Practical LLM-Based Function-Level Automated Program Repair: How Far Are We?

JIAHONG XIANG, Southern University of Science and Technology, China
XIAOYANG XU, Southern University of Science and Technology, China
FANCHU KONG, Southern University of Science and Technology, China
MINGYUAN WU, Southern University of Science and Technology, China
ZIZHENG ZHAN, Kwai Inc., China
HAOTIAN ZHANG, Kwai Inc., China
YUQUN ZHANG*, Southern University of Science and Technology, China

Recently, multiple Automated Program Repair (APR) techniques based on Large Language Models (LLMs) have been proposed to enhance the repair performance. While these techniques mainly focus on the single-line or hunk-level repair, they face significant challenges in real-world applications due to the limited repair task scope and costly statement-level fault localization. However, the more practical function-level APR, which broadens the scope of APR task to fix entire buggy functions and requires only cost-efficient function-level fault localization, remains underexplored.

In this paper, we conduct a comprehensive study of LLM-based function-level APR including investigating the effect of the few-shot learning mechanism and the auxiliary repair-relevant information. Specifically, we adopt six widely-studied LLMs and construct a benchmark on both the Defects4J 1.2 and 2.0 datasets. Our study demonstrates that LLMs with zero-shot learning are already powerful function-level APR techniques, while applying the few-shot learning mechanism leads to disparate repair performance. Moreover, we find that directly applying the auxiliary repair-relevant information to LLMs significantly increases function-level repair performance and even outperforms multiple recent APR techniques. Inspired by our findings, we propose an LLM-based function-level APR technique, namely SRepair, which adopts a dual-LLM framework to leverage the power of the auxiliary repair-relevant information for advancing the repair performance. The evaluation results demonstrate that SRepair can correctly fix 227 single-function bugs in the Defects4J dataset, largely surpassing all previous APR techniques by at least 26%, without the need for the costly statement-level fault location information. Furthermore, SRepair successfully fixes 21 multi-function bugs in the Defects4J dataset, significantly outperforming other state-of-the-art APR techniques.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: Automated Program Repair, Large Language Model

*Yuqun Zhang is the corresponding author.

Authors' Contact Information: Jiahong Xiang, Southern University of Science and Technology, Shenzhen, China, xiangjh2022@mail.sustech.edu.cn; Xiaoyang Xu, Southern University of Science and Technology, Shenzhen, China, 12112620@mail.sustech.edu.cn; Fanchu Kong, Southern University of Science and Technology, Shenzhen, China, 12112822@mail.sustech.edu.cn; Mingyuan Wu, Southern University of Science and Technology, Shenzhen, China, 11849319@mail.sustech.edu.cn; Zizheng Zhan, Kwai Inc., Beijing, China, zhanzizheng@kuaishou.com; Haotian Zhang, Kwai Inc., Beijing, China, zhanghaotian@kuaishou.com; Yuqun Zhang, Southern University of Science and Technology, Shenzhen, China, zhangyq@sustech.edu.cn.

ACM 1557-7392/2018/8-ART111
https://doi.org/XXXXXXX.XXXXXXX

ACM Reference Format:
Jiahong Xiang, Xiaoyang Xu, Fanchu Kong, Mingyuan Wu, Zizheng Zhan, Haotian Zhang, and Yuqun Zhang. 2018. Practical LLM-Based Function-Level Automated Program Repair: How Far Are We?. *ACM Trans. Softw. Eng. Methodol.* 37, 4, Article 111 (August 2018), 35 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Fixing software defects costs developers a significant amount of time and effort [48]. To assist developers in reducing the burden of repairing programs, Automated Program Repair (APR) techniques have been proposed to automatically generate potential patches for buggy programs. Specifically, the learning-based APR techniques which incorporate the learning power to advance the repair performance have been increasingly studied in recent years. For instance, many such techniques [41, 56, 70, 82, 131, 132, 141] utilize Neural Machine Translation (NMT) [104] such that APR is modeled as a translation task where the objective is to transform buggy code into correct code.

More recently, Large Language Models (LLMs) have become largely adopted in various downstream software tasks [44, 45, 50, 62, 67, 105–109, 122, 134] including APR [55, 59, 92, 116, 118, 120, 121] where they have been proven to advance the repair performance [47, 55, 118–120], e.g., the Codex model can fix 32 more bugs than previous APR techniques in the Defects4J 1.2 dataset [119]. Meanwhile, researchers also propose multiple LLM-based APR techniques [59, 68, 112, 116, 120, 121, 125] to further enhance the repair performance. For instance, the state-of-the-art LLM-based APR technique ChatRepair [121] employs a conversational repair mechanism and successfully fixes 162 out of 337 bugs in a crafted Defects4J dataset, causing at least 21.8% gain compared with the previous techniques. However, many LLM-based APR techniques are proposed for the single-line or hunk-level program repair by auto-completing a single line [65] or infilling a hunk of code with context [120] respectively. They typically rely on identifying statement-level program faults, i.e., given fault locations [51, 116, 118, 120, 135] or applying statement-level fault localization techniques such as Gzoltar [112]. Nevertheless, it has been widely argued that accurately identifying statement-level faults can be essentially costly, i.e., demanding fine-grained input or strong assumptions [29, 75, 102], thus potentially limiting the real-world applicability of the single-line or hunk-level APR. On the other hand, the LLM-based function-level APR can be potentially more promising, i.e., applying a generative model such as an LLM to auto-regressively generate the entire patched version of the buggy function by prompting the buggy function into the LLM. To illustrate, first, the function-level APR enables a larger scope of the program repair task—it involves not only the single-line and hunk-level repair tasks, but also a more complicated task which repairs multiple discontinuous lines or hunks within a function [101]. Second, identifying function-level faults tends to be more cost-efficient than identifying statement-level faults, thus rendering the function-level APR more practical in real world [64, 69, 80]. These properties make function-level APR more practical and worthy of investigation, and we use single-function bugs as a controlled subset for systematic empirical evaluation.

While LLM-based function-level APR techniques are more promising, there lacks sufficient study and understanding of them [119], thus potentially hindering the further improved usage of LLMs for APR. Specifically, first, the existing LLM-based APR techniques exhibit significant performance loss for the function-level APR, e.g., incurring a decrease of 33% in ChatRepair and 36.4% in CodexRepair [119, 121] in terms of correctly fixed bugs. Second, the rationale of how such techniques can be affected has not been fully investigated. More specifically, the effectiveness of certain commonly-used mechanism such as the few-shot learning [59, 119, 121], i.e., prompting buggy code and fixed code pair examples that illustrate the function-level APR task and provide repair context for advancing the learning power of models, remains inadequately validated. Additionally, the potential of incorporating the auxiliary repair-relevant information, such as trigger tests and

bug reports, remains underexplored [47, 92, 121]. Thus, there is an urgent need to extensively study the LLM-based function-level APR to further enhance the repair performance.

In this paper, we conduct a comprehensive study on the function-level LLM-based APR to investigate the effects of the few-shot learning and the auxiliary repair-relevant information. Specifically, we adopt six widely-studied LLMs including the state-of-the-art LLMs such as GPT-4, Qwen3, Llama3.1 and StarCoder2 [68, 125, 126, 139] as the study subjects. We also construct a benchmark containing 522 single-function bugs, i.e., the bugs existing within one single function, in the Defects4J dataset. Typically, we build our repair prompt containing the buggy function along with (1) the buggy code and fixed code pair examples to utilize the few-shot learning mechanism and (2) the auxiliary repair-relevant information such as trigger tests for the studied LLMs respectively. In this way, the entire patched functions can be auto-generated and then validated with the Defects4J test suite to derive the plausible patches (which pass all the tests). Our evaluation results show that incorporating the few-shot learning mechanism in the function-level APR actually causes significantly disparate and even negative impacts on the average number of plausible fixes compared with applying the default LLMs only, i.e., from an increase of 6.9% to a decrease of 50.8% among all studied LLMs. Surprisingly, we find that directly applying trigger tests or error messages for prompting can significantly enhance the repair performance, with both achieving a 38.5% increase in the average number of correct fixes. Meanwhile, LLMs which simply adopt project-specific information (trigger tests, error messages, and buggy code comments) outperform multiple recent APR techniques [68, 116, 120, 121] by at least 25.1% without relying on costly statement-level fault location information. Furthermore, while we find that incorporating bug report information improves function-level APR performance, adopting project-specific information can still achieve comparable repair effectiveness without relying on bug reports which cannot be ensured with high quality in practice. Such results indicate the potential of leveraging practical project-specific information to improve function-level APR performance. In our study, over 10 million patches are generated and validated, consuming more than 8,000 GPU and 100,000 CPU hours. To our best knowledge, this is the largest empirical study of LLM-based APR conducted to date.

Inspired by our findings, we propose an LLM-based function-level APR technique, namely SRepair, which adopts a dual-LLM framework to leverage the power of the auxiliary repair-relevant information for advancing the repair performance. In particular, SRepair first adopts a repair suggestion model which employs the Chain of Thought (CoT) technique [114] to generate natural-language repair suggestions. More specifically, SRepair prompts the LLM with the buggy function and the auxiliary repair-relevant information (i.e., trigger tests, error messages, and comments) to identify the root causes of the bugs and generate repair suggestions in natural language accordingly. SRepair then adopts a patch generation model to auto-generate a patched function with the assistance of the repair suggestions. Our evaluation demonstrates that SRepair can correctly fix a total of 227 single-function bugs in our Defects4J dataset, largely surpassing all previous APR techniques, e.g., 26% more than D4C [125] and 40% more than ChatRepair [121], without the need for the costly statement-level fault location information. Moreover, 33 bugs out of them were not fixed by any of the baseline APR techniques adopted in this paper. Surprisingly, SRepair is also capable of repairing 21 multi-function bugs, i.e., bugs existing across multiple functions at the same time, significantly outperforming other state-of-the-art APR techniques.

To summarize, this paper makes the following contributions:

- We perform an extensive study on the LLM-based function-level APR with the impact factors on its performance, paving the way for new directions in future research.

- We find that LLMs with zero-shot learning are already powerful function-level APR techniques. We also find that applying auxiliary repair-relevant information can substantially improve the repair performance for all studied LLMs.
- We propose a new LLM-based function-level APR technique, SRepair, which can achieve remarkable repair performance by correctly fixing 227 single-function bugs, largely surpassing the SOTA techniques, i.e., outperforming D4C [125] by 26% and ChatRepair [121] by 40% in the Defects4J dataset. Moreover, SRepair successfully fixes 21 multi-function bugs, significantly outperforming other state-of-the-art APR techniques.

2 Background & Related Work

2.1 Large Language Model

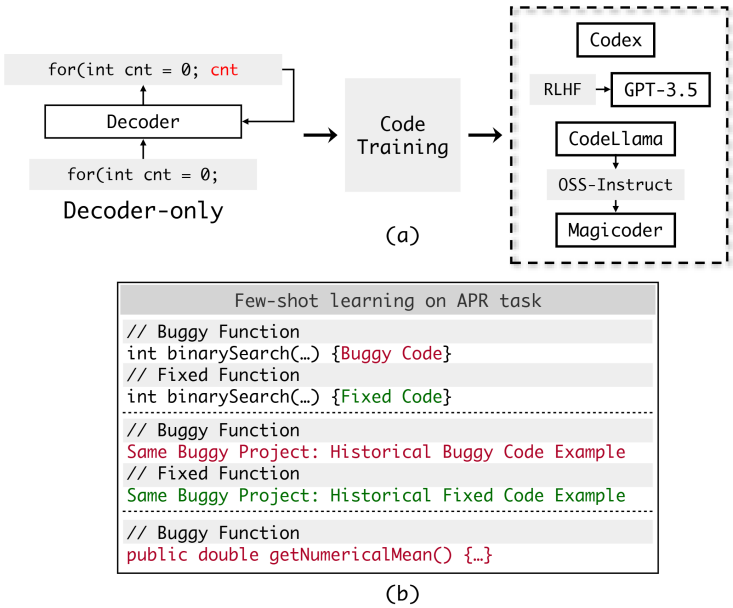

Fig. 1. Decoder-only models and few-shot learning on APR

Large Language Models (LLMs) contain billions of parameters and are trained on petabyte-scale datasets. They are typically built based on the Transformer architecture [111] comprising an encoder for input processing and a decoder for output token generation. In particular, the decoder-only models as shown in Figure 1a have demonstrated superior text comprehension [91] and code generation capabilities [31]. Thus, they have garnered significant interest of researchers and been widely applied to various downstream tasks in software engineering, e.g., test case generation [62, 67], vulnerability detection [44, 45], and program repair [47, 125, 139]. These models when integrating domain-specific knowledge for specific tasks, are often fine-tuned [103] for further improving their performance. For instance, GPT-3.5-Turbo is fine-tuned from the GPT-3.5 [5] base model using extensive conversational data to improve its performance specifically for interactive dialogue tasks. StarCoder2-15B-Instruct-v0.1 [21] and Magicoder [8] are fine-tuned with OSS-Instruct [115] to enhance the code generation performance. While fine-tuning requires significant computational resources and specialized datasets [40], simpler prompting strategies like few-shot learning [34] and Chain of Thought (CoT) [114] which have also been shown effective

are much less costly and thus have been increasingly adopted [27, 138]. Figure 1b illustrates how a typical few-shot learning mechanism is applied in the existing APR techniques. Firstly, the APR task-related examples such as the buggy and fixed code pair of the function `binarySearch` are incorporated into the prompt. Note that the example selection varies among different techniques, e.g., manually crafting examples like `binarySearch` [119] and choosing examples of historical bug fixes within the same buggy project [119, 121]. Next, the target buggy function to be fixed, e.g., `getNumbericalMean()` in the Math-2 bug [12], is also added to the prompt. At last, the resulting prompt is fed to the model to generate patches for the target buggy function. To summarize, the purpose of the few-shot learning mechanism is to enable the model to learn how to handle specific tasks through the examples. However, while multiple LLM-based APR techniques have already incorporated the few-shot learning mechanism [59, 119, 121], its impacts and characteristics remain unexplored.

2.2 Automated Program Repair

Automated Program Repair (APR) techniques [47, 54, 56, 59, 66, 82, 116, 118–120, 123, 131], designed to aid developers in fixing bugs by automatically generating patches, typically follow the Generate-and-Validate (G&V) paradigm [79]. In particular, an APR process refers to locating program faults, generating patches for the buggy locations, and validating such patches against a test suite to determine their plausibility (i.e., whether they could pass all tests). Eventually, these resulting plausible patches are manually reviewed to select the correct fix for the target fault. Notably, the trigger tests in the patch-validating test suite are manually created by developers. During the execution of trigger tests, the unit testing framework, e.g., JUnit [7], can be used to provide the corresponding error messages.

Among the APR techniques, the learning-based techniques [39, 56, 70, 82, 116, 120, 121, 131] that utilize deep learning techniques have recently achieved remarkable performance. Specifically, many such techniques widely adopt the Neural Machine Translation (NMT) [104] techniques which convert APR into a translation task to transform buggy code into correct code. They typically leverage the power of the NMT models through training on extensive datasets containing millions of buggy and fixed code pairs. However, such techniques are highly costly when building well-constructed datasets of buggy and patch code pairs [120] and specific context representation for the NMT models [56]. More recently, Large Language Models (LLMs) have become increasingly adopted in various downstream software tasks including APR. In particular, directly applying models like Codex can already outperform all previous APR techniques [119]. Meanwhile, multiple LLM-based APR techniques have been proposed to further enhance the repair performance. For instance, AlphaRepair [120] applies the pre-trained CodeBERT model with the "cloze-style" APR, i.e., removing the buggy code tokens and applying the LLM to generate the correct ones. Similar as AlphaRepair in adopting the cloze-style repair paradigm, Repilot [116] focuses on synthesizing compilable patches, utilizing the Language Server Protocol to prune infeasible tokens and proactively complete tokens as suggested by the LLM. FitRepair [118] combines LLMs with domain-specific fine-tuning and prompting strategies, fully automating the plastic surgery hypothesis, i.e., the code ingredients to fix bugs usually already exist within the same project. ChatRepair [121] utilizes the conversational repair mechanism based on GPT-3.5 and successfully fixes 162 out of 337 bugs in the Defects4J dataset with the assistance of the rich information from original bug-exposing tests. NTR [52] proposes a two-stage neural template repair framework, fine-tuning LLMs for enhanced template selection and patch generation while resolving template mismatches through probability-based prioritization, achieving 139 fixes in Defects4J. D4C [125] reformulates APR by aligning LLM inference with its training objective, shifting from infilling buggy hunks to full-function refinement, and leveraging artifacts like failed tests and documentation to repair 180

bugs in Defects4J. MORepair [127] leverages multi-objective fine-tuning to jointly optimize patch generation and natural language guidance, enabling LLMs to learn repair logic beyond syntactic transformations and repair. RepairLLaMA [100] combines parameter-efficient fine-tuning with program repair-specific code representations that incorporate fault localization signals, achieving 144 fixes in Defects4J. SWE-bench Lite [58] streamlines the assessment of automated program repair by providing a curated subset of 300 self-contained Python bug-fixing tasks, challenging agents to autonomously diagnose and rectify real-world defects within complex repository contexts. AutoCodeRover [137] utilizes a pipeline of specialized agents for iterative context retrieval and specification inference to guide patch generation. Building on this, SpecRover [98] introduces a reviewer agent to validate patches by referring to inferred specifications. Agentless [117] demonstrates the efficacy of a simpler, non-iterative workflow (e.g., localize, repair, and validate) that deliberately limits the LLM's autonomy. Fan et al. [47] conduct a study to investigate whether APR techniques can correct program errors generated by LLMs, particularly in complex tasks like the LeetCode contests. Another study [119] employs the few-shot learning mechanism and recognizes the ineffectiveness of simply feeding LLMs with only buggy functions as they are not pre-trained for APR. To address this, they create a prompt containing two pairs of buggy and fixed code examples: one manually crafted, and another from the same project or dataset. Then they include the buggy function to be fixed in this prompt, thus activating the function-level APR by providing such a prompt to LLMs. However, their claim that LLMs cannot be directly applied to the function-level APR and the effectiveness of employing the few-shot learning mechanism has not been fully investigated.

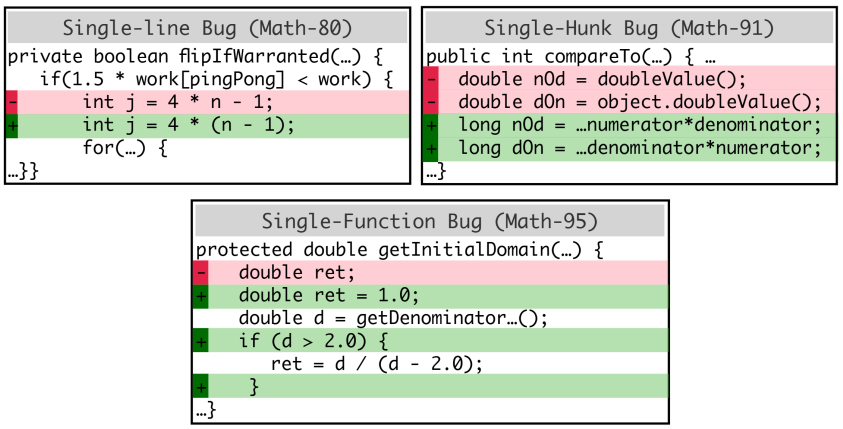

Fig. 2. Bug examples existing in a single line, hunk, or function

The existing LLM-based APR techniques mainly focus on repairing single-line or single-hunk bugs [116, 118, 120, 135], as illustrated in Figure 2. Specifically, the single-line bug Math-80 [13] is contained within a single line of the function `flipIfWarranted` where fixing this line alone can resolve the bug. Such a single-line bug can be fixed by APR techniques focusing on the line-level repair like AlphaRepair [120] given accurate fault locations. Meanwhile, the single-hunk bug Math-91 [14] is contained in a continuous section of code where two contiguous buggy lines are replaced in the fixed version. This kind of bugs can be fixed by the hunk-level APR techniques like Repilot [116] requiring multiple accurate statement-level fault locations. Note that single-line and single-hunk bugs can be considered as part of single-function bugs. On the other hand, the bugs existing in multiple discontinuous sections/lines within a function and requiring simultaneous

edits on them for a fix are also referred to as single-function bugs. For instance, fixing the Math-95 bug [15] requires editing three discontinuous lines simultaneously. It can be easily derived that fixing single-function bugs poses a greater challenge for APR techniques than fixing single-line/hunk bugs, especially when a specific statement-level fault location is not provided, e.g., the state-of-the-art APR techniques like ChatRepair and CodexRepair incur a performance decrease of 33% and 36.4% [121] in terms of the number of correctly fixed bugs for the function-level APR respectively.

As mentioned, single-function bugs actually include single-hunk and single-line bugs as subset, i.e., enabling a larger scope of repair tasks. Multi-function bugs can be viewed as a direct extension of single-function bugs, where the LLM takes multiple buggy functions as input and generates the corresponding fixed functions as output. In addition, multi-hunk APR differs from LLM-based function-level APR in repair granularity: function-level APR operates on entire functions, whereas multi-hunk APR takes multiple buggy hunks as the repair input. Specifically, Hercules [99] focuses on bugs that require similar edits across multiple locations and repairs them simultaneously, successfully fixing 46 bugs in Defects4J including 15 multi-hunk bugs. DEAR [71] tackles multi-location repair using a hierarchical tree-structured LSTM with cycle training to learn coordinated code transformations, repairing 47 bugs in Defects4J. ITER [133] proposes an iterative neural repair paradigm that progressively refines partial patches and constructs multi-location fixes through repeated fault localization, patch generation, and validation, successfully repairing 74 bugs in Defects4J with 9 uniquely fixed multi-location bugs. Moreover, locating function-level bugs tends to be cheaper than locating line-level or hunk-level bugs [29, 69, 80], thus making the function-level APR more practical. Therefore, we consider developing the function-level APR techniques rather promising and worthy being sufficiently investigated. Hence, our main study focuses on single-function bugs to isolate the effectiveness of function-level repair itself. In this setting, single-line and single-hunk bugs and more challenging multi-hunk bugs within one function are covered. Multi-function bugs are further considered as a direct extension of this setting in our approach SRepair.

3 Empirical Study

3.1 Research Questions

We investigate the following research questions for extensively studying the function-level APR along with the factors which can impact its effectiveness.

- **RQ1:** *How does the LLM-based function-level APR perform under the zero-shot and few-shot learning setups?* For this RQ, we attempt to investigate the performance of the LLM-based function-level APR under the default zero-shot learning and explore the performance impact from adopting few-shot learning.
- **RQ2:** *How do different auxiliary repair-relevant information affect the LLM-based function-level repair performance?* For this RQ, we attempt to study the impact from different auxiliary repair-relevant information including bug reports, trigger tests, etc., on the function-level repair performance.

3.2 Study Setup

3.2.1 Study Subjects. We utilize six distinct LLMs as our study subjects, encompassing the widely used state-of-the-art GPT-4 and GPT-3.5-Turbo [119, 121, 125], along with four advanced open-source LLMs including StarCoder2 (publicly released training data, thorough leakage detection, and outstanding performance among code LLMs) [81], Llama3.1 (1M+ downloads on Hugging Face within one month [23]), Magicoder 7B (over 2000 stars on GitHub [8]) and Qwen3 (newly

released and advanced open-source model [126]). Specifically, we adopt gpt-4-0613 [22], gpt-3.5-turbo-1106 [16], Llama-3.1-8B-Instruct [23] and StarCoder2-15B-Instruct-v0.1 [21] since they have conducted the instruction fine-tuning [97] and can better follow the APR prompt instruction. We also employ the MagicoderS-CL [8] model as the version of Magicoder, and adopt qwen3-32b [20] as the version of Qwen3. Due to the page limit, the LLM configuration details are presented in our *GitHub* page [1]. To address RQ2, we further include four recent, widely-studied LLM-based APR techniques: GiantRepair [68], ChatRepair [121], Repilot [116] and AlphaRepair [120] which adopt sample sizes equal to or larger than our study setting (200), e.g., GiantRepair samples 200 patches per bug and ChatRepair samples 500 patches per bug. Note that D4C [125] is excluded from this comparison because its sampling setting differs from our study's configuration of 200 samples per bug. Additionally, we exclude fine-tuning-based function-level APR techniques, as they operate under a distinct technical paradigm—relying on weight updates via domain-specific training—which is orthogonal to the prompting-based inference approaches investigated in our study. Specifically, following prior works [72, 120, 135], we utilize correct fix results gathered from previous papers [116, 118, 120, 121] for comparison.

Notably, our selection is designed to ensure representativeness and complementarity across three key dimensions: (1) adoption in SOTA APR works, (2) model accessibility (closed-source vs. open-source), and (3) model specialization (code-oriented vs. general-purpose). Specifically, first, our selection encompasses LLMs widely adopted by recent SOTA APR techniques. This includes the closed-source series (GPT-4/GPT-3.5-Turbo) utilized in ChatRepair [121], D4C [125], and RepairAgent [33], as well as open-source families (StarCoder and Llama) employed in GiantRepair [68], ensuring a fair comparison with existing baselines. Second, by covering both closed-source commercial APIs and open-source weights (incorporating Qwen3 and Magicoder to further enhance generalizability), we evaluate a comprehensive spectrum of accessibility. Third, the selected models complement each other by contrasting code-oriented models (StarCoder, Magicoder) with general-purpose reasoning models (GPT-4, Llama, Qwen3). Furthermore, while other LLM families exist, we excluded them from this study to maintain experimental feasibility regarding computational costs, and because our selected models capture a broad and representative cross-section of the LLMs frequently employed in recent APR research. In addition, the selected APR baselines include SOTA function-level APR tools across different methodologies and frameworks to ensure representativeness and comparison on function-level repair. For example, among our baselines, ChatRepair [121] employs a conversational mechanism in function-level repair, while GiantRepair [68] repairs buggy functions through patch skeleton extraction and context-aware patch instantiation.

3.2.2 Dataset. We construct our benchmark using both the versions 1.2 and 2.0 of the Defects4J dataset [60] which is the most widely used APR dataset [56, 83, 119] with a collection of a total of 835 real-world bugs extracted from open-source Java projects, comprising both buggy and fixed versions of the source code. Notably, we bound our study scope within 522 single-function bugs including 276 single-hunk ("SH") bugs and 158 single-line ("SL") bugs as shown in Table 1. It should be noted that our collected single-function bugs already include all the single-line and single-hunk bugs studied in previous works [116, 118–121]. Additionally, we also incorporate DebugBench [110], a benchmark designed to mitigate data leakage concerns which contains 4,253 bugs (in Java, C++, and Python) that were programmatically implanted into LeetCode solutions using GPT-4. Following prior baseline work [125], we align with their practice and focus on the 590 single-function logic bugs from DebugBench for our evaluation.

3.2.3 Evaluation Metrics and Correctness Validation. To assess the repair performance, we first follow the standard practice [49, 63, 88] to evaluate the plausible patches that pass all test cases.

Table 1. Statistics of the Dataset

Dataset	Project	# Bugs	SH Bugs	SL Bugs
Defects4J 1.2	Chart	16	12	9
	Closure	105	59	26
	Lang	42	23	13
	Math	74	35	23
	Mockito	24	12	7
	Time	16	6	3
Defects4J 2.0	Cli	28	13	6
	Codec	11	9	8
	Collections	1	1	1
	Compress	36	16	5
	Csv	12	7	4
	Gson	9	5	4
	JacksonCore	13	9	5
	JacksonDatabind	67	26	15
	JacksonXml	5	1	1
	Jsoup	53	38	27
	JxPath	10	4	1
Overall		522	276	158

In particular, those test cases include the trigger tests in the Defects4J dataset designed to expose bugs and relevant tests which can load the classes associated with the buggy functions.

Furthermore, to rigorously distinguish correct fixes from overfitting patches that merely pass the test suite [79, 93], we employ a manual validation process based on *semantic equivalence* to the developer's ground truth, following prior works [53, 77, 79, 82, 85, 119, 120]. Specifically, three authors independently cross-validate each plausible patch guided by a set of established equivalence rules (e.g., regarding control flow refactoring or variable aliasing) [128, 130], ensuring the patch truly resolves the bug. Any disagreements arising during this process are resolved through discussion to reach a final consensus [53, 77], thereby mitigating the potential subjectivity bias in manual analysis. Following prior works [38, 68, 92, 119], we randomly sample 200 patches for each setting (e.g., zero-shot learning) to derive the average number of correct fixes across different models. Moreover, to provide a more detailed breakdown of each model's performance in RQ2, we manually analyze the total number of correct patches generated by each model for our three key settings, as presented in Figure 9. Additionally, DebugBench is assessed using verified patches, following prior work [125], where a patch must pass all LeetCode tests.

3.3 Implementation

We obtain the four studied open-source models from Hugging Face [3] and access GPT-4 and GPT-3.5-Turbo through the APIs [4] provided by OpenAI. Our default setting for patch generation uses the nucleus sampling with top p = 0.9, temperature = 0.8 and 200 samples per bug following prior works [38, 92, 119]. Patches are generated on servers with 128-core 2.6GHz AMD EPYC™ ROME 7H12 CPU, 512 GiB RAM and eight NVIDIA A100 80GB GPUs, running Ubuntu 20.04.6 LTS.

3.3.1 APR input prompt setup. Following prior studies [92, 119], we set the prompt for the LLM utilized in the APR task, as illustrated in Figure 3 to enable the function-level APR. Specifically, we begin with a description of the APR task as '`Provide a fix for the buggy function`'. Next, we

```
// Provide a fix for the buggy function
{Buggy code and fixed code pair examples} OR
{Auxiliary repair-relevant Information}
// Buggy Function
public double getNumericalMean() {
        return (double) (getSampleSize() *
 getNumberOfSuccesses()) / (double) getPopulationSize();}
// Fixed Function
```

Fig. 3. The input prompt for the function-level APR of the Math-2 bug

Table 2. K-shot Settings in RQ1

K	Example Type	Abbreviation
0	N.A.	K_0*(Basic)*
1	Project Example	K_1*(PE)*
1	Binary Search	$K_1(CE_{BNS})$
1	Bubble Sort	$K_1(CE_{BBS})$
1	Fibonacci	$K_1(CE_{FIB})$
2	Project Example & Project Example	K_2*(PE, PE)*
2	Binary Search & Project Example	$K_2(CE_{BNS}, PE)$
2	Bubble Sort & Project Example	$K_2(CE_{BBS}, PE)$
2	Binary Search & Bubble Sort	$K_2(CE_{BNS}, CE_{BBS})$
3	Project Example & Project Example & Project Example	K_3*(PE, PE, PE)*
3	Binary Search & Project Example & Project Example	$K_3(CE_{BNS}, PE, PE)$
3	Binary Search & Bubble Sort & Project Example	$K_3(CE_{BNS}, CE_{BBS}, PE)$
4	Project Example & Project Example & Project Example & Project Example	K_4*(PE, PE, PE, PE)*
4	Binary Search & Bubble Sort & Fibonacci & Project Example	$K_4(CE_{BNS}, CE_{BBS}, CE_{FIB}, PE)$

incorporate the buggy code and fixed code pair examples from the few-shot learning mechanism or the auxiliary repair-relevant information into the prompt. Subsequently, we use '`Buggy Function`' in conjunction with the buggy code, e.g., the Math-2 buggy function `getNumericalMean` [12] in Figure 3, to prompt LLMs with the buggy function to be fixed. Finally, we apply '`Fixed Function`' to guide the LLM in generating a patched function. Notably, we employ the zero-shot learning approach as the default baseline in our study, i.e., adopting no auxiliary repair-relevant information or buggy-fixed code pair examples for prompting.

3.3.2 K-shot learning setups. Table 2 presents our k-shot learning setups. Specifically, we set the zero-shot learning approach, i.e., adopting no pairs of buggy code and fixed code examples (K=0), as our basic setup denoted as K_0*(Basic)*. Moreover, we follow prior work [119] to form our buggy and fixed code pair examples via manually crafted examples (*CE*), including `binarySearch` (CE_{BNS}), `bubbleSort` (CE_{BBS}) and `Fibonacci` (CE_{FIB}), as shown in Figure 4, as well as chosen historical bug fix examples within the same buggy project (*PE*). Figure 1b illustrates how these examples are incorporated into the few-shot learning APR. We thus form our k-shot learning setup variants as K_1*(PE)* with only one chosen historical bug fix example from the same project, $K_1(CE_{BNS})$, $K_1(CE_{BBS})$, $K_1(CE_{FIB})$ with one manually crafted example `binarySearch`, `bubbleSort` and `Fibonacci` respectively, $K_2(CE_{BNS}, PE)$, $K_2(CE_{BBS}, PE)$ with one manually crafted example and one chosen historical bug fix example from the same project, $K_2(CE_{BNS}, CE_{BBS})$ with two manually crafted examples `binarySearch` and `bubbleSort`. We also form K_2*(PE, PE)* with two chosen historical bug fix examples from the same project following the implementation of the prior work for selecting multiple examples [119], utilizing their official implementation [17]. To further investigate the effect of incorporating more examples, we extend our setups to larger K

Fig. 4. Manually crafted code examples

Binary Search

Buggy Function

```
int binarySearch(int[] a, int x){
  ...
  while (l <= r) {
    ...
    if (a[m] < x) l = m;
    else r = m - 1;
  ...
}
```

Fixed Function

```
int binarySearch(int[] a, int x){
  ...
  while (l <= r) {
    ...
    if (a[m] < x) l = m + 1;
    else r = m - 1;
  ...
}
```

Bubble Sort

Buggy Function

```
void bubbleSort(int[] a){
 for (int i=0; i<a.length; i++)
  for (int j=0; j<a.length; j++)
   if (a[j] > a[j+1])
    swap(a, j, j+1);
}
```

Fixed Function

```
void bubbleSort(int[] a){
 for (int i=0; i<a.length-1; i++)
  for (int j=0; j<a.length-1; j++)
   if (a[j] > a[j+1])
    swap(a, j, j+1);
}
```

Fibonacci

Buggy Function

```
int fib(int n){
  if (n <= 1)
      return n;
  return fib(n-1) - fib(n-2);
}
```

Fixed Function

```
int fib(int n){
  if (n <= 1)
      return n;
  return fib(n-1) + fib(n-2);
}
```

values, including K=3 (i.e., $K_3(PE, PE, PE)$, $K_3(CE_{BNS}, PE, PE)$, and $K_3(CE_{BNS}, CE_{BBS}, PE)$) and K=4 (i.e., $K_4(PE, PE, PE, PE)$ and $K_4(CE_{BNS}, CE_{BBS}, CE_{FIB}, PE)$). Notably, since DebugBench does not contain historical bug fix examples (*PE*), only the K1–K3 crafted example settings are applied on DebugBench dataset.

(a) Project-specific Information

Trigger Test

```
public void testMath1021() {
…
for (…) {
  Assert.assertTrue(0 <= sample);
  Assert.assertTrue(sample <= n);
}}
```

Error Message

AssertionFailedError: sample=-50 at HypergeometricDistributionTest.java:297

Comment

/* For population size {@code N}, number of successes {@code m}, and sample*size {@code n}, the mean is {@code n*m/N}. */

(b) Bug Report-relevant Information

Issue Title

Issue 1021: HypergeometricDistribution.sample suffers from integer overflow

Issue Description

"Hi, I have an application which broke when ported from commons math 2.2 to 3.2. It looks like the HypergeometricDistribution.sample() method doesn't work as well as it used to with large integer values, the example code below should return a sample between 0 and 50, but usually returns -50…"

Fig. 5. The bug report and project-specific information in the Math-2 bug

3.3.3 Collecting auxiliary repair-relevant information. In this study, we refer to the auxiliary repair-relevant information as the project-specific information (i.e., trigger tests, error messages, and comments) and bug report from the target buggy project following prior works [66, 121, 131, 133], as the Math-2 bug shown in Figure 5. Specifically, we automatically extract the project-specific information from the buggy project, as the Math-2 bug in Figure 5a following the prior works [92, 121, 131]. More specifically, we first build all the buggy projects and automatically extract all the trigger tests and buggy function comments. Then, for each bug, we execute the trigger tests

Table 3. APR results under different few-shot learning setups

Settings	Plausible Fixes						*Avg. # Correct / Plausible Fixes*	*Coef. of Variation*
	GPT-4	Qwen3	GPT-3.5-Turbo	MAGICODER	StarCoder2	Llama3.1		
K_0*(Basic)*	**205**	**197**	175	**199**	**187**	162	**135 / 188**	0.0872
K_1*(PE)*	200	187	174	157	155	**167**	120 / 173	0.1013
$K_1(CE_{BNS})$	191	180	138	112	161	150	109 / 155	0.1848
$K_1(CE_{BBS})$	184	181	148	114	157	152	110 / 156	0.1635
$K_1(CE_{FIB})$	187	184	145	117	155	158	111 / 158	0.1648
K_2*(PE, PE)*	194	185	**187**	121	147	143	115 / 163	0.1831
$K_2(CE_{BNS}$, *PE)*	181	173	166	100	146	149	107 / 153	0.1906
$K_2(CE_{BBS}$, *PE)*	183	175	164	102	142	147	106 / 152	0.1918
$K_2(CE_{BNS}, CE_{BBS})$	179	171	159	98	143	145	104 / 149	0.1928
K_3*(PE, PE, PE)*	195	179	183	126	151	148	116 / 164	0.1597
$K_3(CE_{BNS}$, *PE, PE)*	189	171	180	120	145	138	110 / 157	0.1713
$K_3(CE_{BNS}, CE_{BBS}$, *PE)*	185	176	167	112	149	143	108 / 155	0.1706
K_4*(PE, PE, PE, PE)*	192	175	172	115	136	132	107 / 154	0.1961
$K_4(CE_{BNS}, CE_{BBS}, CE_{FIB}$, *PE)*	174	163	144	104	126	123	97 / 139	0.1897

and capture the error messages generated by the unit test framework, such as JUnit [7]. Notably, among all 522 single-function bugs in the Defects4J dataset, only 10 missing reports and 2 missing comments. We then leave such auxiliary repair-relevant information empty in our study.

The bug reports are collected from the official issue links of the Defects4J [6] repository. More specifically, following prior works [36, 136], we divide a bug report into two parts, as illustrated in Figure 5b. One is the issue title with averagely around 12 tokens to summarize the type and the cause of the bug (e.g., *"Issue 1021: ... suffers from integer overflow"*). The other is the issue description with averagely 234 tokens which provides detailed conditions, error messages, and reproduction steps, etc. For instance, the issue description in the Math-2 bug report provides a detailed description of the buggy method `HypergeometricDistribution.sample()` with the trigger conditions, i.e., *"with large integer values"*.

For evaluating the impact from the auxiliary repair-relevant information, we form seven different setups. Specifically, for the project-specific information, we form four setups: *PI(TT)* with the trigger test only, *PI(EM)* with the error message only, *PI(BC)* with the buggy function comment only, and *PI(ALL)* with all such information. For bug report-relevant information, we form three setups: *BR(IT)* with the issue title only, *BR(ID)* with the issue description only, and *BR(ALL)* with the whole bug report. Notably, since DebugBench does not contain issue title and issue description, the three settings (*BR(IT)*, *BR(ID)* and *BR(ALL)*) are not applied to the DebugBench dataset.

3.4 Result Analysis

3.4.1 RQ1: the function-level repair performance. Table 3 presents the function-level APR results in terms of the number of plausible and correct fixes. In general, we observe that K_0*(Basic)* achieves the overall optimal plausible fix and correct fix results, i.e., 188 average plausible fixes and 135 average correct fixes out of our collected 522 single-function bugs, outperforming all the rest setups by at least 8.7% (188 vs. 173) and 12.5% (135 vs. 120). In addition, K_0*(Basic)* shows the lowest cross-LLM coefficient of variation in plausible fixes (0.0872) among all settings, indicating more consistent plausible-fix performance across different LLMs. Moreover, Table 4 presents the APR performance on DebugBench, which shows similar results as K_0*(Basic)* outperforming all the rest setups by at least 4.5% (438 vs. 419) with GPT-4 and 5.2% (403 vs. 383) with Qwen3. Such a result indicates that LLMs themselves (with zero-shot learning) are already powerful function-level APR techniques.

Finding 1: LLMs with zero-shot learning are already powerful function-level APR techniques.

Interestingly, we can further observe that applying the few-shot learning mechanism leads to quite disparate plausible fix results across LLMs. For instance, compared with K_0*(Basic)*, while

Table 4. APR results under different few-shot learning setups on DebugBench dataset

Model	K_0(Basic)	$K_1(CE_{BNS})$	$K_2(CE_{BNS}, CE_{BBS})$	$K_3(CE_{BNS}, CE_{BBS}, CE_{FIB})$
GPT-4	**438**	407	419	395
Qwen3	**403**	383	368	361

GPT-3.5-Turbo shows a 6.9% (187 vs. 175) improvement in $K_2(PE, PE)$, MAGICODER shows a 50.8% (98 vs. 199) decline in $K_2(CE_{BNS}, CE_{BBS})$ setting in terms of the number of plausible fixes. Moreover, in terms of aggregate performance, the choice of few-shot examples results in limited variations in the average number of plausible fixes (e.g., ranging from 155 to 173 for K_1), yet individual model sensitivity varies significantly. For instance, MAGICODER suffers a drastic 28.7% drop (157 vs. 112) when switching from project-specific examples ($K_1(PE)$) to constructed ones ($K_1(CE_{BNS})$), whereas other models remain relatively stable. Furthermore, extending the context to K=4 yields no significant improvement over the zero-shot baseline, even for the most capable model GPT-4 (192 vs. 205).

Finding 2: Applying the few-shot learning mechanism in the function-level APR leads to disparate plausible fix results across LLMs.

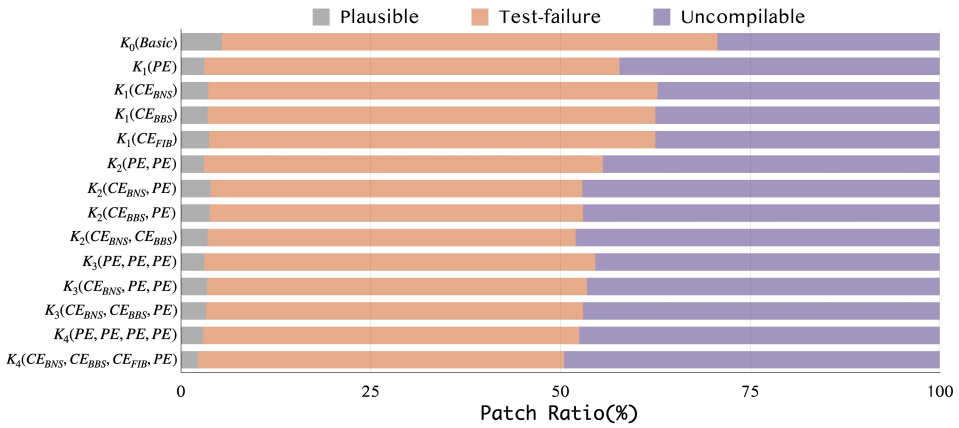

Fig. 6. Patch status averaged across all models under different few-shot learning setups

Furthermore, we present the distribution of plausible, test-failure, and uncompilable patches given the identical total number of generated patches across different LLMs under all setups in Figure 6. Specifically, uncompilable patches are those that fail the compilation process, and the test-failure patches refer to those successfully compiled but fail one or more tests in our Defects4J dataset. Interestingly, we can find that K_0(Basic) achieves the best plausible patch rate 5.4% and the lowest uncompilable patch rate 29.3% among all the k-shot setups, while applying the few-shot learning generates more uncompilable patches, i.e., ranging from 27% to 68.9% more than K_0(Basic).

Finding 3: Applying the few-shot learning mechanism may generate more uncompilable patches than the zero-shot learning mechanism.

Moreover, to further investigate the underlying impact of few-shot learning setups on LLM patch generation, we employ the CodeBLEU score [96] to proxy the magnitude of code modifications by computing the similarity between each generated patch and its corresponding buggy function, as shown in Figure 7. We find that K_0*(Basic)* exhibits both the lowest median CodeBLEU score (0.820) and the largest dispersion of CodeBLEU scores, with an interquartile range (IQR [19]) of 0.18, indicating that in the zero-shot setting the model tends to produce more diverse patches and to apply larger and more varied modifications to the buggy code. In contrast, the few-shot configurations yield higher and more concentrated CodeBLEU distributions, with median scores ranging from 0.849 to 0.937 and IQR values between 0.10 and 0.13, suggesting that the provided examples anchor the model toward more conservative edits that remain closer to the original buggy code. As a result, the diverse zero-shot transformations may increase the chance of a correct fix by covering a wider range of repair attempts.

Finding 4: Zero-shot learning produces more diverse and larger code modifications, while the few-shot configurations tend to generate more conservative patches that stay closer to the original buggy code.

3.4.2 RQ2: performance impact from the auxiliary repair-relevant information. Noticing that since applying zero-shot learning achieves the optimal repair performance among all the k-shot learning techniques as mentioned, we also adopt zero-shot learning in our auxiliary repair-relevant information evaluations and K_0*(Basic)* as a baseline for a fair performance comparison. Table 5 presents the APR results under different auxiliary repair-relevant information setups. We observe that using project-specific information in prompts significantly enhances the repair performance of all models,

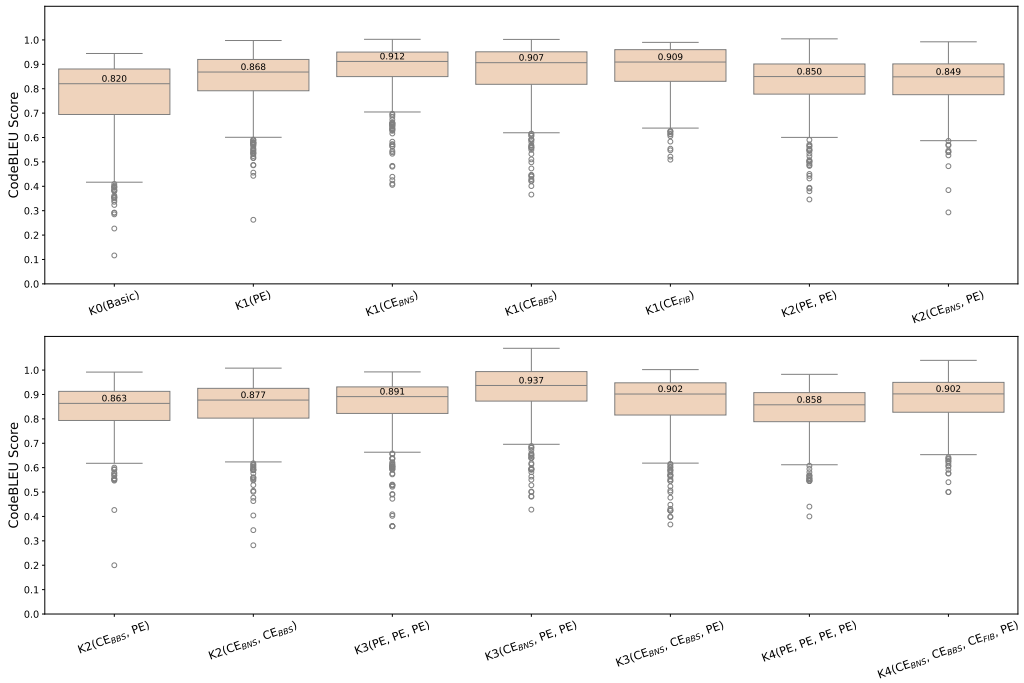

Fig. 7. Distribution of CodeBLEU scores across different few-shot learning setups

Table 5. APR results in different auxiliary repair-relevant information settings

Sources	Settings	Plausible Fixes						*Avg. #Correct / Plausible Fixes*
		GPT-4	Qwen3	GPT-3.5-Turbo	MAGICODER	StarCoder2	Llama3.1	
N.A.	*K_0(Basic)*	205	197	175	199	187	162	135 / 188
Project-specific Information	*PI(BC)*	209	202	185	194	197	186	149 / 196
	PI(EM)	241	236	226	240	246	222	187 / 235
	PI(TT)	260	253	247	235	223	217	187 / 239
	PI(ALL)	**307**	**289**	**273**	**254**	**270**	**242**	**214 / 273**
Bug Report Information	*BR(IT)*	271	262	233	251	243	225	186 / 248
	BR(ID)	307	295	**286**	**279**	269	252	214 / 281
	BR(ALL)	**312**	**303**	285	260	**277**	**264**	**219 / 284**

Table 6. APR results with different auxiliary repair-relevant information on DebugBench datasets

Model	*K_0(Basic)*	*PI(BC)*	*PI(EM)*	*PI(TT)*	*PI(ALL)*
GPT-4	438	461	484	507	**561**
Qwen3	413	425	449	460	**508**

i.e., the average number of correct fixes/plausible fixes rises from 135/188 in *K_0(Basic)* to 149/196 in *PI(BC)*, 187/235 in *PI(EM)*, 187/239 in *PI(TT)*. Notably, *PI(ALL)* achieves an optimal average of 273 plausible fixes and 214 correct fixes, indicating the potential of leveraging as much auxiliary repair-relevant information as possible for enhancing the function-level repair performance. Similar trends are observed on DebugBench, as shown in Table 6. Specifically, with GPT-4, all auxiliary information setups outperform the *K_0(Basic)* setting (438 fixes). While *PI(BC)*, *PI(EM)*, and *PI(TT)* achieve 461, 484, and 507 fixes respectively, *PI(ALL)* notably yields the largest improvement, with a 28.1% gain over *K_0(Basic)* (561 vs. 438). Moreover, we compare the number of correct fixes in the *PI(ALL)* setting with the studied LLM-based APR techniques, as shown in Figure 8. Interestingly, we find that simply adopting *PI(ALL)* already surpasses the studied APR techniques by at least 25.1% (214 vs. 171). We find that this improvement in the number of correct fixes is consistent across all models (Figure 9), e.g., GPT-4's correct fixes increase by 64.9% (244 vs. 148) and Qwen3's by 62.2% (232 vs. 143) when using *PI(ALL)* compared to *K_0(Basic)*.

Finding 5: Directly adopting trigger tests, error messages, and comments from buggy projects can significantly advance the function-level repair performance and even surpass multiple recent APR techniques.

We also attempt to investigate the performance impact from the bug report information on the LLM-based function-level APR. Table 5 shows that using bug report information in prompts leads

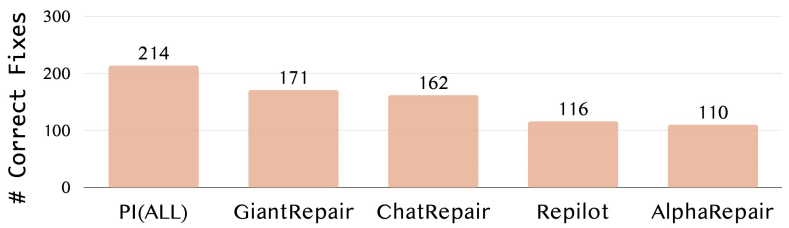

Fig. 8. Correct fix results of *PI(ALL)* setting and APR techniques

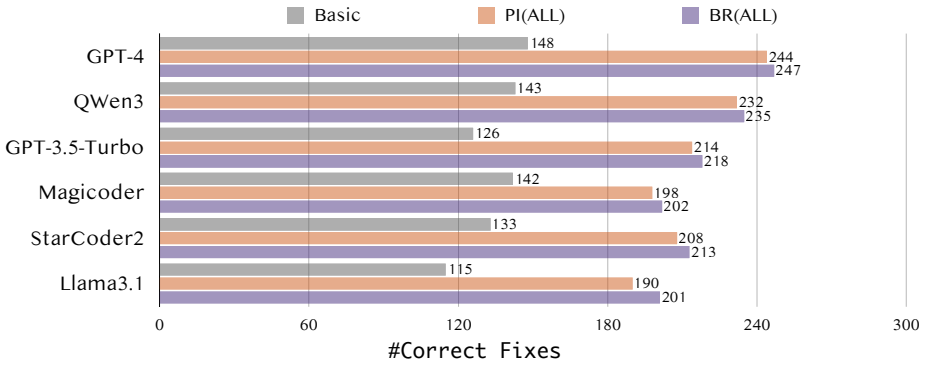

Fig. 9. Correct fix results across different models in different auxiliary repair-relevant information

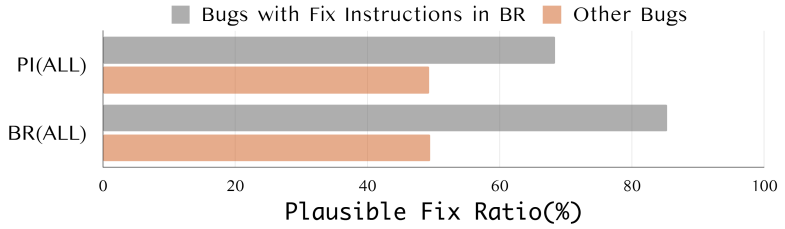

Fig. 10. Plausible fix ratio comparison for bugs with and without explicit fix instructions under *PI(ALL)* and *BR(ALL)* setups.

to an increase for all models, i.e., the number of average correct fixes/plausible fixes increases from 135/188 in K_0*(Basic)* to 186/248 in *BR(IT)*, 214/281 in *BR(ID)*, and 219/284 in *BR(ALL)*. However, upon manually analyzing bug reports, we identify 38 bugs that include explicit patch or fix instructions, such as "to fix, check if the parser's headerMap is null before trying to create the returned map" in the Csv-4's bug report [10]. We then divide our Defects4J dataset into two sets—one consisting of the 38 bugs with fix instructions and another consisting of the remaining 484 bugs—and calculate the average plausible fix ratio, i.e., the ratio of plausible fix bugs to the entire bug set, for all studied models, as shown in Figure 10. Interestingly, compared to the *PI(ALL)* setting, *BR(ALL)* primarily achieves better plausible fix ratio on the 38 bugs that contain fix instructions in their bug reports (85.3% vs. 68.4%), while showing similar repair performance on the rest 484 bugs (51.9% vs. 51.1%).

> *Finding 6: Incorporating bug report information improves function-level APR performance, yet adopting project-specific information achieves comparable repair effectiveness without relying on bug reports.*

To investigate how different auxiliary information sources influence the LLM repair process, Figure 11 presents our manual analysis of each information type across three repair-relevant dimensions: (i) clarifying the Intended Functionality, (ii) revealing the Root Cause, and (iii) providing a Fix Hint. Specifically, all sources contribute to clarifying the Intended Functionality (from 229 to 394 bugs), with buggy function comments providing the strongest support. In contrast, comments rarely help reveal the Root Cause (8 bugs), while other sources contribute significantly (245 to 362 bugs). Explicit Fix Hints are generally rare: issue descriptions provide them most frequently (87

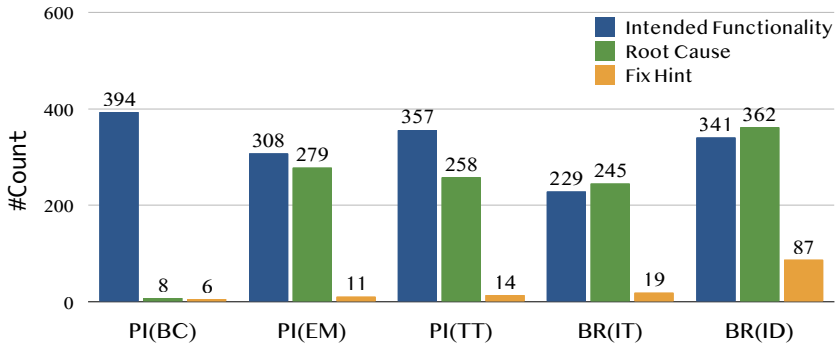

Fig. 11. Distribution of repair-relevant dimensions across auxiliary information sources.

Fig. 12. Illustration of auxiliary repair-relevant information and the resulting fix for JacksonDatabind-108

(a) Auxiliary Information

Function Comment

```
/**
 * Returns JsonNode, or null if parser
 * has no remaining content.
 */
```

Insight: Specifies expected EOF behavior of readTree(JsonParser).

Trigger Test

```
_assertNullTree(reader.readTree(p));
```

Insight: Shows the concrete scenario to trigger the EOF-handling error.

Error Message

```
Should get `null`, instead got:
MissingNode at
EmptyContentAsTreeTest._assertNullTree
```

Insight: Actual behavior contradicts the documented contract.

(b) Fix

Core Patch

```
+ if (t == JsonToken.VALUE_NULL) {
+   resultNode = _config
+       .getNodeFactory()
+       .nullNode();
+ }
```

Root Cause

EOF was handled incorrectly: the parser with no remaining input produced a `MissingNode` instead of the expected null due to the usage of existing `_bindAsTree()`

Fix Idea

Explicitly detect the `VALUE_NULL` token and construct a proper `nullNode()` through the node factory. This ensures EOF behavior aligned with both the function comment specification and the trigger test.

bugs), while others offer them only occasionally (6 to 19 bugs). This indicates that most auxiliary information aids repair not by prescribing concrete patches but by constraining the semantic and diagnostic space for the fix. These observations highlight the complementary nature of the sources, explaining their synergistic effect on enhancing the LLM's repair capabilities.

This synergistic effect is well-illustrated by the JacksonDatabind-108 bug. In this case, multiple information sources converge to make an otherwise infeasible repair possible under the K_0*(Basic)* setting, where the model fails to infer the intended EOF semantics of `readTree(JsonParser)` and

repeatedly retains the incorrect behavior of returning a `MissingNode` instead of `null`. Once auxiliary information is incorporated, however, the repair becomes feasible. As illustrated in Figure 12, the function comment explicitly states that the method must return `null` when no tokens remain, the trigger test shows the concrete code to trigger the bug with `_assertNullTree(reader.readTree(p))`, and the error message exposes the deviation from the expected behavior ("Should get `null`, instead got: `MissingNode`"). Together, these sources clarify the intended specification and pinpoint the mismatch, enabling the model to identify the misuse of `_bindAsTree()` as the root cause and generate the correct patch that handles `VALUE_NULL`. This example demonstrates how auxiliary information transforms implicit design intent into explicit, actionable signals, ultimately allowing LLMs to produce correct repairs that are otherwise unattainable.

Finding 7: The effectiveness of auxiliary repair-relevant information stems not from providing explicit fix hints, but mainly from a synergistic effect where complementary sources collectively constrain the problem space, enabling the LLM to infer the root cause and solution.

4 Approach

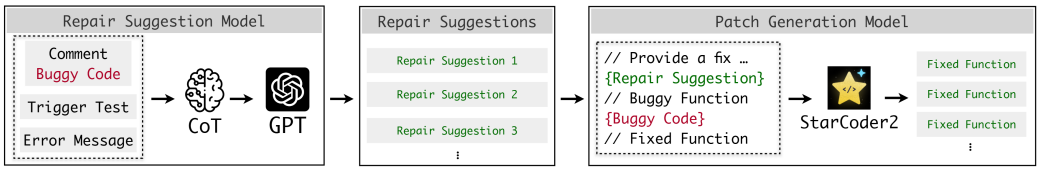

Fig. 13. The SRepair framework

By far, we have demonstrated the power of adopting the auxiliary repair-relevant information in the function-level LLM-based APR, i.e., including such information in the repair prompt along with the buggy function under zero-shot learning. In this section, to further leverage the potential of the auxiliary repair-relevant information, we construct a novel function-level APR technique SRepair (referring to **S**uggestion **Repair**), which adopts a dual-LLM framework and practical project-specific information for advancing the repair performance. Specifically, the design of SRepair is motivated by our empirical findings. Given that zero-shot prompting demonstrates superior and stable performance, SRepair adopts a zero-shot configuration. To further leverage auxiliary information and its synergistic effects, SRepair utilizes a dual-LLM framework to perform structured zero-shot CoT reasoning over project-specific information. In this way, SRepair can progressively analyze multi-source auxiliary information, reason about the root cause of the bug, and produce natural-language repair suggestions before final patch generation. By decomposing the repair process into two coordinated stages, SRepair supports the synthesis of project-specific information prior to final patch generation.

4.1 SRepair Framework

Our Dual-LLM framework is shown in Figure 13 where SRepair first adopts a repair suggestion model which utilizes the learning power of LLM by comprehensively analyzing the auxiliary repair-relevant information via the Chain of Thought (CoT) technique [114]. Then it provides repair suggestions in natural language. Next, SRepair adopts a patch generation model which exhibits its code generation capabilities by generating the entire patched function following the repair suggestions. More specifically, we enable the CoT technique by prompting the LLM to first analyze

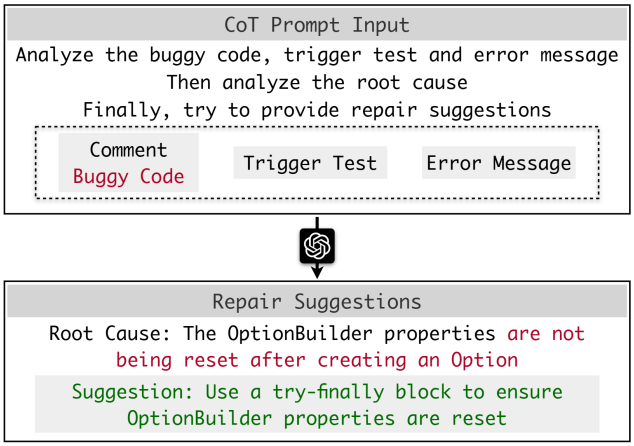

Fig. 14. A Chain of Thought example of the Cli-26 bug

the buggy function and project-specific information, then identify the root cause of the bug, and finally generate repair suggestions in natural language. For instance, as shown in Figure 14, the repair suggestion model first identifies the root cause of the Cli-26 bug [9]: '*are not being reset after creating an Option*', and then generates the correct repair suggestion, '*use a try-finally block*'. Finally, such a suggestion is fed to the patch generation model for generating the patched functions.

4.2 Evaluation

4.2.1 Dataset and Evaluation Metrics. We use four widely studied repair benchmarks: Defects4J [60], QuixBugs [73], DebugBench [110] and SWE-Bench [58]. Specifically, to extensively leverage SRepair's ability in the function-level APR, we include all function-level bugs from Defects4J 1.2 and 2.0, thereby forming a dataset that comprises 522 single-function (SF) bugs and an additional 143 multi-function (MF) bugs, i.e., the bugs existing in multiple functions and requiring simultaneous edits on them for a fix, as shown in Table 7. Additionally, we also evaluate on the QuixBugs dataset which is made up of 40 function-level buggy and fixed versions of classic programming problems in both Python and Java. We also incorporate DebugBench and focus on the 590 single-function bugs, following our study section. Furthermore, we additionally include SWE-bench [58], a real-world and large-scale benchmark derived from high-star Python GitHub project issues. In particular, we use the commonly adopted SWE-bench Lite version [2], which includes 300 high-quality bug-fixing instances.

Following our study section, we utilize plausible patches and correct patches to reflect the repair performance on Defects4J and QuixBugs, and verified patches metric on DebugBench. For SWE-bench, we follow its standard metric [98, 117, 137] and report both the number and the rate of resolved instances, where an instance is considered resolved if the generated patch passes all official tests.

4.2.2 Implementation. In the SRepair implementation, GPT-4 acts as the repair suggestion model due to its superior analytical, coding, and natural language generation abilities [26, 35, 42, 46, 140], especially in *PI(ALL)*. StarCoder2 is adopted as the patch generation model due to its cost-effectiveness and competent code generation ability [37, 43, 61, 81]. Specifically, the prompt templates used for suggestion generation and patch generation are shown in Figure 15. Following the recent SOTA APR technique D4C [125], which enables an efficient patch sampling process for practice, we set the

Table 7. Statistics of SRepair Dataset

Dataset	Project	# Bugs	SF Bugs	MF Bugs
Defects4J 1.2	Chart	25	16	9
	Closure	140	105	35
	Lang	56	42	14
	Math	102	74	28
	Mockito	30	24	6
	Time	22	16	6
Defects4J 2.0	Cli	30	28	2
	Codec	13	11	2
	Collections	2	1	1
	Compress	40	36	4
	Csv	13	12	1
	Gson	12	9	3
	JacksonCore	18	13	5
	JacksonDatabind	85	67	18
	JacksonXml	5	5	0
	Jsoup	58	53	5
	JxPath	14	10	4
Overall		665	522	143

Fig. 15. The prompt templates of SRepair

(a) Suggestion Generation Prompt

Please analyze the buggy code, trigger test, and error message. Then analyse the root cause. Finally, try to provide repair suggestions. Your output format should be "Root Cause: {content}" and "Suggestion {id}: {title}\n{description}"

```
1. Buggy Function: {comment} {buggy_function}
2. Trigger Test: {trigger_src}
3. Error Message: {err_msg}
```

(b) Patch Generation Prompt

Please Generate a fixed version of the function based on the provided root cause, repair suggestion, and buggy function.

```
// Provide a fix for the buggy function.
Root Cause: {root_cause}
Suggestion: {suggestion}
// Buggy Function
{buggy_function_code}
// Fixed Function
```

number of patches sampled by SRepair per bug to 10. In contrast, most SOTA baselines sample more

Table 8. Single-function correct fix result of SRepair

Datasets	Project	SRepair	D4C	GiantRepair	RepairAgent	ChatRepair	Repilot	AlphaRepair	†Others
D4J 1.2	Chart	13	5	8	11	**15**	6	9	14
	Closure	31	28	34	27	**37**	22	23	33
	Lang	22	**25**	14	17	21	15	13	19
	Math	**36**	18	26	29	32	21	21	33
	Mockito	**10**	6	6	6	6	0	5	5
	Time	4	4	1	2	3	2	3	**5**
D4J 2.0	Cli	**15**	13	7	8	5	6	5	8
	Codec	**9**	3	8	9	8	6	6	7
	Collections	**1**	0	0	1	0	1	0	0
	Compress	**16**	11	12	10	2	3	1	4
	Csv	**9**	7	6	6	3	3	1	4
	Gson	2	3	**6**	3	3	1	2	1
	JacksonCore	6	6	**8**	5	3	3	3	2
	JacksonDatabind	21	**24**	15	11	9	8	8	12
	JacksonXml	**1**	1	1	1	1	0	0	1
	Jsoup	**28**	25	18	18	14	18	9	11
	JxPath	3	1	1	0	0	1	1	**4**
D4J 1.2 Total		**116**	86	89	92	114	66	74	109
D4J 2.0 Total		**111**	94	82	72	48	50	36	54
Overall		**227**	180	171	164	162	116	110	163
D4J 1.2 Total$_{FL}$		**94**	80	64	66	81	47	50	-
D4J 2.0 Total$_{FL}$		**72**	61	45	46	29	24	20	-
Overall$_{FL}$		**166**	141	109	112	110	71	70	-
***p*-value**		-	3.72e-4	9.47e-06	7.58e-06	2.21e-07	2.56e-17	1.39e-18	-

†All studied traditional and learning-based APR baselines are grouped as "Others". The column shows their combined results.
FL **indicates end-to-end APR results with fault localization.**

than 200 patches per bug. It should be noted that the design of SRepair could be utilized for repairing multi-function bugs, and its repair process is a straightforward extension of the single-function bug fixing process. For the repair suggestion generation, SRepair analyzes all buggy functions and their corresponding comments, triggered tests, and error messages, allowing the LLM to generate repair suggestions for each buggy function. It then provides the patch generation model with all buggy functions and their repair suggestions in a single prompt, requiring the LLM to generate patches for all buggy functions simultaneously. Additionally, we extend the SRepair implementation to the SWE-Bench benchmark. Following prior SWE-bench works [98, 117, 137], we perform fault localization and test reproduction accordingly. Since the official trigger tests of SWE-bench are unavailable [117], we use the reproduced tests as the trigger-test information in SRepair's suggestion and patch generation processes. For better comparison with SOTA baselines [98, 117] on SWE-Bench, we adopt the same model GPT-4o (gpt-4o-2024-05-13) [89] for SRepair. Notably, the cost of SRepair is calculated by summing the total input and output token costs (including all generated suggestions and patches) across the entire experiment, based on the official API price of closed-source models used (e.g., gpt-4-0613 [22], $30.00/$60.00 per million input/output tokens; gpt-4o-2024-05-13 [89], $5.00/$15.00 per million input/output tokens; gpt-3.5-turbo, $3.00/$6.00 per million input/output tokens), and the referenced token cost of open-source models ($0.26, $0.22 and $0.17 per million tokens for StarCoder2 [28], Llama3.1 [87] and Magicoder [28], respectively; $0.16/$0.64 per million input/output tokens for Qwen3 [94]). For the rest setups, we follow our study section.

4.2.3 Compared Techniques. We compare SRepair against state-of-the-art traditional, learning-based and LLM-based APR baselines. We select 8 traditional and learning-based APR baselines:

TBar [76], SelfAPR [131], RewardRepair [132], Recoder [141], CURE [56], CoCoNuT [82], DLFix [70], and SequenceR [41]. Moreover, we include six recent SOTA LLM-based APR techniques: AlphaRepair [120], Repilot [116], ChatRepair [121], RepairAgent [33], GiantRepair [68] and D4C [125]. Furthermore, we adopt 3 SOTA APR techniques [71, 99, 133] as multi-function repairers for comparing multi-function repair performance. Additionally, we adopt the function-level fault localization technique DepGraph [95] to assess the practical applicability and generalizability of SRepair's end-to-end repair performance against the studied baselines. Specifically, following prior works [72, 72, 120, 135], we utilize correct fix results gathered from previous papers [68, 71, 99, 112, 116, 118, 121, 125, 131, 133, 135] for comparison and we re-implement or re-run the baselines' original setups to obtain missing end-to-end fixing results. In particular, for baselines that did not report end-to-end results (i.e., FitRepair, ChatRepair), we obtained their replication packages and re-ran them with fault localization under their original settings. For D4C and GiantRepair, we adhered to their original settings on Defects4J V1 and re-ran their fault localization on Defects4J V2. Additionally, for the evaluation on SWE-bench, we include three SOTA LLM-based repair baselines: Agentless [117], AutoCodeRover [137], and SpecRover [98], and obtain their results from the original papers for comparison.

4.2.4 Result Analysis. Table 8 presents the APR results for single-function bugs in the Defects4J dataset. Surprisingly, we find that SRepair outperforms all previous LLM-based APR techniques by at least 26.1% (227 vs. 180). We further evaluate the statistical significance of these improvements using McNemar's test [84], which is performed on paired bug-level repair outcomes over all 522 single-function bugs. The results show that the performance gains of SRepair over the compared baselines are statistically significant (p-value < 0.001). Specifically, we can observe that 43.5% of bugs (227/522) in Defects4J can be correctly fixed by SRepair, with the first correct patch being the 1.84th plausible patch generated per bug on average. Such surprising results indicate that SRepair is capable of fixing a significant number of real-world complicated bugs in the function-level APR with limited cost on manually inspecting numerous plausible patches. Notably, repairing 227 single-function bugs with SRepair costs only $50.4, averaging $0.222 per correct fix, which is largely cheaper than SOTA APR techniques, i.e., 47.1% cheaper than ChatRepair ($0.222 vs. $0.42) and 84.5% cheaper than D4C ($0.222 vs. $1.43, based on the revised cost analysis [24]), demonstrating its efficiency as an LLM-based APR technique.

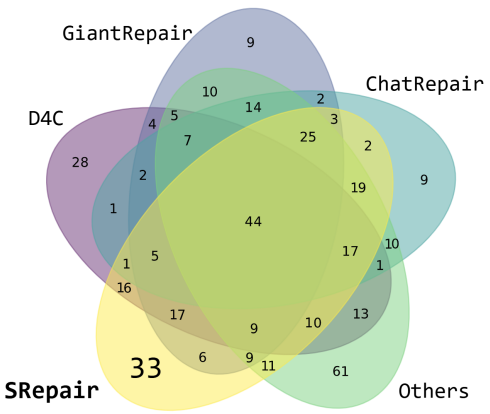

Fig. 16. Correct fix Venn diagram of SRepair with studied baselines in the single-function dataset

Table 9. Correct fixes on QuixBugs datasets

QuixBugs	SRepair	ChatRepair	AlphaRepair	CoCoNut	CURE	Recoder	RewardRepair
Java	**40**	40	28	13	26	17	20
Python	**40**	40	27	19	-	-	-

Figure 16 shows the Venn diagram of the bugs fixed by all studied baselines and SRepair. Specifically, we show the results of three SOTA LLM-based baselines and the combined results of all the other studied APR tools (as "Others"). SRepair correctly fixes 33 out of 522 single-function bugs which cannot be fixed by any APR baseline adopted in this paper. To further analyze the unique fixes of SRepair, we compute the average buggy line count, i.e., the number of buggy lines requiring modification, for the bugs uniquely fixed by SRepair and the compared baselines. We find that the 33 bugs uniquely fixed by SRepair involve 6.24 buggy lines on average, which is substantially higher than the corresponding averages of D4C (3.27), ChatRepair (1.78), and GiantRepair (2.68). This suggests that the bugs uniquely fixed by SRepair generally require more complex modifications. Our manual inspection further shows that SRepair mainly benefits from its dual-LLM CoT framework, which can effectively synthesize multi-source auxiliary information into precise bug analyses and actionable repair suggestions. A representative example is JacksonDatabind-108. As illustrated in Figure 12, the provided auxiliary context synergistically exposes the mismatch between the intended specification and the buggy code, allowing SRepair to systematically analyze these signals and deduce the correct patch. In contrast, the compared baselines often fail on such cases because they either lack sufficiently rich auxiliary information or cannot fully exploit it. By comparison, some bugs uniquely fixed by other tools may be more sensitive to sampling budget. Since SRepair uses only 10 samples per bug, while several baselines mainly use 200–500 samples, these baselines may have a higher chance of generating the exact patch for certain lightweight but highly specific bugs, such as those requiring a particular variable or function name to pass the tests. Moreover, in real-world APR scenarios where perfect fault locations are unavailable, SRepair outperforms the studied baselines by at least 17.7% (166 vs. 141) under end-to-end repair settings integrated with fault localization techniques, as shown in Table 8. Such a result indicates that SRepair not only expands the repair task scope to the more practical function-level APR but also achieves remarkable repair performance without the need for statement-level fault location information. Table 9 shows that SRepair successfully fixes all bugs in the QuixBugs dataset, indicating its superior capability for diverse programming languages. Moreover, Table 10 presents the DebugBench results, where SRepair repairs 556 bugs, outperforming other baselines. Furthermore, Table 11 demonstrates SRepair's superior performance on the challenging repository-level SWE-bench Lite benchmark. SRepair achieves a new state-of-the-art efficacy by resolving 33.7% (101/300) of issues, surpassing recent autonomous agents Agentless (32.0%) and SpecRover (30.7%) while maintaining a lower average API cost ($0.62 vs. $0.70/$0.65).

Taken together, these results suggest that SRepair achieves strong performance across various programming languages and benchmarks. Specifically, Defects4J consists of bugs from large-scale, complex Java projects, where SRepair effectively leverages its function-level repair capabilities through structured CoT reasoning. In contrast, DebugBench and QuixBugs represent more lightweight debugging scenarios; DebugBench spans multiple languages, and QuixBugs includes both Java and Python single-function tasks. SRepair's strong performance on these benchmarks suggests that its two-stage reasoning design generalizes well across languages in function-level repair. Furthermore, SWE-bench Lite introduces a more challenging repository-level Python debugging scenario. By reproducing tests and using the resulting error messages as diagnostic signals, SRepair

Table 10. APR results on DebugBench dataset

Tools	DebugBench-C++	DebugBench-Java	DebugBench-Python	Total
SRepair	**191**	**183**	**182**	**556**
D4C	**181**	**171**	**169**	**521**
ChatRepair	**176**	**163**	**157**	**496**

achieves state-of-the-art performance on this benchmark. Interestingly, Java and Python expose diagnostic signals in different forms: Java repair settings typically involve more compile-time and function-level structural constraints, whereas Python repair, especially in SWE-bench Lite, relies more on runtime feedback such as error messages and repository-level context. Overall, these observations indicate that while repair scenarios vary across languages and benchmarks, SRepair consistently maintains competitive performance and demonstrates robust behavior under diverse language and task settings.

Table 11. Performance of SRepair and baselines on SWE-bench Lite benchmark.

Tools	LLM	% Resolved	Avg. Cost $
AgentLess [117]	GPT-4o	96 (32.0%)	0.70
SpecRover [98]	GPT-4o	92 (30.7%)	0.65
AutoCodeRover [137]	GPT-4	57 (19.0%)	0.43
SRepair	GPT-4o	101 (33.7%)	0.62

We also find that SRepair correctly fixes 21 multi-function bugs, significantly outperforming the multi-function repair baselines , i.e., 12 correct fixes in ITER [133] and 9 in DEAR [71] and Hercules [99], as shown in Figure 17. Interestingly, as shown in Figure 18 where Functions 1 and 2 require information from successfully running Function 3 to determine if they should execute subsequent statements. This poses a significant challenge for APR techniques, as they need to simultaneously alter the return type of Function 3 to boolean and adapt the function calls in Functions 1 and 2. SRepair successfully identifies such a complex function call and generates the correct fix, indicating the power of SRepair on complicated multi-function faults.

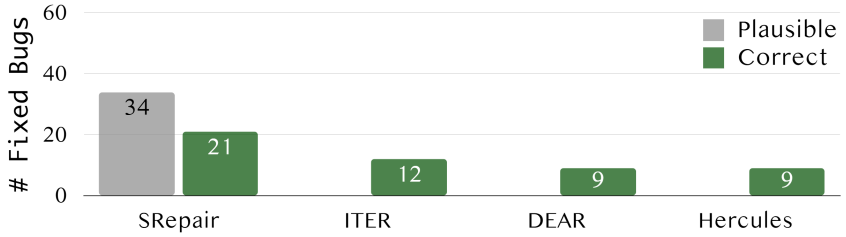

Fig. 17. The APR results of the multi-function bugs in the Defects4J dataset

4.2.5 Ablation and Configuration Study. We further conduct an ablation and configuration study of SRepair, specifically examining its performance on Defects4J single-function bugs from the following aspects: (1) the ablation of the CoT technique and the Dual-LLM framework, and the model capacity respectively; (2) APR performance using different repair suggestion model and patch generation model; (3) different sample size of SRepair. In this section, due to the intensive

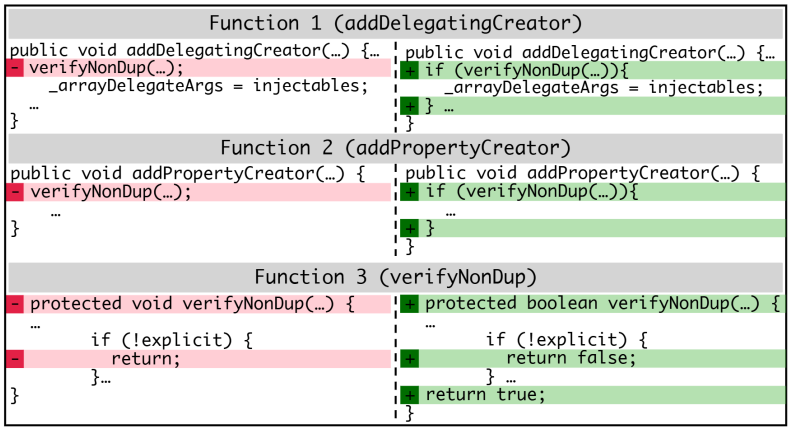

Fig. 18. The multi-function bug JacksonDatabind-69 [11]

manual efforts involved in patch inspection, we analyze the number of plausible fixes produced rather than the number of correct fixes. It should be noted that SRepair utilizes project-specific information as auxiliary repair-relevant information. Hence, we adopt GPT-4 and StarCoder2 with the *PI(ALL)* setting and 10 samples per bug as baselines for comparison. Table 12 shows that the CoT technique and the Dual-LLM framework lead to at least 7.9% (272 vs. 252) and 6.3% (252 vs. 237) gains in repair performance, respectively, with both p-values < 0.001 under McNemar's test [84] on paired bug-level outcomes, indicating that the improvements are statistically significant. Moreover, while fully adopting GPT-4 as both the repair suggestion model and the patch generation model can produce 8% (294 vs. 272) more plausible fixes, it consumes an average of 4.3× ($0.4128 vs. $0.0966) API cost compared with the default SRepair setup. Moreover, Table 13 further presents the average API cost breakdown of different SRepair configurations. We find that enabling the CoT technique under the default GPT-4-StarCoder2 setup increases the average cost per bug by only $0.0215 ($0.0966 vs. $0.0751), whereas fully adopting GPT-4 as both the repair suggestion model and the patch generation model leads to a much larger increase of $0.3162 ($0.4128 vs. $0.0966), suggesting that the larger cost increase is associated more with the patch generation model choice than with the CoT technique. Additionally, we find that the advantage of the SRepair framework could be

Table 12. Configuration study results of SRepair

Setup	Setting	# Plausible	Avg. API $
Default	SRepair (GPT-4 - StarCoder2)	**272**	**0.0966**
Ablation Study	SRepair w/o CoT	252	0.0751
	GPT-4 with *PI(ALL)*	237	0.2957
	StarCoder2 with *PI(ALL)*	203	0.0017
	SRepair (GPT-3.5-Turbo - StarCoder2, Sample n = 200)	303	0.2260
Model Selection	SRepair (GPT-4 - GPT-4)	294	0.4128
	SRepair (GPT-4 - Qwen3)	277	0.0976
	SRepair (GPT-4 - Magicoder)	265	0.0961
	SRepair (Llama3.1 - StarCoder2)	231	0.0023
Sample Size	Sample n = 5	238	0.0523
	Sample n = 100	326	0.7309

generalized to various model configurations. By adopting Llama3.1 as the repair suggestion model and StarCoder2 as the patch generation model, SRepair achieves 231 plausible fixes compared to

Table 13. Average API cost breakdown of different SRepair configurations

SRepair Settings	Model (Input/Output Avg. Cost $)				Total Cost per Bug $
	Suggestion Generation		Patch Generation		
GPT-4 - StarCoder2	GPT-4	0.0267/0.0686	StarCoder2	0.000347/0.000989	0.0966
GPT-4 - StarCoder2 w/o CoT	GPT-4	0.0241/0.0498	StarCoder2	0.000295/0.000952	0.0751
GPT-4 - GPT-4	GPT-4	0.0267/0.0686	GPT-4	0.0401/0.2774	0.4128
GPT-4 - GPT-4 w/o CoT	GPT-4	0.0239/0.0498	GPT-4	0.0353/0.2756	0.3846
StarCoder2 - StarCoder2	StarCoder2	0.000232/0.000292	StarCoder2	0.000336/0.000989	0.00185
StarCoder2 - StarCoder2 w/o CoT	StarCoder2	0.000216/0.000246	StarCoder2	0.000312/0.000964	0.00174

203 when using StarCoder2 with *PI(ALL)*. Furthermore, to address the fairness concern regarding the use of the more advanced GPT-4 model compared to baselines like ChatRepair (which utilizes GPT-3.5-Turbo), we evaluate SRepair using GPT-3.5-Turbo as the repair suggestion model with a comparable sample size of 200—a sample size smaller than ChatRepair. As shown in Table 12, even with a less capable model, this setup achieves 303 plausible fixes. To enable a direct comparison, we manually inspect these patches and identify 235 correct fixes, surpassing the 162 correct fixes reported by ChatRepair [121]. This confirms that the effectiveness of SRepair derives from its framework design rather than merely relying on the advanced model capacity. Notably, by using a sample size of 100 which is smaller than that of most baselines, SRepair achieves 326 plausible fixes, demonstrating the potential to further enhance APR performance. We further show the detailed experimental results under different configurations of SRepair on our GitHub page [1].

5 Discussion

Bug report. While the bug reports associated with dedicated projects like Defects4J are generally of high quality, as reflected by their effectiveness in our evaluation results, their quality may vary substantially in real-world settings [32, 66, 74], and they may contain invalid, irreproducible, incomplete, or even misleading contents. Meanwhile, trigger tests [78, 124, 131] effectively help reproduce the observed failure and expose fault-triggering conditions. Error messages [86, 131] can be automatically obtained from test outputs and reveal the fault-triggering boundary conditions. Comments provide function descriptions added by developers [92]. Specifically, recent studies on the bug-report-driven benchmark SWE-bench have shown that issue descriptions are not always reliable as the sole repair signal. Agentless reports that SWE-bench Lite contains issues with insufficient or misleading descriptions [117], and the construction of SWE-bench Verified by OpenAI was similarly motivated by the observation that a substantial portion of SWE-bench instances are underspecified or have problematic test specifications [90]. These observations suggest that relying on bug reports alone may be insufficient in practice. Moreover, recent SWE-bench approaches [30, 98, 113, 117, 129] such as Agentless [117] explicitly generate bug-reproducing tests from the issue description and use the resulting execution feedback for patch validation. Our results further show that SRepair can effectively leverage the reproduced trigger test and the corresponding error message as project-specific information to improve repair performance. Although bug reports are typically the primary directly available input in bug-report-driven benchmarks such as SWE-bench, execution-validated trigger tests and the resulting error messages are highly valuable for APR because they are grounded in actual program execution and directly reflect the observed failure conditions. Therefore, we recommend the utilization of project-specific information in LLM-based APR techniques to improve repair performance.

Data leakage. We study the fully open-source LLM StarCoder2 because its training data is publicly available with thorough data leakage analysis [18] such that we could find out whether the code snippet associated with each Defects4J bug is included in StarCoder2's training data. Eventually, we rule out the 122 bugs present in StarCoder2's training dataset and form Defects4J-clean dataset with

the rest 400 bugs, creating a clean, uncontaminated dataset. The detailed procedure for identifying data leakage bugs and statistics of Defects4J-clean are provided on our GitHub page [1].

We evaluate the number of plausible fixes generated by StarCoder2 under various experimental settings for both Defects4J and Defects4J-clean datasets. The results demonstrate consistent trends across both datasets, i.e., the number of plausible fixes in Defects4J/Defects4J-clean increase to 270/195 in *PI(ALL)* and 277/204 in *BR(ALL)*, and decrease to 146/102 in K_2*(CE, PE)* from 187/138 in K_0*(Basic)*, validating our findings in the study section.

Furthermore, we perform an evaluation on 590 single-function bugs on the DebugBench [110] dataset following prior work [125] and find that SRepair outperforms D4C with 556 vs. 521 bugs fixed. We also perform an evaluation on the newly released dataset ConDefects, a collection of programming contest bugs obtained after the knowledge cut-off date of gpt-4-0613 [25] to alleviate the data leakage concern. Following prior work [121], we remove duplicated buggy programs or tasks and obtain 321 and 330 Java and Python bugs respectively. We find that SRepair consistently outperforms ChatRepair and AlphaRepair with at least 18.1% (287 vs. 243) and 12.8% (272 vs. 241) more correct fixes on the Java and Python datasets respectively. We also evaluate SRepair fully constructed by StarCoder2 on the Defects4J and GrowingBugs [57] datasets, which also show the same trends as in our GitHub page [1]. These results further validate our findings and approach, effectively mitigating the risk of data leakage affecting its performance.

Assessing statement-level fault localization needs. We utilize the ground-truth statement-level fault location information following previous work [92] and find that while including the statement-level fault location (*sfl*) information enhances the repair performance, the extent of this improvement can be potentially compromised with the addition of the auxiliary repair-relevant information. For instance, while K_0*(Basic)* with *sfl* achieves a performance improvement of 22.2% (165 vs. 135) on the average number of correct fixes compared to K_0*(Basic)*, such an improvement shrinks to 5.5% (231 vs. 219) when comparing *BR(ALL)* with *sfl* over *BR(ALL)*, and 4.2% (223 vs. 214) when comparing *PI(ALL)* with *sfl* over *PI(ALL)*. Additional details, including the labeling method for statement-level fault locations and the complete performance results across all auxiliary repair-relevant information settings, are available in our GitHub page [1].

Few-shot learning in APR. Our empirical study suggests that the impact of few-shot learning in function-level APR is highly inconsistent and model-dependent (Finding 2). One possible explanation for such disparate performance is a potential anchoring effect induced by the provided code exemplars. As demonstrated in Figure 7, the zero-shot setting (K_0) exhibits the lowest median CodeBLEU score (0.820) and the largest interquartile range (0.18), indicating a diverse and broad search space for patches. In contrast, few-shot settings consistently yield higher and more concentrated CodeBLEU distributions (median scores ranging from 0.849 to 0.937, IQR between 0.10 and 0.13). This suggests that few-shot exemplars might anchor the models toward conservative edits that remain lexically and structurally closer to the original buggy function. While this anchoring could be beneficial when the provided example closely aligns with the target bug's repair pattern, it risks narrowing the search space and potentially misguiding the model when there is a mismatch. This restricted exploration may contribute to degraded patch diversity, disparate performance across different LLMs, and the observed increase in uncompilable patches (as shown in Figure 6). To mitigate these potential limitations and further explore few-shot learning's utility in future work, researchers could investigate adaptive few-shot prompting, which dynamically selects semantically aligned exemplars based on the target bug's characteristics. Furthermore, inspired by Finding 5 and Finding 7, another potential strategy is to place more emphasis on diagnostic auxiliary context (e.g., trigger tests and error messages), rather than relying solely on code-level exemplars. Unlike few-shot code examples that might lexically restrict patch generation, such auxiliary information

may provide more direct semantic guidance for repair, helping the LLM infer the correct fix without substantially compromising the diversity of the generated code.

6 Threats to Validity

Threats to internal validity. One potential threat arises from our manual validation process, which differentiates between plausible patches and those that are semantically correct. To address this concern, three authors cross-validated the plausible patches generated by different models in our study and SRepair by comparing them to those created by developers, ensuring the robustness and reliability of the evaluation.

Another threat is the risk of data leakage if the developer patches were included in the original training data. We discuss how to address this threat in Section 5.

An additional threat lies in the trigger tests adopted in SRepair where the LLMs might have recognized the trigger tests and manipulated them to pass all tests, creating seemingly plausible patches. Our SRepair's Dual-LLM mechanism effectively mitigates this threat, as the repair suggestion model only suggests bug fixes without trigger test information, keeping the patch generation model isolated from such data. Moreover, the selection of compared APR techniques in our study introduces another internal validity threat. To mitigate this, we carefully selected four recent, well-designed LLM-based APR techniques with sample sizes equal to or larger than those used in our study settings.

Furthermore, the disparity in the backbone models used by different techniques poses a threat to the fairness of our comparison. Specifically, SRepair employs the advanced GPT-4 model, whereas some baselines like ChatRepair utilize the GPT-3.5-Turbo model. While replicating these baselines with GPT-4 would resolve this discrepancy, the prohibitive cost—estimated at over $6,000 for ChatRepair due to its intensive iterative sampling mechanism—renders this infeasible. To mitigate this threat, we instead evaluated SRepair using the same GPT-3.5-Turbo model under a comparable sampling budget. As detailed in Section 4.2.5, SRepair still demonstrates superior performance over the baseline even with the less capable model, confirming that our improvements derive from the framework design rather than solely from the underlying model's capabilities.

Threats to external validity. The main threat to external validity lies in our evaluation datasets used which may not well generalize our experimental results. To mitigate this, we evaluate our approach on both the popular Defects4J 1.2 and 2.0 datasets where we include all their single-function bugs in our study. Furthermore, we extend our investigation to multi-function bugs in our SRepair evaluation. We also evaluate SRepair on the QuixBugs, ConDefects, SWE-Bench and DebugBench, datasets, which contain Java, Python and C++ bugs, to validate its generalizability.

An additional threat to external validity is that not all types of project-specific information are guaranteed to be available for every bug in real-world settings. This threat can be partially mitigated in bug-report-driven scenarios by reproducing trigger tests from bug reports and executing them to obtain the corresponding error messages. Our experiments on SWE-bench further show that SRepair can effectively leverage such reproduced trigger tests and error messages in this setting.

Threats to construct validity. The threat to construct validity mainly lies in the metrics used. To mitigate this, we adopt the widely-used plausible patches and correct patches along with their distributions.

7 Conclusion

In this paper, we conduct a comprehensive study on the function-level LLM-based APR. Our study reveals that LLMs with zero-shot learning are powerful function-level APR techniques. Moreover, directly applying the auxiliary repair-relevant information to LLMs significantly increases the function-level repair performance. Inspired by our findings, we design a Dual-LLM framework

utilizing the Chain of Thought technique, named SRepair, which achieves remarkable repair performance by correctly fixing 227 single-function bugs in the Defects4J dataset, surpassing D4C by 26% and ChatRepair by 40%. Furthermore, SRepair successfully fixes 21 multi-function bugs in Defects4J, significantly outperforming other state-of-the-art APR techniques.

Data Availability

All study results, evaluation details, and source code of the SRepair implementation are presented in the *GitHub* page [1].

Acknowledgments

This work is partially supported by the National Natural Science Foundation of China (Grant No. 62372220). It is also partially supported by Kuaishou.

References

[1] 2024. Github Repository. https://github.com/GhabiX/SRepair.
[2] 2024. SWE-bench Lite. https://www.swebench.com/lite.html.
[3] 2024-02-29. Hugging Face. https://huggingface.co.
[4] 2024-02-29. OpenAI API. https://openai.com/api.
[5] 2024-02-29. OpenAI Platform. https://platform.openai.com/docs/models/gpt-3-5-turbo.
[6] 2024-03. Defects4J-GitHub. https://github.com/rjust/defects4j.
[7] 2024-03. Junit-5. https://junit.org/junit5/.
[8] 2024-03-04. ise-uiuc/magicoder. https://huggingface.co/ise-uiuc/Magicoder-S-CL-7B.
[9] 2024-03-17. Defects4j Cli-26. https://github.com/rjust/defects4j/blob/master/ framework/projects/Cli/patches/26.src.patch.
[10] 2024-03-17. Defects4j Csv-4. https://issues.apache.org/jira/browse/CSV-100.
[11] 2024-03-17. Defects4j JacksonDatabind-69. https://github.com/rjust/defects4j/blob/ master/framework/projects/JacksonDatabind/patches/69.src.patch.
[12] 2024-03-17. Defects4j Math-2. https://github.com/rjust/defects4j/blob/master/ framework/projects/Math/patches/2.src.patch.
[13] 2024-03-17. Defects4j Math-80. https://github.com/rjust/defects4j/blob/master/ framework/projects/Math/patches/80.src.patch.
[14] 2024-03-17. Defects4j Math-91. https://github.com/rjust/defects4j/blob/master/ framework/projects/Math/patches/91.src.patch.
[15] 2024-03-17. Defects4j Math-95. https://github.com/rjust/defects4j/blob/master/ framework/projects/Math/patches/95.src.patch.
[16] 2024-03-17. Flagship Model gpt-3.5-turbo-1106. https://platform.openai.com/docs/models/gpt-3-5-turbo.
[17] 2024-03-17. Implementation of multiple example selection. https://zenodo.org/records/7592886/files/patch.tar.gz?download=1.
[18] 2025. Data portraits. https://stack.dataportraits.org/.
[19] 2025. Interquartile range.
[20] 2025. Qwen3-32b huggingface. https://huggingface.co/Qwen/Qwen3-32B.
[21] 2025-03-05. bigcode/starcoder2-15b-instruct-v0.1. https://huggingface.co/bigcode/starcoder2-15b-instruct-v0.1.
[22] 2025-03-05. Flagship Model gpt-4. https://platform.openai.com/docs/models/gpt-4.
[23] 2025-03-05. meta-llama/Llama-3.1-8B-Instruct huggingface. https://huggingface.co/meta-llama/Llama-3.1-8B-Instruct.
[24] 2025-03-08. D4C cost. https://github.com/CUHK-Shenzhen-SE/D4C/issues/3.
[25] 2025-03-09. GPT-4 Cut-off Date. https://learn.microsoft.com/en-us/azure/ai-services/openai/concepts/models?tabs=global-standard%2Cstandard-chat-completions#gpt-4.
[26] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).
[27] Monica Agrawal, Stefan Hegselmann, Hunter Lang, Yoon Kim, and David Sontag. 2022. Large language models are few-shot clinical information extractors. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*. 1998–2022.

[28] Cyfuture AI. 2025. Open-source model price. https://cyfuture.ai/serverless-inferencing.
[29] Gabin An and Shin Yoo. 2022. FDG: a precise measurement of fault diagnosability gain of test cases. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 14–26.
[30] Anthropic. 2025. Raising the Bar on SWE-Bench Verified with Claude 3.5 Sonnet. https://www.anthropic.com/engineering/swe-bench-sonnet. Accessed: 2025-09-11.
[31] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models.
[32] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. 2008. What makes a good bug report?. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. 308–318.
[33] Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. 2024. Repairagent: An autonomous, llm-based agent for program repair. *arXiv preprint arXiv:2403.17134* (2024).
[34] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, and et al. Kaplan. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems*, Vol. 33. 1877–1901.
[35] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, et al. 2023. Sparks of artificial general intelligence: Early experiments with gpt-4. *arXiv preprint arXiv:2303.12712* (2023).
[36] Oscar Chaparro, Jing Lu, Fiorella Zampetti, Laura Moreno, Massimiliano Di Penta, Andrian Marcus, Gabriele Bavota, and Vincent Ng. 2017. Detecting missing information in bug descriptions. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*. 396–407.
[37] Lingjiao Chen, Matei Zaharia, and James Zou. 2023. Frugalgpt: How to use large language models while reducing cost and improving performance. *arXiv preprint arXiv:2305.05176* (2023).
[38] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé De Oliveira Pinto, and et al Kaplan. 2021. Evaluating large language models trained on code. *arXiv (Cornell University)* (July 2021). doi:10.48550/arxiv.2107.03374
[39] Xiangping Chen, Xing Hu, Yuan Huang, He Jiang, Weixing Ji, Yanjie Jiang, Yanyan Jiang, Bo Liu, Hui Liu, Xiaochen Li, et al. 2025. Deep learning-based software engineering: progress, challenges, and opportunities. *Science China Information Sciences* 68, 1 (2025), 111102.
[40] Yukang Chen, Shengju Qian, Haotian Tang, Xin Lai, Zhijian Liu, Song Han, and Jiaya Jia. [n. d.]. LongLoRA: Efficient Fine-tuning of Long-Context Large Language Models. In *The Twelfth International Conference on Learning Representations*.
[41] Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2019. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering* 47, 9 (2019), 1943–1959.
[42] Liying Cheng, Xingxuan Li, and Lidong Bing. 2023. Is gpt-4 a good data analyst?. In *Findings of the Association for Computational Linguistics: EMNLP 2023*. 9496–9514.
[43] Jonathan Cordeiro, Shayan Noei, and Ying Zou. 2024. An empirical study on the code refactoring capability of large language models. *arXiv preprint arXiv:2411.02320* (2024).
[44] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In *Proceedings of the 32nd ACM SIGSOFT international symposium on software testing and analysis*. 423–435.
[45] Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. 2024. Large language models are edge-case generators: Crafting unusual programs for fuzzing deep learning libraries. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–13.
[46] Jessica López Espejel, El Hassane Ettifouri, Mahaman Sanoussi Yahaya Alassan, El Mehdi Chouham, and Walid Dahhane. 2023. GPT-3.5, GPT-4, or BARD? Evaluating LLMs reasoning ability in zero-shot setting and performance boosting through prompts. *Natural Language Processing Journal* 5 (2023), 100032.
[47] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. 2023. Automated repair of programs from large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1469–1481.
[48] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. 2018. Automatic software repair: A survey. In *Proceedings of the 40th International Conference on Software Engineering*. 1219–1219.
[49] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. Deepfix: Fixing common c language errors by deep learning. In *Proceedings of the aaai conference on artificial intelligence*, Vol. 31.
[50] Yirui He, Ziyao He, Syed Fatiul Huq, and Sam Malek. 2026. ReFLAIR: Detecting Responsive Layout Reflow Issues using Multimodal Generative AI. *Proceedings of the ACM on Software Engineering* 3, FSE (2026). doi:10.1145/3808136
[51] Kai Huang, Xiangxin Meng, Jian Zhang, Yang Liu, Wenjie Wang, Shuhao Li, and Yuqing Zhang. 2023. An empirical study on fine-tuning large language models of code for automated program repair. In *2023 38th IEEE/ACM International*

Conference on Automated Software Engineering (ASE). IEEE, 1162–1174.
[52] Kai Huang, Jian Zhang, Xiangxin Meng, and Yang Liu. 2024. Template-guided program repair in the era of large language models. In *Proceedings of the 47th International Conference on Software Engineering, ICSE*. 367–379.
[53] Jiajun Jiang et al. 2023. KNOD: A knowledge-driven data-centric approach for automated program repair. In *Proceedings of the IEEE/ACM 45th International Conference on Software Engineering (ICSE)*.
[54] Jiajun Jiang, Fengjie Li, Zijie Zhao, Zhirui Ye, Mengjiao Liu, Bo Wang, Hongyu Zhang, and Junjie Chen. 2025. Boosting Redundancy-based Automated Program Repair by Fine-grained Pattern Mining. In *2025 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 85–97.
[55] 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.
[56] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. Cure: Code-aware neural machine translation for automatic program repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1161–1173.
[57] Yanjie Jiang, Hui Liu, Xiaoqing Luo, Zhihao Zhu, Xiaye Chi, Nan Niu, Yuxia Zhang, Yamin Hu, Pan Bian, and Lu Zhang. 2022. BugBuilder: An Automated Approach to Building Bug Repository. *IEEE Transactions on Software Engineering* (2022), 1–22. doi:10.1109/TSE.2022.3177713
[58] 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).
[59] Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. 2023. 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*. 1646–1656.
[60] 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.
[61] Sungmin Kang, Juyeon Yoon, Nargiz Askarbekkyzy, and Shin Yoo. 2024. Evaluating diverse large language models for automatic and general bug reproduction. *IEEE Transactions on Software Engineering* (2024).
[62] Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023. Large language models are few-shot testers: Exploring llm-based general bug reproduction. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2312–2323.
[63] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 802–811.
[64] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. 2016. Practitioners' expectations on automated fault localization. In *Proceedings of the 25th international symposium on software testing and analysis*. 165–176.
[65] Sophia D Kolak, Ruben Martins, Claire Le Goues, and Vincent Josua Hellendoorn. 2022. Patch generation with language models: Feasibility and scaling behavior. In *Deep Learning for Code Workshop*.
[66] Anil Koyuncu, Kui Liu, Tegawendé F Bissyandé, Dongsun Kim, Martin Monperrus, Jacques Klein, and Yves Le Traon. 2019. iFixR: Bug report driven program repair. In *Proceedings of the 2019 27th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 314–325.
[67] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K Lahiri, and Siddhartha Sen. 2023. Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 919–931.
[68] Fengjie Li, Jiajun Jiang, Jiajun Sun, and Hongyu Zhang. 2024. Hybrid automated program repair by combining large language models and program analysis. *ACM Transactions on Software Engineering and Methodology* (2024).
[69] Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. 2019. Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization. In *Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis*. 169–180.
[70] 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.
[71] 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.
[72] 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*.
[73] Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. 2017. QuixBugs: A multi-lingual program repair benchmark set based on the Quixey Challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN international conference on systems, programming, languages, and applications: software for humanity*. 55–56.
[74] Chen Liu, Jinqiu Yang, Lin Tan, and Munawar Hafiz. 2013. R2Fix: Automatically generating bug fixes from bug reports. In *2013 IEEE Sixth international conference on software testing, verification and validation*. IEEE, 282–291.

[75] Kui Liu, Anil Koyuncu, Tegawendé F Bissyandé, Dongsun Kim, Jacques Klein, and Yves Le Traon. 2019. You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems. In *2019 12th IEEE conference on software testing, validation and verification (ICST)*. IEEE, 102–113.

[76] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F Bissyandé. 2019. TBar: Revisiting template-based automated program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 31–42.

[77] Yu Liu et al. 2025. Repatt: A representation-based framework for automated program repair. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME)*. Explicitly describes the 'three authors independent validation' protocol.

[78] Fan Long and Martin Rinard. 2015. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 166–178.

[79] Fan Long and Martin Rinard. 2016. An analysis of the search spaces for generate and validate patch generation systems. In *Proceedings of the 38th International Conference on Software Engineering*. 702–713.

[80] Yiling Lou, Ali Ghanbari, Xia Li, Lingming Zhang, Haotian Zhang, Dan Hao, and Lu Zhang. 2020. Can automated program repair refine fault localization? a unified debugging approach. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 75–87.

[81] 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).

[82] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. Coconut: combining context-aware neural translation models using ensemble for program repair. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*. 101–114.

[83] Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. 2017. Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset. *Empirical Software Engineering* 22 (2017), 1936–1964.

[84] Quinn McNemar. 1947. Note on the sampling error of the difference between correlated proportions or percentages. *Psychometrika* 12, 2 (1947), 153–157.

[85] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th international conference on software engineering*. 691–701.

[86] Ali Mesbah, Andrew Rice, Emily Johnston, Nick Glorioso, and Edward Aftandilian. 2019. Deepdelta: learning to repair compilation errors. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 925–936.

[87] Meta. 2025. Llama 3.1 Price. https://www.llama.com/llama3_1.

[88] 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.

[89] OpenAI. 2024. Hello GPT-4o. https://openai.com/index/hello-gpt-4o.

[90] OpenAI. 2024. Introducing SWE-bench, verified. https://openai.com/index/introducing-swe-bench-verified/. Accessed on 2025-06-23.

[91] Fabio Petroni, Tim Rocktäschel, Sebastian Riedel, Patrick Lewis, Anton Bakhtin, Yuxiang Wu, and Alexander Miller. 2019. Language Models as Knowledge Bases?. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. 2463–2473.

[92] Julian Aron Prenner, Hlib Babii, and Romain Robbes. 2022. Can OpenAI's codex fix bugs? an evaluation on QuixBugs. In *Proceedings of the Third International Workshop on Automated Program Repair*. 69–75.

[93] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. 2015. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA)*.

[94] Qwen. 2025. qwen3-32b Price. https://modelstudio.console.alibabacloud.com/?tab=doc#/doc/?type=model&url=2840914_2&modelId=qwen3-32b.

[95] Md Nakhla Rafi, Dong Jae Kim, An Ran Chen, Tse-Hsun Chen, and Shaowei Wang. 2024. Towards Better Graph Neural Network-based Fault Localization Through Enhanced Code Representation. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1937–1959.

[96] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297* (2020).

[97] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, and et al. Sootla. 2023. Code llama: Open Foundation Models for code. *arXiv (Cornell University)* (Aug. 2023). doi:10.48550/arxiv.2308.12950

[98] Haifeng Ruan, Yuntong Zhang, and Abhik Roychoudhury. 2024. Specrover: Code intent extraction via llms. *arXiv preprint arXiv:2408.02232* (2024).

[99] Seemanta Saha et al. 2019. Harnessing evolution for multi-hunk program repair. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 13–24.

[100] André Silva, Sen Fang, and Martin Monperrus. 2025. Repairllama: Efficient representations and fine-tuned adapters for program repair. *IEEE Transactions on Software Engineering* (2025).

[101] Victor Sobreira, Thomas Durieux, Fernanda Madeiral, Martin Monperrus, and Marcelo de Almeida Maia. 2018. Dissection of a bug dataset: Anatomy of 395 patches from defects4j. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 130–140.

[102] Ezekiel Soremekun, Lukas Kirschner, Marcel Böhme, and Mike Papadakis. 2023. Evaluating the impact of experimental assumptions in automated fault localization. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 159–171.

[103] Chi Sun, Xipeng Qiu, Yige Xu, and Xuanjing Huang. 2019. How to fine-tune bert for text classification?. In *Chinese computational linguistics: 18th China national conference, CCL 2019, Kunming, China, October 18–20, 2019, proceedings 18*. Springer, 194–206.

[104] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to Sequence Learning with Neural Networks. In *Advances in Neural Information Processing Systems*, Vol. 27.

[105] Hanzhuo Tan, Weihao Li, Xiaolong Tian, Siyi Wang, Jiaming Liu, Jing Li, and Yuqun Zhang. 2025. SK2Decompile: LLM-based Two-Phase Binary Decompilation from Skeleton to Skin. *arXiv preprint arXiv:2509.22114* (2025).

[106] Hanzhuo Tan, Qi Luo, Ling Jiang, Zizheng Zhan, Jing Li, Haotian Zhang, and Yuqun Zhang. 2024. Prompt-based code completion via multi-retrieval augmented generation. *ACM Transactions on Software Engineering and Methodology* (2024).

[107] Hanzhuo Tan, Qi Luo, Jing Li, and Yuqun Zhang. 2024. Llm4decompile: Decompiling binary code with large language models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*. 3473–3487.

[108] Hanzhuo Tan, Xiaolong Tian, Hanrui Qi, Jiaming Liu, Siyi Wang, GAO Zuchen, Qi Luo, Jing Li, and Yuqun Zhang. [n. d.]. Decompile-Bench: Million-Scale Binary-Source Function Pairs for Real-World Binary Decompilation. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.

[109] Hanzhuo Tan, Chunpu Xu, Jing Li, Yuqun Zhang, Zeyang Fang, Zeyu Chen, and Baohua Lai. 2024. Hicl: Hashtag-driven in-context learning for social media natural language understanding. *IEEE transactions on neural networks and learning systems* 36, 4 (2024), 7037–7050.

[110] Runchu Tian, Yining Ye, Yujia Qin, Xin Cong, Yankai Lin, Yinxu Pan, Yesai Wu, Haotian Hui, Weichuan Liu, Zhiyuan Liu, et al. 2024. Debugbench: Evaluating debugging capability of large language models. *arXiv preprint arXiv:2401.04621* (2024).

[111] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) *(NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 6000–6010.

[112] Weishi Wang, Yue Wang, Shafiq Joty, and Steven CH Hoi. 2023. Rap-gen: Retrieval-augmented patch generation with codet5 for automatic program repair. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 146–158.

[113] Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. 2024. Openhands: An open platform for ai software developers as generalist agents. *arXiv preprint arXiv:2407.16741* (2024).

[114] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems* 35 (2022), 24824–24837.

[115] Yuxiang Wei, Zhe Wang, Jiawei Liu, Ye Ding, and Lingming Zhang. 2023. MagicOder: Source code is all you need. *arXiv (Cornell University)* (Dec. 2023). doi:10.48550/arxiv.2312.02120

[116] Yuxiang Wei, Chunqiu Steven Xia, and Lingming Zhang. 2023. Copiloting the copilots: Fusing large language models with completion engines for automated program repair. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 172–184.

[117] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2025. Demystifying llm-based software engineering agents. *Proceedings of the ACM on Software Engineering* 2, FSE (2025), 801–824.

[118] Chunqiu Steven Xia, Yifeng Ding, and Lingming Zhang. 2023. The Plastic Surgery Hypothesis in the Era of Large Language Models. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 522–534.

[119] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated program repair in the era of large pre-trained language models. In *Proceedings of the 45th International Conference on Software Engineering (ICSE 2023)*.

Association for Computing Machinery.

[120] Chunqiu Steven Xia and Lingming Zhang. 2022. Less training, more repairing please: revisiting automated program repair via zero-shot learning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 959–971.

[121] Chunqiu Steven Xia and Lingming Zhang. 2024. Automated Program Repair via Conversation: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT. In *Proceedings of the 33th ACM SIGSOFT International Symposium on Software Testing and Analysis*.

[122] Jiahong Xiang, Wenxiao He, Xihua Wang, Hongliang Tian, and Yuqun Zhang. 2026. Evaluating and Improving Automated Repository-Level Rust Issue Resolution with LLM-based Agents. *arXiv preprint arXiv:2602.22764* (2026).

[123] Yingfei Xiong and Bo Wang. 2022. L2S: A framework for synthesizing the most probable program under a specification. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 3 (2022), 1–45.

[124] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise condition synthesis for program repair. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 416–426.

[125] Junjielong Xu, Ying Fu, Shin Hwei Tan, and Pinjia He. 2024. Aligning the Objective of LLM-based Program Repair. *arXiv preprint arXiv:2404.08877* (2024).

[126] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388* (2025).

[127] Boyang Yang, Haoye Tian, Jiadong Ren, Hongyu Zhang, Jacques Klein, Tegawendé F Bissyandé, Claire Le Goues, and Shunfu Jin. 2026. Morepair: Teaching llms to repair code via multi-objective fine-tuning. *ACM Transactions on Software Engineering and Methodology* 35, 2 (2026), 1–38.

[128] Jun Yang et al. 2023. PraPatch: A comprehensive patch correctness comparison dataset. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

[129] 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. *Advances in Neural Information Processing Systems* 37 (2024), 50528–50652.

[130] Jun Yang and Yuehan Wang. 2023. Patch manual inspection rules. https://github.com/claudeyj/patch_correctness/blob/master/semantic_equivalence_rules.md. Accessed: 2024.

[131] He Ye, Matias Martinez, Xiapu Luo, Tao Zhang, and Martin Monperrus. 2022. Selfapr: Self-supervised program repair with test execution diagnostics. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–13.

[132] He Ye, Matias Martinez, and Martin Monperrus. 2022. Neural program repair with execution-based backpropagation. In *Proceedings of the 44th International Conference on Software Engineering*. 1506–1518.

[133] He Ye and Martin Monperrus. 2024. ITER: Iterative Neural Repair for Multi-Location Patches. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–13.

[134] Zhengran Zeng, Hanzhuo Tan, Haotian Zhang, Jing Li, Yuqun Zhang, and Lingming Zhang. 2022. An extensive study on pre-trained models for program understanding and generation. In *Proceedings of the 31st ACM SIGSOFT international symposium on software testing and analysis*. 39–51.

[135] Quanjun Zhang, Chunrong Fang, Tongke Zhang, Bowen Yu, Weisong Sun, and Zhenyu Chen. 2023. Gamma: Revisiting template-based automated program repair via mask prediction. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 535–547.

[136] Tao Zhang, Jiachi Chen, He Jiang, Xiapu Luo, and Xin Xia. 2017. Bug report enrichment with application of automated fixer recommendation. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. IEEE, 230–240.

[137] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. Autocoderover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1592–1604.

[138] Zhuosheng Zhang, Aston Zhang, Mu Li, George Karypis, Alex Smola, et al. [n. d.]. Multimodal Chain-of-Thought Reasoning in Language Models. *Transactions on Machine Learning Research* ([n. d.]).

[139] Jiuang Zhao, Donghao Yang, Li Zhang, Xiaoli Lian, Zitian Yang, and Fang Liu. 2024. Enhancing automated program repair with solution design. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 1706–1718.

[140] Shen Zheng, Yuyu Zhang, Yijie Zhu, Chenguang Xi, Pengyang Gao, Zhou Xun, and Kevin Chang. 2024. Gpt-fathom: Benchmarking large language models to decipher the evolutionary path towards gpt-4 and beyond. In *Findings of the Association for Computational Linguistics: NAACL 2024*. 1363–1382.

[141] Qihao Zhu, Zeyu Sun, Yuan-an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. 2021. A syntax-guided edit decoder for neural program repair. In *Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*. 341–353.

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009