

Rethinking Automated Program Repair: The Impact of Bug Complexity, Fault Localization, and LLM Cost-Efficiency

Junchi Liu ✉ 

Colorado State University, Fort Collins, CO, US

Ali Bigdeli ✉ 

Colorado State University, Fort Collins, CO, US

Roya Daneshi ✉ 

Colorado State University, Fort Collins, CO, US

Atu Ambala ✉

Colorado State University, Fort Collins, CO, US

Sudipto Ghosh ✉ 

Colorado State University, Fort Collins, CO, US

Fabio Santos ✉ 

Colorado State University, Fort Collins, CO, US

Abstract

Background. Software bugs remain a critical challenge in development, necessitating effective Automated Program Repair (APR) techniques. While Large Language Model (LLM)-based APR systems have shown promise, prior studies primarily focus on overall repair effectiveness. The effects of bug complexity, fault localization, reasoning settings, and repair cost-effectiveness remain insufficiently explored.

Aims. This study presents a comprehensive empirical analysis of LLM-based APR, focusing on how repair performance is shaped by bug complexity, fault localization, reasoning settings, and costs.

Method. We construct a curated dataset by collecting algorithmic bugs from AtCoder, a competitive programming platform. We evaluate two APR techniques (ChatRepair and CodeCorrector) using three LLMs (DeepSeek, GPT, and Llama), with multiple model variants and reasoning settings, and examine their performance across diverse levels of bug complexity and localization strategies through a multi-dimensional empirical framework and statistical analysis.

Results. Although structurally complex bugs and imprecise fault localization make repair more challenging, LLM-based APR techniques still achieve competitive repair effectiveness. Imprecise fault localization can substantially enlarge the performance gap between APR techniques. Furthermore, higher-cost LLMs and stronger reasoning settings do not consistently yield better cost-efficiency, revealing a nontrivial trade-off between repair effectiveness and computational cost. We further observe that the impact of reasoning strategies varies considerably across different LLM families, affecting both repair effectiveness and cost-efficiency.

Conclusions. Over 50% of moderately complex bugs can be repaired by low-cost LLM-based APR techniques. The repair effectiveness gap between APR techniques becomes larger as fault localization becomes less precise. GPT-5 repairs 7 and 39 more complex bugs than DeepSeek-V4-pro and DeepSeek-V3.2, respectively; whereas the total repair cost of DeepSeek-V3.2 across the non-reasoning and reasoning stages is only approximately 4.6% and 7.7% of GPT-5 and DeepSeek-V4-pro, respectively.

2012 ACM Subject Classification Software and its engineering

Keywords and phrases Automated Program Repair, Large Language Models, Fault Localization, Bug Complexity, Cost-efficiency



© Junchi Liu, Ali Bigdeli, Roya Daneshi, Atu Ambala, Sudipto Ghosh, and Fabio Santos; licensed under Creative Commons License CC-BY 4.0

20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).

Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;

Article No. 40; pp. 40:1–40:21



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Digital Object Identifier 10.4230/LIPIcs.ESEM.2026.40

Category Technical Track Paper

Related Version *Full Version*: <https://arxiv.org/abs/2608.14065>

Supplementary Material *Software (Source Code)*: <https://github.com/Leo6-sys/Rethink-APR-Empirical-Study-2026>, archived at `swb:1:dir:9923ceba2a738718f2d1c91d7b8a6d8023a388db`

Dataset (Dataset): <https://doi.org/10.5281/zenodo.20257267>

1 Introduction

In recent years, the rapid advancement of Generative AI, driven by intense competition among technology companies, has significantly accelerated the development of Large Language Models (LLMs) [29]. Current LLMs have shown substantial improvements in their ability to understand, reason, and generate programs [47, 3, 41]. As a result, they have been increasingly applied to a wide range of software engineering tasks, including code generation, code summarization, and bug detection [14, 22, 18].

In the context of Automated Program Repair (APR), LLM-based approaches have demonstrated significant advantages over traditional techniques, achieving strong performance across widely used benchmarks such as Defects4J [19], Bugs.jar [40], and QuixBugs [24]. Notably, many APR techniques achieve near-perfect performance on QuixBugs and IntroClass [12], small-scale algorithmic datasets primarily composed of simple, single-hunk bugs, suggesting that these datasets may be insufficiently challenging for evaluating the effectiveness of modern LLM-based APR approaches. Also, as LLMs exhibit strong performance in code understanding and reasoning, recent studies have increasingly explored end-to-end or self-directed APR approaches that reduce reliance on externally provided fault localization (FL) information [17, 21]. Nevertheless, the impact of the absence of FL on repair effectiveness has not been systematically evaluated in the agent era. To address these unexplored gaps, our study introduces a more challenging algorithmic dataset, featuring bugs with varied complexity levels, and uses the tools of Git [43], srcML [6], and GumTreeDiff [11] to systematically and automatically categorize bugs based on their complexity, enabling a fine-grained empirical analysis of repair performance across different bug complexity levels as well as under accurate, vague, or no fault localization settings.

Additionally, the reliance on commercial LLM APIs produces non-negligible computational costs [44]. APR tasks often require generating numerous candidate patches [23, 51], resulting in substantial output token usage, particularly under reasoning-intensive settings [49, 13]. Consequently, different LLM choices and reasoning settings can lead to significant cost discrepancies. Such costs are often overlooked in prior APR evaluations, despite being critical for practical deployment. In our study, we incorporate cost-efficiency metrics and standardize the unit to \$1 for fair comparison. Specifically, we define Repaired Bugs/\$ and Attempts/\$ as the numbers of successfully repaired bugs and repair attempts divided by the total API cost, respectively, where the cost is calculated from input/output token usage and the corresponding API prices.

We present a comprehensive evaluation of two LLM-based APR techniques across three LLMs, including multiple versions and reasoning settings. We employ multi-dimensional metrics (number of compilable patches, plausible patches, and repaired bugs, Top- k , consistency, and cost-efficiency metrics), and statistical tests to address the following research questions (RQs):

RQ1. *How does bug complexity affect the repair performance of LLM-based APR?*

RQ2. *To what extent does fault localization influence the effectiveness of LLM-based APR?*

RQ3. *How do LLM choices and reasoning settings impact the repair performance–cost trade-off in APR?*

The results of this study show that:

- Repair performance varies substantially across bug complexity, with the simplest complexity bugs being consistently the easiest to fix, as expected, but repair performance remains at a relatively considerable level as complexity increases, suggesting that LLM-based APR approaches maintain a stable reasoning ability even on structurally complex bugs.
- FL plays a critical role in improving both repair effectiveness and patch consistency. Although more precise localization generally leads to better repair performance, appropriate APR frameworks can still maintain relatively stable repair effectiveness in settings with vague or missing localization, suggesting potential for practical end-to-end APR scenarios. Also, the impact of FL precision on repair effectiveness decreases as bug complexity increases.
- More costly LLMs and reasoning strategies do not always produce proportional repair improvements; the increased cost should be considered seriously, and reasoning strategies do not consistently yield ideal outcomes, revealing a nontrivial trade-off between performance and efficiency.

Our contributions include:

- (i) We empirically analyze the impact of bug complexity and fault localization on LLM-based APR performance, and incorporate unit-standardized cost-efficiency metrics to quantify the trade-off between repair effectiveness and computational cost. We also provide an extensible experimentation infrastructure to support future studies, replication efforts, and systematic evaluation of LLM-based APR techniques.
- (ii) Our study offers novel insights into how repair effectiveness varies with bug complexity and reveals the nontrivial cost implications of different LLM choices and reasoning settings. In particular, by systematically evaluating settings with accurate, vague, and no fault localization, we demonstrate the feasibility and potential of end-to-end LLM-based APR.
- (iii) Additionally, our results across multiple LLMs and reasoning strategies provide practical guidance for selecting appropriate LLMs under the trade-off between the repair performance and economic cost, highlighting the importance of balancing repair performance and computational cost.

2 Background and Related Work

This section provides a relevant review of the literature and highlights the limitations of existing studies.

2.1 Previous Evaluations of LLM-based APR

Recent evaluations of APR are primarily based on the Defects4J Benchmark [19, 26], which has become the dominant benchmark for APR evaluation. However, heavy reliance on a single benchmark can limit the diversity of bug types and weaken the evaluation of APR approaches' generalizability. Nevertheless, several variants of Defects4J have been proposed, including DEFECTS4J-TRANS [20], and MuBench [33]. These benchmarks still primarily

contain software-engineering-oriented bugs [39]. Meanwhile, algorithm-oriented datasets such as QuixBugs [24], IntroClass, and ManyBugs [12] have been widely utilized to evaluate the generalizability of APR techniques. However, the difficulty of these datasets is often insufficient to challenge recent LLM-based APR approaches [46], thereby leading to an over-estimation of their effectiveness. In addition, the continued reuse of long-standing benchmark datasets introduces data leakage issues [39], potentially leading to biased evaluation results. Furthermore, Bean et al. [4] point out that the lack of statistical testing in the majority of benchmarks reduces the reliability of empirical findings.

We collect recent real-world bugs from AtCoder [15] as an extension of algorithm-oriented datasets. Then, Git, srcML, and GumTreeDiff are utilized to categorize bugs based on complexity. Finally, we conduct an empirical evaluation of LLM-based APR under different bug complexity and FL settings, with a cost-efficiency and statistical analysis.

2.2 APR Techniques

APR techniques have advanced significantly with the development of LLMs' code-reasoning and code-generation capabilities [52]. With the success of LLMs in code reasoning and generation [16], recent studies have increasingly explored their applications in APR. Xia and Zhang [46] introduced ChatRepair, which iteratively generates and refines patches by leveraging feedback from both failed and successful repair attempts, and has been widely viewed as a representative baseline in subsequent studies. For instance, Yin et al. [48] proposed ThinkRepair, which uses Chain-of-Thought (CoT) prompting [45] to enable reasoning prior to patch generation. Bouzenia et al. [5] introduced RepairAgent, the first autonomous agent-based framework for APR. Li et al. [21] further proposed CodeCorrector, an end-to-end framework that first analyzes programs to infer repair directions and then generates patches, achieving the best performance on the Defects4J v1.2 dataset.

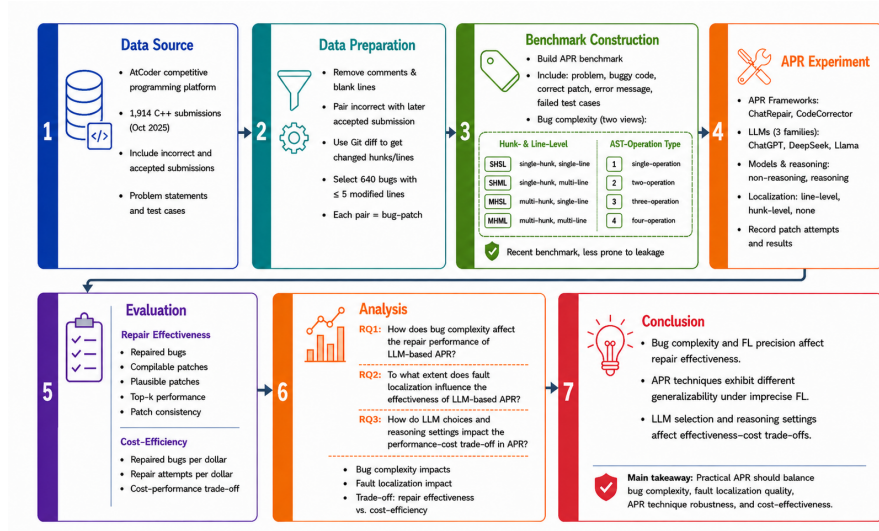
In this study, we select ChatRepair as a representative LLM-based APR approach and CodeCorrector as a recent LLM-based method that demonstrates strong performance in prior studies.

2.3 Evaluation Metrics

Existing APR evaluation metrics primarily focus on repair effectiveness, including the number of repaired bugs, $Top - k$, compilable patches, and plausible patches [10]. To further assess patch quality, Dai et al. [7] introduced the consistency metric, which measures the extent of changes between patches and buggy programs. However, these metrics fail to capture cost-efficiency, which has become increasingly important in the empirical study of LLM-based APR. Although recent studies have begun to report costs [46, 5, 21], they use inconsistent units, such as cost per bug, cost per successful repair, or cost per attempt, making it difficult to perform fair, direct comparisons across approaches. This lack of a unified cost metric limits the systematic evaluation of performance-cost trade-offs. Thus, we standardize costs to a common monetary unit, enabling a direct trade-off between repair performance and cost.

3 Research Design

In this section, we discuss the research methodology.



■ **Figure 1** Flow of the research method.

3.1 Experimental Infrastructure

The experimental infrastructure is deployed on computing servers with isolated execution environments to ensure controlled access and protected communication. Shared storage and distributed computing resources are utilized to support large-scale APR experiments across multiple machines, improving the overall experimental efficiency. The experimental infrastructure consists of five components: the dataset, APR techniques, LLMs, additional tools, and Python scripts.

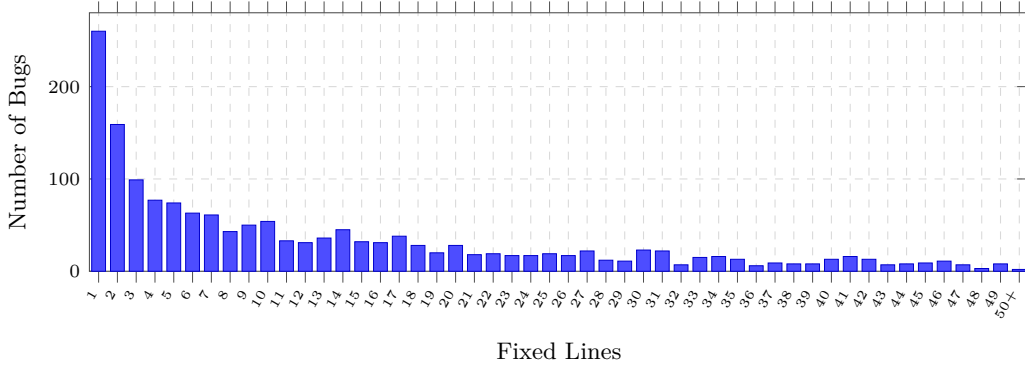
The dataset is constructed from AtCoder and includes problem descriptions, pairs of incorrect and correct C++ programs, and corresponding test cases (Section 3.3). The tools include the APR tools, along with Git, srcML, and GumTreeDiff. The AtCoder Library (ACL) is included as an external dependency to support compilation. Custom Python scripts are used for data analysis and dataset filtering. All the above components are maintained within our isolated environment.

We utilize multiple LLMs in our study: gpt-5-2025-08-07 (GPT-5) [32], gpt-4o-mini-2024-07-18 (GPT-4o-mini) [31], DeepSeek-V3.2 [8], DeepSeek-V4-pro [9], Llama-3.1-8B [2], and Llama-3.1-405B [1]. GPT-5 and GPT-4o-mini use the OpenAI API. DeepSeek-V3.2 and DeepSeek-V4-pro use the DeepSeek API. Llama-3.1 is deployed in our experimentation infrastructure.

3.2 Experimental Workflow

Figure 1 illustrates the overall experimental infrastructure and workflow of our study. We first collect buggy and corrected programs from AtCoder and construct an automated benchmark and evaluation pipeline for LLM-based APR experiments (details are provided in subsection 3.3).

Next, we analyze bug complexity from multiple perspectives. Git-based differencing is utilized to extract hunk-line complexity information, while srcML [6] and GumTreeDiff [11] are employed to measure structural complexity at the Abstract Syntax Tree (AST) level (details are provided in subsection 3.4).



■ **Figure 2** Distribution of Fixed Size Across Raw Data.

Based on the constructed benchmark and complexity analysis, we deploy APR frameworks and integrate multiple LLM APIs with different reasoning settings to conduct automated repair experiments under FL settings (details are provided in subsection 3.5).

Finally, we perform a comprehensive empirical evaluation to investigate the impact of bug complexity, fault localization, and cost-efficiency trade-offs on the performance of LLM-based APR techniques. The corresponding findings and experimental analyses are presented in section 4.

3.3 Dataset

Existing APR benchmarks have been widely used, raising concerns about data contamination in the evaluation of LLM-based APR. To mitigate this risk, we construct our dataset from the recent AtCoder submissions rather than relying on existing benchmarks. We focus on C++ programs because comparing FL settings and bug complexity across programming languages with different characteristics could introduce additional bias. This design enables a more controlled evaluation of LLM-based APR.

In this study, we use AtCoder, a Japanese competitive programming platform, and collect 1,914 C++ source code submissions in October, 2025. Our data comes from regular contests, which typically include algorithmically challenging problems designed to assess intermediate-to-advanced programming skills. We pair every incorrect program and correct program from the same participant as a pair of a bug and a correct patch (bug-patch pair). For each bug-patch pair, we use the `git diff` command to generate a diff file that captures the corresponding code changes.

Before generating diff files using Git, we remove all comments and redundant blank lines. The diff files reflect line-level code changes, where ‘-’ denotes deleted (buggy) lines N_- , and ‘+’ denotes added (patch) lines N_+ . For each hunk, the number of fixed lines is defined as $N_{\text{fixed}} = \max(N_+, N_-)$.

The distribution of bug-fix sizes across the original 1,914 C++ files is shown in Figure 2. Based on preliminary experiments comparing different fixed line thresholds on samples drawn from the full dataset, we observe that APR becomes considerably more difficult and computationally expensive as the number of fixed lines increases. Considering both repair feasibility and computational cost, we set a threshold of five fixed lines to focus on small-scope program repair. Finally, 640 bugs remain for our empirical study.

■ **Table 1** Hunk-line complexity.

Bug Type	# Bugs
Single Hunk Single Line	251
Single Hunk Multi Line	226
Multi Hunk Single Line	67
Multi Hunk Multi Line	96
Total	640

■ **Table 2** AST operation-type complexity.

Bug Type	# Bugs
Single Operation	278
Two Operations	158
Three Operations	121
Four Operations	83
Total	640

3.4 Bug Complexity

Buggy hunk-line-level complexity. The diff files generated by Git not only identify buggy and patched lines but also organize code changes into hunks. We use Git to generate diff files for every bug-patch pair. Based on the number of hunks and the number of fixed lines within each hunk, we categorize bugs into four categories: Single-Hunk-Single-Line (SHSL), Single-Hunk-Multi-Line (SHML), Multi-Hunk-Single-Line (MHSL), and Multi-Hunk-Multi-Line (MHML). The categorization rule is defined in Equation 1. The number of hunks can be obtained directly from Git diff files. If any hunk contains more than one fixed line, the bug is classified as a multi-line bug.

$$\begin{cases} \text{SHSL,} & H = 1 \wedge \forall h_i \in H, L(h_i) = 1 \\ \text{SHML,} & H = 1 \wedge \exists h_i \in H, L(h_i) > 1 \\ \text{MHSL,} & H > 1 \wedge \forall h_i \in H, L(h_i) = 1 \\ \text{MHML,} & H > 1 \wedge \exists h_i \in H, L(h_i) > 1 \end{cases} \quad (1)$$

where H denotes the total number of hunks in a bug, h_i represents the i -th hunk ($1 \leq i \leq H$), and $L(h_i)$ denotes the number of fixed lines in hunk h_i .

AST operation-type complexity. We employ srcML to construct XML-based structural representations of programs [6] and further utilize GumTreeDiff to extract AST-level edit operations between buggy and corrected program pairs [11]. Unlike hunk-line-level complexity, this metric measures structural differences at the AST node level, providing a finer-grained view of bug complexity.

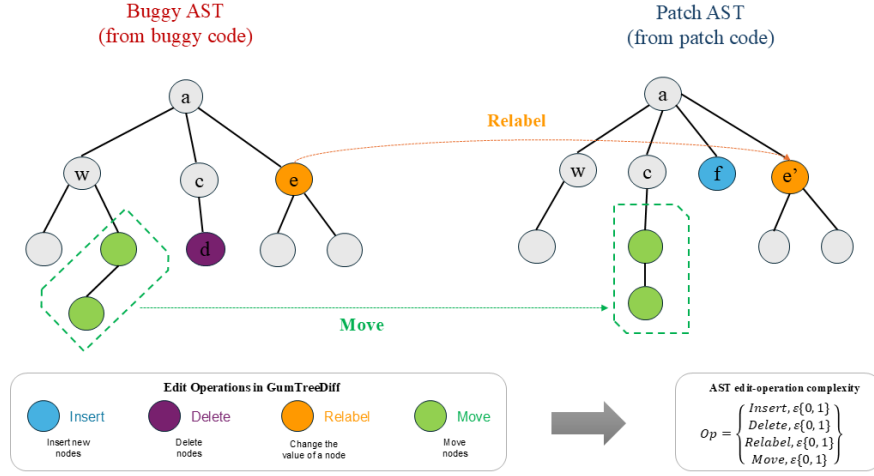
The AST edit distance [42] between a buggy and a correct program is defined by four edit operations: insert, delete, relabel, and move. In our study, bugs are further categorized based on the diversity of edit operations required for repair. Figure 3 illustrates how the AST edit-operation-type complexity is categorized.

Table 1 and Table 2 present the number of bugs in each type under the hunk-line-level and AST operation-type complexity, respectively. We conduct empirical evaluations to analyze how bug complexity affects LLM-based APR performance.

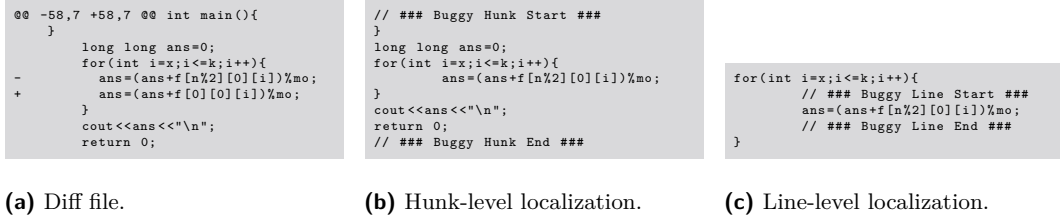
3.5 Fault Localization And APR Settings

We use three FL strategies to explore their impact on LLM-based APR: buggy line-level FL, buggy hunk-level FL, and no FL. Figure 4 presents the examples of line-level FL and hunk-level FL according to the diff file.

Different LLMs provide different mechanisms for controlling reasoning behavior. (a) DeepSeek-V3.2 provides two model variants: *DeepSeek-V3.2-chat* and *DeepSeek-V3.2-reasoner*. For DeepSeek-V4-pro, the reasoning mode can be enabled or disabled through



■ **Figure 3** AST operation-type complexity.



■ **Figure 4** Fault Localization Strategy Examples.

the `thinking_type` parameter in the DeepSeek API. Additionally, the `reasoning_effort` parameter can be configured as `high` or `max`. (b) GPT models do not provide separate chat or reasoner variants; instead, reasoning intensity can only be controlled through the `reasoning_effort` parameter, with settings such as `medium`, `high`, and `xhigh`. For Llama models, there is currently no explicit mechanism to directly control reasoning settings.

Furthermore, the temperature for all LLMs is set to 1.0 [38] for all experiments. For ChatRepair, we set the maximum number of conversation rounds to 3 [46]. For both ChatRepair and CodeCorrector, we provide the problem background, the buggy code, the error message, and at most 3 failed test cases. We retain the original prompts reported in the corresponding studies [46, 21], and we do not perform model-specific parameter tuning, allowing the comparison to focus on the influence of the APR frameworks, FL strategies, and LLM effectiveness.

3.6 Experimental Design

To balance experimental comprehensiveness and computational cost, we adopt a three-stage experimental design.

First, we evaluate two APR tools with three different low-cost LLMs on the complete set of 640 bugs under line-level FL: *DeepSeek-V3.2-chat*, *GPT-4o-mini*, and *Llama-3.1-8B*. The maximum number of repair rounds is set to 10 as in [34]. We conduct a comprehensive empirical analysis of various APR techniques and LLM combinations with respect to repair effectiveness and cost-efficiency.

Next, we select the best-performing LLM from the first stage and further evaluate it on all 640 bugs at line-level, hunk-level, and no-FL. The results are then compared across three FL strategies to analyze the impact of different FL precision on two LLM-based APR techniques. For the first two stages, reasoning mode is disabled across all LLMs, allowing us to identify bugs that can already be repaired under non-reasoning configurations before evaluating more computationally expensive reasoning-enabled models.

To further investigate the repair effectiveness and cost-efficiency of advanced LLMs and reasoning settings under more challenging repair scenarios, we adopt a filtering strategy to focus the evaluation on bugs with moderate repair difficulty while controlling the computational cost of evaluating advanced LLMs. Specifically, we exclude SHSL bugs, single-operation-type bugs, and four-operation-type bugs. In addition, bugs that can already be successfully repaired by at least five APR techniques–LLM combinations in the first-stage experiments are removed. Consequently, the remaining benchmark focuses on bugs with moderate structural complexity and lower repairability under low-cost APR configurations. After filtering, 211 bugs remain for empirical analysis. We further select the better-performing APR technique from the previous stages as the representative APR method for subsequent experiments.

In this stage, we evaluate four advanced LLMs: *DeepSeek-V4-pro*, *DeepSeek-V3.2*, *GPT-5*, and *Llama-3.1-405B*. The maximum number of repair rounds is increased to 13. During the first 10 rounds, reasoning mode is disabled. If a bug remains unfixed, reasoning mode is enabled for the final three rounds. For *GPT-5*, *DeepSeek-V3.2*, and *DeepSeek-V4-pro*, the reasoning effort is configured as `high`.

Experimental Token Consumption. To facilitate the reproduction of our experiments, we report the total input and output tokens consumed by each evaluated LLM that incurred API costs. *DeepSeek-V3.2* consumed 116.5M input tokens and 42.7M output tokens. *GPT-4o-mini* used 49.5M input tokens and 14.4M output tokens. *DeepSeek-V4-pro* consumed 6.3M input tokens and 9.9M output tokens, *GPT-5* consumed 3.4M input tokens and 7.7M output tokens, and *Llama-3.1-405B* consumed 6.3M input tokens and 1.7M output tokens.

API Response Time. We also report the approximate response time per repair request. Without reasoning mode, all evaluated LLMs require approximately 1–3 minutes per request. With reasoning enabled, *DeepSeek-V3.2* and *DeepSeek-V4-pro* required approximately 40–60 minutes per request, while *GPT-5* required about 30 minutes.

4 Results and Empirical Analysis

In this section, we show the results of our empirical experiment.

4.1 APR Performance With Different LLMs

Table 3 presents the repair performance of CodeCorrector and ChatRepair with three base LLMs under the non-reasoning setting. In this experiment, line-level FL is provided within the incorrect programs, and the maximum repair round is set to 10. CodeCorrector stops after generating the first plausible patch, whereas ChatRepair continues generating candidate patches.

Overall, *DeepSeek-V3.2-chat* achieves substantially stronger repair effectiveness than *GPT-4o-mini* across both APR frameworks. When integrated with *DeepSeek-V3.2-chat*, both CodeCorrector and ChatRepair repair more than twice as many bugs as they do with *GPT-4o-mini* (295, 292 vs. 126, 140), whereas *Llama-3.1-8B* demonstrates very limited repair behavior: only 11 bugs were corrected with CodeCorrector and 50 with ChatRepair.

■ **Table 3** Comparison of repair effectiveness between CodeCorrector and ChatRepair frameworks using three base LLMs on the full benchmark.

	CodeCorrector			ChatRepair		
	DeepSeek-V3.2-chat	GPT-4o-mini	Llama-3.1-8B	DeepSeek-V3.2-chat	GPT-4o-mini	Llama-3.1-8B
Repaired Bugs	295/640	126/640	11/640	292/640	140/640	50/640
Attempts	4003	5373	6305	6400	6400	6400
Unique Patches	3117(77.9%)	4425(82.4%)	5833(92.5%)	2623(41.0%)	3774(59.0%)	5111(80.0%)
Compilable Patches	2830(70.7%)	3899(72.6%)	1466(23.3%)	2523(39.4%)	3511(54.9%)	3122(48.8%)
Plausible Patches	295	126	11	421	247	76
Repaired Bugs / \$	47.62	17.86	–	71.43	20.41	–
Attempts / \$	646.17	761.48	–	1565.56	932.94	–

In addition, DeepSeek-V3.2-chat exhibits noticeably better cost-efficiency. DeepSeek-V3.2-chat repairs 47.62 bugs per dollar under CodeCorrector, compared with only 17.86 bugs per dollar for GPT-4o-mini, representing approximately $2.7\times$ higher cost-efficiency. Similarly, under ChatRepair, DeepSeek achieves 71.43 repaired bugs per dollar, whereas GPT-4o-mini repairs only 20.41 bugs per dollar, corresponding to approximately $3.5\times$ higher cost-efficiency.

Finding 1: Among relatively low-cost LLMs, DeepSeek-V3.2-chat demonstrates the strongest overall APR effectiveness, achieving significantly better repair results and cost-efficiency than GPT-4o-mini, whereas Llama-3.1-8B exhibits limited repair effectiveness despite its negligible inference cost.

Bug Complexity. We then analyze how bug complexity affects overall repairability, where a bug is considered repaired if any evaluated APR technique and LLM configuration can fix it. Table 4 and Table 5 present the pairwise 2×2 contingency tables and Pearson chi-square test results for the numbers of fixed and unfixed bugs across different bug complexity categories. Pairwise chi-square tests are conducted to assess whether differences in repair rates across complexity categories are statistically significant. Following [27], we consider differences statistically significant when $p < 0.05$, while $p < 0.005$ indicates sufficient evidence against the null hypothesis.

SHSL and single-operation-type bugs exhibit the lowest structural complexity and consequently achieve the highest repair rates among all categories, reaching 78.5% and 75.2%, respectively. Furthermore, all pairwise comparisons between these two categories and the remaining complexity categories yield $p < 0.005$, providing strong statistical evidence that SHSL and single-operation-type bugs are substantially easier for APR techniques to repair. However, the SHML, MHSL, MHML, two-operation, and three-operation bugs also achieve repair rates above 50%.

For hunk-line-level complexity, no statistically significant difference in repair is observed among multi-line bug categories ($p > 0.05$). In contrast, although no statistically significant difference is observed between two-operation-type and three-operation-type bugs ($p = 0.78$), four-operation-type bugs exhibit significantly lower repair effectiveness than other operation-type categories ($p < 0.05$). Thus, AST operation complexity further distinguishes highly complex bugs, with statistical analysis showing that four-operation bugs exhibit the most complex repair structure.

■ **Table 4** Pairwise 2×2 contingency tables for chi-square tests.

(a) Hunk-line Complexity Comparisons								
SHSL vs SHML			SHSL vs MHSL			SHSL vs MHML		
Type	Fixed	Unfixed	Type	Fixed	Unfixed	Type	Fixed	Unfixed
SHSL	197(78.5%)	54	SHSL	197(78.5%)	54	SHSL	197(78.5%)	54
SHML	113(50.0%)	113	MHSL	39(58.2%)	28	MHML	48(50.0%)	48
SHML vs MHSL			SHML vs MHML			MHSL vs MHML		
Type	Fixed	Unfixed	Type	Fixed	Unfixed	Type	Fixed	Unfixed
SHML	113(50.0%)	113	SHML	113(50.0%)	113	MHSL	39(58.2%)	28
MHSL	39(58.2%)	28	MHML	48(50.0%)	48	MHML	48(50.0%)	48

(b) AST Operation Complexity Comparisons								
Single vs Two			Single vs Three			Single vs Four		
Type	Fixed	Unfixed	Type	Fixed	Unfixed	Type	Fixed	Unfixed
Single	209(75.2%)	69	Single	209(75.2%)	69	Single	209(75.2%)	69
Two	90(57.0%)	68	Three	66(54.5%)	55	Four	32(38.6%)	51
Two vs Three			Two vs Four			Three vs Four		
Type	Fixed	Unfixed	Type	Fixed	Unfixed	Type	Fixed	Unfixed
Two	90(57.0%)	68	Two	90(57.0%)	68	Three	66(54.5%)	55
Three	66(54.5%)	55	Four	32(38.6%)	51	Four	32(38.6%)	51

■ **Table 5** Pairwise Pearson Chi-square test results across different bug complexity categories.

	SHSL	SHML	MHSL	MHML		1-op	2-op	3-op	4-op
SHSL	–	$p < 0.0001$	$p = 0.0013$	$p < 0.0001$	1-op	–	$p = 0.0001$	$p < 0.0001$	$p < 0.0001$
SHML	–	–	$p = 0.30$	$p = 1.000$	2-op	–	–	$p = 0.78$	$p = 0.010$
MHSL	–	–	–	$p = 0.38$	3-op	–	–	–	$p = 0.035$
MHML	–	–	–	–	4-op	–	–	–	–

(a) Hunk-line complexity

(b) AST operation type complexity

Darker cells indicate smaller p-values.

Finding 2: SHSL and single-operation-type bugs are the easiest to repair, whereas four-operation-type bugs remain the most challenging. The repair rates of moderate-complexity categories exceed 50%, indicating that LLM-based APR techniques maintain strong repair effectiveness. Although four-operation-type bugs exhibit lower repair performance, their repair rates remain considerable. These results suggest that low-cost LLM-based APR techniques retain substantial repair effectiveness even for structurally complex bugs.

RQ1 Summary. Bug complexity significantly affects repair effectiveness. Chi-square test results suggest three approximate levels of repair difficulty across different bug complexity categories. Overall, repair effectiveness decreases with increasing bug complexity, while the simplest bugs remain substantially easier to repair. Although four-operation bugs are the hardest to repair, APR techniques still maintain considerable repair capability for structurally complex bugs. Over 50% of moderate-complexity bugs can still be repaired successfully. These results demonstrate the potential of low-cost LLM-based APR techniques for handling challenging real-world repair scenarios.

■ **Table 6** Comparison of repair effectiveness between CodeCorrector and ChatRepair frameworks using three Fault Localization strategies on the full benchmark.

	CodeCorrector			ChatRepair		
	line-level	hunk-level	w/o FL	line-level	hunk-level	w/o FL
Repaired Bugs	295/640	179/640	112/640	292/640	237/640	205/640
Attempts	4003	4986	5480	6400	6400	6400
Unique Patches	3117(77.9%)	4346(87.2%)	4012(73.2%)	2623(41.0%)	4741(74.1%)	3261(51.0%)
Compilable Patches	2830(70.7%)	3503(70.3%)	3756(68.5%)	2523(39.4%)	4073(63.6%)	3174(49.6%)
Plausible Patches	295	179	112	421	542	299
Repaired Bugs / \$	47.62	16.60	18.87	71.43	24.41	47.62
Attempts / \$	646.17	462.52	923.18	1565.56	659.13	1486.64

■ **Table 7** Comparison of repair consistency under different FL strategies.

FL Strategy	CodeCorrector	ChatRepair
Line-level FL	0.85	0.92
Hunk-level FL	0.75	0.83
w/o FL	0.54	0.85

4.2 Fault Localization Evaluation

Given DeepSeek-V3.2-chat’s best performance in the previous experiment, we further evaluate the impact of different FL strategies on APR performance using this LLM with both APR techniques. In this experiment, the maximum repair round is still set to 10.

Table 6 shows that CodeCorrector and ChatRepair achieve nearly identical repair effectiveness under line-level FL, with CodeCorrector even repairing three more bugs. However, when only hunk-level FL or no FL is provided, ChatRepair demonstrates consistently greater stability and generalizability. Under hunk-level FL, ChatRepair repairs 58 more bugs than CodeCorrector, and this gap further increases to 93 bugs when FL is unavailable.

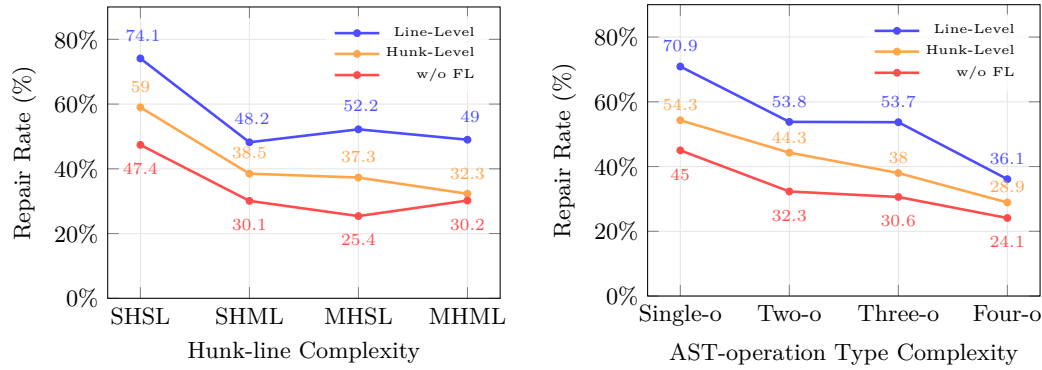
Finding 3: *The performance gap between APR techniques increases markedly as FL becomes less precise, indicating that repair stability and generalizability are critical for real-world APR scenarios. Different APR techniques may achieve similar repair rates under precise line-level FL, but their repair rates diverge substantially under imprecise FL settings.*

We further introduce the *code consistency rate* (CCR) to measure how well repaired programs preserve the original code’s structure. $CCR = r/k$ defines the consistency metric, where k denotes the total number of code lines in the repaired program, and r denotes the number of preserved code lines after repair.

According to Table 7, line-level FL achieves the highest patch consistency across all FL settings for both APR techniques. ChatRepair generally generates patches with higher structural consistency than CodeCorrector, particularly without FL (0.85 vs. 0.54).

Finding 4: *FL not only improves repair effectiveness but also helps generate patches that remain more structurally consistent with the original programs. The design of APR techniques also significantly impacts the structural consistency of generated patches.*

We use logistic regression [28] to examine the extent to which reduced FL precision affects the likelihood of successful repair. We set line-level FL as the reference category and analyze each complexity level independently. The results are reported using odds ratios (ORs) and corresponding p -values. Smaller OR values reflect a greater reduction in the odds of successful repair.



■ **Figure 5** Repair rate under different bug complexity categories.

■ **Table 8** Logistic regression results of FL effects on repair success under different complexity categories.

(a) Hunk-line complexity categories.

Scope	Hunk-level FL		No FL	
	OR	p	OR	p
Overall	0.582	< .001	0.399	< .001
SHSL	0.502	< .001	0.315	< .001
SHML	0.672	0.037	0.462	< .001
MHSL	0.544	0.084	0.311	0.002
MHML	0.497	0.019	0.451	0.008

(b) AST-operation complexity categories.

Scope	Hunk-level FL		No FL	
	OR	p	OR	p
Overall	0.582	< .001	0.399	< .001
single-op	0.489	< .001	0.336	< .001
two-op	0.683	0.092	0.409	< .001
three-op	0.528	0.015	0.379	< .001
four-op	0.719	0.321	0.561	0.092

Figure 5 shows that repair rates consistently decrease across all complexity categories as FL becomes less precise (blue vs. orange vs. red). According to Table 8, both hunk-level FL and no-FL settings yield overall p -values smaller than 0.001, indicating that FL precision has a statistically significant effect on overall repair effectiveness.

We further observe that the simplest bugs (SHSL and single-operation bugs) appear sensitive to reductions in FL precision, as both hunk-level FL and no-FL settings produce very low odds ratios, also with $p < 0.001$. In contrast, the highest complexity bugs (four-operation bugs) exhibit the least noticeable repair degradation under hunk-level FL and no-FL settings, as the odds ratios remain relatively higher and the FL degradation effects are not statistically significant ($p = 0.092 > 0.05$).

Finding 5: Repair rates consistently decrease as FL becomes less precise across all bug complexity categories, indicating that FL precision remains important regardless of bug complexity. However, as bug complexity increases, repair effectiveness becomes less dependent on precise FL, suggesting it is more likely reliant on structural understanding and reasoning of program logic.

RQ2 Summary. FL still plays a critical role in the bug repair process. More precise FL not only improves overall repair rates but also helps APR techniques generate more consistent patches. Furthermore, APR techniques with similar repair performance under the same FL strategy can exhibit substantially different effectiveness as FL becomes increasingly imprecise, resulting in progressively larger performance gaps among APR approaches. These results highlight the importance of the stability and generalizability of APR techniques for handling realistic bug-repair scenarios. Moreover, as bug structures become increasingly complex, the relative importance of FL precision decreases, indicating the growing significance of reasoning and program understanding.

■ **Table 9** The Advanced LLMs Performance.

	Repaired Bugs	Attempts	Repaired Bugs / \$	Attempts / \$
GPT-5	175/211	666	2.94	11.19
GPT-5-reasoning	3/36	106	0.15	5.24
DeepSeek-V3.2-chat	75/211	1550	75.75	1565.56
DeepSeek-V3.2-reasoner	64/136	318	23.97	119.10
DeepSeek-V4-pro	90/211	1475	4.17	68.29
DeepSeek-V4-pro-reasoner	81/121	232	3.16	9.05
Llama-3.1-405B	49/211	1801	-	-

4.3 APR With Advanced LLMs

ChatRepair demonstrates overall better performance than CodeCorrector in sections 4.1 and 4.2, so this study chooses ChatRepair to evaluate performance and cost-efficiency for advanced LLM-based approaches. To reduce API cost, ChatRepair stops when generating the first plausible patch. Subsection 3.6 describes the experimental environment, and Table 9 presents the results of ChatRepair combined with four LLMs and two reasoning settings.

Cost analysis shows that inference costs increase substantially when more advanced LLMs are adopted. DeepSeek-V3.2-chat supports approximately 1,565.56 repair attempts per dollar in non-reasoning mode and 119.10 in reasoning mode, whereas DeepSeek-V4-pro decreases to 68.29 in non-reasoning mode, and 9.05 in reasoning mode. GPT-5 supports only 11.19 attempts per dollar in non-reasoning mode and 5.24 attempts per dollar in reasoning mode. These results show that the cost increases considerably as larger and reasoning-oriented LLMs are used. Compared with the advanced LLMs, DeepSeek-V3.2 is much cheaper than GPT-5 and DeepSeek-V4-pro. Therefore, both the choice and reasoning settings of LLMs heavily impact the cost-efficiency.

The performance of different LLMs varies substantially under non-reasoning and reasoning settings. Figure 6 (a) presents the number of additionally repaired bugs achieved by each LLM when reasoning mode is enabled.

GPT-5 repairs 175 bugs under non-reasoning mode, achieving the highest number of repaired bugs among all evaluated models, while DeepSeek-V4-pro and DeepSeek-V3.2 repair 90 and 75 bugs, respectively. However, when the reasoning effort is increased to high, GPT-5 repairs only 3 additional bugs. In contrast, enabling high-reasoning-mode effort allows DeepSeek-V3.2 and DeepSeek-V4-pro to repair 64 and 81 additional bugs, respectively.

Finding 6: *Both LLM selection and reasoning settings have significant impacts on repair effectiveness and cost, while the effectiveness of reasoning strategies differs across LLM families. Enabling reasoning mode provides only limited repair improvement for GPT-5, whereas DeepSeek-V3.2 and DeepSeek-V4-pro benefit considerably more from reasoning-based settings, repairing many bugs that cannot be fixed under non-reasoning configurations.*

Figure 6 (b) presents the repair rates of different LLMs and reasoning settings under the cost metric of attempts per dollar (Attempts/\$). Combined with Figure 6 (a), GPT-5 repairs 175 bugs under the non-reasoning setting, which is higher than all other evaluated LLMs under both reasoning and non-reasoning settings (171 for DeepSeek-V4-pro and 139 for DeepSeek-V3.2). Moreover, GPT-5 under the non-reasoning setting is even cheaper than DeepSeek-V4-pro under the reasoning setting in terms of attempts per dollar (11.19 vs. 9.05). Although GPT-5 under the non-reasoning setting is more expensive than DeepSeek-V3.2 under the reasoning setting, the repair rate is markedly higher.

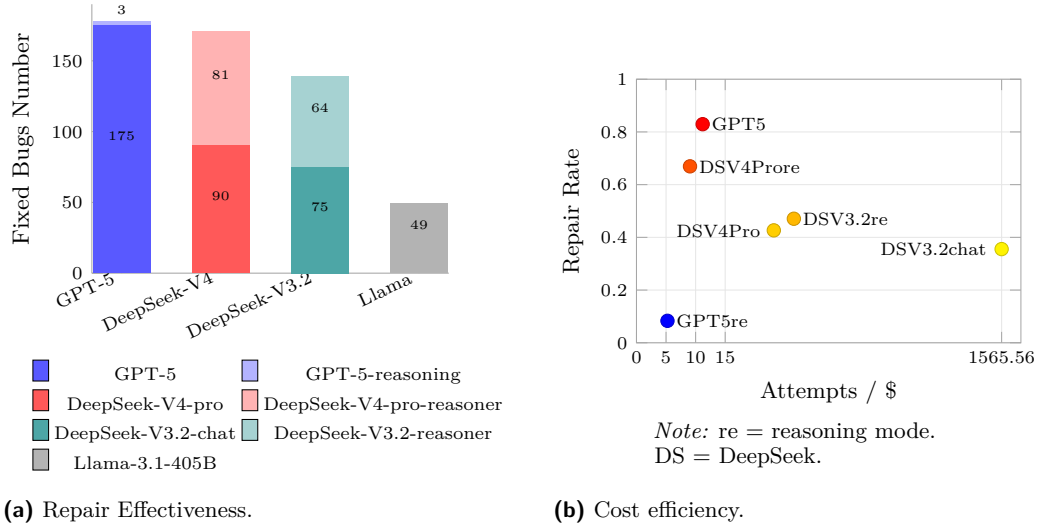


Figure 6 Repair effectiveness and cost efficiency of advanced LLMs.

Figure 6 (b) shows that under the non-reasoning setting, DeepSeek-V3.2 achieves competitive repair performance compared with DeepSeek-V4-pro, while maintaining a substantially lower cost. Therefore, using DeepSeek-V3.2 in non-reasoning mode for overall bug repair, and subsequently applying GPT-5 in non-reasoning mode to repair the remaining unresolved bugs, provides a more effective trade-off between repair performance and cost.

Finding 7: *GPT-5 achieves the strongest overall repair effectiveness, whereas DeepSeek-V3.2-chat demonstrates significantly better cost-efficiency. Combining cost-efficient LLMs with more advanced LLMs for unresolved bugs can therefore achieve a better balance between repair effectiveness and cost-efficiency in LLM-based APR techniques.*

Table 10 presents the *Top – k* metric for both APR techniques across three LLM families and three different FL strategies. The first five repair rounds usually resolve the largest proportion of bugs across evaluated APR techniques, LLMs, and FL settings. However, the marginal repair gain decreases considerably as additional repair rounds are introduced, with later rounds contributing only a limited number of newly repaired bugs.

Finding 8: *The majority of bugs are repaired within the first five repair rounds, whereas the marginal repair effectiveness decreases significantly in later rounds.*

RQ3 Summary. Using advanced LLMs and reasoning modes improves repair rates and enables APR techniques to fix more structurally complex bugs. However, these improvements come with substantially higher costs, making the trade-off between repair performance and cost increasingly important. Moreover, the benefits of reasoning modes vary across different LLM families. In our experiments, GPT-5 shows the smallest gain from reasoning, whereas the DeepSeek series repairs many additional bugs that cannot be fixed in non-reasoning mode. We further find that applying cost-efficient LLMs first, and then using advanced LLMs to repair the remaining unresolved bugs, can achieve a better balance between repair effectiveness and cost.

■ **Table 10** Repair performance (Top- k) across APR configurations. Values are reported as incremental fixes and cumulative total in parentheses.

APR	LLMs	FL	Top-1	Top-2	Top-3	Top-4	Top-5	Top-10
CC	L3.1-8B	ll	8	2 (10)	1 (11)	0 (11)	0 (11)	0 (11)
	4o-mini	ll	80	23 (103)	8 (111)	6 (117)	4 (121)	5 (126)
	DS-V3.2	ll	191	49 (240)	18 (258)	14 (272)	6 (278)	17 (295)
	DS-V3.2	hl	106	32 (138)	12 (150)	12 (162)	2 (164)	15 (179)
	DS-V3.2	w/o FL	80	15 (95)	6 (101)	1 (102)	2 (104)	8 (112)
CR	L3.1-8B	ll	15	5 (20)	1 (21)	9 (30)	6 (36)	14 (50)
	4o-mini	ll	67	16 (83)	9 (92)	14 (106)	9 (115)	25 (140)
	DS-V3.2	ll	194	30 (224)	8 (232)	19 (251)	8 (259)	33 (292)
	DS-V3.2	hl	93	38 (131)	22 (153)	26 (179)	16 (195)	42 (237)
	DS-V3.2	w/o FL	128	30 (158)	14 (172)	11 (183)	7 (190)	15 (205)
CR adv.LLM	L3.1-405B	ll	15	12 (27)	0 (27)	5 (32)	3 (35)	14 (49)
	GPT-5	ll	103	43 (146)	16 (162)	4 (166)	5 (171)	4 (175)
	DS-V3.2	ll	45	8 (53)	4 (57)	3 (60)	3 (63)	12 (75)
	DS-V4-pro	ll	31	26 (57)	5 (62)	5 (67)	7 (74)	16 (90)
	GPT-5(re)	ll	1	0 (1)	2 (3)	—	—	—
	DS-V3.2(re)	ll	33	19 (52)	12 (64)	—	—	—
	DS-V4-pro(re)	ll	58	15 (73)	8 (81)	—	—	—

Note: CC=CodeCorrector, CR=ChatRepair, L3.1 = Llama-3.1, 4o-mini = GPT-4o-mini, DS = DeepSeek, re = reasoning, w/o = without, ll = line-level FL, hl = hunk-level FL.

5 Discussion

We investigated the effectiveness of LLM-based APR under different bug complexity categories and fault localization strategies. The results reveal several insights regarding the repair effectiveness and cost-efficiency of the evaluated LLM-based APR techniques, providing useful guidance for future APR research and practical deployment.

Impacts of bug complexity. Our results show that bug complexity substantially affects the effectiveness of LLM-based APR techniques. The simplest bugs achieve much higher repair rates, while structurally complex bugs containing multiple buggy lines or diverse AST edit operations remain more challenging.

The repair performance across different bug complexity levels suggests that future APR evaluations could pay more attention to complex repair scenarios rather than to single-line or single-operation bugs. Structurally complex bugs may better reflect some of the challenges encountered in practical APR tasks. New benchmarks and standard performance metrics could consider evaluations across different bug categories, domains, and complexity levels to better characterize tool performance [10, 36, 39].

Impacts of fault localization. Precise FL not only improves repair effectiveness, but also helps APR techniques generate patches that are more consistent with the original program logic. Such consistency may facilitate the understanding, validation, and maintenance of repaired programs.

Additionally, existing LLM-based APR methods are affected differently by less precise FL information. This difference highlights robustness to imprecise FL as an important requirement for practical APR techniques [10]. Since real-world FL is often noisy and incomplete, methods that maintain stable repair performance when the precision of FL decreases are more likely to be effective in practice. For complex bugs, the relatively smaller effect of FL precision indicates that FL may not be the only bottleneck. Repairing these bugs may also depend on reasoning ability and deeper program-understanding capabilities.

The Trade-off Between Repair Effectiveness and Cost-efficiency. Using more advanced LLMs and reasoning modes substantially increases computational cost, while often improving repair effectiveness. Based on the results presented in section 4.3, GPT-5 achieves the highest repair effectiveness, while DeepSeek-V3.2-chat provides the most cost-efficient repair performance. Our results suggest that such a two-stage strategy could be a promising direction for future investigation.

The marginal repair efficiency drops as the number of repair rounds increases [34, 50], which further impacts overall cost, especially when using advanced LLMs such as GPT-5 and DeepSeek-V4-pro in reasoning mode. Our results also suggest that selecting an appropriate maximum number of repair rounds is important for cost-efficiency.

6 Threats to Validity

External Validity. Our dataset is collected solely from the competitive programming platform AtCoder and focuses only on C++ programs, which are typically shorter but more algorithm-oriented [24, 12] than industrial software systems. Thus, the findings of our study may not generalize fully to real-world software projects with larger codebases, complex dependencies, and practical software engineering constraints. Furthermore, we evaluate only two APR techniques and three LLM series, and thus, the best-performing approaches identified in this study should not be regarded as definitive state-of-the-art APR solutions.

The evaluated LLMs have different knowledge cutoff dates, and we collect benchmark data after 2025-10-01, which is after the knowledge cutoff date of the GPT and Llama models we used. DeepSeek did not publish its knowledge cutoff date, so it remains difficult to completely rule out the possibility that some competitive programming solutions or related code patterns were exposed during model pretraining [30, 37].

Internal Validity. We set the temperature to 1.0 for all evaluated LLMs and used high reasoning-effort settings instead of the maximum settings for GPT-5, DeepSeek-V3.2-reasoner, and DeepSeek-V4-pro due to their substantial time costs. Different parameter configurations and reasoning settings may lead to variations in repair performance. In addition, API-based LLMs may produce nondeterministic outputs across different runs, which could also affect repair results. Furthermore, different prompting strategies, repair rounds, and stopping criteria may influence the effectiveness and cost-efficiency of APR.

Construct Validity. We determine repaired bugs mainly based on whether the generated patches can pass all collected test cases. However, the test cases may not be sufficiently comprehensive to fully validate the semantic correctness of repaired programs. Therefore, some plausible patches may still contain hidden defects or overfitting issues.

7 Conclusion and Future Work

This work conducts an empirical study on how bug complexity and FL affect LLM-based APR techniques. We utilize Git, srcML, and GumTreeDiff to categorize bug complexity from both hunk-line-level and AST-operation-level perspectives, and to evaluate APR effectiveness under accurate, vague, and no fault-localization settings.

The experimental results show that structurally complex bugs and imprecise FL often reduce repair effectiveness. Nevertheless, well-designed LLM-based APR techniques still maintain considerable repair rates under these challenging conditions. Furthermore, different LLMs exhibit clear trade-offs between repair effectiveness and cost-efficiency, highlighting the importance of balancing repair effectiveness and cost for practical APR tools.

Future work may develop a benchmark that can automatically collect, filter, and categorize bug complexity. Also, we will consider practical factors such as API response latency [35, 25] and overall inference time when evaluating the efficiency of LLM-based APR systems. We plan to explore additional LLM families, such as Gemini and Claude, to further evaluate the effectiveness of LLM-based APR. Finally, the agent-based APR frameworks should be further investigated, as agent-based techniques have become increasingly popular across various software engineering and reasoning tasks in recent years.

Data Availability Statement

The dataset and source code used in this study are publicly available at the following links:

Dataset: <https://doi.org/10.5281/zenodo.20257267>

Code: <https://github.com/Leo6-sys/Rethink-APR-Empirical-Study-2026>

References

- 1 Meta AI. Llama-3.1-405b. <https://huggingface.co/meta-llama/Meta-Llama-3.1-405B>, 2024.
- 2 Meta AI. Llama-3.1-8b. <https://huggingface.co/meta-llama/Llama-3.1-8B>, 2024.
- 3 Aram Bahrini, Mohammadsadra Khamoshifar, Hossein Abbasimehr, Robert J. Riggs, Maryam Esmaeili, Rastin Mastali Majdabadkohne, and Morteza Pasehvar. ChatGPT: Applications, opportunities, and threats. In *2023 Systems and Information Engineering Design Symposium (SIEDS)*, pages 274–279, 2023. doi:10.1109/SIEDS58326.2023.10137850.
- 4 Andrew M Bean, Ryan Othniel Kearns, Angelika Romanou, Franziska Sofia Hafner, Harry Mayne, Jan Bartzner, Negar Foroutan Eghlidi, Chris Schmitz, Karolina Korgul, Hunar Batra, et al. Measuring what matters: Construct validity in large language model benchmarks. *Advances in Neural Information Processing Systems*, 38, 2026. doi:10.52202/085713-0590.
- 5 Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. RepairAgent: An autonomous, llm-based agent for program repair. In *Proceedings of the 47th IEEE/ACM International Conference on Software Engineering*, pages 2188–2200, 2025. doi:10.1109/ICSE55347.2025.00157.
- 6 Michael L. Collard, Michael J. Decker, and Jonathan I. Maletic. srcML: An infrastructure for the exploration, analysis, and manipulation of source code: A tool demonstration. In *Proceedings of the 29th IEEE International Conference on Software Maintenance*, pages 516–519, 2013. doi:10.1109/ICSM.2013.85.
- 7 Zhenlong Dai, Bingrui Chen, Zhuoluo Zhao, Xiu Tang, Sai Wu, Chang Yao, Zhipeng Gao, and Jingyuan Chen. Less is more: Adaptive program repair with bug localization and preference learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 128–136, 2025. doi:10.1609/aaai.v39i1.31988.
- 8 DeepSeek-AI. Deepseek-v3.2. <https://api-docs.deepseek.com/news/news251201>, 2025.
- 9 DeepSeek-AI. Deepseek-v4. <https://api-docs.deepseek.com/news/news260424>, 2026.
- 10 Sena Dikici and Turgay Tugay Bilgin. Advancements in automated program repair: a comprehensive review. *Knowledge and Information Systems*, 67(6):4737–4783, 2025. doi:10.1007/s10115-025-02383-9.
- 11 Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, pages 313–324, 2014. doi:10.1145/2642937.2642982.
- 12 Claire Le Goues, Neal Holtschulte, Edward K. Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. The ManyBugs and IntroClass benchmarks for automated repair of c programs. *IEEE Transactions on Software Engineering*, 41(12):1236–1256, 2015. doi:10.1109/TSE.2015.2454513.

- 13 Tingxu Han, Zhenting Wang, Chunrong Fang, Shiyu Zhao, Shiqing Ma, and Zhenyu Chen. Token-budget-aware LLM reasoning. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 24842–24855, 2025. doi:10.18653/v1/2025.findings-acl.1274.
- 14 Nam Huynh and Beiyu Lin. Large language models for code generation: A comprehensive survey of challenges, techniques, evaluation, and applications. *arXiv preprint arXiv:2503.01245*, 2025. doi:10.48550/arXiv.2503.01245.
- 15 AtCoder Inc. Atcoder. <https://atcoder.jp/>.
- 16 Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation. *ACM Transactions on Software Engineering and Methodology*, 35(2):1–72, 2026. doi:10.1145/3747588.
- 17 Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. InferFix: End-to-end program repair with LLMs. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1646–1656, 2023. doi:10.1145/3611643.3613892.
- 18 Sathvik Joel, Jie JW Wu, and Fatemeh H. Fard. A survey on LLM-based code generation for low-resource and domain-specific programming languages. *ACM Transactions on Software Engineering and Methodology*, 2025. doi:10.1145/3770084.
- 19 René Just, Darioush Jalali, and Michael D. Ernst. Defects4J: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 437–440. ACM, 2014. doi:10.1145/2610384.2628055.
- 20 Fengjie Li, Jiajun Jiang, Jiajun Sun, and Hongyu Zhang. Evaluating the generalizability of llms in automated program repair. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering: New Ideas and Emerging Results*, pages 91–95, 2025. doi:10.1109/ICSE-NIER66352.2025.00024.
- 21 Yingling Li, Muxin Cai, Junjie Chen, Yang Xu, Lei Huang, and Jianping Li. Context-aware prompting for llm-based program repair. *Automated Software Engineering*, 32(2):42, 2025. doi:10.1007/s10515-025-00512-w.
- 22 Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, et al. Competition-level code generation with AlphaCode. *Science*, 378(6624):1092–1097, 2022. doi:10.1126/science.abq1158.
- 23 Bo Lin, Shangwen Wang, Ming Wen, Liqian Chen, and Xiaoguang Mao. One size does not fit all: Multi-granularity patch generation for better automated program repair. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1554–1566, 2024. doi:10.1145/3650212.3680381.
- 24 Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. 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*, pages 55–56, 2017. doi:10.1145/3135932.3135941.
- 25 Fan Liu, Wen-Shuo Chao, Naiqiang Tan, and Hao Liu. Bag of tricks for inference-time computation of llm reasoning. *Advances in Neural Information Processing Systems*, 38, 2026.
- 26 Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. Automatic repair of real bugs in java: A large-scale experiment on the Defects4J dataset. *Empirical Software Engineering*, 22(4):1936–1964, 2017. doi:10.1007/s10664-016-9470-4.
- 27 Mary L McHugh. The chi-square test of independence. *Biochemia medica*, 23(2):143–149, 2013. doi:10.11613/BM.2013.018.
- 28 Scott Menard. *Applied logistic regression analysis*. SAGE publications, 2001. doi:10.4135/9781412983433.
- 29 Humza Naveed, Asad Ullah Khan, Shi Qiu, Muhammad Saqib, Saeed Anwar, Muhammad Usman, Naveed Akhtar, Nick Barnes, and Ajmal Mian. A comprehensive overview of large language models. *ACM Transactions on Intelligent Systems and Technology*, 16(5):1–72, 2025. doi:10.1145/3744746.

- 30 Shiwen Ni, Xiangtao Kong, Chengming Li, Xiping Hu, Ruifeng Xu, Jia Zhu, and Min Yang. Training on the benchmark is not all you need. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 24948–24956, 2025. doi:10.1609/aaai.v39i23.34678.
- 31 OpenAI. Gpt-4o mini. <https://platform.openai.com/docs/models/gpt-4o-mini>, 2024.
- 32 OpenAI. Gpt-5. <https://platform.openai.com/docs/models/gpt-5>, 2025.
- 33 Yicheng Ouyang, Jun Yang, and Lingming Zhang. Benchmarking automated program repair: An extensive study on both real-world and artificial bugs. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 440–452, 2024. doi:10.1145/3650212.3652140.
- 34 Mayur Amarnath Palavalli and Mark Santolucito. Using a feedback loop for llm-based infrastructure as code generation. *arXiv preprint arXiv:2411.19043*, 2024. doi:10.48550/arXiv.2411.19043.
- 35 Gyeongmuk Park, Seonghyeon Lee, and Yeonsu Park. Minimizing response latency in LLM-based agent systems: A comprehensive survey. *IEEE Access*, 14:26140–26168, 2026. doi:10.1109/ACCESS.2026.3664226.
- 36 Meghana Puvvadi, Sai Kumar Arava, Adarsh Santoria, Sesha Sai Prasanna Chennupati, and Harsha Vardhan Puvvadi. Coding agents: A comprehensive survey of automated bug fixing systems and benchmarks. In *2025 IEEE 14th International Conference on Communication Systems and Network Technologies (CSNT)*, pages 680–686, 2025. doi:10.1109/CSNT64827.2025.10968728.
- 37 Daniel Ramos, Claudia Mamede, Kush Jain, Paulo Canelas, Catarina Gamboa, and Claire Le Goues. Are large language models memorizing bug benchmarks? In *2025 IEEE/ACM International Workshop on Large Language Models for Code (LLM4Code)*, pages 1–8, 2025. doi:10.1109/LLM4Code66737.2025.00005.
- 38 Matthew Renze. The effect of sampling temperature on problem solving in large language models. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 7346–7356, 2024. doi:10.18653/v1/2024.findings-emnlp.432.
- 39 Joseph Renzullo, Pemma Reiter, Westley Weimer, and Stephanie Forrest. Automated program repair: Emerging trends pose and expose problems for benchmarks. *ACM Computing Surveys*, 57(8):1–18, 2025. doi:10.1145/3704997.
- 40 Ripon K. Saha, Yingjun Lyu, Wing Lam, Hiroaki Yoshida, and Mukul R. Prasad. Bugs.jar: A large-scale, diverse dataset of real-world java bugs. In *Proceedings of the 15th International Conference on Mining Software Repositories*, pages 10–13. ACM, 2018. doi:10.1145/3196398.3196473.
- 41 Sandeep Singh Sengar, Affan Bin Hasan, Sanjay Kumar, and Fiona Carroll. Generative artificial intelligence: A systematic review and applications. *Multimedia Tools and Applications*, 84(21):23661–23700, 2025. doi:10.1007/s11042-024-20016-1.
- 42 Yewei Song, Cedric Lothritz, Xunzhu Tang, Tegawendé Bissyandé, and Jacques Klein. Re-visiting code similarity evaluation with abstract syntax tree edit distance. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 38–46, 2024. doi:10.18653/v1/2024.acl-short.3.
- 43 Diomidis Spinellis. Git. *IEEE Software*, 29(3):100–101, 2012. doi:10.1109/MS.2012.61.
- 44 Chi Wang, Xueqing Liu, and Ahmed Hassan Awadallah. Cost-effective hyperparameter optimization for large language model generation inference. In *Proceedings of the Second International Conference on Automated Machine Learning*, volume 224 of PMLR, pages 21/1–17, 2023. URL: <https://proceedings.mlr.press/v224/wang23b.html>.
- 45 Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837, 2022. doi:10.52202/068431-1800.
- 46 Chunqiu Steven Xia and Lingming 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*, pages 819–831, 2024. doi:10.1145/3650212.3680323.

- 47 Fengli Xu, Qian Yue Hao, Chenyang Shao, Zefang Zong, Yu Li, Jingwei Wang, Yunke Zhang, Jingyi Wang, Xiaochong Lan, Jiahui Gong, et al. Toward large reasoning models: A survey of reinforced reasoning with large language models. *Patterns*, 6(10):101370, 2025. doi:10.1016/j.patter.2025.101370.
- 48 Xin Yin, Chao Ni, Shaohua Wang, Zhenhao Li, Limin Zeng, and Xiaohu Yang. ThinkRepair: Self-directed automated program repair. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1274–1286, 2024. doi:10.1145/3650212.3680359.
- 49 Murong Yue, Jie Zhao, Min Zhang, Liang Du, and Ziyu Yao. Large language model cascades with mixture of thought representations for cost-efficient reasoning. In *The Twelfth International Conference on Learning Representations*, 2024. doi:10.48550/arXiv.2310.03094.
- 50 Yuze Zhao, Zhenya Huang, Yixiao Ma, Rui Li, Kai Zhang, Hao Jiang, Qi Liu, Linbo Zhu, and Yu Su. Repair: Automated program repair with process-based feedback. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 16415–16429, 2024. doi:10.18653/v1/2024.findings-acl.973.
- 51 Xin Zhou, Bowen Xu, Kisub Kim, DongGyun Han, Hung Huu Nguyen, Thanh Le-Cong, Junda He, Bach Le, and David Lo. Leveraging large language model for automatic patch correctness assessment. *IEEE Transactions on Software Engineering*, 50(11):2865–2883, 2024. doi:10.1109/TSE.2024.3452252.
- 52 Fida Zubair, Maryam Al-Hitmi, and Cagatay Catal. The use of large language models for program repair. *Computer Standards & Interfaces*, 93:103951, April 2025. doi:10.1016/j.cs.2024.103951.