

# APR-Router: Complexity-Aware Cascade Routing for Cost-Effective LLM-Based Program Repair

Zhixiang Zhang<sup>1</sup>, Zhicheng Bao<sup>1</sup>, Xiaoyu Li<sup>1</sup>, Jiaojiao Jiang<sup>1\*</sup>, Qinghua Lu<sup>2</sup>, Yulei Sui<sup>1</sup>, Liming Zhu<sup>2</sup>

<sup>1</sup>The University of New South Wales, Australia

<sup>2</sup>CSIRO, Australia

zhixiang.zhang1@student.unsw.edu.au, {zhicheng.bao, xiaoyu.li2, jiaojiao.jiang, y.sui}@unsw.edu.au  
{qinghua.lu, liming.zhu}@data61.csiro.au

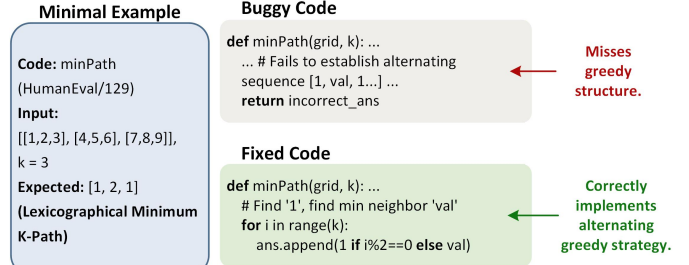
**Abstract**—LLM-based automated program repair (APR) encompasses strategies ranging from lightweight zero-shot prompting to costly agentic workflows, yet most systems apply a fixed strategy to every defect. Through a systematic empirical study, we establish three new findings. First, no single repair strategy dominates across all complexity levels. Second, repair success depends on domain-specific code characteristics. Third, a cascade of just three attempts closes most of the repair gap left by any single strategy, with sharply diminishing returns beyond. Based on these observations, we propose APR-Router, a complexity-aware routing framework that formulates APR strategy selection as a sequential repair problem. APR-Router operates in three stages. First, it extracts a 114-dimensional feature vector spanning five families. Second, it predicts per-strategy repair effectiveness with a LambdaRank-based ranker and per-strategy token costs with independent regressors. Third, it executes strategies in decreasing order of a repair efficiency index grounded in sequential search theory, stopping at the first success. Extensive experiments demonstrate that APR-Router improves pass rate by +8.7 to +17.4 percentage points and Aggregate Intelligence Quotient by +8.3% to +22.3% over the strongest APR baseline, and consistently outperforms adapted generic LLM routing methods, with routing overhead less than 0.1% of a single LLM inference call.

**Index Terms**—Automated program repair, large language model, multi-objective optimization

## I. INTRODUCTION

Recent advances in large language models (LLMs) have given rise to a rich and rapidly expanding spectrum of automated program repair (APR) strategies. A recent survey [1] catalogues 62 LLM-based repair systems published between 2022 and 2025, revealing a broad design space that ranges from single-turn prompting to autonomous multi-step agents. At one end of this spectrum, lightweight zero-shot methods [2], [3] consume only hundreds of tokens per repair attempt, while at the other end, agentic workflows [4] can consume tens of thousands of tokens per issue. Between these extremes lie retrieval-augmented few-shot prompting [5], analogy-augmented methods [6] and iterative feedback methods [7]–[11], each occupying a distinct point in the cost-effectiveness space.

At deployment scale, LLM inference cost has become the primary constraint on the practical adoption of LLM-based APR, with per-issue monetary cost swinging from sub-cent to



(a) A minimal example of an APR task.



(b) Comparison of four representative LLM-based APR strategies on the example task.

Fig. 1. A motivating example showing defect-specific differences across LLM-based APR strategies.

about \$10 in heavy multi-turn scaffolds [12]. Although per-token prices are decreasing fast, total inference expenditure continues to rise because token consumption grows faster than unit prices fall. As Figure 1 illustrates, the same bug can be repaired cheaply by one strategy, expensively by another, or not at all by a third, revealing large per-instance variation in cost-effectiveness that a fixed workflow cannot exploit. Yet the vast majority of existing APR systems apply a uniform repair pipeline to every defect [7], over-serving simple bugs with expensive methods and under-serving complex bugs with lightweight ones, resulting in systematic computational waste.

The challenge of balancing quality against cost is well studied in the LLM routing literature. HybridLLM [13] introduces quality-aware routing between a cheap and an expensive model. RouteLLM [14] trains preference-based routers that reduce GPT-4 calls by up to 85%. Dekoninck et al. [15] unify routing (one-shot model selection) and cascading (sequential model execution) into a theoretically optimal framework.

\*Corresponding author: Jiaojiao Jiang (jiaojiao.jiang@unsw.edu.au)

A recent survey [16] covers more than 30 such systems, and RouterBench [17] establishes the Aggregate Intelligence Quotient (AIQ) as a standardized cost-quality metric. These methods all target essentially the same scenario, where the candidates are LLMs of different sizes (e.g., a cheap 7B model versus an expensive 70B model), larger models are uniformly stronger but more expensive, and the router’s only job is to decide whether a query is hard enough to justify calling the expensive model.

APR strategy selection, however, does not fit this scenario. Through a systematic empirical study (Section II), we show that APR candidates are structurally heterogeneous workflows, and their effectiveness does not grow monotonically with cost. The most expensive agentic workflow is not always the best repairer. Concretely, the best-performing strategy shifts across complexity bands, so no fixed selection rule can be universally optimal. We further find that different strategies fix largely disjoint bug subsets, leaving a large complementarity gap. One-shot selection cannot close this gap, but a short cascade of two to three attempts can close most of it. Finally, code-structural features predict token cost well but relate to repair success only weakly and non-linearly, requiring domain-specific feature engineering rather than the embedding-based classifiers used in generic routing.

To address these challenges, we propose APR-Router, a complexity-aware routing framework that formulates APR strategy selection as a sequential repair selection problem. APR-Router operates in three stages. First, the feature extraction stage computes a 114-dimensional feature vector spanning five families. Second, the performance prediction stage uses a single LambdaRank-based ranker to predict each candidate strategy’s repair likelihood and a set of per-strategy regressors to predict the corresponding token cost. Third, the cascade routing stage combines these two predictions into a repair efficiency index grounded in sequential search theory, which ranks strategies by their expected repair gain per unit of token cost. The router then executes strategies in this ranked order as a short cascade, stopping at the first success.

The main contributions of this study are as follows.

- 1) We present a systematic empirical study that establishes the alternation of strategy dominance across complexity levels, the strong complementarity among APR strategies, and the non-linear relationship between code features and repair success, collectively motivating a cascade-based routing approach.
- 2) Motivated by these observations, we formulate APR strategy selection as a sequential repair selection problem and propose a predictive, cascade-based routing framework that estimates per-strategy repair effectiveness and token cost from code features and selects strategies by a cost-effectiveness index derived from sequential search theory. To our knowledge, this is the first cascade-routing framework specifically designed for APR.
- 3) We construct a curated open dataset that maps buggy programs to repair outcomes and token consumption across ten representative APR methods, and design a 114-

dimensional feature vector with five families, of which three are APR-specific and have no counterpart in the generic LLM routing literature.

- 4) Extensive experiments on two standard benchmarks demonstrate that APR-Router improves pass rate by +8.7 to +17.4 percentage points and AIQ by +8.3% to +22.3% over the strongest individual baseline, and consistently outperforms generic LLM routing methods including HybridLLM and RouteLLM, with all improvements statistically significant across all four configurations.

## II. EMPIRICAL STUDY

In this section, we present an empirical investigation that motivates the design of APR-Router.

**Experimental Setup.** For the correlation and localized trend analyses (Sections II-A and II-B), we use three representative strategies spanning distinct paradigms, CEDAR (few-shot) [5], RLCE (RAG) [18], and CEGPR (zero-shot) [11], on the Run-BugRun [19] benchmark with the Llama-3-8B backbone, characterizing each bug by 22 Radon-extracted structural features. For the complementarity analysis (Section II-C), we extend to all ten candidate APR methods and average results across four configurations (two LLMs  $\times$  two datasets). Unless otherwise noted, token cost sums prompt and completion tokens, and a repair is counted as successful iff the produced patch passes the full unit-test suite for that bug. This criterion is semantic, alternative variable names or functionally equivalent implementations are accepted as long as all tests pass.

### A. Global Correlation Trends

Figure 2 reports the Pearson correlation coefficients  $r$  between 22 structural code features and two performance indicators. Statistical significance levels are marked ( $***p<0.001$ ,  $**p<0.01$ ,  $*p<0.05$ ). Positive correlations appear in ■ red shades, while negative correlations appear in ■ blue shades.

Token cost shows a strong positive linear correlation with most code features. In particular, AST Nodes ( $r=0.68$ ), LOC ( $r=0.78$ ), and Raw LOC ( $r=0.77$ ) exhibit the highest correlations, indicating that larger or more syntactically complex code consistently incurs higher token consumption. In contrast, the repair success rate exhibits weak or negative correlations with the same features, e.g., AST Nodes ( $r=-0.47$ ) and LOC ( $r=-0.48$ ), with several others near zero. This asymmetry suggests that repair success is not linearly explained by global code size or complexity.

**Finding 1.** Code features strongly predict token cost but only weakly correlate with repair success rate. Repair performance is feature-dependent yet non-linear, and global linear summaries are insufficient.

### B. Localized Trend Analysis

To further investigate the non-linear effects revealed in Figure 2, we analyze the local behavior of repair performance across different code complexity levels. We group code samples according to their normalized AST node count. For each

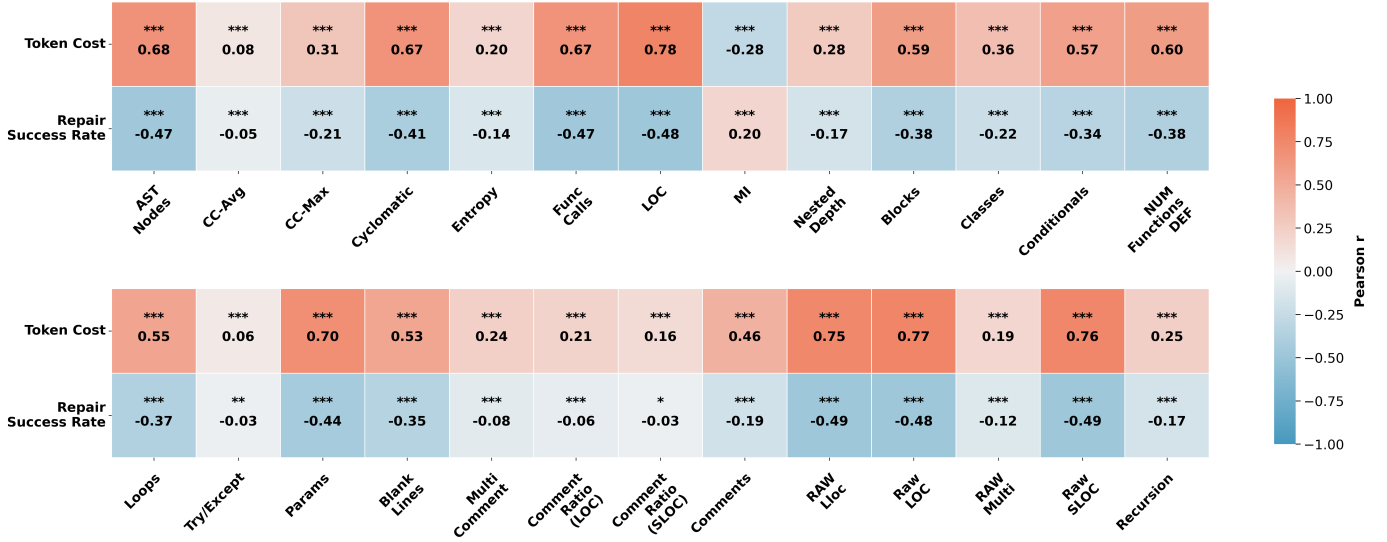


Fig. 2. Pearson correlations  $r$  between structural code features and two performance indicators.

interval  $[n-\delta, n+\delta]$ , we compute the mean repair success rate of each method and apply locally weighted smoothing to obtain continuous trend curves. The resulting patterns are shown in Figure 3.

Figure 3 plots locally smoothed pass-rate curves against the AST-node count for three repair methods. Two key patterns emerge. First, each method’s curve is non-monotonic, with clear peaks and troughs across the complexity axis. Second, the alternation of shaded bands, each indicating the per-bin winner with the highest mean pass rate, establishes a region-wise ordering rather than a single global ranking. CEGPR dominates in low-complexity regions where concise zero-shot prompts suffice, CEDAR peaks in mid-complexity ranges where few-shot exemplars provide the most guidance, and RLCE recovers competitiveness in certain high-complexity bands where additional retrieved context compensates for program scale.

**Finding 2.** Each method’s pass rate varies non-monotonically with AST-node density, and the leading method alternates across complexity ranges.

### C. Strategy Complementarity Analysis

Findings 1 and 2 establish that no single strategy dominates across all complexity regions, motivating per-instance strategy routing. A natural follow-up question is whether selecting a single best strategy per bug is sufficient, or whether significant repair potential remains beyond what any one-shot selector can capture.

To investigate this, we extend the analysis to all ten candidate APR methods and compute the union pass rate as a function of cascade depth  $k$ . For each configuration, we rank the ten methods by their individual pass rate in descending order and compute the fraction of test-set bugs that are

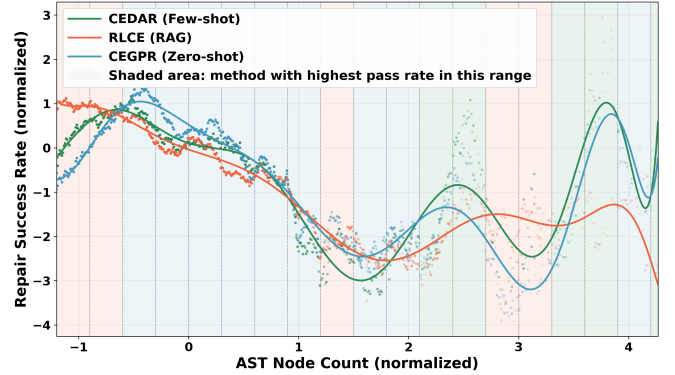


Fig. 3. Normalized repair success rate across different AST node count levels for three repair methods.

successfully repaired by at least one of the top- $k$  methods. Figure 4 reports the mean and standard deviation across the four configurations.

The best single method, identified by ranking all ten candidates by mean pass rate, achieves 55.9%. Expanding the pool to the top two methods raises the union pass rate to 63.4%, and the top three together reach 67.4%. The oracle ceiling, defined as the pass rate when all ten methods are available, is 72.5%, leaving a complementarity gap of 16.6 pp between the best individual method and the theoretical upper bound. The marginal gain curve (right axis of Figure 4) reveals a characteristic diminishing-returns pattern. The second method contributes +7.6 pp, the third adds +3.9 pp, and each subsequent method contributes less than 2 pp. Consequently, the first three methods alone close 69.1% of the gap to the oracle (11.5 out of 16.6 pp), while the remaining seven methods collectively account for only the last 30.9%. This sharp decay implies that a small cascade depth captures the

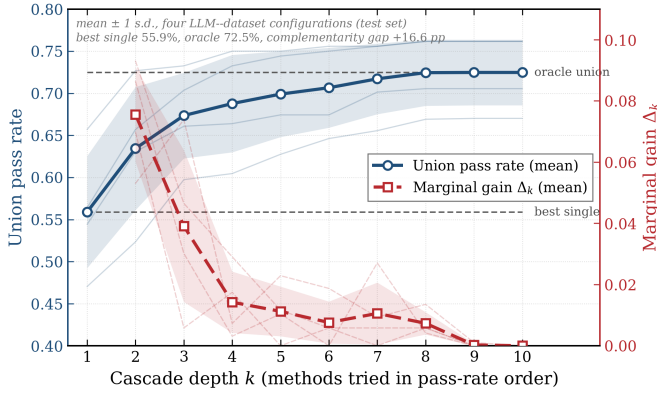


Fig. 4. Union pass rate (left axis) and marginal gain  $\Delta_k$  (right axis) versus cascade depth  $k$ .

majority of the available headroom.

**Finding 3.** APR strategies exhibit strong complementarity. The best single method captures only 77.1% of the oracle repair potential (55.9% vs. 72.5%). The top three methods close 69.1% of this gap, while marginal gains decay rapidly beyond  $k=3$ , motivating a short, intelligently ordered cascade over one-shot selection.

### III. RELATED WORK

In this section, we discuss prior work on LLM-based automated program repair (APR) and model routing.

#### A. LLM-Based Automated Program Repair

LLMs have fundamentally reshaped APR [2], [20]. Yang et al. [1] classify LLM-based APR into four paradigms—fine-tuning, prompting, procedural pipelines, and agentic frameworks—with retrieval-augmented generation (RAG) and analysis-augmented generation (AAG) as orthogonal enhancement layers. We adopt this taxonomy to situate the ten repair methods in APR-Router.

1) *Prompting Paradigm*: Prompting methods keep model parameters frozen and rely solely on prompt design. Zero-shot prompting feeds buggy code directly to an LLM with a repair instruction [2]. AlphaRepair [3] replaces fault-localized lines with `HOLE` markers for cloze-style repair. Adding RAG, CEDAR [5] retrieves similar bug-fix pairs as few-shot demonstrations. Adding AAG, CEGPR [11] augments prompts with MaxSAT-based program sketches and counterexample feedback from a verifier-driven synthesis loop.

2) *Procedural Pipeline Paradigm*: Procedural pipelines embed LLM calls within script-controlled multi-step workflows. SRepair [6] first prompts the LLM to analyze root causes via chain-of-thought reasoning, then generates patches based on the analysis (+AAG). ChatRepair [7] drives repair through generate–test–feedback loops, progressively refining patches with failure information (+AAG). Self-Debug [8] inserts a code-explanation step before each iteration, enabling

the LLM to self-identify logical errors (+AAG). ContrastRepair [10] augments feedback with minimal-difference passing/failing test pairs to sharpen fault localization (+AAG). ThinkRepair [9] retrieves CoT reasoning traces from a pre-collected knowledge base and iteratively refines patches with test feedback (+RAG+AAG).

3) *Agentic Framework Paradigm*: RepairAgent [4] grants the LLM full control over planning and tool invocation, autonomously gathering context and repairing bugs including multi-hunk and cross-file defects.

#### B. Model Routing and Selection

RouteLLM [14] formalized the problem of dynamically routing queries between a stronger and a weaker LLM, training routers on human preference data to reduce cost while preserving response quality. RouterBench [17] provides a standardized evaluation framework with over 405K precomputed inference outcomes and introduces the AIQ metric that jointly measures performance and cost. Our work adopts AIQ and APGR as primary evaluation metrics.

### IV. METHOD

APR-Router is a predictive routing framework that selects, for each incoming bug, an ordered sequence of repair strategies maximizing cost-effectiveness. During offline training, all candidate strategies are executed on a labeled corpus to collect repair outcomes and token costs, from which a success ranker and per-strategy cost regressors are learned. At inference, the trained models estimate each strategy’s expected success probability and token cost from code-level features, compute a repair efficiency index, and execute strategies in index-ranked order until the first success (Algorithm 1 and Figure 5). The remainder of this section formalizes the problem (Section IV-A), defines the index and cascade (Section IV-B), and describes the learning components (Section IV-C).

#### A. Problem Formulation

Motivated by the findings in Section II, we formulate APR strategy selection as a sequential repair selection problem.

**Sequential Policy.** Let  $\mathcal{B}$  denote a corpus of buggy programs and  $\mathcal{M} = \{m_1, \dots, m_n\}$  the set of  $n$  candidate LLM-based APR strategies. For each bug  $b \in \mathcal{B}$ , a feature extractor  $\phi$  produces a vector  $\mathbf{z} = \phi(b) \in \mathbb{R}^d$ .

Applying strategy  $m$  to bug  $b$  yields a binary random variable  $y_m(b) \in \{0, 1\}$ , where  $y_m(b) = 1$  if and only if the strategy produces a patch that passes all unit tests under the chosen execution protocol, and  $y_m(b) = 0$  otherwise. We write

$$y_m(b) \sim \text{Bernoulli}(p_m(\mathbf{z}))$$

only to model this pass/fail event compactly, where

$$p_m(\mathbf{z}) = \Pr[y_m(b) = 1 \mid \mathbf{z}].$$

This notation does not introduce an additional sampling step beyond executing the repair strategy; rather, it formalizes the stochastic success of LLM-based repair attempts so that the expected behavior of the cascade can be analyzed.

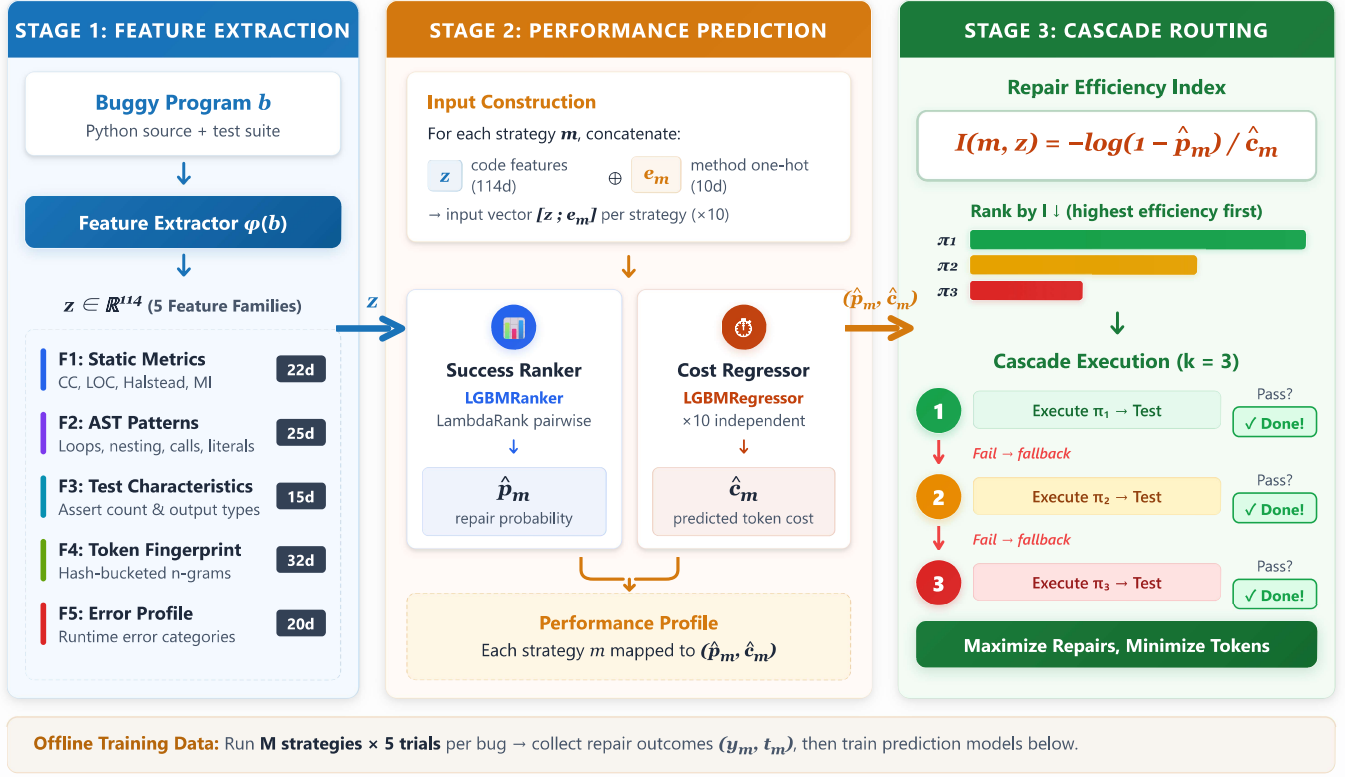


Fig. 5. The Overall Architecture of APR-Router

Motivated by Findings 2 and 3, we therefore define a routing policy  $\pi$  that maps  $\mathbf{z}$  to an *ordered* sequence of up to  $k$  strategies:

$$\pi(\mathbf{z}) = (\pi_1(\mathbf{z}), \pi_2(\mathbf{z}), \dots, \pi_k(\mathbf{z})), \quad \pi_i(\mathbf{z}) \in \mathcal{M}, \quad (1)$$

which are executed in order, stopping at the first success. Let  $\tau$  denote the index of the first successful attempt, setting  $\tau = k+1$  if all attempts fail. The outcome and cumulative cost are then:

$$\text{pass}(\pi, b) = \mathbf{1}[\tau \leq k], \quad C(\pi, b) = \sum_{i=1}^{\min(\tau, k)} c_{\pi_i}(b). \quad (2)$$

This formulation is an instance of the classical sequential search problem [21], in which independent trials with heterogeneous success probabilities and costs are ordered to minimize the expected cost of finding the first success. The optimal policy executes trials in decreasing order of a priority index derived from the success-to-cost ratio. We exploit this result in Section IV-B by defining a *repair efficiency index* that serves as the cascade ordering criterion.

**Objective.** The goal is to find  $\pi^*$  that maximizes corpus-level quality-adjusted cost-effectiveness. Following RouterBench [17], we adopt the Aggregate Intelligence Quotient (AIQ) as the primary metric:

$$\text{AIQ}(\pi) = \bar{q}(\pi) \cdot \frac{2(c_{\max} - \bar{c}(\pi) - c_{\min})}{2(c_{\max} - c_{\min})}, \quad (3)$$

where  $\bar{q}(\pi)$  and  $\bar{c}(\pi)$  are the average pass rate and token cost over the test set, and  $c_{\min}$ ,  $c_{\max}$  are the observed cost bounds across all evaluated configurations. The cost factor equals 1.0 when  $\bar{c} = c_{\min}$  and decreases to 0.5 when  $\bar{c} = c_{\max}$ , so that policies with equal pass rates are distinguished by efficiency.

### B. Repair Efficiency Index and Cascade

The sequential policy defined in Section IV-A requires a scoring function to rank candidate strategies and an execution protocol to carry out the cascade. We address both below.

**Repair Efficiency Index.** For each strategy  $m \in \mathcal{M}$  and a bug with features  $\mathbf{z}$ , let  $\hat{p}_m(\mathbf{z})$  and  $\hat{c}_m(\mathbf{z})$  denote the predicted success probability and token cost, respectively. The design of the scoring function follows directly from the structure of sequential repair. When the cascade executes strategies  $\pi_1, \dots, \pi_k$  independently, the overall probability of at least one successful repair is

$$1 - \prod_{i=1}^k (1 - p_{\pi_i}(\mathbf{z})). \quad (4)$$

Applying the negative logarithm to the complementary survival term yields  $\sum_{i=1}^k -\log(1 - p_{\pi_i})$ , which decomposes the joint repair success into additive per-strategy contributions. This additivity is a well-known property of the cumulative hazard function in reliability theory [22] and of the self-information in information theory [23]. Each term  $-\log(1 - p_m)$  can thus be interpreted as the repair information

---

**Algorithm 1** APR-Router Inference Procedure

---

**Require:** Bug  $b$ , feature extractor  $\phi$ , ranker  $R$ , regressors  $\{r_m\}$ , cascade depth  $k$

- 1:  $\mathbf{z} \leftarrow \phi(b)$  {extract features}
- 2: **for** each method  $m \in \mathcal{M}$  **do**
- 3:    $s_m \leftarrow R(\mathbf{z}, \mathbf{e}_m)$  {ranking score}
- 4:    $\hat{c}_m \leftarrow r_m(\mathbf{z})$  {predicted token cost}
- 5: **end for**
- 6:  $\hat{p}_m \leftarrow \text{softmax}(\{s_m\})$  for all  $m$
- 7:  $I_m \leftarrow (-\log(1 - \hat{p}_m))^\alpha / \hat{c}_m^{1-\alpha}$  for all  $m$
- 8:  $\pi \leftarrow \text{argsort\_desc}(\{I_m\})$  {cascade order}
- 9:  $C_{\text{total}} \leftarrow 0$
- 10: **for**  $i = 1$  to  $k$  **do**
- 11:   Execute strategy  $\pi_i$  on bug  $b$
- 12:    $C_{\text{total}} \leftarrow C_{\text{total}} + c_{\pi_i}(b)$
- 13:   **if**  $y_{\pi_i}(b) = 1$  **then**
- 14:     **return** (pass=1,  $C_{\text{total}}$ )
- 15:   **end if**
- 16: **end for**
- 17: **return** (pass=0,  $C_{\text{total}}$ )

---

contributed by strategy  $m$ , measuring its marginal contribution to overall repair success. Unlike the raw probability  $p$ , this quantity grows rapidly as  $p \rightarrow 1$ , assigning greater weight to strategies with high success rates. Normalizing by the predicted token cost, we define the repair efficiency index as

$$I_\alpha(m, \mathbf{z}) = \frac{(-\log(1 - \hat{p}_m(\mathbf{z})))^\alpha}{\hat{c}_m(\mathbf{z})^{1-\alpha}}. \quad (5)$$

The weight  $\alpha \in [0, 1]$  balances success information against token cost. The index measures the expected repair information per unit of token cost, and it generalizes the classical  $p/c$  priority index from sequential search theory [21] by replacing the linear success term with its information-theoretic counterpart. Ordering strategies by decreasing  $I$  therefore maximizes the cumulative repair information acquired per token in the early cascade rounds, which is the cascade-level analogue of the cost-effectiveness objective in Eq. (3).

**Cascade Procedure.** Unlike existing APR systems that repeatedly sample from a single strategy [7], [8], our cascade exploits the cross-strategy complementarity established in Finding 3 by trying distinct strategies in decreasing order of  $I(m, \mathbf{z})$ , stopping at the first success or after  $k$  attempts. Algorithm 1 presents the complete inference workflow of APR-Router. All ranking and ordering steps rely solely on predicted values, making the procedure deployment-realistic, while token costs accumulate from actual executions. The cascade depth  $k=3$  is selected empirically. Section V reports sensitivity analysis over  $k \in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$ .

### C. Learning the Index Components

Computing the repair efficiency index  $I(m, \mathbf{z})$  in Eq. (5) requires two predicted quantities for every candidate strategy: the success probability  $\hat{p}_m(\mathbf{z})$  and the token cost  $\hat{c}_m(\mathbf{z})$ . Both predictions are conditioned on a feature vector  $\mathbf{z} =$

$\phi(b) \in \mathbb{R}^{114}$  extracted from the buggy program  $b$ . This section describes the feature design, the success ranking model, and the cost regressor.

**1) APR-Specific Feature Engineering:** The feature vector  $\mathbf{z}$  comprises 114 dimensions organized into five families, each capturing a distinct aspect of the buggy program that influences token cost or the relative effectiveness of different repair strategies.

**F1: Static Code Metrics (22 dim).** Computed by the Radon library on the buggy source, this family includes cyclo-matic complexity, Halstead measures, lines of code, comment density, and the composite maintainability index. LOC alone correlates strongly with average token cost, and the mean token cost across LOC terciles rises monotonically from 1,439 to 3,064, making program scale the primary cost driver and a foundational input to the cost predictor.

**F2: AST Structural Patterns (25 dim).** We parse the buggy code into an AST and extract binary indicators for control-flow constructs (loops, conditionals, recursion, comprehensions) together with counts of arithmetic operators, function calls, and literal types, providing finer structural detail than aggregate complexity scores. Critically, different strategies respond differently to the same structural signal. The Spearman correlation between maximum nesting depth and pass rate is  $-0.098$  for zero-shot but only  $-0.001$  for analysis-augmented SRepair, meaning deep nesting disproportionately degrades simple strategies while leaving sophisticated ones largely unaffected. This differential sensitivity is precisely the signal that enables per-instance routing.

**F3: Test-Code Characteristics (15 dim).** We parse the test suite to extract assertion counts, the distribution of expected output types (integer, string, list, boolean), and keyword-argument counts in test calls. The dominant output type shifts the optimal strategy. On RunBugRun, integer-typed outputs favor Self-Debug and ThinkRepair, while string-typed outputs favor CEDAR and CEGPR, showing that test-side signals carry strategy-discriminative information beyond what code features alone provide.

**F4: Token Distribution Fingerprint (32 dim).** We tokenize the buggy source into unigrams and bigrams, hash each token into one of 32 buckets using a deterministic blake2b hash with a fixed key, accumulate counts, and L2-normalize. These hashed n-grams capture coarse lexical regularities that correlate with repair difficulty and strategy suitability. In practice, code containing library-specific identifiers, nested indexing patterns, container-manipulation idioms, or dense API usage tends to induce different repair behavior from code dominated by simple arithmetic and shallow control flow. **F5: Runtime Error Profile (20 dim).** For datasets with execution diagnostics, we extract binary indicators for error categories (syntax error, timeout, return mismatch, non-zero exit) along with counts and string-length statistics of error fields. Error category is a strong strategy predictor. On RunBugRun, Self-Debug performs best on syntax errors, SRepair on timeouts, and CEGPR on return mismatches, with the best versus worst pass-rate gap reaching 0.60 on timeout bugs, the largest among

all error types.

Among these five families, F1 and F2 capture general-purpose code characteristics commonly used in software analytics, whereas F3 through F5 are APR-specific designs with no counterpart in existing LLM routing literature, which typically operates on natural-language query embeddings. An incremental feature-family ablation confirms each family contributes positively.

Among these five families, F1 and F2 capture general-purpose code characteristics commonly used in software analytics, whereas F3 through F5 are APR-specific designs with no counterpart in existing LLM routing literature. We keep the representation at 114 dimensions by functional coverage rather than by indiscriminate expansion: F1 mainly explains token cost, F2 captures structural difficulty, F3 exposes signals from the test suite, F4 provides lexical priors, and F5 summarizes observed failure modes. Consistent gains from feature-family ablations and the 94.7% average cross-configuration transfer reported in Section V suggest that the learned routing patterns are portable rather than overfit to a single benchmark–model combination.

2) *LambdaRank-Based Success Ranking*: The cascade defined in Section IV-B requires a ranking over strategies for each bug. We frame success prediction as a learning-to-rank problem [24] and train a single LightGBM ranker [25] with the LambdaRank pairwise loss.

**Data Construction.** The ranker’s input combines code-level features with a strategy identity component. Specifically, each bug’s feature vector  $\mathbf{z} \in \mathbb{R}^d$  is concatenated with a one-hot strategy indicator  $\mathbf{e}_m \in \{0, 1\}^{|\mathcal{M}|}$ , producing an augmented vector  $[\mathbf{z}; \mathbf{e}_m] \in \mathbb{R}^{d+|\mathcal{M}|}$ . Because the strategy indicator differs for each method, the complete input vector is distinct for every (bug, strategy) pair, and each bug is therefore represented by  $|\mathcal{M}|$  rows. These rows form a query group with binary labels  $y_m(b) \in \{0, 1\}$ , within which the ranker learns to place successful strategies above unsuccessful ones. This design enables the model to capture strategy-specific response patterns from the shared code features through the interaction between  $\mathbf{z}$  and  $\mathbf{e}_m$ .

**Pairwise Training Objective.** The cascade executes strategies in a fixed order and stops at the first success. What matters is the *relative ordering* of strategies within each bug, not the absolute calibration of individual probabilities. LambdaRank is aligned with this requirement, as its pairwise loss directly penalizes inversions between successful and unsuccessful strategies within each query group. Section V empirically confirms that the ranking-based architecture outperforms classification-based alternatives.

#### Pseudo-Probability Conversion.

At inference, the ranker outputs a raw score

$$s_m = R(\mathbf{z}, \mathbf{e}_m)$$

for each strategy. To compute the repair efficiency index in Eq. (5), we apply a per-bug softmax to obtain normalized

pseudo-probabilities

$$\hat{p}_m(z) = \frac{\exp(s_m)}{\sum_{j \in \mathcal{M}} \exp(s_j)},$$

where  $\mathcal{M}$  denotes the candidate strategy set for the given bug. These values should be interpreted as relative preference scores within the candidate set for the same bug, rather than as calibrated absolute success probabilities across bugs.

This distinction is important for hard or even unfixable bugs, where the top-ranked strategy may still have a low absolute chance of success. In APR-ROUTER, however, the purpose of softmax is not probability calibration, but order-preserving normalization for the index computation: LambdaRank is trained for within-bug ordering, and the cascade depth is capped at

$$k = 3,$$

which limits the token cost of ranking mistakes. We therefore use softmax as a practical normalization layer over ranking scores rather than as a claim of globally calibrated success estimation.

3) *Per-Strategy Cost Regression*: The second component of the index is the predicted token cost  $\hat{c}_m(\mathbf{z})$ . Unlike success prediction, where a shared ranker benefits from cross-strategy comparisons, cost prediction is better served by independent models. The token cost distributions of different strategies occupy fundamentally different ranges, from approximately 200 tokens for a zero-shot prompt to over 12,000 for an agentic workflow. A shared regressor would conflate these heterogeneous distributions into a single function, degrading accuracy across all strategies. We therefore train a separate LightGBM regressor for each strategy  $m \in \mathcal{M}$ , yielding  $|\mathcal{M}|$  independent models. Each regressor takes the code feature vector  $\mathbf{z}$  as input and predicts the expected token consumption of strategy  $m$  on bug  $b$ .

To handle the right-skewed cost distribution, we apply a  $\log(1+x)$  transformation to the training targets after clipping to the 2nd–98th percentile range computed on the training set. At inference, the inverse transform  $\exp(\cdot) - 1$  maps predictions back to the original token scale.

## V. EXPERIMENTS

In this section, we evaluate APR-Router’s performance. We first detail the experimental setup, followed by assessments of strategy selection effectiveness (RQ1), design choices (RQ2), efficiency and scalability (RQ3), and sensitivity analysis (RQ4).

### A. Experimental Setup

**Baselines.** Following the taxonomy proposed by Yang et al. [1], we select ten representative APR methods covering all major paradigms as both candidate strategies and evaluation baselines. For zero-shot repair, we include a naive zero-shot prompting baseline [2] and AlphaRepair [3]. For few-shot prompting, we use CEDAR [5]. For chain-of-thought analysis, we include SRepair. For constraint-guided repair, we use CEGPR [11]. For iterative feedback-based repair,

TABLE I  
COMPARISON OF APR-ROUTER WITH TEN REPAIR BASELINES ACROSS FOUR CONFIGURATIONS.

| Method                                           | Qwen2.5-Coder-7B         |                  |                   |                   |                 | LLaMA-3-8B               |                  |                   |                   |                 |
|--------------------------------------------------|--------------------------|------------------|-------------------|-------------------|-----------------|--------------------------|------------------|-------------------|-------------------|-----------------|
|                                                  | PR $\uparrow$            | Tok $\downarrow$ | Cost $\downarrow$ | AIQ $\uparrow$    | APGR $\uparrow$ | PR $\uparrow$            | Tok $\downarrow$ | Cost $\downarrow$ | AIQ $\uparrow$    | APGR $\uparrow$ |
| <b>Dataset: Refactory</b>                        |                          |                  |                   |                   |                 |                          |                  |                   |                   |                 |
| <i>Individual APR Methods</i>                    |                          |                  |                   |                   |                 |                          |                  |                   |                   |                 |
| Zero-Shot                                        | 37.79 $\pm 7.6$          | 287              | 1.72              | .378              | 0.00            | 31.98 $\pm 6.7$          | 268              | 1.21              | .320              | 0.00            |
| AlphaRepair                                      | 40.12 $\pm 7.3$          | 442              | 2.65              | .399              | 0.08            | 36.63 $\pm 7.0$          | 430              | 1.94              | .364              | 0.16            |
| CEDAR                                            | 52.33 $\pm 7.0$          | 923              | 5.54              | .511              | 0.49            | <b>61.63</b> $\pm 7.3$   | 918              | 4.13              | <b>.600</b>       | 0.97            |
| SRepair                                          | <b>66.86</b> $\pm 7.3$   | 913              | 5.48              | <b>.654</b>       | 0.98            | 57.56 $\pm 7.3$          | 948              | 4.27              | .560              | 0.84            |
| CEGPR                                            | 60.47 $\pm 7.6$          | 1,116            | 6.70              | .587              | 0.76            | 56.40 $\pm 7.3$          | 1,145            | 5.15              | .544              | 0.79            |
| ChatRepair                                       | 51.74 $\pm 7.3$          | 944              | 5.67              | .505              | 0.47            | 53.49 $\pm 7.3$          | 980              | 4.41              | .519              | 0.70            |
| Self-Debug                                       | 60.47 $\pm 7.6$          | 1,663            | 9.98              | .575              | 0.74            | 56.40 $\pm 7.6$          | 1,597            | 7.19              | .533              | 0.78            |
| ThinkRepair                                      | 50.58 $\pm 7.3$          | 1,281            | 7.68              | .488              | 0.42            | 55.81 $\pm 7.6$          | 1,301            | 5.86              | .535              | 0.77            |
| ContrastRepair                                   | 51.16 $\pm 7.3$          | 1,021            | 6.13              | .498              | 0.45            | 61.63 $\pm 7.3$          | 1,043            | 4.70              | .597              | 0.97            |
| RepairAgent                                      | 58.14 $\pm 7.6$          | 14,283           | 85.70             | .291              | 0.35            | 58.14 $\pm 7.6$          | 12,496           | 56.20             | .291              | 0.44            |
| <i>Generic LLM Routing Baselines<sup>†</sup></i> |                          |                  |                   |                   |                 |                          |                  |                   |                   |                 |
| Random                                           | 53.01 $\pm 7.8$          | 2,297            | 13.80             | .492              | 0.49            | 53.06 $\pm 7.6$          | 2,134            | 9.60              | .490              | 0.66            |
| Always-Best                                      | 66.86 $\pm 7.3$          | 913              | 5.48              | .654              | 0.98            | 55.81 $\pm 7.6$          | 1,301            | 5.86              | .535              | 0.77            |
| HybridLLM                                        | 59.88 $\pm 7.3$          | 679              | 4.07              | .591              | 0.75            | 54.07 $\pm 7.3$          | 1,092            | 4.92              | .523              | 0.72            |
| RouteLLM                                         | 66.86 $\pm 7.6$          | 924              | 5.54              | .653              | 0.98            | 63.95 $\pm 7.3$          | 928              | 4.18              | .622              | 1.05            |
| Cost-Cascade                                     | <b>67.44</b> $\pm 7.3$   | 1,173            | 7.04              | <b>.653</b>       | 0.99            | <b>65.70</b> $\pm 7.0$   | 1,237            | 5.57              | <b>.631</b>       | 1.09            |
| <b>APR-Router</b>                                | <b>75.58</b> $\pm 6.4$   | 2,063            | 12.40             | <b>.708</b>       | <b>1.22</b>     | <b>79.07</b> $\pm 6.4$   | 2,028            | 9.12              | <b>.734</b>       | <b>1.47</b>     |
| vs. $\uparrow$                                   | $\uparrow 8.7\text{pp}$  |                  |                   | $\uparrow 8.3\%$  | $p < 10^{-3}$   | $\uparrow 17.4\text{pp}$ |                  |                   | $\uparrow 22.3\%$ | $p < 10^{-7}$   |
| <b>Dataset: RunBugRun</b>                        |                          |                  |                   |                   |                 |                          |                  |                   |                   |                 |
| <i>Individual APR Methods</i>                    |                          |                  |                   |                   |                 |                          |                  |                   |                   |                 |
| Zero-Shot                                        | 44.02 $\pm 3.3$          | 509              | 3.05              | .440              | 0.12            | 36.63 $\pm 3.0$          | 470              | 2.11              | .366              | 0.05            |
| AlphaRepair                                      | 42.87 $\pm 3.1$          | 730              | 4.38              | .425              | 0.00            | 36.11 $\pm 2.9$          | 725              | 3.26              | .358              | 0.00            |
| CEDAR                                            | 46.83 $\pm 3.2$          | 1,484            | 8.90              | .452              | 0.39            | 44.12 $\pm 3.0$          | 1,422            | 6.40              | .425              | 0.71            |
| SRepair                                          | <b>52.55</b> $\pm 3.1$   | 1,191            | 7.14              | <b>.513</b>       | 0.98            | 42.35 $\pm 3.2$          | 1,171            | 5.27              | .412              | 0.56            |
| CEGPR                                            | 51.20 $\pm 3.3$          | 1,693            | 10.20             | .491              | 0.82            | 44.85 $\pm 3.1$          | 1,749            | 7.87              | .426              | 0.77            |
| ChatRepair                                       | 48.07 $\pm 3.3$          | 1,491            | 8.95              | .464              | 0.52            | 45.16 $\pm 3.0$          | 1,468            | 6.60              | .434              | 0.80            |
| Self-Debug                                       | 52.55 $\pm 3.2$          | 2,216            | 13.30             | .494              | 0.94            | <b>46.93</b> $\pm 3.0$   | 2,212            | 9.95              | <b>.437</b>       | 0.93            |
| ThinkRepair                                      | 45.47 $\pm 3.1$          | 2,028            | 12.20             | .431              | 0.25            | 43.60 $\pm 3.0$          | 2,020            | 9.09              | .409              | 0.65            |
| ContrastRepair                                   | 47.24 $\pm 3.3$          | 1,483            | 8.90              | .456              | 0.44            | 45.16 $\pm 3.0$          | 1,466            | 6.60              | .434              | 0.80            |
| RepairAgent                                      | 47.35 $\pm 3.2$          | 14,797           | 88.80             | .237              | 0.23            | 43.50 $\pm 3.2$          | 13,189           | 59.30             | .218              | 0.34            |
| <i>Generic LLM Routing Baselines<sup>†</sup></i> |                          |                  |                   |                   |                 |                          |                  |                   |                   |                 |
| Random                                           | 47.72 $\pm 3.2$          | 2,757            | 16.50             | .440              | 0.46            | 42.71 $\pm 3.1$          | 2,581            | 11.60             | .392              | 0.56            |
| Always-Best                                      | 52.55 $\pm 3.1$          | 1,191            | 7.14              | .513              | 0.98            | 45.16 $\pm 3.0$          | 1,466            | 6.60              | .434              | 0.80            |
| HybridLLM                                        | 51.20 $\pm 3.2$          | 968              | 5.81              | .504              | 0.85            | 45.27 $\pm 3.0$          | 1,995            | 8.98              | .426              | 0.80            |
| RouteLLM                                         | 52.76 $\pm 3.1$          | 1,192            | 7.15              | .515              | 1.00            | 45.16 $\pm 3.0$          | 1,469            | 6.61              | .434              | 0.80            |
| Cost-Cascade                                     | <b>59.52</b> $\pm 3.2$   | 1,601            | 9.61              | <b>.572</b>       | 1.65            | <b>51.30</b> $\pm 3.1$   | 1,624            | 7.31              | <b>.490</b>       | 1.34            |
| <b>APR-Router</b>                                | <b>64.20</b> $\pm 3.1$   | 3,077            | 18.50             | <b>.584</b>       | <b>2.01</b>     | <b>59.73</b> $\pm 3.1$   | 3,281            | 14.80             | <b>.531</b>       | <b>1.94</b>     |
| vs. $\uparrow$                                   | $\uparrow 11.7\text{pp}$ |                  |                   | $\uparrow 13.9\%$ | $p < 10^{-10}$  | $\uparrow 12.8\text{pp}$ |                  |                   | $\uparrow 21.5\%$ | $p < 10^{-10}$  |

<sup>†</sup> marks the strongest individual baseline per configuration; where multiple methods tie at the top pass rate, one is marked as representative. PR = Pass Rate (%); Tok = Token Cost; Cost = Avg. Cost ( $\times 10^{-5}$  USD).  $\pm$  values are 95% bootstrap CI half-widths (1,000 resamples). Improvements and McNemar  $p$  are vs.  $\uparrow$ .

we include ChatRepair [7], Self-Debug [8], ThinkRepair, and ContrastRepair. Finally, RepairAgent [4] represents the agentic workflow paradigm.

**LLM Configuration.** We use Qwen2.5-Coder-7B-Instruct<sup>1</sup> and Meta-Llama-3-8B-Instruct<sup>2</sup> as backbone models, with temperature 0.2, top- $p$  0.95. We run 5 trials per method-bug pair to mitigate stochasticity.

**Benchmarks.** We evaluate APR-Router on two single-file Python repair benchmarks. RunBugRun [19] contains 9,608 bugs from competitive-programming tasks, where the buggy program is paired with a unit-test suite and the defect is typically expressed at the function or single-file logic level.

Refactory [26] contains 1,711 bugs from student programming assignments drawn from five introductory tasks. Across the two benchmarks, the evaluated faults are primarily logic-level repair targets such as incorrect control flow, wrong expressions or constants, faulty function-call usage, and output mismatches. We split each split 9:1 at the bug-instance level (seed 42) and stratified on the zero-shot pass/fail outcome to keep the repairable-bug proportion consistent across splits. Pairing each dataset with each backbone LLM yields four configurations, {Qwen, Llama}  $\times$  {Refactory, RunBugRun}.

**Labels.** For each method-bug pair, we define two supervision signals under the same five-trial execution protocol used throughout our empirical evaluation. Any Pass is a binary label set to 1 if at least one of five independent decoding trials passes all unit tests, and 0 otherwise; this signal is

<sup>1</sup><https://huggingface.co/Qwen/Qwen2.5-Coder-7B-Instruct>

<sup>2</sup><https://www.llama.com/models/llama-3>

used to supervise the success ranker. Avg Total Tokens is the mean prompt plus completion token consumption across the same five trials and is used for cost regression after log-transformation. Accordingly, all reported pass-rate results in Sections V-B to V-E are also computed under the Any Pass.

**Evaluation Metrics.** We adopt the AIQ [17] defined in Eq. (3) as the primary metric. As a secondary metric, we report the Aggregate Performance Gain Ratio (APGR) [14], which normalizes the pass-rate improvement against the gap between the weakest and strongest single-strategy baselines and applies the same cost factor as AIQ:

$$\text{APGR} = \frac{\bar{q} - q_{\text{weak}}}{q_{\text{strong}} - q_{\text{weak}}} \cdot \frac{2c_{\text{max}} - \bar{c} - c_{\text{min}}}{2(c_{\text{max}} - c_{\text{min}})}, \quad (6)$$

where  $q_{\text{weak}}$  and  $q_{\text{strong}}$  are the pass rates of the weakest and strongest baselines. APGR equals 0 when the router matches the weak baseline and 1 when it matches the strong baseline at minimum cost; values above 1 indicate gains beyond the strong baseline. In addition, we report pass rate, average tokens, and average cost in USD<sup>3</sup>.

**Statistical Testing.** We report 95% bootstrap confidence intervals with 1,000 resamples for all pass rates and apply McNemar’s test for pairwise comparisons, as the outcome per bug is binary.

**Execution Environment.** All experiments run on an HPC cluster with NVIDIA H200 GPUs, CUDA 13.0, Python 3.13, and vLLM (v0.18.0). Seeds are fixed at 42 for reproducibility. The complete implementation and data are provided in the supplementary materials.

### B. RQ1: Strategy Selection Effectiveness

We compare APR-Router against ten individual APR methods spanning all major repair paradigms and four generic LLM routing baselines. These routing baselines, including HybridLLM and RouteLLM, are originally designed to dispatch queries to different *model tiers*, e.g., routing between a cheap small model and an expensive large model. They do not natively support selecting among APR methods. We therefore re-implement their core selection logic to operate over our ten candidate strategies using the same 114 features. Table I reports all five metrics across four configurations, 2 LLMs  $\times$  2 datasets. Each configuration is evaluated independently with a separate trained router. We mark the strongest individual baseline  $\dagger$  and the strongest routing baseline  $*$  within each configuration and measure all improvements against these per-configuration bests.

**Comparison with individual APR methods.** As shown in the top section of Table I, the strongest individual method varies across configurations. SRepair is among the top methods on both Qwen configurations, while CEDAR ties for first with ContrastRepair on Llama–Refactory and Self-Debug leads on Llama–RunBugRun. Combined with the finding in Section II that the leading method also alternates across complexity bands within a single configuration, this cross-configuration

variation further confirms that no fixed strategy choice can be universally optimal.

APR-Router consistently achieves the highest pass rate and AIQ in all four configurations. The pass-rate improvement over the strongest individual baseline  $\dagger$  ranges from +8.7 pp on Qwen–Refactory to +17.4 pp on Llama–Refactory, with corresponding AIQ gains of +8.3% to +22.3%. All improvements are statistically significant at McNemar  $p < 10^{-3}$ . On Llama–Refactory, the router achieves 79.07% pass rate, reaching 99.3% of the theoretical oracle upper bound of 79.65% as discussed in Section V-E.

**Comparison with LLM routing baselines.** We further compare APR-Router against four LLM routing methods re-implemented for APR strategy selection. These include Random selection, Always-Best-Single that always picks the strategy with the highest training-set AIQ, HybridLLM [13] that falls back from cheap to expensive methods via a learned threshold, and RouteLLM [14] that uses a preference-trained binary classifier. We also include Cost-Cascade, which tries methods in ascending token-cost order until one succeeds.

Among all routing baselines, Cost-Cascade performs best, achieving 51.30–67.44% pass rate by exploiting the correlation between low cost and high success probability for easy bugs. However, its fixed cost-based ordering ignores per-bug feature signals and cannot differentiate among methods with similar costs. APR-Router surpasses Cost-Cascade by +4.7 to +13.4 pp in pass rate and by +2.1 to +16.3% in AIQ across configurations. RouteLLM and HybridLLM are designed for binary model-tier selection and lack the expressiveness to rank among ten heterogeneous strategies. This limits their effectiveness, and they neither consistently beats the Always-Best-Single heuristic across all four configurations.

**Answer to RQ1:** APR-Router improves pass rate by +8.7 to +17.4 pp and AIQ by +8.3% to +22.3% over the strongest individual baseline, with McNemar  $p < 0.001$  in all four configurations. It also outperforms all adapted LLM routing methods, including Cost-Cascade by +4.7 to +13.4 pp in pass rate. This confirms that domain-specific feature-driven routing yields large gains over both fixed strategies and generic routing heuristics.

### C. RQ2: Design Choices

APR-Router involves two key design decisions, namely the model architecture for the ranker and the selection rule that maps predicted scores to a strategy. Table II reports 4-configuration averages for each design variant, and Figure 6 visualizes the cascade depth sensitivity. All  $\Delta\text{AIQ}$  values are measured against the strongest individual baseline per configuration.

**Model architecture.** Table II(a) compares three predictor architectures. Per-Method trains ten independent classifiers, one per strategy. Multi-Target trains a single shared classifier with method one-hot encoding. Learning-to-Rank (LTR) trains a LambdaRank-based ranker that directly optimizes pairwise ordering among strategies. When each architecture selects

<sup>3</sup>DeepInfra pricing for Llama-3-8B; Nebius pricing for Qwen-Coder-7B.

only a single strategy per bug without retry, i.e., the one-shot setting, all three perform similarly, with LTR achieving a slight edge at +7.3%  $\Delta$ AIQ. The gap widens substantially under cascade  $k=3$ , where the router tries up to three strategies in sequence. LTR reaches 69.65% pass rate and 0.639 AIQ, outperforming Per-Method by +1.1% and Multi-Target by +2.7% in AIQ. This advantage arises because LambdaRank directly optimizes the pairwise ranking of passing methods above failing ones within each bug’s query group, which aligns with the cascade’s need to place the correct strategy as early as possible in the attempt sequence.

**Selection rule and cascade depth.** Table II(b) compares selection rules under the LTR architecture. We evaluate three one-shot rules that each select a single strategy per bug. Success Maximization picks the strategy with the highest predicted success probability, ignoring cost entirely. Efficiency Ratio selects the strategy that maximizes the ratio of predicted success to token cost, treating cost as a soft penalty. Budget-Constrained imposes a hard token ceiling at  $1.5\times$  the training-set median and picks the highest-success strategy within that limit. Among these, Efficiency Ratio performs best at 0.590 AIQ by balancing success probability against token cost.

While one-shot rules provide a reasonable baseline, cascade rules consistently outperform them. Cascade  $k=1$  produces results identical to Efficiency Ratio, confirming internal consistency. Moving from  $k=1$  to  $k=3$  yields a large jump in both pass rate, from 60.86% to 69.65%, and AIQ, from 0.590 to 0.639. This demonstrates that allowing up to three ordered attempts substantially improves repair coverage.

Figure 6 further illustrates the trade-off between cascade depth and cost-effectiveness. Pass rate increases monotonically with  $k$ , as deeper cascades provide more chances to find a working strategy. However, AIQ peaks at  $k=3$  and then declines because additional attempts accumulate token costs that outweigh the diminishing pass-rate gains. The pass-rate improvement from  $k=3$  to  $k=10$  is only +3.6pp, while token cost nearly quadruples from 2,612 to 10,078. This makes  $k=3$  the optimal operating point that maximizes cost-adjusted quality.

**Answer to RQ2:** The Learning-to-Rank architecture with cascade  $k=3$  achieves the best overall design. It outperforms alternative architectures by up to +2.7% AIQ under cascade selection. Cascade  $k=3$  is the AIQ sweet spot, beyond which token overhead erodes cost-effectiveness despite continued pass-rate gains.

#### D. RQ3: Efficiency and Scalability

Beyond repair accuracy, we examine APR-Router’s runtime overhead, cascade behavior, and training cost.

**Runtime comparison.** Table III(a) compares APR-Router against three alternative usage patterns. The best single-method baseline always applies the strongest individual strategy per configuration, achieving 56.99% pass rate at 2.6s per bug. APR-Router improves pass rate to 69.65% and AIQ from

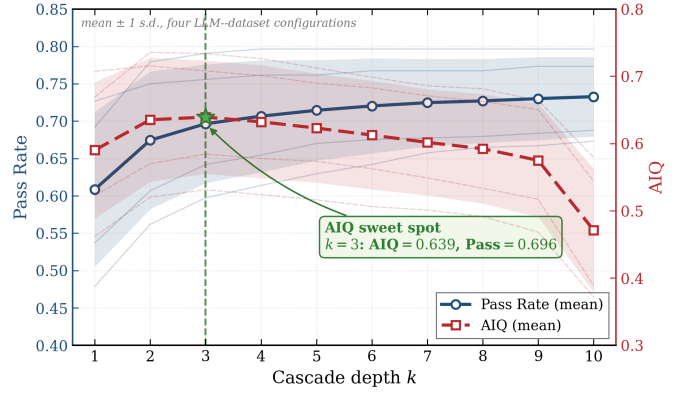


Fig. 6. Cascade depth trade-off of the APR-Router.

TABLE II  
DESIGN CHOICE ANALYSIS (4-CONFIGURATION AVERAGE).

| Variant                         | PR (%) $\uparrow$ | Tok $\downarrow$ | AIQ $\uparrow$ | APGR $\uparrow$ | $\Delta$ AIQ  |
|---------------------------------|-------------------|------------------|----------------|-----------------|---------------|
| <b>(a) Model Architecture</b>   |                   |                  |                |                 |               |
| <i>Best one-shot rule</i>       |                   |                  |                |                 |               |
| Per-Method                      | 59.23             | 992              | .579           | 1.03            | +4.8%         |
| Multi-Target                    | 61.15             | 1,388            | .591           | 1.18            | +6.9%         |
| LTR                             | 61.17             | 1,243            | .593           | 1.16            | +7.3%         |
| <i>Cascade <math>k=3</math></i> |                   |                  |                |                 |               |
| Per-Method                      | 67.25             | 2,026            | .632           | 1.51            | +14.9%        |
| Multi-Target                    | 65.49             | 1,750            | .622           | 1.39            | +12.7%        |
| <b>LTR</b>                      | <b>69.65</b>      | <b>2,612</b>     | <b>.639</b>    | <b>1.66</b>     | <b>+16.5%</b> |
| <b>(b) Selection Rule (LTR)</b> |                   |                  |                |                 |               |
| <i>One-shot rules</i>           |                   |                  |                |                 |               |
| max_pass                        | 61.48             | 2,217            | .572           | 1.14            | +3.5%         |
| cost_eff                        | 60.86             | 1,188            | .590           | 1.13            | +6.8%         |
| budget                          | 59.76             | 1,049            | .583           | 1.05            | +5.2%         |
| <i>Cascade rules</i>            |                   |                  |                |                 |               |
| cascade $k=1$                   | 60.86             | 1,188            | .590           | 1.13            | +6.8%         |
| cascade $k=2$                   | 67.44             | 1,948            | .635           | 1.53            | +15.6%        |
| <b>cascade <math>k=3</math></b> | <b>69.65</b>      | <b>2,612</b>     | <b>.639</b>    | <b>1.66</b>     | <b>+16.5%</b> |
| cascade $k=4$                   | 70.66             | 3,250            | .632           | 1.68            | +15.2%        |
| cascade $k=5$                   | 71.45             | 3,864            | .623           | <b>1.70</b>     | +13.6%        |

0.560 to 0.665. In sequential mode, wall-clock time increases to 5.2s per bug because the cascade tries up to three strategies one after another. In parallel mode, where all cascade candidates are launched concurrently, wall-clock time drops to 3.0s per bug without sacrificing any pass rate or AIQ compared to the sequential cascade. Exhaustive evaluation of all ten methods reaches 73.27% pass rate but consumes 24,378 tokens and 48.8s per bug, yielding the lowest AIQ of 0.366. RepairAgent consumes 13,691 tokens per bug for only 51.78% pass rate.

Table III(b) breaks down the router’s own overhead. Feature extraction takes 4.0ms per bug and LightGBM prediction takes 0.02ms per bug. Compared to a single LLM inference call at 4,876ms, the total routing overhead is less than 0.1% of the repair cost. This confirms that the router adds negligible latency to the repair pipeline.

**Cascade attempt distribution.** Table IV and Figure 7 show that on average 60.9% of bugs are fixed on the first

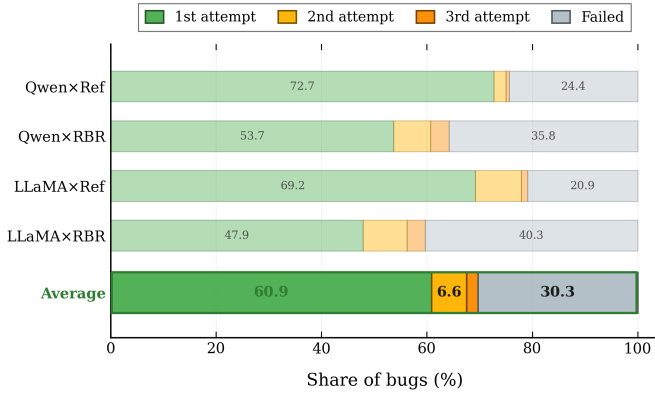


Fig. 7. Cascade attempt distribution ( $k=3$ ).

attempt, 8.8% on the second or third, and 30.3% remain unfixed, yielding an average depth of 1.72 attempts per bug. The high first-try rate reflects the quality of the learned ranking, while the additional 8.8% shows that the cascade’s value lies in compensating for local ranking errors rather than in exhaustive retry. First-try rates are higher on Refactory (72.7% Qwen, 69.2% Llama) than on RunBugRun (53.7% and 47.9%), consistent with the greater difficulty of competitive-programming bugs.

**Training data investment.** Table V(a) reports the one-time cost of collecting training data for each configuration. Building the training set requires running all ten strategies on the training corpus for five trials each. This amounts to 160–1,200M tokens per configuration, corresponding to \$9.6–\$72 at current API pricing. The return on this investment is substantial. For instance, LLaMA×Refactory spends \$9.60 on training data and gains +17.4pp in pass rate. At deployment, each additional automated fix saves roughly \$4.17 (assuming \$50/hr developer time and 5 min per unfixed bug), so the training investment is recovered within 20–140 deployed bugs.

Table V(b) shows that this training cost can be further amortized through cross-configuration transfer. A router trained on one configuration and tested on another retains 94.7% of the in-configuration pass rate on average. Same-dataset cross-model transfer is particularly strong, e.g., training on Qwen×Refactory and testing on LLaMA×Refactory retains 94.9% because the bugs are identical and only the LLM’s repair behavior changes. This means that switching backbone models on the same bug corpus does not require retraining, and the one-time collection cost is effectively shared across all compatible configurations.

**Answer to RQ3:** APR-Router’s routing overhead is less than 0.1% of a single LLM call. The cascade fixes 60.9% of bugs on the first attempt, with an average depth of 1.72. Training data collection costs \$9.6–\$72 per configuration, is recovered within 20–140 deployed bugs, and can be amortized across configurations with 94.7% average cross-transfer retention.

TABLE III  
DEPLOYMENT EFFICIENCY AND ROUTER OVERHEAD (4-CONFIG AVERAGE).

| Scenario                      | PR (%)↑ | Tok↓   | AIQ↑ | APGR↑ | Time (s)      |      |
|-------------------------------|---------|--------|------|-------|---------------|------|
|                               |         |        |      |       | /bug          | /fix |
| (a) Deployment Scenarios      |         |        |      |       |               |      |
| Best Single <sup>a</sup>      | 56.99   | 1,308  | .560 | 0.98  | 2.6           | 4.9  |
| APR-Router                    | 69.65   | 2,612  | .665 | 1.73  | 5.2           | 7.8  |
| RepairAgent                   | 51.78   | 13,691 | .372 | 0.49  | 27.4          | 53.8 |
| Exhaustive                    | 73.27   | 24,378 | .366 | 1.07  | 48.8          | 67.4 |
| (b) Router Overhead Breakdown |         |        |      |       |               |      |
| Feature extraction            |         |        |      |       | 4.00 ms/bug   |      |
| LightGBM prediction + ranking |         |        |      |       | 0.02 ms/bug   |      |
| Single LLM inference call     |         |        |      |       | 4,876 ms/call |      |
| Router overhead / LLM call    |         |        |      |       | < 0.1%        |      |

<sup>a</sup> Always applies the strongest single method per configuration.

TABLE IV  
CASCADE ATTEMPT DISTRIBUTION FOR  $k=3$ .

| Config.        | Attempt Outcome (%) |     |     |      | Fixed | Tried |
|----------------|---------------------|-----|-----|------|-------|-------|
|                | 1st                 | 2nd | 3rd | Fail |       |       |
| Qwen × Ref     | 72.7                | 2.3 | 0.6 | 24.4 | 75.6  | 1.52  |
| Qwen × RBR     | 53.7                | 7.0 | 3.5 | 35.8 | 64.2  | 1.86  |
| LLaMA × Ref    | 69.2                | 8.7 | 1.2 | 20.9 | 79.1  | 1.53  |
| LLaMA × RBR    | 47.9                | 8.3 | 3.5 | 40.3 | 59.7  | 1.96  |
| <b>Average</b> | 60.9                | 6.6 | 2.2 | 30.3 | 69.7  | 1.72  |

TABLE V  
TRAINING COST AND CROSS-CONFIGURATION TRANSFERABILITY.

| Config.                                  | Bugs               | Tokens (M) | Cost    | ΔPR     |
|------------------------------------------|--------------------|------------|---------|---------|
| <b>(a) Training Data Collection Cost</b> |                    |            |         |         |
| Qwen × Ref                               | 1,539              | 180        | \$10.80 | +8.7pp  |
| Qwen × RBR                               | 8,647              | 1,200      | \$72.00 | +11.7pp |
| LLaMA × Ref                              | 1,539              | 160        | \$9.60  | +17.4pp |
| LLaMA × RBR                              | 8,647              | 1,100      | \$66.00 | +12.8pp |
| <b>(b) Cross-Config Transfer</b>         |                    |            |         |         |
| Train                                    | Test Configuration |            |         |         |
|                                          | Q×Ref              | Q×RBR      | L×Ref   | L×RBR   |
| Qwen × Ref                               | 100%               | 98.7%      | 94.9%   | 98.4%   |
| Qwen × RBR                               | 96.1%              | 100%       | 93.4%   | 100.4%  |
| LLaMA × Ref                              | 93.8%              | 90.3%      | 100%    | 95.6%   |
| LLaMA × RBR                              | 88.5%              | 96.3%      | 90.4%   | 100%    |

Legend: ≥95% (green), 90–95% (yellow), <90% (red). Avg off-diag: **94.7%**

#### E. RQ4: Sensitivity Analysis

We examine the sensitivity of APR-Router to its distance from the theoretical optimum, the cost-efficiency weight  $\alpha$ , and the statistical significance of its improvements.

**Oracle analysis.** Table VI(a) compares APR-Router against an oracle that selects the best method per bug with perfect foresight. The router captures 94.8% of the oracle pass rate on average, reaching 99.3% on LLaMA×Refactory where only one additional bug separates the router from the theoretical ceiling. The gap is widest on LLaMA×RunBugRun at 88.7%, reflecting the greater difficulty of competitive-programming

TABLE VI  
ORACLE ANALYSIS, AND STATISTICAL ROBUSTNESS.

| (a) Oracle Upper Bound |               |        |         |       |
|------------------------|---------------|--------|---------|-------|
| Config.                | Pass Rate (%) |        | Capture |       |
|                        | Oracle        | Router | PR      | AIQ   |
| Qwen $\times$ Ref      | 77.33         | 75.58  | 97.7%   | 92.5% |
| Qwen $\times$ RBR      | 68.78         | 64.20  | 93.3%   | 85.7% |
| LLaMA $\times$ Ref     | 79.65         | 79.07  | 99.3%   | 93.3% |
| LLaMA $\times$ RBR     | 67.33         | 59.73  | 88.7%   | 79.9% |
| Average                | 73.27         | 69.64  | 94.8%   | 87.8% |

| (b) Statistical Robustness |               |                  |        |                           |
|----------------------------|---------------|------------------|--------|---------------------------|
| Config.                    | vs. $\dagger$ | Cliff's $\delta$ | Effect | McNemar $p$               |
| Qwen $\times$ Ref          | SRepair       | 0.267            | small  | $< 10^{-3}$ $\checkmark$  |
| Qwen $\times$ RBR          | SRepair       | 0.233            | small  | $< 10^{-10}$ $\checkmark$ |
| LLaMA $\times$ Ref         | CEDAR         | 0.128            | negl.  | $< 10^{-7}$ $\checkmark$  |
| LLaMA $\times$ RBR         | Self-Debug    | 0.173            | small  | $< 10^{-10}$ $\checkmark$ |

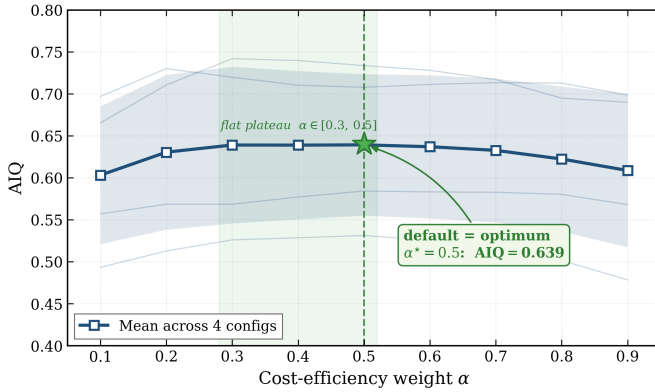


Fig. 8. Cascade AIQ versus the cost-efficiency weight  $\alpha$

bugs. The AIQ capture rate is lower at 87.8% because the cascade accumulates token costs from failed attempts that the oracle avoids by always selecting correctly on the first try.

**Cost-efficiency weight  $\alpha$ .** The weight  $\alpha$  in the cost-efficiency score balances predicted success probability against token cost. Figure 8 shows AIQ across  $\alpha \in [0.1, 0.9]$  averaged over four configurations. AIQ forms a flat plateau over  $\alpha \in [0.3, 0.5]$  and peaks at the default  $\alpha=0.5$  with  $\text{AIQ}=0.639$ , degrading at both extremes where the router either over-saves cost or over-chases pass rate. The plateau confirms that APR-Router is robust to the exact choice of  $\alpha$ .

**Statistical robustness.** Table VI(b) confirms that APR-Router’s improvements are statistically significant in all four configurations (McNemar  $p < 10^{-3}$ ), with small Cliff’s  $\delta$  effect sizes.

**Answer to RQ4:** The router captures 94.8% of the oracle pass rate. The cost-efficiency weight  $\alpha$  is robust across  $[0.3, 0.5]$ , and all improvements are statistically significant at  $p < 0.001$ .

## VI. CONCLUSION

In this paper, we propose APR-Router, the first cascade-based routing framework for LLM-based automated program repair. Our key insight is that no single repair strategy dominates across all defects, yet a short, intelligently ordered sequence can close the majority of the gap to the theoretical oracle. APR-Router operationalizes this insight through domain-specific feature engineering and a learned cost-effectiveness ranking, enabling adaptive per-bug strategy selection with negligible overhead. Experiments confirm that APR-Router consistently and significantly outperforms both fixed repair strategies and generic LLM routing methods. In the future, we plan to extend APR-Router to cross-model cascades.

## VII. THREATS TO VALIDITY

**Internal.** LLM outputs are stochastic. We mitigate this with 5 trials per method–bug pair and a stratified 9:1 bug-level split with a fixed seed. The “Any Pass” label is generous relative to single-trial deployment, but it is standard in multi-trial APR evaluation [4], [7] and preserves relative rankings across all methods. Our integrity audit programmatically verifies that no ground-truth solution is accessible during feature extraction.

**External.** Both benchmarks consist of single-file Python programs. Repository-level bugs involve multi-file dependencies that our current features do not capture. However, the routing principle that no single strategy dominates is likely to hold even more strongly at repository scale. Our feature design is modular and extensible to richer program representations. Similarly, our 7B/8B backbones may not reflect frontier-model behavior, but cost-sensitive routing becomes *more* valuable at larger scales where per-token prices are 10–100 $\times$  higher, and our cross-transfer results (94.7% retention) suggest that routing signals generalize across backbones.

## ACKNOWLEDGEMENT

We thank the anonymous reviewers for their helpful comments and suggestions. We acknowledge the use of AI-based assistants during the research and writing process to support brainstorming and improve the clarity of the manuscript. All substantive ideas, arguments, interpretations, and conclusions presented in this article are the authors’ own. This research includes computations using the computational cluster Katana supported by Research Technology Services at UNSW Sydney [27]. We also thank Craig Atkinson, Patrick Tung, and the rest of the Research Technology Services team at UNSW Sydney for their technical support.

## REFERENCES

- [1] B. Yang, Z. Cai, F. Liu, B. Le, L. Zhang, T. F. Bissyandé, Y. Liu, and H. Tian, “A survey of llm-based automated program repair: Taxonomies, design paradigms, and applications,” 2025. [Online]. Available: <https://arxiv.org/abs/2506.23749>
- [2] C. S. Xia, Y. Wei, and L. Zhang, “Automated program repair in the era of large pre-trained language models,” in *Proceedings of the 45th International Conference on Software Engineering*, ser. ICSE ’23. IEEE Press, 2023, p. 1482–1494. [Online]. Available: <https://doi.org/10.1109/ICSE48619.2023.00129>

- [3] C. S. Xia and L. Zhang, “Less training, more repairing please: revisiting automated program repair via zero-shot learning,” in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 959–971. [Online]. Available: <https://doi.org/10.1145/3540250.3549101>
- [4] I. Bouzenia, P. Devanbu, and M. Pradel, “Repairagent: An autonomous, llm-based agent for program repair,” in *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*, ser. ICSE ’25. IEEE Press, 2025, p. 2188–2200. [Online]. Available: <https://doi.org/10.1109/ICSE55347.2025.00157>
- [5] N. Nashid, M. Sintaha, and A. Mesbah, “Retrieval-based prompt selection for code-related few-shot learning,” in *Proceedings of the 45th International Conference on Software Engineering*, ser. ICSE ’23. IEEE Press, 2023, p. 2450–2462. [Online]. Available: <https://doi.org/10.1109/ICSE48619.2023.00205>
- [6] J. Xiang, X. Xu, F. Kong, M. Wu, Z. Zhang, H. Zhang, and Y. Zhang, “How far can we go with practical function-level program repair?” 2024. [Online]. Available: <https://arxiv.org/abs/2404.12833>
- [7] C. S. Xia and L. Zhang, “Automated program repair via conversation: Fixing 162 out of 337 bugs for \$0.42 each using chatgpt,” in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2024. New York, NY, USA: Association for Computing Machinery, 2024, p. 819–831. [Online]. Available: <https://doi.org/10.1145/3650212.3680323>
- [8] X. Chen, M. Lin, N. Schärli, and D. Zhou, “Teaching large language models to self-debug,” in *The Twelfth International Conference on Learning Representations*, 2024. [Online]. Available: <https://openreview.net/forum?id=KuPixIqPiq>
- [9] X. Yin, C. Ni, S. Wang, Z. Li, L. Zeng, and X. Yang, “Thinkrepair: Self-directed automated program repair,” in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2024. New York, NY, USA: Association for Computing Machinery, 2024, p. 1274–1286. [Online]. Available: <https://doi.org/10.1145/3650212.3680359>
- [10] J. Kong, X. Xie, M. Cheng, S. Liu, X. Du, and Q. Guo, “Contrastrepair: Enhancing conversation-based automated program repair via contrastive test case pairs,” *ACM Trans. Softw. Eng. Methodol.*, vol. 34, no. 8, Oct. 2025. [Online]. Available: <https://doi.org/10.1145/3719345>
- [11] P. Orvalho, M. Janota, and V. M. Manquinho, “Counterexample guided program repair using zero-shot learning and maxsat-based fault localization,” in *Proceedings of the Thirty-Ninth AAAI Conference on Artificial Intelligence and Thirty-Seventh Conference on Innovative Applications of Artificial Intelligence and Fifteenth Symposium on Educational Advances in Artificial Intelligence*, ser. AAAI’25/IAAI’25/EAAI’25. AAAI Press, 2025. [Online]. Available: <https://doi.org/10.1609/aaai.v39i1.32046>
- [12] Anonymous, “How do coding agents spend your money? analyzing and predicting token consumptions in agentic coding tasks,” in *Submitted to ACL Rolling Review - January 2026*, 2026, under review. [Online]. Available: <https://openreview.net/forum?id=ZQPRZBxWEG>
- [13] D. Ding, A. Mallick, C. Wang, R. Sim, S. Mukherjee, V. Rühle, L. V. S. Lakshmanan, and A. H. Awadallah, “Hybrid LLM: Cost-efficient and quality-aware query routing,” in *The Twelfth International Conference on Learning Representations*, 2024. [Online]. Available: <https://openreview.net/forum?id=02f3mUtqnM>
- [14] I. Ong, A. Almahairi, V. Wu, W.-L. Chiang, T. Wu, J. E. Gonzalez, M. W. Kadous, and I. Stoica, “RouteLLM: Learning to route LLMs from preference data,” in *The Thirteenth International Conference on Learning Representations*, 2025. [Online]. Available: <https://openreview.net/forum?id=8sSqNntaMr>
- [15] J. Dekoninck, M. Baader, and M. Vechev, “A unified approach to routing and cascading for LLMs,” in *First Workshop on Scalable Optimization for Efficient and Adaptive Foundation Models*, 2025. [Online]. Available: <https://openreview.net/forum?id=qe8BfREMrb>
- [16] C. Varangot-Reille, C. Bouvard, A. Gourru, M. Ciancone, M. Schaeffer, and F. Jacquenet, “Doing more with less: A survey on routing strategies for resource optimisation in large language model-based systems,” 2025. [Online]. Available: <https://arxiv.org/abs/2502.00409>
- [17] Q. J. Hu, J. Bieker, X. Li, N. Jiang, B. Keigwin, G. Ranganath, K. Keutzer, and S. K. Upadhyay, “Routerbench: A benchmark for multi-LLM routing system,” in *Agentic Markets Workshop at ICML 2024*, 2024. [Online]. Available: <https://openreview.net/forum?id=IVXmV8Uxwh>
- [18] Y. Chen, J. Wu, X. Ling, C. Li, Z. Rui, T. Luo, and Y. Wu, “When large language models confront repository-level automatic program repair: How well they done?” in *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, ser. ICSE-Companion ’24. New York, NY, USA: Association for Computing Machinery, 2024, p. 459–471. [Online]. Available: <https://doi.org/10.1145/3639478.3647633>
- [19] J. A. Prenner and R. Robbes, “Runbugrun: An executable dataset for automated program repair,” *Empirical Softw. Engg.*, vol. 31, no. 4, Mar. 2026. [Online]. Available: <https://doi.org/10.1007/s10664-025-10790-3>
- [20] Q. Zhang, C. Fang, Y. Xie, Y. Zhang, Y. Yang, W. Sun, S. Yu, and Z. Chen, “A survey on large language models for software engineering,” *Science China Information Sciences*, 2025.
- [21] M. L. Weitzman, “Optimal search for the best alternative,” *Econometrica*, vol. 47, no. 3, pp. 641–654, 1979.
- [22] J. F. Lawless, *Statistical Models and Methods for Lifetime Data*, 2nd ed. Wiley, 2003.
- [23] T. M. Cover and J. A. Thomas, *Elements of Information Theory*, 2nd ed. Wiley, 2006.
- [24] C. J. C. Burges, “From RankNet to LambdaRank to LambdaMART: An overview,” *Microsoft Research Technical Report MSR-TR-2010-82*, 2010.
- [25] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu, “Lightgbm: a highly efficient gradient boosting decision tree,” in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS’17. Red Hook, NY, USA: Curran Associates Inc., 2017, p. 3149–3157.
- [26] Y. Hu, U. Z. Ahmed, S. Mehtaev, B. Leong, and A. Roychoudhury, “Refactoring based program repair applied to programming assignments,” in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 388–398.
- [27] “Katana,” 2010. [Online]. Available: <https://doi.org/10.26190/669x-a286>