the right choice. This raises a central question: *when should a failed program be repaired through debugging, and when is it better to abandon the current draft and generate a new solution?*

To answer this question, we systematically compare debugging and regeneration across a diverse collection of naturally occurring model failures. Our analysis reveals that neither strategy consistently dominates the other (Figure 1). Instead, repair effectiveness depends strongly on initial solution quality. When the generated program already captures much of the correct logic, debugging is often more successful because it preserves useful structure while making targeted modifications. In contrast, when the initial solution is fundamentally flawed and requires substantial algorithmic revision, regeneration is more effective because it can explore entirely new solution trajectories. Moreover, repaired programs produced through debugging exhibit substantially higher test-case overlap with the original buggy solution than those produced through regeneration, suggesting that debugging primarily preserves and refines existing behavior, whereas regeneration is more likely to explore alternative solution paths. These findings suggest that successful code repair requires determining whether the current implementation provides a useful foundation for correction or should be abandoned. This suggests that repair strategy should depend on the quality of the existing program.

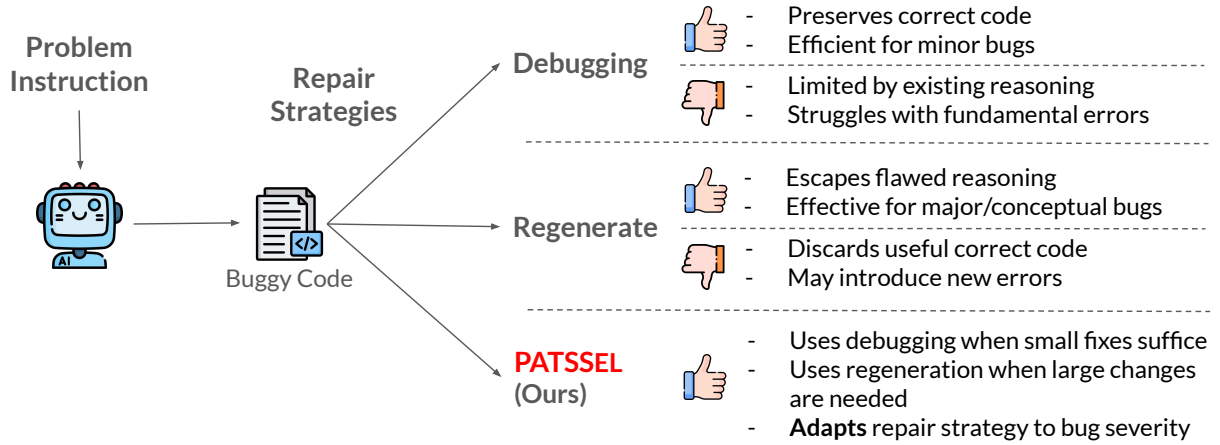


Figure 1: Various repair strategies for fixing buggy code generated by coding agents exhibit complementary strengths and weaknesses. PATSSSEL achieves strong performance by adaptively selecting debugging or regeneration based on the severity of the underlying bug.

Motivated by these observations, we propose PATSSSEL (Program-Adaptive Test-guided Repair Strategy Selection), a test-guided repair selection framework using execution feedback. Rather than committing to a single repair strategy, PATSSSEL first explores both repair paths by constructing two competing candidates for each buggy program: a debugged candidate that refines the existing implementation and a regenerated candidate produced directly from the original problem specification. To determine which candidate is more likely to be correct, PATSSSEL generates verifier tests and executes both candidates on them. The resulting execution traces provide behavioral evidence for comparing the relative reliability of alternative repairs. Because gold evaluation tests are unavailable at deployment time, the verifier tests are produced by a learned unit-test generator trained from execution feedback. By selecting repairs based on their observed behavior under generated tests, PATSSSEL adaptively chooses between debugging and regeneration without requiring oracle test suites.

We evaluate PATSSSEL on LeetCodeDataset (Xia et al., 2025), LiveCodeBench (Jain et al., 2024) and HE+Fix (Prasad et al., 2025; Muennighoff et al., 2024), and using Qwen3-8B (Yang et al., 2025) and Llama3-8B (Grattafiori et al., 2024) as repair models and compare it against single-strategy repair baselines that perform either regeneration-only or debugging-only repair with the same inference budget (two repair attempts) and the same routing mechanism used by PATSSSEL. Using gold test cases for routing, PATSSSEL achieves average absolute improvements of 2.7% with Qwen3-8B and

3.2% with Llama3-8B over the best single-strategy baselines across the three benchmarks.

We further find that stronger verifiers lead to larger gains over single-strategy baselines, highlighting the importance of accurate verification for repair strategy selection. Analyzing performance across pass-rate buckets reveals that PATSSSEL is effective across diverse bug severities. Compared to regeneration, the largest improvements occur in the highest pass-rate bucket, where debugging is particularly effective at making localized fixes. In contrast, compared to debugging, the largest gains occur in the lowest pass-rate bucket, where regeneration is better suited to correcting severely flawed solutions. These findings suggest that PATSSSEL successfully combines the complementary strengths of debugging and regeneration rather than relying on a single repair strategy. This complementarity holds across verifiers. Finally, we observe that the benefits of PATSSSEL persist with a stronger external debugger, showing that the framework is robust to debugger choice.

2 Related Work

Code Repair and Self-Debugging. Recent work has demonstrated that execution feedback provides an effective supervision signal for improving code models, enabling reinforcement learning from unit-test outcomes and other verifiable signals (Le et al., 2022; Shao et al., 2024; Kulkarni and Srikumar, 2025). Building on this paradigm, a growing body of work studies automated repair and self-debugging of LLM-generated programs. Prior approaches improve repair models through iterative

refinement, synthetic debugging trajectories, or execution-grounded training objectives (Fan et al., 2023; Jiang et al., 2025; Pham et al., 2025; Sonwane et al., 2025). While these methods demonstrate that many incorrect programs can be successfully repaired, self-repair is not uniformly reliable: improvements are highly instance-dependent, and repeated refinement may fail to recover correct solutions (Olausson et al., 2024). Motivated by these findings, we study a complementary question: rather than assuming that every failed program should be repaired through debugging, we investigate when debugging is preferable to regeneration and how a system can choose between these two repair operations.

Test Generation and Verification for Code Models. Generated tests have become an important mechanism for supervising, evaluating, and selecting code solutions. UTGen (Prasad et al., 2025) showed that execution feedback can be used to train a unit-test generator whose tests support program verification and reranking. ACECoder (Zeng et al., 2025) scales code RL with automated test-case synthesis and execution-grounded rewards. CURE (Wang et al., 2025) co-evolves a coder and a unit-test generator through execution feedback, while SOLVER (Lin et al., 2025) improves solvers and verifiers through solve-verify self-play. These approaches show that generated tests can provide useful behavioral signals even when gold evaluation tests are unavailable. We build on this line of work by training a unit-test generator using execution-grounded feedback. However, our goal differs from prior approaches. Rather than using generated tests solely to improve or verify a single candidate solution, we use them to serve as behavioral evidence for selecting between a debugged candidate and a regenerated candidate, providing a practical approximation to oracle repair-strategy selection without access to hidden evaluation tests.

3 Regeneration vs. Debugging

In this section, we investigate whether debugging is the most effective default strategy for repairing buggy code. To enable a comprehensive comparison across diverse bug types, we first construct an evaluation dataset using outputs generated by models of varying sizes (Section 3.1). We then compare two repair strategies, debugging and regeneration, and analyze their relative performance by initial program quality in Section 3.2.

3.1 Experimental Setup

Constructing Buggy Programs. To obtain a diverse collection of buggy programs, we generate solutions for problems in the LeetCodeDataset (Xia et al., 2025) using Qwen 2.5 (Hui et al., 2024) models at six scales: 0.5B, 1.5B, 3B, 7B, 14B, and 32B parameters. Using models of varying capabilities produces a broad spectrum of errors, ranging from simple implementation mistakes to more substantial reasoning failures. For each problem, we execute the generated solution against the reference test cases and retain only programs that fail at least one test. Each resulting instance consists of the problem specification, the generated buggy program, the reference tests, and the corresponding execution outcomes. We partition the collected instances into training and evaluation sets for the analyses and experiments presented in the remainder of the paper. More details in Appendix A.

Models. We experiment using Qwen3 8B (Yang et al., 2025) and Llama3-8B (Grattafiori et al., 2024) as base models for regeneration and debugging to evaluate generalization across repair-model families. To ensure that differences between regeneration and debugging are not solely attributable to the debugging capability of the underlying model, we additionally evaluate an RL-trained debugger specialized for code repair. Details of the debugger training procedure are provided in Section 4.2.

3.2 Motivating Analysis

For each buggy program, we compare debugging and regeneration using the same base Qwen3-8B model, allowing us to isolate the effect of the repair strategy. *Debugging* conditions on the problem description, the buggy program, and a failing test case, and asks the model to modify the existing code to fix the bug. In contrast, *regeneration* conditions only on the problem specification and generates a new solution from scratch without access to the buggy program. We evaluate both repaired programs on the same reference test suite and compare their performance across different levels of initial program quality.

Figure 3 shows that the relative effectiveness of debugging and regeneration depends strongly on the original pass rate of the buggy program. Regeneration substantially outperforms debugging for low-quality programs, with a 25.8-point advantage in Q1. However, this advantage steadily decreases as the quality of the original program improves,

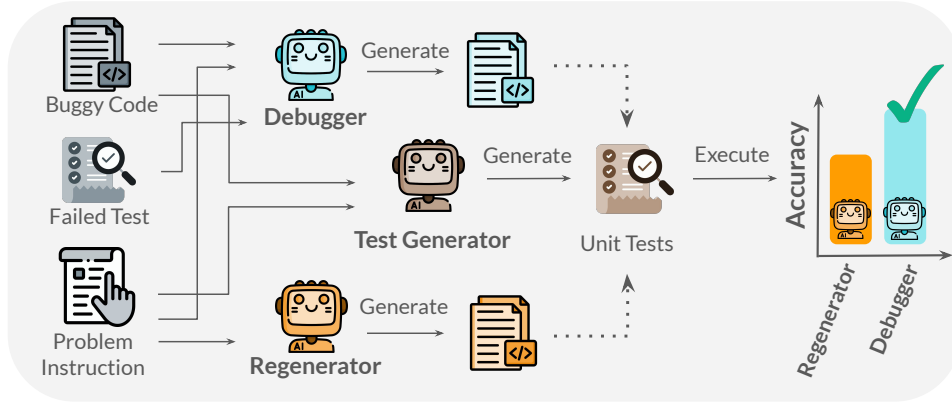


Figure 2: Overview of PATSSEL. Given a buggy code, failed test, and problem instruction, we generate two candidate repairs using complementary strategies: debugging and regeneration. We also use a test generator to produce verifier unit tests. Both repair candidates are executed on the generated tests, and the candidate achieving the higher pass rate is selected as the final output.

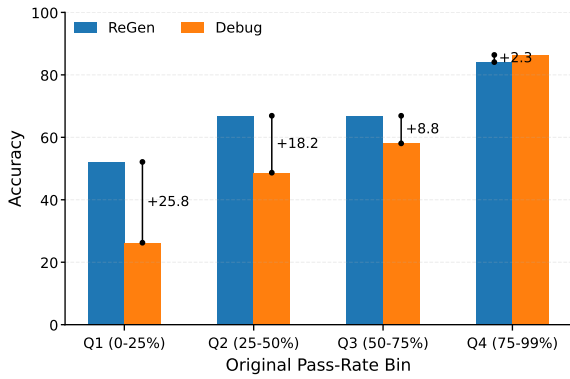


Figure 3: Performance of regeneration and debugging across original pass-rate bins (Q1: 0–25, Q2: 25–50, Q3: 50–75, Q4: 75–99). We compare one ReGen and one Debug attempt using the same base Qwen3-8B model, isolating the effect of repair strategy.

and in the highest pass-rate bucket (Q4), debugging instead outperforms regeneration by 2.3 points. These results suggest that regeneration is more effective when the initial solution contains substantial errors, whereas debugging becomes preferable when the existing program already provides a strong foundation for repair.

4 Program-Adaptive Test-guided Repair Strategy Selection

In this section, we introduce PATSSEL (Program-Adaptive Test-guided Repair Strategy Selection), a test-guided repair framework that treats code repair as a repair-strategy selection problem, to combine the complementary strengths of debugging and regeneration as shown in Section 3.2. Rather

than committing to either strategy a priori, PATSSEL generates both a debugging-based repair and a regeneration-based repair, then uses generated verifier tests to determine which candidate is more likely to be correct (Figure 2). We first describe the overall inference procedure of PATSSEL in Section 4.1. Although PATSSEL is compatible with any debugging, generation, and test-generation models including the base model, we additionally train specialized debugger and unit-test generation models to maximize performance. The training procedure for the debugger is described in Section 4.2, and the unit-test generator is described in Section 4.3, including training objectives.

4.1 Repair Strategy Selection at Test-Time

As shown in Figure 2, at inference time the system receives a problem instruction x and a buggy code y_0 . Rather than committing to a single repair strategy, it generates two competing repair candidates: a debugged candidate (y_{dbg}) that refines the existing implementation and a regenerated candidate (y_{gen}) generated by resampling from the problem specification.

The *regeneration* branch produces a fresh solution: $y_{\text{gen}} \sim \pi_{\text{gen}}(\cdot \mid x)$ conditioning only on the problem instruction. The *debugging* branch produces a repaired solution: $y_{\text{dbg}} \sim \pi_{\text{dbg}}(\cdot \mid x, y_0, t)$. The failing test t is drawn from the same test source used for verification in each experimental setting: Base-generated tests, RL-generated tests, or the gold test suite for the Base, RL, and Gold settings, respectively.

A unit-test generator then gets the problem in-

struction and buggy code as input and produces unit tests: $Z = \{z_1, \dots, z_m\}$. Both repair candidates are executed against the unit tests Z , producing pass-rate estimates:

$$s_{\text{gen}} = p(y_{\text{gen}}, Z), \quad s_{\text{dbg}} = p(y_{\text{dbg}}, Z).$$

The final repair is selected by comparing these pass rates and choosing the one with higher pass rate.

4.2 RL Debugger Training

The debugger is trained to improve an existing buggy program. Each training instance consists of a problem instruction x , a buggy code y_0 , a reference test suite Z , and at least one failing test case $z \in Z$. The debugger receives the buggy program and the failing test as input and generates a repaired program $\hat{y}$. The repaired program is then executed against the full reference test suite Z .

We train the debugger with GRPO (Shao et al., 2024) using an execution-based reward that measures pass-rate improvement over the original buggy program. For each training instance, the debugger samples a group of candidate repairs $\{\hat{y}_1, \dots, \hat{y}_G\}$ from the current policy. For a candidate repair $\hat{y}_i$, the reward is

$$R_{\text{debug}}(\hat{y}_i; y_0, Z) = 1000 \times (p(\hat{y}_i, Z) - p(y_0, Z)),$$

where $p(\hat{y}_i, Z)$ denote the fraction of tests in Z passed by program $\hat{y}_i$. This reward directly encourages functional improvement over the starting buggy program y_0 . A repair receives positive reward when $\hat{y}_i$ fixes more tests than the original code y_0 , zero reward when it leaves the pass rate unchanged, and negative reward when it introduces regressions. Invalid outputs that cannot be parsed into the required function or class receive no useful reward because they fail execution.

4.3 RL Unit-Test Generator Training

As gold test cases are typically unavailable at inference time, to obtain verifier tests for estimating program correctness and selecting between repair strategies, we train a unit-test generator. The unit-test generator produces executable tests that serve as a routing signal for PATSSEL. We train this component using GRPO (Shao et al., 2024). Each training instance consists of a problem specification x , a buggy program y_0 , a reference solution, starter code, and the program entry point. The test generator receives the problem specification, starter

code, buggy program, and entry point and generates a suite of executable unit tests. The generated tests are then executed on the reference solution to compute the reward based on three objectives: (1) producing a valid executable test suite, (2) generating a sufficient number of informative tests, and (3) ensuring that the generated assertions agree with the behavior of the reference implementation.

Let $\rho(G) = c/|G|$, where c is the number of generated assertions in generated unit tests G that pass on the reference solution. With minimum suite size τ , we use

$$R_{\text{test}} = \begin{cases} -200, & \text{invalid}(G), \\ 0, & |G| < \tau, \\ 1000\rho(G) + 20 \min(c, 10), & \text{otherwise.} \end{cases} \quad (1)$$

The first term rewards semantic correctness of the generated tests, while the second gives a small bonus for producing multiple correct assertions. The minimum-assertion constraint discourages degenerate suites that contain only one easy test. The resulting test generator learns to produce executable behavioral probes rather than free-form explanations or informal test descriptions (Prasad et al., 2025; Wang et al., 2025).

5 Experiments

5.1 Setup

Evaluation Datasets. We evaluate on three code-repair benchmarks: LeetCodeDataset, LiveCodeBench (LCB), and HE+Fix. HE+Fix (Prasad et al., 2025; Muennighoff et al., 2024) directly provides buggy functions together with reference solutions, entry points, and test suites. We also evaluate on the LCB self-repair setting using the release_v1 split, which consists of 400 problems released between May 2023 and March 2024 (Jain et al., 2024). Unlike HE+Fix, LCB consists of std-in/stdout programming tasks and does not provide buggy programs. To construct repair instances, we first generate solutions using Qwen3 models (Yang et al., 2025) at four scales (1.7B, 4B, 8B, and 14B), execute them on the official test cases, and retain the failing solutions as buggy programs. Details of LeetCodeDataset construction are provided in Section 3.1 and more details of dataset is in Appendix A. For LeetCodeDataset and LCB, we further partition evaluation instances into four pass-rate quartiles (Q1–Q4) based on the buggy program’s initial pass rate.

Training Details. PATSSEL consists of three components: a generation model (GEN) that regenerates solutions from the problem specification, a debugging model (DEBUG) that repairs buggy code, and a test-generation model (TEST GEN) that produces verifier unit tests. For both GEN and DEBUG, if not specified, we use the base Qwen3-8B model (Base) and an RL-trained Qwen3-8B debugger (RL), respectively. For TEST GEN, we use Qwen3-4B-Instruct-2507 and its RL-trained counterpart. For the Llama experiments, we use Llama3-8B (Grattafiori et al., 2024) as the common backbone for GEN, DEBUG, and TEST GEN, and separately train Llama3-8B debugger and test-generator models using the corresponding objectives in Sections 4.2 and 4.3. In addition, for analysis experiments, we evaluate a stronger debugging model, FrogMini-14B-2510 (Sonwane et al., 2025), an RL-trained model built on top of Qwen3-14B.

Both learned components are trained with GRPO using HuggingFace TRL (von Werra et al., 2020). The debugger is optimized using the pass-rate improvement reward (Section 4.2), while the test generator is optimized with the test correctness reward defined in Section 4.3. For both models, we train LoRA adapters with rank 16, alpha 32, and dropout 0.05 on the attention and MLP projection layers. We train for one epoch with learning rate 10^{-5} , batch size 1, gradient accumulation 2, KL coefficient $\beta = 0.02$, and 4 sampled completions per prompt. For debugger training, we use a maximum prompt length of 2048 tokens and completion length of 1024 tokens; for test-generator training, we use 3072 and 1800 tokens, respectively.

Baselines. We compare PATSSEL against two single-strategy repair baselines: GEN, which regenerates a solution from the problem specification, and DEBUG, which repairs the given buggy program. To control for inference-time compute, we evaluate all methods under setting where compute cost matches. Specifically, each baseline produces two independent candidates using the same repair strategy, and the final prediction is selected using the same unit-test-based routing procedure employed by PATSSEL. Consequently, all compared methods use the same number of model calls and incur comparable inference costs.

Metrics. We report accuracy, where accuracy is computed from the benchmark reference-test outcomes of the selected repair. For Q1–Q4, scores are micro-averaged over all reference tests within

each bucket, while MICRO AVG. is computed over all reference tests in the full evaluation set. For PATSSEL, the regenerated and debugged candidates are selected using their verifier-test pass rates. To *match the inference budget*, ReGen and Debug each generate *two* independent candidates and use the same verifier-based selection procedure.

5.2 Results

Table 1 reports accuracy on LeetCodeDataset (Xia et al., 2025), LiveCodeBench (LCB) (Jain et al., 2024), and HE+Fix (Prasad et al., 2025; Muenighoff et al., 2024). LeetCodeDataset serves as the in-domain benchmark used for training both the RL debugger DEBUG and RL test generator TEST GEN, while HE+Fix and LCB evaluate out-of-domain generalization. For the generation model, we use the corresponding base Qwen3-8B or Llama3-8B model. We compare PATSSEL against single-strategy baselines that rely exclusively on regeneration or debugging. We further evaluate three routing settings depending on TEST GEN, the verifier used for candidate selection: Base (the pretrained test generator), RL (the RL-trained test generator), and Gold (the ground-truth evaluation tests). We discuss the Qwen3-8B results in detail below; the Llama3-8B results exhibit the same overall trend, showing that the benefits of repair-strategy selection transfer across models.

Routing consistently outperforms single-strategy repair. Across all benchmarks and verifier settings, PATSSEL, which routes between regeneration and debugging, consistently achieves stronger or comparable performance than relying on either regeneration or debugging alone, while using the same compute budget and routing. This demonstrates that PATSSEL effectively leverages the complementary strengths of both approaches.

On HE+Fix and LiveCodeBench, using gold verifier tests, PATSSEL improves accuracy by 6.7% and 2.8% over ReGen (routing between two regenerated candidates), and by 2.3% and 3.4% over Debug (routing between two debugging attempts), respectively. Similar gains are observed on LeetCodeDataset, where PATSSEL improves accuracy by 3.1% over ReGen and 10.1% over Debug. The gains largely persist when replacing gold tests with verifier tests generated by our RL-trained test generator. On HE+Fix, PATSSEL improves by 5.9% over ReGen and 3.1% over Debug, while on LiveCodeBench it improves by 2.7% over ReGen

Model	Test Gen	Strategy	LeetCodeDataset					LiveCodeBench					HE+Fix	
			Q1	Q2	Q3	Q4	Micro Avg.	Q1	Q2	Q3	Q4	Micro Avg.	Micro Avg.	
Qwen3-8B	Base	ReGen	57.6	69.7	68.4	87.9	70.7	11.5	46.2	60.3	71.0	22.2	89.1	
		Debug	37.6	58.6	69.5	90.1	63.6	9.9	45.7	66.5	90.1	22.4	92.7	
		PATSSSEL	53.5	67.0	71.6	90.7	70.4	11.7	51.9	66.8	91.2	24.5	94.7	
	RL	ReGen	58.3	67.8	70.4	88.4	71.0	12.1	50.7	58.2	67.4	22.8	89.3	
		Debug	39.9	59.0	66.7	89.1	63.3	11.0	50.3	68.9	89.0	23.9	92.1	
			PATSSSEL	54.7	67.4	71.0	90.2	70.6	12.9	55.8	68.3	82.2	25.5	95.2
Llama3-8B	Base	ReGen	33.0	40.0	45.1	47.4	40.9	11.2	36.6	36.2	51.4	19.9	76.3	
		Debug	10.9	30.8	33.9	41.6	28.5	5.8	12.9	23.2	90.1	15.5	54.7	
		PATSSSEL	33.1	42.8	42.7	49.2	41.6	11.0	35.6	46.4	97.2	24.6	77.4	
	RL	ReGen	33.0	40.0	45.2	47.4	40.9	8.4	27.0	31.9	45.8	16.0	74.2	
		Debug	10.9	30.4	34.3	42.7	28.8	5.2	11.9	21.7	90.1	14.9	47.9	
			PATSSSEL	33.1	42.8	42.8	49.2	41.6	10.0	27.7	37.1	95.7	21.9	77.4
	Gold	ReGen	35.1	46.0	50.7	61.5	47.6	12.4	42.6	41.4	58.6	22.6	80.1	
		Debug	19.8	37.4	47.6	55.1	38.9	7.0	15.8	28.6	91.4	17.2	48.5	
		PATSSSEL	40.3	50.0	52.1	60.8	50.3	13.8	45.5	54.3	95.7	28.3	81.3	

Table 1: Overall performance on LeetCodeDataset, LiveCodeBench, and HE+Fix using Qwen3-8B and Llama3-8B repair models. TEST GEN indicates the model used to generate verifier tests for routing, while STRATEGY denotes the repair approach. All methods use the same inference budget and routing mechanism. *ReGen* and *Debug* each generate two candidates using regeneration or debugging, respectively, and select the candidate with the higher verifier-test pass rate. PATSSSEL generates one candidate from each strategy and selects the better candidate based on verifier-test pass rate.

and 1.6% over Debug. On LeetCodeDataset, performance remains comparable to ReGen (-0.4%) while maintaining a substantial 7.3% advantage over Debug. Even a simple zero-shot test generator based on the base model is sufficient to obtain similar benefits. Under this setting, PATSSSEL improves by 5.6% and 2.0% over ReGen and Debug on HE+Fix, respectively; by 2.3% over ReGen and 2.1% over Debug on LiveCodeBench; and achieves performance comparable to ReGen (-0.3%) while outperforming Debug by 6.8% on LeetCodeDataset.

Taken together, these results indicate that regeneration and debugging succeed on complementary subsets of failures when using PATSSSEL. Consequently, selecting between the two strategies is more effective than committing to either strategy alone, even when both approaches use the *same number of repair attempts and the same routing budget*. Moreover, this complementarity generalizes across benchmarks and remains observable with automatically generated verifier tests.

Stronger verifier quality leads to larger gains from PATSSSEL. Comparing different verifier models (TEST GEN) reveals that as verifier quality

improves, the performance gap between PATSSSEL and single-strategy repair baselines widens. For example, on LiveCodeBench, the average improvement of PATSSSEL over single-strategy repair is 3.1% when using gold verifier tests, compared to 2.2% when using the Base Test Gen. This suggests that higher-quality verifier tests enable more accurate routing decisions and allow PATSSSEL to better exploit the complementary strengths of regeneration and debugging. At the same time, the RL-trained Test Gen consistently outperforms the Base Test Gen and substantially narrows the gap to oracle routing with gold tests. This result indicates that training the verifier with execution feedback produces reliable routing signals, enabling effective strategy selection without requiring access to evaluation test cases at inference time.

Routing is effective across a wide range of bug severities. The bucketed results (Q1–Q4) on LeetCodeDataset and LiveCodeBench, where instances are grouped by the original program pass rate, show that PATSSSEL consistently outperforms both single-strategy repair approaches across all levels of bug severity. The largest gains over ReGen are observed in the highest pass-rate bucket (Q4),

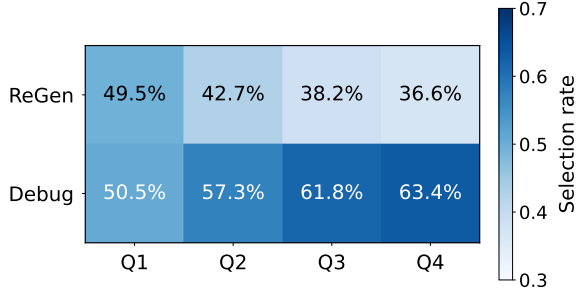


Figure 4: Oracle routing preferences across original pass-rate buckets. Colors indicate the proportion of instances for which regeneration (ReGen) or debugging (Debug) is preferred when routing decisions are made using gold test cases.

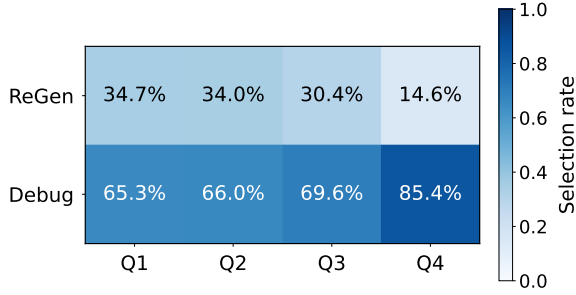


Figure 5: Routing preferences between regeneration (ReGen) and debugging (Debug) using the RL-trained test generator. Colors indicate the proportion of instances assigned to each repair strategy.

where PATSSEL benefits from the debugging strategy, which we showed in Section 3.2 to be particularly effective for localized fixes. Conversely, the largest gains over Debug occur in the lowest pass-rate bucket (Q1), where PATSSEL leverages the strength of regeneration on more severely flawed solutions. Taken together, these results suggest that regeneration and debugging offer complementary strengths across a broad spectrum of bug severities, rather than being limited to specific failure regimes.

6 Analysis

6.1 Routing Decisions by Bug Type

Figure 4 and Figure 5 show the routing preferences between regeneration (ReGen) and debugging (Debug) across original pass-rate buckets using either gold test cases (oracle routing) or tests generated by the RL-trained verifier. We observe that both routing policies increasingly favor debugging as the original pass rate increases, with the strongest preference for debugging in Q4 (75–99% pass rate) and the weakest preference in Q1 (0–25% pass rate). This behavior is consistent with our earlier findings

Selection Method	Q1	Q2	Q3	Q4	Micro Avg.
TF-IDF Value Router	54.7	64.4	68.9	83.1	67.5
Preference-Gap SFT Router	45.6	60.4	62.6	86.9	63.6
SFT Router	48.1	57.7	65.4	78.3	62.1
PATSSEL (Base Test Gen)	53.5	67.0	71.6	90.7	70.4
PATSSEL (RL Test Gen)	54.7	67.4	71.0	90.2	70.6
PATSSEL (Gold Tests)	60.0	71.7	76.0	91.3	74.5

Table 2: Comparison with outcome-independent routers on LeetCodeDataset using Qwen3-8B as the repair model. The trained routers select between regeneration and debugging before observing the resulting repair candidates, whereas PATSSEL selects between generated candidates using verifier-test execution.

on the relative strengths of regeneration and debugging across different bug severities, suggesting that the routers successfully adapt the repair strategy to the quality of the initial solution. Compared to oracle routing (Figure 4), however, the RL-trained verifier (Figure 5) exhibits a stronger overall bias toward debugging, particularly in the lower pass-rate buckets where regeneration is often more effective. This tendency likely explains part of the performance gap between RL-based routing and oracle routing reported in Table 1. Nevertheless, the overall routing trends remain highly consistent across the two settings, indicating that generated tests provide sufficient signal for effective repair-strategy selection. As a result, PATSSEL is able to leverage the complementary strengths of both repair strategies, favoring debugging for near-correct solutions while relying more on regeneration when larger modifications are required.

6.2 Can repair strategy be predicted without execution?

The routing trends in Section 6.1 raise a natural question: can the preferred repair strategy be predicted from the original repair instance before observing the resulting candidates? To investigate this, we train three outcome-independent routers on LeetCodeDataset (Table 2). The TF-IDF Value Router represents the problem, buggy code, and failing test with TF-IDF features compressed using SVD and predicts expected ReGen and Debug outcomes. We also train two LLM-based routers: the SFT Router directly predicts the preferred strategy, while the Preference-Gap SFT Router additionally models the performance gap between them. Although all three learn useful routing signals, the strongest TF-IDF router reaches 67.5 average ac-

Verifier	Micro Avg.	Gap to Gold	Wrong Route (%)	Weak/ Non-disc.
Base Test Gen	70.4	4.10	17.5%	49/70
RL Test Gen	70.6	3.94	17.5%	48/70
Gold Tests	74.5	—	—	—

Table 3: Routing error analysis for PATSSEL on 400 LeetCodeDataset instances using Qwen3-8B. GAP TO GOLD measures the difference in Micro Avg. relative to gold-test routing, while WRONG ROUTE is the percentage of instances whose routing decision differs from gold-test routing. WEAK/NON-DISC. reports how many wrong routing decisions are caused by generated verifier tests that are weak or fail to distinguish ReGen from Debug.

curacy, below PATSSEL with Base (70.4) or RL-trained (70.6) verifier tests. This indicates that such input-level signals are informative but insufficient: repair success also depends on the particular candidates produced at test time. By generating both candidates and comparing their observed behavior through execution, PATSSEL can use evidence that outcome-independent routers cannot access in advance.

6.3 When does verifier-based routing fail?

Although generated verifier tests outperform outcome-independent routing, they remain below oracle selection with gold tests. On 400 LeetCodeDataset instances, both Base and RL-generated tests disagree with gold routing on 70 cases (17.5%), corresponding to Micro Avg. gaps of 4.10 and 3.94 points, respectively (Table 3).

The dominant failure mode is insufficiently discriminative tests: 70.0% Base and 68.6% RL routing errors occur when generated tests fail to distinguish ReGen and Debug despite a clear difference under gold tests. Thus, the main bottleneck is whether verifier tests expose behavioral differences between competing repairs.

6.4 Generalization to Other Stronger Debugger Models

To evaluate the generalizability of PATSSEL beyond our own debugger, we replace it with FrogMini-14B (Sonwane et al., 2025), a stronger debugger trained on top of Qwen3-14B. Table 4 reports results using Qwen3-8B for regeneration and FrogMini-14B for debugging.

Consistent with Section 5.2, PATSSEL achieves the best average performance under all three routing settings. With gold tests, it improves over Re-

Test Gen	Strategy	Q1	Q2	Q3	Q4	Micro Avg.
Base	ReGen	57.6	69.7	68.4	87.9	70.7
	Debug	43.5	60.5	66.3	90.6	64.8
	PATSSEL	57.8	71.5	78.8	88.5	73.3
RL	ReGen	58.3	67.8	70.4	88.4	71.0
	Debug	42.8	61.0	67.5	90.2	64.9
	PATSSEL	57.1	73.5	72.9	87.8	72.1
Gold	ReGen	59.4	69.1	73.7	86.2	71.4
	Debug	51.3	64.1	73.0	90.7	69.4
	PATSSEL	66.8	75.2	80.3	92.4	78.2

Table 4: Performance of PATSSEL on LeetCodeDataset using FrogMini-14B as the debugger and Qwen3-8B as the regenerator. We evaluate three routing settings: Base-generated tests (Base), RL-generated tests (RL), and gold test cases (Gold). All methods use the same inference budget of choosing from two repair trials.

Gen by 6.8% and Debug by 8.8%; gains also persist with Base and RL-generated tests. Thus, PATSSEL transfers to a stronger external debugger and is not tied to our trained debugging model.

7 Conclusion

In this paper, we show that debugging and regeneration possess complementary strengths for code repair, with neither strategy being uniformly optimal across all failures. Motivated by the observation, we introduced PATSSEL, a test-guided framework that dynamically selects between the two repair strategies using verifier tests and their execution outcomes. Across LeetCodeDataset, HE+Fix, and LiveCodeBench, PATSSEL consistently outperforms single strategy baselines, debugging-only and regeneration-only, under the same inference-time compute budget. These gains persist across verifier settings and repair models. Our results highlight the importance of adaptive repair strategy selection, suggesting that effective code repair depends not only on stronger repair models but also on choosing the right repair pathway for each failure instance.

Limitations

Due to the computational cost of running multiple repair strategies and verifier evaluations, our experiments focus on two 8B-scale repair models, Qwen3-8B and Llama3-8B, rather than a broader range of model sizes. However, we observe that the strength of PATSSEL generalizes over multiple benchmarks and with different debugger and test-generation models.

Acknowledgments

This work was supported by NSF-AI Engage Institute DRL-2112635, NSF-CAREER Award 1846185, a Capital One Research Award, and an Apple PhD Fellowship. The views contained in this article are those of the authors and not of the funding agency.

References

- Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. [Teaching large language models to self-debug](#). *Preprint*, arXiv:2304.05128.
- Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. 2023. [Automated repair of programs from large language models](#). *Preprint*, arXiv:2205.10583.
- Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, and 542 others. 2024. [The llama 3 herd of models](#). *Preprint*, arXiv:2407.21783.
- Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, and 5 others. 2024. [Qwen2.5-coder technical report](#). *Preprint*, arXiv:2409.12186.
- Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. [Live-codebench: Holistic and contamination free evaluation of large language models for code](#). *Preprint*, arXiv:2403.07974.
- Xue Jiang, Yihong Dong, Mengyang Liu, Hongyi Deng, Tian Wang, Yongding Tao, Rongyu Cao, Binhua Li, Zhi Jin, Wenpin Jiao, Fei Huang, Yongbin Li, and Ge Li. 2025. [Coderl+: Improving code generation via reinforcement with execution semantics alignment](#). *Preprint*, arXiv:2510.18471.
- Atharv Kulkarni and Vivek Srikumar. 2025. [Reinforcing code generation: Improving text-to-sql with execution-based learning](#). *Preprint*, arXiv:2506.06093.
- Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven C. H. Hoi. 2022. [Coderl: Mastering code generation through pretrained models and deep reinforcement learning](#). *Preprint*, arXiv:2207.01780.
- Cheryl Lee, Chunqiu Steven Xia, Longji Yang, Jentse Huang, Zhouruixing Zhu, Lingming Zhang, and Michael R. Lyu. 2025. [UniDebugger: Hierarchical multi-agent framework for unified software debugging](#). In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 18237–18266, Suzhou, China. Association for Computational Linguistics.
- Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. 2017. [Quixbugs: a multi-lingual program repair benchmark set based on the quixey challenge](#). In *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity, SPLASH Companion 2017*, page 55–56, New York, NY, USA. Association for Computing Machinery.
- Zi Lin, Sheng Shen, Jingbo Shang, Jason Weston, and Yixin Nie. 2025. [Learning to solve and verify: A self-play framework for code and test generation](#). *Preprint*, arXiv:2502.14948.
- Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro von Werra, and Shayne Longpre. 2024. [Octopack: Instruction tuning code large language models](#). *Preprint*, arXiv:2308.07124.
- Theo X. Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. 2024. [Is self-repair a silver bullet for code generation?](#) *Preprint*, arXiv:2306.09896.
- Minh V. T. Pham, Huy N. Phan, Hoang N. Phan, Cuong Le Chi, Tien N. Nguyen, and Nghi D. Q. Bui. 2025. [Swe-synth: Synthesizing verifiable bug-fix data to enable large language models in resolving real-world bugs](#). *Preprint*, arXiv:2504.14757.
- Archiki Prasad, Elias Stengel-Eskin, Justin Chih-Yao Chen, Zaid Khan, and Mohit Bansal. 2025. [Learning to generate unit tests for automated debugging](#). *Preprint*, arXiv:2502.01619.
- Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. 2024. [Deepseekmath: Pushing the limits of mathematical reasoning in open language models](#). *Preprint*, arXiv:2402.03300.
- Atharv Sonwane, Isadora White, Hyunji Lee, Matheus Pereira, Lucas Caccia, Minseon Kim, Zhengyan Shi, Chinmay Singh, Alessandro Sordoni, Marc-Alexandre Côté, and Xingdi Yuan. 2025. [Bugpilot: Complex bug generation for efficient learning of swe skills](#). *Preprint*, arXiv:2510.19898.
- Leandro von Werra, Younes Belkada, Lewis Tunstall, Edward Beeching, Tristan Thrush, Nathan Lambert, Shengyi Huang, Kashif Rasul, and Quentin Galouédec. 2020. [TRL: Transformers Reinforcement Learning](#).

Yinjie Wang, Ling Yang, Ye Tian, Ke Shen, and Mengdi Wang. 2025. [Co-evolving llm coder and unit tester via reinforcement learning](#). *Preprint*, arXiv:2506.03136.

Yunhui Xia, Wei Shen, Yan Wang, Jason Klein Liu, Huifeng Sun, Siyue Wu, Jian Hu, and Xiaolong Xu. 2025. [Leetcodedataset: A temporal dataset for robust evaluation and efficient training of code llms](#). *Preprint*, arXiv:2504.14655.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, and 41 others. 2025. [Qwen3 technical report](#). *Preprint*, arXiv:2505.09388.

Huaye Zeng, Dongfu Jiang, Haozhe Wang, Ping Nie, Xiaotong Chen, and Wenhui Chen. 2025. [Acecoder: Acing coder rl via automated test-case synthesis](#). *Preprint*, arXiv:2502.01718.

Dataset	Split	Problems	Avg. Buggy Code Accuracy
LeetCodeDataset	Train	3000	48.0
	Dev	400	49.5
	Test	400	50.5
HE+Fix	Test	158	37.8
LiveCodeBench	Test	813	15.3

Table 5: Dataset statistics. Each LeetCodeDataset problem pairs a buggy program with one selected failing unit test. Problems denotes the number of training or evaluation problems in each split. Avg. Buggy Acc. is the average full-suite accuracy of the buggy code before repair. The LeetCodeDataset dev and test splits are each further divided into four pass-rate groups: Q1 (0–25%), Q2 (25–50%), Q3 (50–75%), and Q4 (75–99%), with 100 problems in each group.

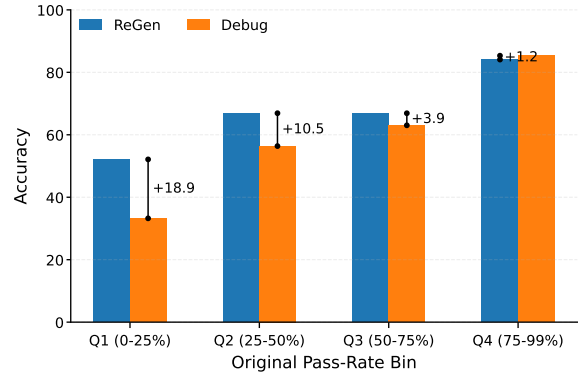


Figure 6: Performance of regeneration and debugging with an RL-trained debugger across original pass-rate bins (Q1: 0–25, Q2: 25–50, Q3: 50–75, Q4: 75–99). ReGen uses the base Qwen3-8B model, while Debug uses the RL-trained Qwen3-8B debugger. RL training narrows the gap between the two strategies, while the overall trend across initial program quality remains.

A Details of Dataset

Table 5 shows the statistics of dataset used for training and evaluation. HE+Fix and LeetCodeDataset are under MIT license and LiveCodeBench is under Creative Commons license. We focused on English.

B More analysis on Regeneration vs. Debugging

B.1 Performance with an RL-Trained Debugger

We additionally examine whether the relationship between regeneration and debugging observed in Section 3.2 persists when using a debugger explicitly trained for code repair. We keep regeneration

fixed to the base Qwen3-8B model and replace the base debugging model with our RL-trained Qwen3-8B debugger.

Figure 6 shows that RL training substantially strengthens debugging, particularly in the lower and intermediate pass-rate bins, narrowing the gap between debugging and regeneration. Nevertheless, the overall trend remains unchanged: regeneration performs better for lower-quality programs (Q1–Q3), while debugging becomes preferable in the highest pass-rate bucket (Q4). The regeneration advantage decreases from 18.9 points in Q1 to 3.9 points in Q3, while the RL-trained debugger outperforms regeneration by 1.2 points in Q4. These results show that the dependence of repair strategy on initial program quality persists even with a stronger specialized debugger, further motivating adaptive selection between regeneration and debugging.

C Additional Results

C.1 Comparison on QuixBugs-Python

To further evaluate PATSSEL against existing code-repair approaches, we report results on QuixBugs-Python (Lin et al., 2017) in Table 6. UniDebugger-Lite (Lee et al., 2025) repairs all 40 problems using GPT-4o with five repair attempts. In comparison, PATSSEL uses the substantially smaller Qwen3-8B model and only two repair candidates, repairing 33/40 problems with either Base- or RL-generated verifier tests and 35/40 with gold verifier tests. The gold-verifier setting also exceeds several larger standalone code models evaluated with 20 samples, including CodeLlama-34B and LLaMA2-70B (33/40), DeepSeekCoderV2 (25/40), Gemini-1.5-Flash (29/40), and GPT-3.5-Turbo (34/40). Although the compared approaches differ in model scale and inference budget, these results show that PATSSEL remains competitive on QuixBugs-Python while using an 8B repair model and only two candidate repairs.

C.2 Compute-Matched Repair Budget

Our main comparisons control for candidate-generation budget: ReGen and Debug each generate two candidates, while PATSSEL generates one candidate from each repair strategy. In all three cases, the final candidate is selected using the same verifier-based criterion. Table 7 additionally reports single-attempt ReGen and Debug performance for context. While using a second candidate

Method / Model	Setting	Correct (Out of 40)
CodeLlama-34B	20 samples	33
LLaMA2-70B	20 samples	33
DeepSeekCoderV2	20 samples	25
Gemini-1.5-Flash	20 samples	29
GPT-3.5-Turbo	20 samples	34
Claude-3.5-Sonnet	20 samples	40
GPT-4o	20 samples	39
UniDebugger-Lite (GPT-4o)	5 attempts	40
PATSSEL (Qwen3-8B)	Base Test Gen	33
PATSSEL (Qwen3-8B)	RL Test Gen	33
PATSSEL (Qwen3-8B)	Gold Tests	35

Table 6: Comparison of code-repair performance on QuixBugs-Python. PATSSEL uses Qwen3-8B with two repair candidates; other methods are shown with their corresponding inference settings.

Verifier / Budget	Strategy	Q1	Q2	Q3	Q4	Micro Avg.
No verification / 1 attempt	Base ReGen	52.1	66.9	66.9	84.1	67.3
No verification / 1 attempt	RL Debug	33.2	56.4	63.0	85.3	59.3
Gold Tests / 2 attempts	ReGen	59.4	69.1	73.7	86.2	71.4
Gold Tests / 2 attempts	Debug	44.8	60.1	70.1	87.7	64.4
Gold Tests / 2 attempts	PATSSEL	60.0	71.7	76.0	91.3	74.5

Table 7: Effect of candidate-generation budget on LeetCodeDataset using Qwen3-8B as the repair model. The main two-candidate comparison is compute-matched: ReGen and Debug allocate both candidates to a single strategy, whereas PATSSEL allocates one candidate to each strategy.

improves the available repair budget, the controlled two-candidate comparison shows that PATSSEL achieves the strongest average performance (74.5), outperforming two ReGen attempts (71.4) and two Debug attempts (64.4). Thus, the improvement of PATSSEL is not due to generating more repair candidates, but to allocating the same two-candidate budget across complementary regeneration and debugging strategies.

D More Details of Training

Given the computational cost of training, we use one training run for each reported configuration.

D.1 Details of GRPO Training

We train both learned components with Group Relative Policy Optimization (GRPO; Shao et al., 2024). For each prompt q , the current policy samples a group of outputs, each output is executed, and the resulting scalar rewards are normalized within the group to form relative advantages. This is a natural fit for code repair and test generation because

both tasks provide execution-based rewards, but do not require a single gold text target. We use the same GRPO training procedure for the debugger and the unit-test generator, while changing only the reward function: the debugger is rewarded for improving the buggy program’s reference-test pass rate, and the test generator is rewarded for producing executable tests that agree with the reference solution.

D.2 Unit Test Generator

The test generator model generates a test suite in the form “def check(candidate):” with assertions over the candidate program. The test-generator reward has three goals. First, the output must have the correct structure. It must define a valid check(candidate) function containing executable assertions. Second, the suite should contain enough tests to provide a useful behavioral signal. Third, the expected outputs in the generated assertions must agree with the reference solution.

E Prompts

This section lists the prompt templates used for regeneration, debugging, and unit-test generation. Placeholders such as <PROBLEM_SPECIFICATION>, <BUGGY_FUNCTION_OR_CLASS>, and <FAILING_UNIT_TEST> are replaced with instance-specific fields at inference time.

F LLM Usage

In our research, we employed LLMs as writing assistance. During writing, LLMs were utilized for sentence-level refinement and grammatical polishing. All AI-generated suggestions were carefully reviewed and edited by the authors to maintain the coherency and accuracy.

G Future Work

Our experiments focus on algorithmic code repair with executable candidates and verifier tests. Future work should extend PATSSEL to repository-level repair, where larger contexts, dependencies, and execution environments complicate the choice between localized debugging and broader regeneration. We also plan to investigate outcome-independent strategy prediction. While our learned routers show that the original repair instance contains useful signal, more principled predictors based on model-internal or uncertainty signals may

enable strategy selection before candidate execution, reducing the need for candidate comparison.

Regeneration prompt

You are an expert Python programmer. You will be given a question (problem specification) and will generate a correct Python program that matches the specification and passes all tests.

Question:
<PROBLEM_SPECIFICATION>

Format: You will use the following starter code to write the solution to the problem and enclose your code within delimiters.
```python  
<STARTER\_CODE>  
```

Answer: (use the provided format with backticks)

Figure 7: Regeneration prompt template. The problem specification and starter code are filled in for each example.

Debugging system prompt

You are a Python bug-fixing assistant.

You MUST respond with the exact structure below. Every section is REQUIRED.
Write naturally and with as much detail as needed (no length limit).

- 1) CODE INTENT
- 2) BUG REVIEW
- 3) FIX PLAN
- 4) FINAL FIX

Output ONLY the fixed function/class in exactly ONE Python fenced code block.
The code block MUST be the LAST thing in the response.
Keep the same function/class name and signature.
Do NOT change imports.

Figure 8: System prompt used for the debugging branch.

Debugging user prompt

Fix the buggy function/class so that the failing unit test passes, also try to fix any other issues in the code.
- Keep the same function/class name and signature.
- Do NOT change imports.

Problem description:
<PROBLEM_DESCRIPTION>

Buggy function/class:
<BUGGY_FUNCTION_OR_CLASS>

Failing unit test:
<FAILING_UNIT_TEST>

Figure 9: User prompt used for the debugging branch. The failing unit test is sampled from the same test source used by the verifier in the corresponding experimental setting.

Unit-test generation system prompt

You are a Python unit-test generation assistant for programming problems.
Your job is to generate executable Python tests in exactly the dataset style.

REQUIRED OUTPUT FORMAT:

- Output ONLY one Python code block.
- Inside the code block, define exactly one function:

```
def check(candidate):  
    assert candidate(...) == ...  
    assert candidate(...) == ...  
    ...
```

Rules:

- Use the provided entry point implicitly through `candidate`.
- Do not include imports.
- Do not include any text outside the code block.
- Generate AT LEAST 10 assert statements.
- Generate diverse tests that aim for broad behavior coverage of the provided problem.
- Include easy, edge, boundary, duplicate, large-value, adversarial, empty/singleton, and structure-specific cases when applicable.
- Every assert must be deterministic.
- If no valid output exists for an input, you may use `None` only when consistent with the problem behavior.

Figure 10: System prompt used for the unit-test generator.

Unit-test generation user prompt

Problem description:
<PROBLEM_DESCRIPTION>

Starter code:
<STARTER_CODE>

Buggy function/class:
<BUGGY_FUNCTION_OR_CLASS>

Entry point:
<ENTRY_POINT>

Generate Python tests in the dataset format now.
You MUST generate at least 10 assert statements inside def check(candidate).
Generate diverse tests; do not output one simple test case.
All expected outputs must be correct with respect to the problem description.

Figure 11: User prompt used for the unit-test generator.