LLM-based Agents for Automated Bug Fixing: How Far Are We?

Xiangxin Meng
ByteDance
Beijing, China
mengxiangxin.1219@bytedance.com

Zexiong Ma
ByteDance
Beijing, China
mazexiong@bytedance.com

Pengfei Gao
ByteDance
Beijing, China
gaopengfei.se@bytedance.com

Chao Peng*
ByteDance
Beijing, China
pengchao.x@bytedance.com

Abstract

Large language models (LLMs) and LLM-based Agents have been applied to fix bugs automatically, demonstrating the capability in addressing software defects by engaging in development environment interaction, iterative validation and code modification. However, systematic analysis of these agent systems remain limited, particularly regarding performance variations among top-performing ones. In this paper, we examine six repair systems on the SWE-bench Verified benchmark for automated bug fixing. We first assess each system's overall performance, noting the instances solvable by all or none of these systems, and explore the capabilities of different systems. We also compare fault localization accuracy at file and code symbol levels and evaluate bug reproduction capabilities. Through analysis, we concluded that further optimization is needed in both the LLM capability itself and the design of Agentic flow to improve the effectiveness of Agents in bug fixing.

CCS Concepts

• **Software and its engineering → Software verification and validation**.

Keywords

Bug fixing, Fault localization, Bug reproduction, Coding agents, Large language models

ACM Reference Format:
Xiangxin Meng, Zexiong Ma, Pengfei Gao, and Chao Peng. 2026. LLM-based Agents for Automated Bug Fixing: How Far Are We?. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3744916.3773269

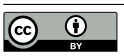
1 Introduction

Large Language Models (LLMs) [62] are advanced machine learning models trained on vast amounts of textual data, capable of understanding and generating human-like text. LLM-based Agents [52] are systems that utilize large language models to interact with the environment and accomplish specific tasks. Recently, LLM-based Agents have demonstrated significant influence in automated bug fixing in code repositories [19, 56, 61]. Thanks to the powerful natural language processing capabilities of LLMs, these agents can efficiently understand and analyze source code and its associated natural language descriptions, such as user-submitted issue descriptions and code comments. Additionally, through dynamic interaction with local environments (e.g., via terminal), LLM-based agents can retrieve useful information from the code repository, perform code editing and execution, and iteratively validate repair results, thereby improving the accuracy and efficiency of bug fixes. This combination of LLM and environmental feedback has made bug fixing more efficient and feasible than ever before, providing revolutionary new tools for software maintenance and development.

Researchers from both industry [1–4, 6, 32, 34] and academia [42, 44, 61] have developed LLM-based agent systems to locate and fix bugs in code repositories. To evaluate the fault localization and repair capabilities of LLMs and various agent systems, Jimenez et.al [17] proposed the evaluation datasets SWE-bench, with derived versions SWE-bench Lite (a subset of the full benchmark), and SWE-bench Verified (human annotated subset of SWE-bench published recently). These datasets contain real bugs from code repositories and can verify the correctness of the patches generated by Agents through unit tests. Recently, these datasets have become the most influential benchmarks in the field of automated bug fixing, attracting both academic and industrial participants to compete on the corresponding leaderboard[1], with new submissions typically every one or a half week.

However, no work has systematically analyzed the fault localization and repair capabilities of LLM-based Agents or the performance differences among various tools within these Agent systems. Regarding the SWE-bench Verified dataset itself, due to the quality of issue descriptions[2] and the complexity of the logical dependencies related to the defects, some instances in the benchmark are easier for Agents to fix, while others are more difficult [53]. As for the design of the systems, different designs exhibit different planning, reasoning, and problem solving capabilities, i.e. some systems adopting static approaches [5] and others adopting dynamic approaches [61]. We have also observed significant differences in cases that each system can solve. Therefore, analyzing the solving capabilities of LLM-based Agents on specific instances can not only

*Corresponding author.



This work is licensed under a Creative Commons Attribution 4.0 International License.
ICSE '26, Rio de Janeiro, Brazil
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2025-3/2026/04
https://doi.org/10.1145/3744916.3773269

[1] https://www.swebench.com/
[2] An issue description is a concatenated string of "Issue Title" and "Description".

help us better understand the current performance of Agents but also provide comparative insights for future research directions.

We collected the six systems (i.e. W&B Programmer [6], Blackbox AI Agent [1], CodeStory Midwit Agent [2], Learn-by-interact [42], Devlo [3], Emergent E1 [4] with top performances on the SWE-bench Verified Leaderboard and conducted a comprehensive analysis of the performance differences of each system. First, we evaluated the overall performance of LLM-based Agents in bug fixing, including statistics on the instances that all six systems can solve and those that none can solve, and analyzed the reasons behind these results. Next, we investigated the performance differences in fault localization among different systems and their causes, compiling file-level and code symbol-level localization accuracy rates. Finally, we analyzed the impact of bug reproduction on bug fixing.

Through data analysis, we have summarized several insights. To improve bug fixing, it is essential to enhance the model's reasoning ability, enabling it to accurately identify bug-related information within an issue and reduce noise interference. For multiple potential repair locations, the model should leverage its reasoning capabilities to determine the most relevant location. From the Agentic flow's perspective, there should be a strong focus on the quality of the issue and attention to multiple suspicious locations in the stack trace. The design should include mechanisms to verify the completeness of patches and consider their global impact. Mechanisms should also be implemented to mitigate the randomness of the model's output or effectively utilize its diversity. In fault localization, achieving code symbol-level accuracy is more critical than file-level and line-level. During the reproduction process, ensuring the correctness of reproductions is crucial, as incorrect reproductions can result in the failure of the entire solving process.

Contributions: To our knowledge, this is the first work to:

- Study the effectiveness of LLM-based Agents in automatic bug fixing for code repositories.
- Examine the effectiveness of different LLM-based Agents in fault localization and analyze their differences.
- Investigate the impact of bug reproduction on bug fixing of LLM-based Agents.
- Summarize the current issues and future research directions for LLM-based Agents in bug fixing.

Paper Organization The remainder of this paper is organized as follows: Section 2 explains the background. Section 3 describes the study design. Section 4 presents the analysis results and findings. Section 5 discusses the analysis results and findings. Section 6 reports the threats to validity. Section 7 discusses related work, and Section 8 concludes the paper.

2 Background

In this section, we first introduce SWE-bench benchmark and then we introduce the leading LLM-based bug fixing systems.

2.1 SWE-bench Benchmark

SWE-Bench [17] is a comprehensive benchmark designed to evaluate LLMs on complex real-world software engineering tasks sourced from GitHub issues and corresponding pull requests across 12 popular Python repositories. This benchmark addresses the limitations of existing coding benchmarks such as HumanEval [11] by presenting tasks that require models to understand and coordinate changes across large codebases involving multiple functions and files. The benchmark includes 2,294 task instances and emphasizes the need for models to interact with execution environments and handle long contexts, showcasing the challenges that real-world software engineering problems pose to current LLMs. Their evaluations reveal that even the best-performing models at the time of publication, such as Claude 2, achieve a success rate of only 1.96%, highlighting significant room for improvement.

After the release of the benchmark, OpenAI's testing identified several challenging or possibly unsolvable tasks within SWE-bench benchmark, which may cause the benchmark to systematically underestimate the autonomous software engineering capabilities of models. To address these issues, OpenAI collaborated with the authors of SWE-bench to develop a new release of the benchmark that should provide more accurate evaluations, named SWE-bench Verified[3], which contains 500 instances with higher quality. It retains the diversity of SWE-bench but is easier to evaluate. After the release of SWE-bench Verified, it quickly became the most popular benchmark for evaluating large language models and Agent systems on coding performance.

2.2 Leading LLM-based Bug Fixing Systems

LLM-based Bug Fixing Systems are systems built on Large Language Models (LLMs) that can automatically edit code repositories to fix bugs based on issue reports. Bug fixing is a highly resource-intensive task in software development, requiring developers to reproduce the bugs reported in issue reports, precisely locate defective code snippets within large code repositories, understand the cause of errors, and implement fixes. Automating bug fixing has long attracted widespread attention in both academia and industry. Given the strong logical reasoning and coding capabilities demonstrated by LLMs, numerous works have explored the development of automated bug fixing tools based on LLMs. In this paper, we study six leading LLM-based Bug Fixing Systems, comparing their differences in system design and performance in automated bug fixing, analyzing the shortcomings and limitations of existing systems, and providing direction for future work in building adaptive, high-reliability automated bug fixing systems. Below this section, we will introduce these systems from a technical perspective:

W&B Programmer [6], developed by wandb, leverages OpenAI's o1 model with a high reasoning mode for each step and editing decision. It incorporates a GPT-4o based memory component to compress the agent's history, a custom Python code editor optimized for model context, and the ability to register "auto-commands" that execute after each edit. To enhance performance, the agent employs five parallel rollouts per instance, followed by a "crosscheck" step using o1 as a tie-breaker to select the best rollout.

Blackbox AI Agent [1] developed by Blackbox, is a sophisticated tool designed to enhance productivity and streamline workflows. The agent is equipped with a suite of powerful tools, including `get_repo_structure`, `open_file`, `create_file`, `search_file`, and the ability to execute `bash commands`. These tools enable the

[3] https://openai.com/index/introducing-swe-bench-verified/

agent to efficiently navigate, manipulate, and manage code repositories and files. LLMs leverage these tools in a ReAct (Reasoning and Acting) format to perform a variety of tasks. It can filter and extract relevant content, generate file paths, and validate patches by executing tests. This approach ensures that the agent not only retrieves the necessary information but also verifies the integrity and functionality of the code through rigorous testing.

CodeStory Midwit Agent [2], an agent released by CodeStory. They incorporate a suite of specialized tools, including `list_files`, `open_files`, `str_replace_editor`, `attempt_completion`, `ripgrep_search`, and `terminal_access` for Agent. It utilizes Monte Carlo Tree Search[10] to ensure efficient exploration of the candidate space and filters trajectories by calculating the mean reward of the trajectory's steps. This approach enhances the performance of the Agent system on SWE-Bench tasks.

Learn-by-interact [42] employs an agent that directly interacts with code repositories in offline mode to gather extensive high-quality interaction trajectories. It leverages in-context learning to strategically integrate these historical interaction patterns into prompts. This enables the agent to systematically reference and adapt prior successful debugging procedures when addressing novel software defects, resulting in progressively improved bug resolution effectiveness through accumulated experiential knowledge.

Devlo [3], an agent released by Devlo, providing tools such as `exploring_repository_structure`, `creating_file`, `modifying code`, and `run_script`. Additionally, Devlo employs rigorous testing mechanisms such as `reproduction_test` and `edge_case_test` to meticulously filter and validate candidate patches, ensuring robust and reliable outcomes.

Emergent E1 [4] initially approaches the problem by engaging multiple expert agents, each equipped with tools such as `viewing`, `creating_file`, and `str_replace`. Subsequently, the final patch is selected from among multiple candidate patches.

In this paper, we will analyze the localization accuracy of different localization strategies and the impact of different components on the final patch generation accuracy, providing guidance for building more reliable bug fixing systems in future work.

3 Study Design

In this section, we introduce the research questions and the design of issue quality assessment criteria.

3.1 Research Questions

RQ1. Effectiveness of Systems: How does the LLM-based Agent currently perform in automatic bug fixing in code repositories?

In the SWE-bench Verified leaderboard, resolve rates of different systems vary significantly, with substantial differences in instances each system can and cannot solve, often due to issue description quality and system design. Analyzing why high-quality descriptions sometimes fail while low-quality ones succeed is necessary. We will analyze differences in instances solved by various systems, showing how many are resolved by all systems and how many by none. Similar to the assessment done by Agentless on the SWE-bench Lite dataset, we will design an automated method for quality assessment based on state-of-the-art large language models for the SWE-bench Verified dataset. We will investigate why many high-scoring issues are unresolved by any system, while some low-scoring issues are resolved by all systems.

RQ2. Effectiveness of FL: How do different systems perform in Fault Localization and what are the reasons for their differences?

Fault localization is crucial in bug fixing since accurate localization increases the probability of fixing the bug. We need to investigate the differences in fault localization effectiveness among different systems. Based on the ground truth, we will compile statistics on the proportion of successfully localized faulty files and code symbols (including classes, functions, methods, and top-level statements) for each system in each SWE-bench Verified instance.

RQ3. Effectiveness of Reproduction: How effective is the bug reproduction task currently?

Bug reproduction is vital in bug fixing and dynamic debugging. It aids fault localization through error messages and validates patches. Higher-quality reproduction scripts provide more accurate information, increasing the probability of fixing the bug. Investigating the effectiveness of bug reproduction is crucial. Due to the difficulty of collecting bug reproduction scripts from various repair systems, we designed an Agent repair system with a mainstream repair process. We collected the content and execution commands of reproduction scripts, subsequently evaluating their quality based on output differences before and after applying the golden patch.

3.2 Issue Quality Assessment Criteria

We design a scoring mechanism based on five metrics, allowing us to evaluate the quality of different issue descriptions across multiple dimensions. Specifically, we drew inspiration from the five metrics used in Agentless [53] for evaluating the quality of issue descriptions: Quality of Bug Locations (File/Function/Line), Quality of Reproducible Examples, and Quality of Resolve Solutions. Agentless systematically evaluated the quality of issue descriptions for 300 instances in SWE-bench Lite benchmark using these five metrics. Since SWE-bench Verified was recently proposed, there has been no research conducting similar evaluations on it so far. To fill this gap, we designed clear scoring criteria and utilized the DeepSeek-R1[4] model, one of the models with the best reasoning abilities and cost-effectiveness, to implement the five-dimensional quality assessment for instances in SWE-bench Verified. One point to note is that the Function-level Fault Localization granularity proposed in Agentless was replaced with a similar but broader granularity of code symbol-level Fault Localization in our study, where code symbols include classes, functions, class methods, and top-level statements. We explain the condensed scoring criteria for the five dimensions as follows, and full prompts for the LLM to make the quality assessments can be obtained in our homepage[5].

Quality of File-level Fault Localization: We identify edited code files based on the golden patch (human-written patch, considered ground truth) and treat them as *golden files*. We then input the *issue description* and the corresponding *golden files* to the LLM to determine the extent to which golden files appear in issue descriptions. The scoring criteria are:

[4]https://www.deepseek.com/

[5]https://github.com/mxx1219/icse26_empirical_bug_fixing

- **StackTrace (3)**. The issue description contains stack trace information, which contains the modified file name.
- **Keyword (2)**. The issue description contains the file name and its path prefix. The path prefix is required to uniquely identify the file entity. For example, for *path/to/file_reader.py*, either *path/to/file_reader.py* or *path/to/file_reader* is acceptable. File path and name may appear in different locations but both must be present in the issue description.
- **Natural Language (1)**. The issue description provides the file name in natural language without path prefix. The file name cannot be split - *file_reader.py* or *file_reader* is valid, but *file* and *reader* appearing separately does not qualify.
- **No Information (0)**. This applies when keywords appear separately without forming the complete file name, for example, *file* and *reader* appear individually rather than as *file_reader*, or when no relevant keywords appear.

Since a single case may contain multiple golden files, the LLM is required to evaluate each file based on the aforementioned scoring criteria, with the maximum score ultimately being selected as the annotation result for that case.

Quality of Symbol-level Fault Localization: Similar to file-level localization, we extract modified code symbols (classes, functions, methods) from golden patches via AST analysis, storing them in *file_path::symbol_name* format and providing them to the LLM with issue descriptions. As a case may also involve multiple symbol modifications, its final score is the maximum of all symbol-level evaluations. The scoring criteria are:

- **StackTrace (3)**. The issue description contains stack trace information, which contains the target code symbol name.
- **Keyword (2)**. The issue description includes both the symbol name and its associated file path. While a code symbol may have multiple definitions across different files, the *file_path::symbol_name* combination enables more accurate localization. Notably, we relax the requirement for *complete* file paths since the presence of code symbols in issue descriptions itself provides sufficient contextual constraints.
- **Natural Language (1)**. The issue description mentions the code symbol name in natural language without file path. Following the same principle as file-level localization, these symbol names must remain intact as atomic units, e.g., the symbol *get_name* would not be considered matched if only its constituent parts (*get* or *name*) appear independently.
- **No Information (0)**. The issue description contains neither direct nor sufficiently indicative references to the target symbol. Partial or indirect mentions (e.g., separate components like *get* and *name* appearing without the complete symbol *get_name*) are treated as no symbol information.

Quality of Line-level Fault Localization: We use an out-of-the-box tool named delta[6] to convert the standard diff format golden patch into a *show-line golden patch*, displaying line numbers before and after the patch at the beginning of each line. This annotation aids the model in understanding code modifications and accurately matching line numbers. We then provide the issue description and the show-line golden patch to the LLM. The scoring criteria are:

- **StackTrace (3)**: The description must contain a stack trace explicitly including at least a pair of <file_name, line_number> corresponding to the modified code locations in the patch.
- **Keyword (2)**: At least one pair of <file_name, line_number> appears in the issue text, but not appears in a stack trace.
- **Natural Language (1)**: The issue description contains a natural language explanation of the buggy line(s), but does not specify the exact line number(s). For example, if the patch contains *result = get_content(url, proxy, timeout)*, the description including *result = get_content(url, proxy, timeout)* or *get_content(url, proxy, timeout)* qualifies. Isolated identifiers like *proxy* or *timeout* do not qualify.
- **No Information (0)**: No useful line-related information appears in the issue description.

Quality of Reproducible Examples: We provide only the issue description, without additional ground truth information. Consequently, our approach only requires the LLM to detect the presence of reproducible examples, without evaluating their semantic correctness. The scoring criteria are:

- **Contains REs (3)**. The issue description includes complete, directly runnable code to reproduce the problem without additional setup.
- **Contains Partial REs (2)**. The description provides partial code, containing the main part needed but lacking some information, requiring extra context or setup to run.
- **Info in NL (1)**. The description provides a detailed, step-by-step explanation in natural language on how to reproduce the problem, without code snippets.
- **Not Enough Info (0)**. There is insufficient information to reproduce the problem.

Quality of Resolve Solutions: The model receives the issue description and the golden patch. Scoring criteria are:

- **Exact Patch (3)**. The issue description provides a code solution that is exactly the same as the correct patch. If there is any semantic discrepancy or if the code snippet only covers part of the standard patch rather than the entire content, this option should not be selected.
- **Complete Steps in NL (2)**. The issue description provides a complete natural language solution that is semantically consistent with the correct patch. One could reasonably write the correct patch based on this description without additional guidance. If the description only covers part of the standard patch, this option should not be selected.
- **Some Steps in NL (1)**. The issue description provides a partial natural language solution that is not semantically complete compared to the provided patch. One could write part of the correct patch based on this description but not all. This occurs when users have incomplete understanding and provide partial solutions.
- **No Solution (0)**. The issue description does not provide any solution description - the user only points out the problem without suggesting how to fix it.
- **Misleading Information (-1)**. The issue description includes the user's guess on how to solve the problem, but this guess is completely wrong and off track compared to the correct patch, making it misleading information.

[6]https://github.com/dandavison/delta

3.3 Reliability of the Scoring System

To validate the reliability of the scoring mechanism using DeepSeek R1, we performed a cross-validation against deterministic regex rules designed to identify structured cues such as stack traces in issue descriptions. Specifically, we analyzed all 500 issues and found disagreements in 34 cases. A detailed inspection revealed that DeepSeek substantially outperforms regex-based methods. Regex rules, though precise, often fail to generalize across diverse stack trace formats, which vary by Python version, bug-tracking tool, and runtime environment (e.g., Jupyter Notebook). Furthermore, many issue reports describe stack traces or error locations using natural language rather than standardized formatting. In such cases, DeepSeek's semantic understanding allows it to successfully identify valuable debugging signals that regex rules miss. This result supports the use of DeepSeek R1 as a more robust and adaptable scoring mechanism for assessing issue quality.

4 Analysis & Results

We will sequentially present the analysis results and insights for RQ1 through RQ3.

4.1 RQ1: Effectiveness of Systems

We analyzed the versions of cases that each of the six tools can individually solve, as well as the differences between the cases that each tool can resolve, as shown in Figure 1. The histogram at the top of the figure shows the number of cases each tool can resolve in the SWE-bench Verified dataset. Specifically, ranked from lowest to highest, these are Emergent E1, Devlo, Learn-by-interact, Midwit Agent, Blackbox AI Agent, W&B Programmer, resolving 286, 291, 301, 311, 314, and 323 cases, respectively. W&B Programmer performs the best, achieving a 13.0% performance improvement over Emergent E1 and addressing 64.6% of the total 500 cases in the SWE-bench Verified dataset. Compared to the popular APR benchmark Defects4J [18] over the past decade, SWE-bench Verified introduces stricter usage protocols, prohibiting participants from leveraging dynamic evaluation results generated by closely related failing test cases as feedback information for filtering patches. This test case set can only be utilized as a quality standard once the patch generation process has been completed. In this context, considerable of FL methods based on dynamic test execution information (e.g., spectrum-based and mutation-based methods) cannot be used, adding further difficulty to problem detection and resolution. This strict protocol undoubtedly aligns more closely with real-world development scenarios, where repair tools must rely almost solely on issues raised by users and the current state of the code repository to devise solutions. Against this backdrop, W&B Programmer's ability to address 64.6% of cases underscores its advanced capabilities and utility in real-world development environments.

In Figure 1, the histogram to the right of the tool names presents the count of case versions that different tool combinations can address. Each column indicates the number of case versions solvable by the tools marked with black dots but not by those marked with gray dots. For instance, in the first column, only W&B Programmer is marked with a black dot, while the other five tools are marked with gray dots, indicating that W&B Programmer can solve a unique set of 8 cases that none of the other five tools can handle. Similarly, the seventh column demonstrates that W&B Programmer and Blackbox AI Agent together can solve 3 cases that the other four tools cannot address. The final column shows that there are 181 cases that can be solved by each of 6 tools.

Analysis of Case Solvability. Among the 500 cases in SWE-bench Verified, 404 cases are solvable by at least one of the six tools, while 96 cases remain unsolved by any tool. Furthermore, 181 cases can be solved by all six tools (represented by the last column in Figure 1). We hypothesize that the issue descriptions for these 404 solvable cases are generally of higher quality, whereas the 96 cases that none of the tools can resolve likely exhibit lower-quality issue descriptions. To validate this hypothesis, we conducted a significance analysis of issue quality differences using five metrics. For quantitative comparison, we assigned a value to each option for each metric, as described in Section 3 (e.g., 3 for StackTrace). The only point to note is that we assigned a score of -1 to the Misleading option in the Quality of Resolve Solutions metric. This ensures that a score of 0 consistently represents "no information" across all metrics. Hence, the scoring range was set from -1 to 3 instead of 0 to 4, making scores across different indicators comparable.

To examine the validity of our hypothesis, we calculated the mean values for the five indicators across two case sets: the set of 96 cases unsolved by any tool (referred to as the "No-one-resolve Set") and the set of 181 cases solvable by all tools (referred to as the "All-resolve Set"). These results are shown in Table 1. For each indicator, a higher score indicates that the corresponding issue description provides more complete and detailed information. It is evident that the scores for all five indicators are consistently higher in the All-resolve Set compared to the No-one-resolve Set. This suggests that issue quality significantly influences the effectiveness of resolution methods, underscoring the importance of crafting clear and comprehensive issue descriptions to improve resolution rates from the outset.

Across both case sets, the most significant performance gap lies in the *Quality of Resolve Solutions* metric, whose average score in the All-resolve Set is 1.95 times that of the No-one-resolve Set. This suggests that, at least within the SWE-bench Verified dataset, leveraging symptom-based positive modifications can markedly increase the likelihood of resolving an issue. The next most notable differences are observed in symbol-level, file-level, and line-level bug localization, with score ratios of 1.30×, 1.26×, and 1.22×, respectively. The marginal variance among these three implies that code symbol-level localization may offer relatively greater utility. In contrast, the *Quality of Reproducible Examples* shows a negligible difference (1.01×) between sets, indicating that this metric has limited influence on improving the resolvability of bug-fix systems.

In the above analysis, we validated our hypothesis by calculating differences across five metrics that represent issue quality in the All-resolve Set and No-one-resolve Set, showing a clear disparity in issue quality between cases that can be resolved and those that cannot. This reflects an overall trend, though some outliers may exist. On one hand, we observed several high-scoring cases in the No-one-resolve Set; we aim to investigate why, despite relatively complete information in these issues, no tools were able to resolve them. On the other hand, we also identified some low-scoring cases within the All-resolve Set, which may further highlight the capability of repair tools to independently find and gather necessary

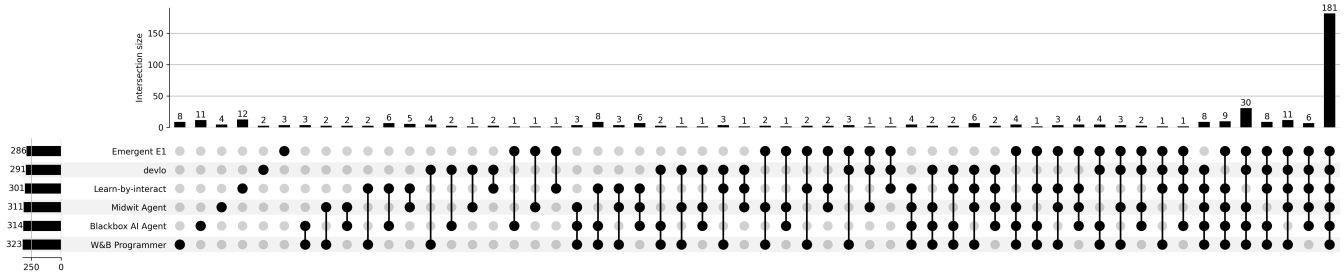


Figure 1: Analysis of Performance of State-of-the-art Techniques on Issue Resolve Task

Table 1: Issue Quality Analysis for All-resolve Cases and No-one-resolve Cases.

Case Sets	Quality of Reproducible Examples	Quality of Resolve Solutions	Quality of File-level Locations	Quality of Symbol-level Locations	Quality of Line-level Locations	Avg Scores
No-one-resolve Set (96 cases)	2.01	0.656	1.052	0.906	0.813	1.087
All-resolve Set (181 cases)	2.022	1.276	1.326	1.177	0.994	1.359
All-resolve Set / No-one-resolve Set	101%	195%	126%	130%	122%	125%

information. To this end, we collected the top 5 cases with the highest average scores across the five metrics in the No-one-resolve Set, and the 5 cases with the lowest average scores in the All-resolve Set, as shown in Table 2. It is evident that cases with a relatively high average score but cannot be successfully resolved generally have lower resolve solution quality. Hence, this feature appears to be the most critical factor influencing whether an issue can be resolved, which is consistent with our previous conclusions. For cases with relatively low average scores but can be resolved by all repair systems, their feature distributions vary significantly. Some provide different granular levels of bug location information, while others include reproducible examples and resolve solutions. A case-by-case analysis is needed to draw conclusions.

We first conducted a manual inspection of the trajectories of the five high-scoring cases that could not be repaired and summarized the following insights:

Insights: (1) Enhancing the understanding of root causes is crucial. The model's understanding should extend beyond the location where the issue occurs (i.e., the symptoms) and include deeper reasoning about the relationship between multiple suspicious locations in the issue title or stack trace and the root cause. This would help prioritize the root cause location more effectively. (2) Improving the ability to generate patches for related locations and to verify patch completeness offers a feasible approach to further enhance issue resolution effectiveness from a holistic perspective.

Subsequently, we conducted a manual inspection of the trajectories of 5 low-scoring cases that could be repaired by all 6 repair systems, and summarized the following insights:

Insights: (3) The manually crafted patch may not be the only solution to the issue; multiple resolution strategies may exist. The model may attempt to generate a semantically correct or partially correct patch based on the suspicious locations already highlighted in the issue. (4) LLMs and agents demonstrate strong capabilities for discovering relevant information. They can deeply analyze the natural language issue description, iteratively extracting high-relevance code segments over multiple interactions.

As shown in Figure 1, we also discovered that Learn-by-interact [42] was able to independently repair 12 cases, the highest among the six repair systems. However, there were 30 cases that could be repaired by any of the other five systems, but not by Learn-by-interact, making it the most noteworthy in this aspect.

Learn-by-interact uses an agent that interfaces with offline code repositories to collect extensive, high-quality interaction trajectories. It strategically integrates these historical interactions into prompts via in-context learning, enabling the agent to reference and adapt successful debugging methods to new software issues. This systematic approach results in progressively better bug resolution as experiential knowledge accumulates. In contrast, the other five systems focus on agent process design without leveraging prior experience, which is a relatively novel idea in Learn-by-interact. The significant design differences between the two sets of systems appear to lead to varied impacts on different cases, as the experiment results suggest.

One insight is that generating patch candidates using diverse workflow designs within the process and the voting on the patches via their validity after generation, potentially enhancing the final effectiveness as the number of overall fixable cases increases.

Table 2: Scores of 5 top-ranked cases in No-one-resolve Set and 5 bottom-ranked cases in All-resolve Set.

No-one-resolve Set						
Case IDs	**Scores of Reproducible Examples**	**Scores of Resolve Solutions**	**Score s of File-level Locations**	**Scores of Symbol-level Locations**	**Scores of Line-level Locations**	**Avg Scores**
astropy-13977	3	2	3	3	3	2.8
astropy-14182	3	0	3	3	3	2.4
django-13513	2	3	2	3	2	2.4
django-16667	3	0	3	3	3	2.4
django-16938	3	0	3	3	3	2.4
All-resolve Set						
django-10914	1	0	0	0	1	0.4
sympy-22456	0	0	1	1	0	0.4
django-13516	1	0	0	0	1	0.4
sphinx-8721	1	0	1	0	0	0.4
django-15499	0	2	0	0	0	0.4

Insights: (5) Using designs that deviate from mainstream processes might help to uncover more repairable cases. For instance, leveraging experience gained from historical repair processes could aid in solving difficult instances. (6) Integrating different repair strategies into the entire workflow to generate a wider variety of patch candidates, and then voting on the validity of these patches, might enhance the final repair effectiveness.

To validate the generalizability of these observations, we further examined ten additional cases. We have to limit the scope of manual analysis due to two primary constraints. First, beyond certain score thresholds, issues no longer met the criteria for being considered extreme, reducing their diagnostic value. Second, each case required in-depth comprehension of the underlying codebase and task context, and the corresponding agent trajectories could span up to 200,000 tokens. This made detailed inspection resource-intensive. To ensure reliability, two authors independently reviewed each case and reconciled their assessments through discussion.

4.2 RQ2: Effectiveness of FL

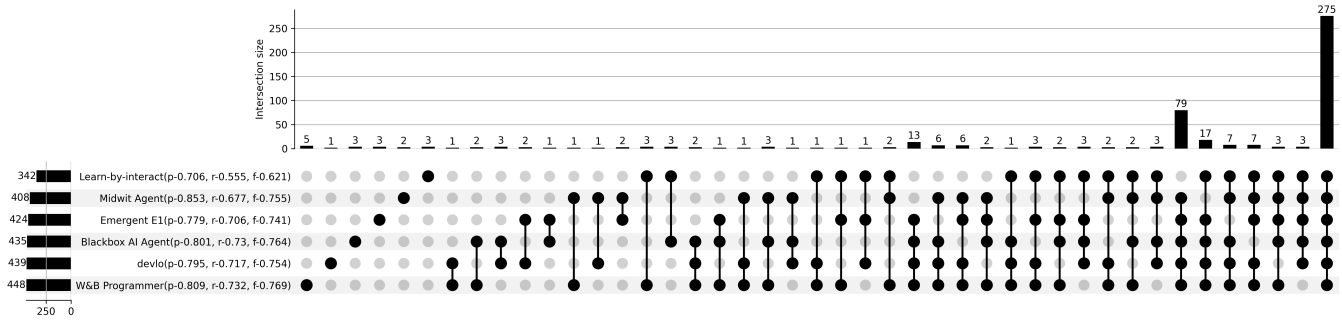
Fault localization (FL) is a critical step in resolving issues, and only when the faulty code element is accurately identified can the model generate a semantically correct patch. In this RQ, we examine the effectiveness of six methods in FL. As run trajectories of most repair techniques lack intermediate FL results or provide insufficient localization granularity (e.g., only offering file-level FL information, without finer-grained code symbol-level or line-level details), we extract the localization information in different levels from the final submitted patch to fairly assess each tool's localization capability. Specifically, we downloaded the final patch set submitted by each tool on the official SWE-bench Verified website, then parsed the formatted patch files. From the perspective of the files pre-modification (pre-patch code files), we recorded the files and code symbols that were modified. We did not conduct line-level experiments because we believe that individual lines of code do not adequately represent a complete functional module, and therefore, have limited reference value. We applied the same processing to the golden patches.

After obtaining the above processed data, we created Figure 2 and Figure 3 to illustrate the hit rate of FL at the file and code symbol levels, respectively. The meaning of the different modules in the figures is similar to Figure 1, except that Figure 3 is oriented towards the FL task.

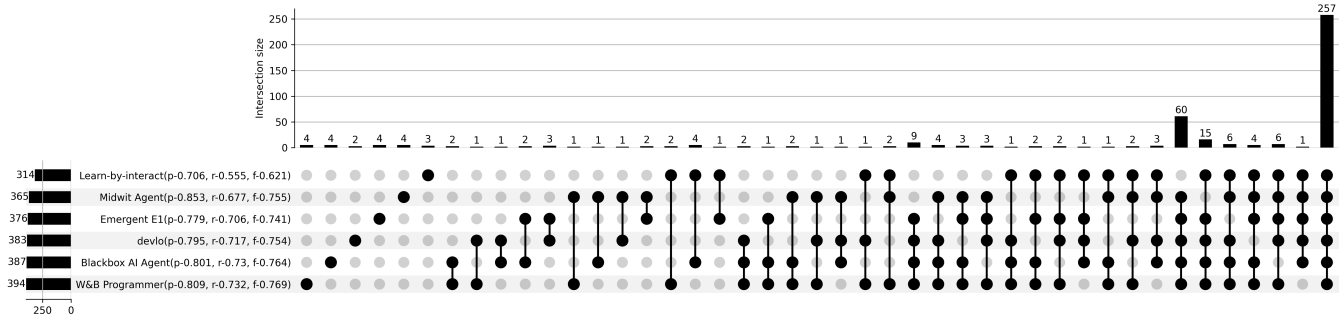
Furthermore, we analyzed the localization performance of the six tools based on their definition of a hit. As shown in Figure 2, the W&B Programmer outperformed others in the file-level localization task, successfully localizing at least one buggy file in 448 cases (Figure 2a), and all buggy files in 394 cases (Figure 2b). In the "Cover-at-least-one-file" setup, W&B Programmer is followed by Devlo, Blackbox AI Agent, Emergent E1, Midwit Agent, and finally Learn-by-interact with 342 cases localized. This indicates a 31.0% improvement for W&B Programmer over Learn-by-interact. In the "Cover-all-files" scenario, only Devlo and Blackbox AI Agent swapped their positions, while the rest remained in the same order. The top-ranked tool, W&B Programmer, showed a 25.5% improvement over the lowest-ranked Learn-by-interact.

Additionally, since each instance may have multiple buggy files and each system may predict a list of files rather than a single file, we compiled the prediction accuracy, recall, and F1-score for each system at the code file level. This data is provided after the name of each system in Figure 2. Based on the F1-score, the systems, sorted from lowest to highest, are Learn-by-interact, Emergent E1, Devlo, Midwit Agent, Blackbox AI Agent, and W&B Programmer. Excluding Learn-by-interact, the relative order of the remaining systems aligns with their ranking in the issue resolution task, indicating a strong positive correlation between accurate file-level localization and overall bug-fixing performance.

Next, we assess FL performance at the code symbol level. In the "Cover-at-least-one-symbol" scenario, the top bug-fixing system is W&B Programmer, successfully localizing 396 cases. The lowest bug-fixing system is Learn-by-interact, with 308 cases localized, resulting in a 28.6% improvement for W&B Programmer. In the "Cover-all-symbols" scenario, W&B Programmer localized 50 more cases than Learn-by-interact, showing a 20% improvement. Compared to file-level results, code symbol-level localization shows more variance from the issue resolution task, and the abilities of various indicators differ. For example, W&B Programmer ranks highest in F1-score for code symbol prediction, making it an

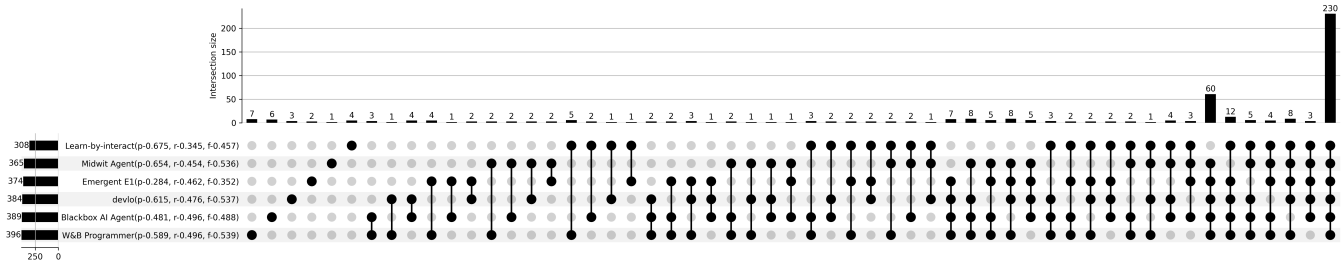


(a) Statistics of samples where at least one buggy file is hit

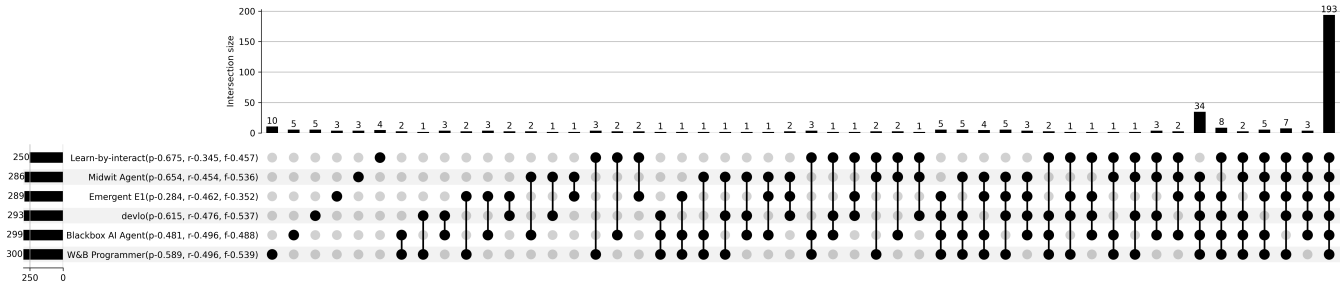


(b) Statistics of samples where all buggy files are hit

Figure 2: Analysis of Performance of State-of-the-art Techniques on File-level FL Task



(a) Statistics of samples where at least one buggy code symbol is hit



(b) Statistics of samples where all buggy code symbols are hit

Figure 3: Analysis of Performance of State-of-the-art Techniques on Code Symbol-level FL Task

all-around performer, while Midwit Agent excels in prediction precision. Although Blackbox AI Agent does not rank in the top three for precision and recall, it remains the second-best in localization from an instance perspective.

We believe these differences are primarily due to the varying localization strategies at the code symbol granularity employed by different models. Some repair systems might focus on recalling more code symbols to provide more flexibility for subsequent patch generation. Others might aim to precisely identify the first erroneous code symbols to avoid misguidance in subsequent repair processes, without overly considering the completeness of code symbol recall and associated modifications. Despite these differences, we still observe a positive correlation between code symbol-level FL results and bug-fixing results. Additionally, compared to the file-level, there is still considerable room for improvement in symbol-level localization effectiveness in the future.

It is easy to observe that in localization experiments, Learn-by-interact consistently ranks last and performs significantly below the other five repair systems. However, in the bug-fixing task, it ranks fourth. To investigate this, we constructed three case sets: 301 cases that Learn-by-interact successfully repaired (Set-Repair), 342 cases where it successfully localized at least one erroneous file (Set-File-Localization), and 308 cases where it localized at least one erroneous code symbol (Set-Symbol-Localization).

The intersection of Set-Repair and Set-File-Localization is 281 cases, indicating that if Learn-by-interact can successfully localize at least one erroneous file, the probability of ultimately resolving the case is 82.2(281/342)%. The intersection of Set-Repair and Set-Symbol-Localization is 262 cases, showing that if it can successfully localize at least one erroneous code symbol, the probability of resolving the case is 85.1%(262/308).

In comparison, these probabilities for W&B Programmer are 69.6% and 72.7%, respectively. Thus, while Learn-by-interact's success rate in predicting the correct files and code symbols is minimal, once a prediction is successful, the subsequent repair success rate is remarkably high. This is why Learn-by-interact can achieve relatively good repair outcomes despite its poor FL performance.

Insights: (7) Existing methods have achieved strong performance in file-level FL, but there remains room for improvement in code symbol-level localization tasks. (8) Both coarse-grained file-level FL results and fine-grained code symbol-level FL results are positively correlated with final bug-fixing performance.

4.3 RQ3: Effectiveness of Bug Reproduction

Under most of top-leading bug-fixing systems, the core process involves searching for suspicious codes, generating bug reproduction scripts, successfully reproducing the issue, and modifying the functional code until the issue is resolved (i.e., passing the reproduction testings). The generation of bug reproduction scripts is of paramount importance for the model's understanding of the problem and for verifying the correctness of the generated patches. Therefore, in this RQ, we plan to specifically investigate the significance of this task. An initial idea was to extract the reproduction scripts generated during the repair process from the trajectories submitted by the top six repair methods. However, we encountered numerous difficulties during the attempts.

Reproduction scripts within a project have different names and locations, complicating accurate filename extraction through rule-based methods. Additionally, trajectories do not provide final versions of reproduction files, as repair systems often modify initially visible contents using code editing tools. These modifications remain partially visible while full text stays obscured. The proprietary nature and diverse designs of these tools make understanding their functions challenging. Moreover, execution commands vary considerably. Some methods generate new Python files with a `main` entry point, while others alter existing test files and execute them using frameworks like `pytest`. Extracting these commands from trajectories proves difficult. Finally, problem reproduction processes may involve environment alterations, such as installing dependencies via `pip` or exporting environment variables, which can impede effective script execution if unrecorded.

Having abandoned extracting and analyzing reproduction scripts from existing repair systems, we implemented an Agent with a standard process to preserve the final reproduced script during runtime, advancing subsequent investigations. Specifically, we designed a repair framework with two Agents, named RepoFixer. Compared with other SOTA bug-fixing systems, Agentless employs a single-step, phase-separated inference process, while SWE-agent uses a multi-step ReAct-based framework. Our implementation aligns more closely with the latter but simplifies the agent architecture to retain only core components, avoiding heavy optimizations to better generalize insights for bug reproduction tasks. In RepoFixer, the first Agent, called Searcher, takes as input the issue description from each sample in the SWE-bench Verified dataset and the current commit state of the code repository. We instruct the Searcher via system prompts to perform code symbol granularity FL tasks and, through multi-turn conversations, continuously search and read issue-related context, ultimately returning a list of suspicious code symbols.

Our second Agent, called Fixer, takes as input the issue description and the suspicious code symbols list returned by Searcher, aiming to generate a patch resolving the issue. In the system prompt, we outline the Fixer's workflow: first, generate a problem reproduction script, run it, and self-assess whether the issue is successfully reproduced based on results. If unsuccessful, continue modifying the reproduction script until the issue is deemed successfully reproduced. Then, generate the patch and verify using the reproduction script whether the modified code successfully resolves the issue (i.e., the previously reproducible issue no longer occurs). If successful, save the patch and conclude; otherwise, continue repairing the patch and running the reproduction script until the issue is resolved or the predefined dialogue turn limit is reached. Regardless of patch generation success, we save the latest edited reproduction script and corresponding execution commands.

During this process, we utilized the powerful Claude 3.5 Sonnet model (October 2024 version)[7] as the base model to implement the Agent process. We used Claude's official toolkits, namely `bash` and `str_replace_editor`, as the Agent runtime toolkits and utilized the standard Function Call interface for model requests. Post-experimental statistics show that RepoFixer successfully resolved 255 issues in total, accounting for 51% of all issues. While we did

[7] https://www.anthropic.com/news/3-5-models-and-computer-use

not adopt very heavy sampling or regression testing strategies, and complex mechanisms such as self-reflection, it appears that there remains some gap in performance compared to the top-ranked methods. However, the current focus of our research question is on investigating the quality of reproduction scripts generated by Agent systems. To explore this, we designed the following experiment.

We executed each collected reproduction script in its respective Docker environment. First, we ran the script on the original repository prior to applying the golden patch, logging the output as `before_fix_status`. Then, in a fresh Docker instance, we applied the golden patch (i.e., the validated correct patches from the SWE-bench Verified dataset) and reran the same script, recording the output as `after_fix_status`. A natural assumption is that if the output differs before and after applying the golden patch, the script is likely related to the issue fixed by the patch. Therefore, reproduction scripts that produce such status differences can be regarded as higher-quality, whereas those showing no difference appear less relevant to the target issue.

Based on this, we collected experimental results. Out of 500 instances generating 500 reproduction scripts, 384 scripts produced status differences before and after the golden patch application to the repository, representing 76.8%. We term reproduction scripts exhibiting this property as "related reproduction scripts." We hypothesize that the higher the quality of the reproduction information in the issue description, the more likely the model is to generate related reproduction scripts. For this, we conducted detailed statistics. Among 194 samples containing "Contains REs," 156 produced related reproduction scripts, accounting for 80.4%; among 169 samples containing "Contains Partial REs," 135 produced related reproduction scripts, accounting for 79.9%; among 87 samples where the information is in natural language ("Info in NL"), 61 produced related reproduction scripts, accounting for 70.1%; and among 50 samples marked "Not Enough Info," 32 produced related reproduction scripts, accounting for 64%. It can be observed that as the quality of reproduction descriptions in the issue decreases, the proportion of generating related reproduction scripts also decreases. Thus, for users, accurately and comprehensively recording the process of triggering an issue in code form is of significant value for automating the resolution of the issue.

Insights: (9) The most effective bug-fixing systems currently employ a process based on bug reproduction, patch generation, and patch correctness verification. Among these, the bug reproduction process holds significant value. (10) Drafting clear and comprehensive bug reproduction cases within the issue can greatly reduce the difficulty of reproducing the bug, consequently enhancing the efficiency of end-to-end bug-fixing tasks.

5 Discussion

5.1 Large Language Model

From the LLM perspective, it is necessary to further enhance the model's reasoning ability so that it can accurately identify information related to the bug within the issue, thereby reducing the interference of noise. Additionally, for multiple potential repair locations, the model should utilize its reasoning capability to select the location most relevant to the issue.

5.2 Agentic Flow

Agents should especially focus on the quality of the issue and pay attention to multiple suspicious locations in the stack trace. The Agentic flow design should include mechanisms to check the completeness of patches and consider the global impact of the fixes. During the use of the model, mechanisms should be established to either avoid the randomness of the model's output or make full use of the diversity in the model's output. The effectiveness of fault localization is important for bug-fixing tasks, finer-grained code symbol level fault localization task still has significant room for improvement. During the reproduction process, it is necessary to increase the proportion of the dynamic repair process, centered on bug reproduction, patch generation, and patch validation. The quality of bug reproduction scripts should be improved by introducing more relevant artifacts and increasing model reflection.

5.3 General Applicability

While our study uses the SWE-bench Verified benchmark with only Python repositories, we believe the analytical insights generalize beyond this setting. The benchmark is widely used by organizations such as OpenAI and Anthropic, and the analyzed systems rely on language-agnostic tools, with no signs of Python-specific tuning. Nevertheless, we recognize the importance of multilingual benchmarks and are developing a broader dataset for future evaluations.

5.4 Qualitative Evaluations from Developers

While we evaluated patch correctness via test suite validation, we acknowledge it may not fully capture patch quality from a developer's perspective. To supplement this, we manually analyzed 30 patches per system, evaluating code style, readability, and adherence to development norms. We found that agent-based workflows enable context-aware understanding, produce syntactically correct code, reuse existing structures effectively, and often yield clean, human-readable patches with meaningful comments. These results indicate that modern LLM-based agents can generate patches that are both correct and stylistically aligned with professional standards. More comprehensive evaluation with developer feedback remains future work.

6 Threats to Validity

Fail-to-Pass Tests: SWE-bench uses Fail-to-Pass (F2P) tests to verify patch correctness. However, F2P tests may lack comprehensiveness, allowing patches to pass F2P and be deemed correct without fully addressing the issue. This is a common problem in APR and LLM evaluation based on unit tests [31]. In this context, we assume patch correctness when F2P tests pass. We also call for academic community contributions to improve test cases in the SWE-bench evaluation dataset, making evaluation results more reliable.

Uncertainty of LLM: The output of LLMs is stochastic, leading to a probabilistic nature for whether an instance is solved. In this work, we directly analyzed the patches and trajectories submitted by various systems, assuming that the results submitted to SWE-bench represent the best performance of the Agent. Furthermore, conducting multiple experiments for each system to eliminate stochasticity is impractical in terms of both cost and accessibility. As these systems are typically closed-source, we cannot standardize their model

and decoding configurations or conduct a controlled ablation using identical LLM backbones. Thus, differences in model capacity inherently contribute to variations in performance, a factor we acknowledge explicitly as a limitation of our comparative analysis.

LLM-based Issue Quality Assessment: To ensure that the LLM's evaluation results are as close as possible to those of software experts, we took the following measures: (1) We selected the DeepSeek-R1 model, which has top-tier inference performance, as the evaluator; (2) Initially, we conducted model annotations and manual annotations on a subset of the SWE-bench Verified dataset, continuously refining the scoring prompts based on discrepancies until they aligned with human expectations; (3) We used DeepSeek-R1 to perform five annotations for each metric of each instance and selected the option with the highest frequency as the final choice.

Reliability of Reproduction Scripts: While our approach leverages output divergence as an indication of successful bug reproduction, inspired by mutation-based fault localization [26], this method may still misclassify some non-triggering scripts as successful reproductions. Incorporating additional checks, such as verifying log patterns or exception types, could theoretically reduce such misclassification. However, given the practical complexity of implementing and reliably automating these checks within limited agent interactions, we consciously accepted this limitation to balance practicality and effectiveness. Future work could explore incorporating such checks to more precisely measure reproduction accuracy and further mitigate this validity threat.

7 Related Work

7.1 Large Language Models

Large language models (LLMs) are highly advanced pre-trained language models. These models undergo initial unsupervised training on vast amounts of corpus, followed by fine-tuning for specific tasks to enhance performance.

Language models are classified into three categories based on their architecture: encoder-only models [14], decoder-only models [39], and encoder-decoder models [45]. Most existing LLMs for code utilize the transformer architecture's encoders, known for their exceptional learning capabilities and scalability. Regardless of their architecture, most models can be fine-tuned with task-specific data to enhance performance [29].

Large language models (LLMs) have become a promising choice for various software engineering tasks due to their impressive performance in both code generation and understanding [57], such as program synthesis [31, 46–48, 64], code translation [58, 59], program repair [16, 30, 54], fault detection and localization [13, 41], incident analysis [9, 12], code summarization [15] and testing [43]. For example, Codex [11], StarCoder [33], and DeepSeek-Coder [63] are notable code-specific LLMs developed through extensive training on large datasets of open-source code snippets. Additionally, instruction-following code-specific LLMs such as DeepSeek-Coder-Instruct [63] and Magicoder [50] have been created using instruction-tuning methods to enhance their utility in coding tasks.

7.2 Fault Localization and Program Repair

Fault localization (FL) [51] techniques aim to discover and analyze the location and causes of faults, which can be categorized into dynamic and static approaches. Dynamic techniques, such as spectrum-based fault localization (SBFL) [7, 8] and mutation-based fault localization (MBFL) [40], analyze the dynamic execution information to determine fault locations, though they are resource-intensive. Static FL techniques [35] determine fault locations through semantic or syntactic analysis at the bug report or source code level, offering fast detection with low resource consumption. Advanced FL techniques, such as multiple fault localization (MFL) and combined dynamic and static methods, have emerged to find more errors [20, 37, 55].

Automated program repair (APR) [24] has attracted significant attention over the past decade. APR techniques aim to generate patches for buggy programs to pass given test suites. These techniques can be categorized into search-based [25, 36], semantics-based [21, 22, 38], and pattern/learning-based approaches [27, 28, 60]. Search-based APR techniques like GenProg [23] use predefined code mutation operators to generate patches, while semantics-based APR techniques generate patches by solving repair constraints based on test suite specifications. Learning-based APR techniques, such as those utilizing deep learning models, train on large code repositories to predict correct patches. Recent work uses LLMs for APR, often focusing on constructing APR-specific prompts to guide LLMs in generating patches for buggy program statements [54].

7.3 Agents for Software Development

The emergence and popularity of agent-based frameworks have led to the development of agent-based approaches for solving software engineering tasks. Devin and its open-source counterpart OpenDevin [49] are among the first end-to-end LLM agent-based frameworks. These frameworks use agents for planning based on user requirements and enable agents to iteratively perform tasks using tools like file editors, terminals, and web search engines. SWE-agent [56], for example, designs a custom agent-computer interface (ACI) allowing LLM agents to interact with the repository environment through actions such as reading, editing files, and running bash commands. AutoCodeRover [61] provides LLM agents with specific APIs to effectively identify locations needing modification to resolve issues. Numerous other agent-based approaches have been developed, both in open-source and commercial products.

8 Conclusion

In this paper, we analyzed the top 6 bug-fixing systems on SWE-bench Verified. We conducted detailed analyses of the performance of LLM-based Agents in automatic bug fixing for code repositories, and the performance of different systems in different levels of fault localization. Moreover, we developed a mainstream workflow bug-fixing agent to evaluate the overall performance on bug reproduction tasks. The analysis results indicate that to further enhance the capabilities of LLM-based Agents in bug fixing, future efforts should focus on improving the reasoning ability of LLMs, and the Agentic flow design, considering the quality of issues, stack traces, and the correctness of reproductions.

Data availability

The code and experimental data in this paper are accessible on this page: *https://github.com/mxx1219/icse26_empirical_bug_fixing*

References

[1] 2025. Blackbox AI Agent. https://www.blackbox.ai/.
[2] 2025. Code Story Midwit Agent. https://codestory.ai/.
[3] 2025. Devlo. https://devlo.ai/.
[4] 2025. Emergent E1. https://emergent.sh/.
[5] 2025. Gru. https://gru.ai/.
[6] 2025. W&B Programmer. https://github.com/wandb/programmer.
[7] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan JC Van Gemund. 2009. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software* 82, 11 (2009), 1780–1792.
[8] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. 2007. On the accuracy of spectrum-based fault localization. In *Testing: Academic and industrial conference practice and research techniques-MUTATION (TAICPART-MUTATION 2007)*. IEEE, 89–98.
[9] Toufique Ahmed, Supriyo Ghosh, Chetan Bansal, Thomas Zimmermann, Xuchao Zhang, and Saravan Rajmohan. 2023. Recommending root-cause and mitigation steps for cloud incidents using large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1737–1749.
[10] Antonis Antoniades, Albert Örwall, Kexun Zhang, Yuxi Xie, Anirudh Goyal, and William Wang. 2024. SWE-Search: Enhancing Software Agents with Monte Carlo Tree Search and Iterative Refinement. *arXiv preprint arXiv:2410.20285* (2024).
[11] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).
[12] Yinfang Chen, Huaibing Xie, Minghua Ma, Yu Kang, Xin Gao, Liu Shi, Yunjie Cao, Xuedong Gao, Hao Fan, Ming Wen, et al. 2024. Automatic root cause analysis via large language models for cloud incidents. In *Proceedings of the Nineteenth European Conference on Computer Systems*. 674–688.
[13] Xiaohu Du, Ming Wen, Jiahao Zhu, Zifan Xie, Bin Ji, Huijun Liu, Xuanhua Shi, and Hai Jin. 2024. Generalization-Enhanced Code Vulnerability Detection via Multi-Task Instruction Fine-Tuning. *arXiv preprint arXiv:2406.03718* (2024).
[14] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).
[15] Mingyang Geng, Shangwen Wang, Dezun Dong, Haotian Wang, Ge Li, Zhi Jin, Xiaoguang Mao, and Xiangke Liao. 2024. Large language models are few-shot summarizers: Multi-intent comment generation via in-context learning. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–13.
[16] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of code language models on automated program repair. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1430–1442.
[17] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770* (2023).
[18] René Just, Darioush Jalali, and Michael D Ernst. 2014. 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*. 437–440.
[19] Sungmin Kang, Bei Chen, Shin Yoo, and Jian-Guang Lou. 2023. Explainable automated debugging via large language model-driven scientific debugging. *arXiv preprint arXiv:2304.02195* (2023).
[20] Yunho Kim, Seokhyeon Mun, Shin Yoo, and Moonzoo Kim. 2019. Precise learn-to-rank fault localization using dynamic and static features of target programs. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 28, 4 (2019), 1–34.
[21] Xuan-Bach D Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. JFIX: semantics-based repair of Java programs via symbolic PathFinder. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 376–379.
[22] Xuan-Bach D Le, David Lo, and Claire Le Goues. 2016. Empirical study on synthesis engines for semantics-based program repair. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 423–427.
[23] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2011. Genprog: A generic method for automatic software repair. *Ieee transactions on software engineering* 38, 1 (2011), 54–72.
[24] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated program repair. *Commun. ACM* 62, 12 (2019), 56–65.
[25] Dongcheng Li, W Eric Wong, Mingyong Jian, Yi Geng, and Matthew Chau. 2022. Improving search-based automatic program repair with Neural Machine Translation. *IEEE Access* 10 (2022), 51167–51175.
[26] Xia Li and Lingming Zhang. 2017. Transforming programs and tests in tandem for fault localization. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–30.
[27] Yi Li, Shaohua Wang, and Tien N Nguyen. 2020. Dlfix: Context-based code transformation learning for automated program repair. In *Proceedings of the ACM/IEEE 42nd international conference on software engineering*. 602–614.
[28] Yi Li, Shaohua Wang, and Tien N Nguyen. 2022. Dear: A novel deep learning-based approach for automated program repair. In *Proceedings of the 44th international conference on software engineering*. 511–523.
[29] Bo Lin, Shangwen Wang, Zhongxin Liu, Yepang Liu, Xin Xia, and Xiaoguang Mao. 2023. Cct5: A code-change-oriented pre-trained model. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1509–1521.
[30] Bo Lin, Shangwen Wang, Ming Wen, Liqian Chen, and Xiaoguang Mao. 2024. 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*.
[31] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2024. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems* 36 (2024).
[32] Yizhou Liu, Pengfei Gao, Xinchen Wang, Chao Peng, and Zhao Zhang. 2024. MarsCode Agent: AI-native Automated Bug Fixing. *arXiv preprint arXiv:2409.00899* (2024).
[33] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, et al. 2024. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173* (2024).
[34] Yingwei Ma, Qingping Yang, Rongyu Cao, Binhua Li, Fei Huang, and Yongbin Li. 2024. How to Understand Whole Software Repository? arXiv:2406.01422 [cs.SE] https://arxiv.org/abs/2406.01422
[35] Xiaoguang Mao, Yan Lei, Ziying Dai, Yuhua Qi, and Chengsong Wang. 2014. Slice-based statistical fault localization. *Journal of Systems and Software* 89 (2014), 51–62.
[36] Ben Mehne, Hiroaki Yoshida, Mukul R Prasad, Koushik Sen, Divya Gopinath, and Sarfraz Khurshid. 2018. Accelerating search-based program repair. In *2018 IEEE 11th international conference on software testing, verification and validation (ICST)*. IEEE, 227–238.
[37] Neelofar Neelofar, Lee Naish, Jason Lee, and Kotagiri Ramamohanarao. 2017. Improving spectral-based fault localization using static analysis. *Software: Practice and Experience* 47, 11 (2017), 1633–1655.
[38] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. Semfix: Program repair via semantic analysis. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 772–781.
[39] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474* (2022).
[40] Mike Papadakis and Yves Le Traon. 2015. Metallaxis-FL: mutation-based fault localization. *Software Testing, Verification and Reliability* 25, 5-7 (2015), 605–628.
[41] Yihao Qin, Shangwen Wang, Yiling Lou, Jinhao Dong, Kaixin Wang, Xiaoling Li, and Xiaoguang Mao. 2024. AgentFL: Scaling LLM-based Fault Localization to Project-Level Context. *arXiv preprint arXiv:2403.16362* (2024).
[42] Hongjin Su, Ruoxi Sun, Jinsung Yoon, Pengcheng Yin, Tao Yu, and Sercan Ö Arık. 2025. Learn-by-interact: A Data-Centric Framework for Self-Adaptive Agents in Realistic Environments. *arXiv preprint arXiv:2501.10893* (2025).
[43] Maolin Sun, Yibiao Yang, Yang Wang, Ming Wen, Haoxiang Jia, and Yuming Zhou. 2023. SMT solver validation empowered by large pre-trained language models. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1288–1300.
[44] Wei Tao, Yucheng Zhou, Wenqiang Zhang, and Yu Cheng. 2024. MAGIS: LLM-Based Multi-Agent Framework for GitHub Issue Resolution. *arXiv preprint arXiv:2403.17927* (2024).
[45] Zhao Tian, Junjie Chen, Qihao Zhu, Junjie Yang, and Lingming Zhang. 2022. Learning to construct better mutation faults. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–13.
[46] Chaozheng Wang, Junhao Hu, Cuiyun Gao, Yu Jin, Tao Xie, Hailiang Huang, Zhenyu Lei, and Yuetang Deng. 2023. How practitioners expect code completion?. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1294–1306.
[47] Shangwen Wang, Mingyang Geng, Bo Lin, Zhensu Sun, Ming Wen, Yepang Liu, Li Li, Tegawendé F Bissyandé, and Xiaoguang Mao. 2023. Natural language to code: How far are we?. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 375–387.
[48] Shangwen Wang, Bo Lin, Zhensu Sun, Ming Wen, Yepang Liu, Yan Lei, and Xiaoguang Mao. 2023. Two birds with one stone: Boosting code generation and code search via a generative adversarial network. *Proceedings of the ACM on Programming Languages* 7, OOPSLA2 (2023), 486–515.
[49] Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. 2024. OpenDevin: An Open Platform for AI Software Developers as Generalist Agents. *arXiv preprint arXiv:2407.16741* (2024).

[50] Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2023. Magicoder: Source code is all you need. *arXiv preprint arXiv:2312.02120* (2023).
[51] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A survey on software fault localization. *IEEE Transactions on Software Engineering* 42, 8 (2016), 707–740.
[52] Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, et al. 2023. The rise and potential of large language model based agents: A survey. *arXiv preprint arXiv:2309.07864* (2023).
[53] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2024. Agentless: Demystifying LLM-based Software Engineering Agents. *arXiv preprint arXiv:2407.01489* (2024).
[54] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated program repair in the era of large pre-trained language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1482–1494.
[55] Xi Xiao, Yuqing Pan, Bin Zhang, Guangwu Hu, Qing Li, and Runiu Lu. 2021. ALBFL: A novel neural ranking model for software fault localization via combining static and dynamic features. *Information and Software Technology* 139 (2021), 106653.
[56] John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. Swe-agent: Agent-computer interfaces enable automated software engineering. *arXiv preprint arXiv:2405.15793* (2024).
[57] Kang Yang, Xinjun Mao, Shangwen Wang, Tanghaoran Zhang, Bo Lin, Yanlin Wang, Yihao Qin, Zhang Zhang, and Xiaoguang Mao. 2023. Enhancing Code Intelligence Tasks with ChatGPT. *arXiv preprint arXiv:2312.15202* (2023).
[58] Zhen Yang, Fang Liu, Zhongxing Yu, Jacky Wai Keung, Jia Li, Shuo Liu, Yifan Hong, Xiaoxue Ma, Zhi Jin, and Ge Li. 2024. Exploring and unleashing the power of large language models in automated code translation. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1585–1608.
[59] Zeliang Yu, Ming Wen, Xiaochen Guo, and Hai Jin. 2024. Maltracker: A Fine-Grained NPM Malware Tracker Copiloted by LLM-Enhanced Dataset. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*.
[60] Quanjun Zhang, Chunrong Fang, Yuxiang Ma, Weisong Sun, and Zhenyu Chen. 2023. A survey of learning-based automated program repair. *ACM Transactions on Software Engineering and Methodology* 33, 2 (2023), 1–69.
[61] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. Autocoderover: Autonomous program improvement. *arXiv preprint arXiv:2404.05427* (2024).
[62] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. 2023. A survey of large language models. *arXiv preprint arXiv:2303.18223* (2023).
[63] Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y Wu, Yukun Li, Huazuo Gao, Shirong Ma, et al. 2024. DeepSeek-Coder-V2: Breaking the Barrier of Closed-Source Models in Code Intelligence. *arXiv preprint arXiv:2406.11931* (2024).
[64] Chen Zhu-Tian, Zeyu Xiong, Xiaoshuo Yao, and Elena Glassman. 2024. Sketch Then Generate: Providing Incremental User Feedback and Guiding LLM Code Generation through Language-Oriented Code Sketches. *arXiv preprint arXiv:2405.03998* (2024).