

Received 21 June 2026, accepted 8 July 2026, date of publication 16 July 2026, date of current version 24 July 2026.

Digital Object Identifier 10.1109/ACCESS.2026.3714292

RESEARCH ARTICLE

Exploring Generalizable Automated Program Repair With Large Language Models

VIOLA CAMPOS¹, RIDWAN SHARIFFDEEN², (Senior Member, IEEE),
ADRIAN ULGES¹, AND YANNIC NOLLER³, (Member, IEEE)

¹RheinMain University of Applied Sciences, 65195 Wiesbaden, Germany

²Sonar, Austin, TX 78701, USA

³Ruhr University Bochum, 44801 Bochum, Germany

Corresponding author: Yannic Noller (yannic.noller@acm.org)

This work was supported in part by the German Federal Ministry of Research, Technology and Space (BMFTR), under Project “SLIMDOC” (FKZ: 13HAW15PX4).

ABSTRACT Automated Program Repair (APR) proposes bug fixes to aid developers in maintaining software. The state of the art in this domain focuses on LLMs, leveraging their strong capabilities to comprehend specifications in natural language and to generate program code. However, despite the APR community’s research achievements and industry deployments, APR still cannot generalize broadly. In this work, we present an intensive empirical evaluation of LLMs’ capabilities in APR. We evaluate a diverse set of 13 recent open and closed models. In particular, we explore language-agnostic repair by utilizing benchmarks for Java, JavaScript, Python, and PHP. Besides the generalization across languages and levels of patch complexity, we also investigate the effects of fault localization (FL). Our key results include: 1) Different LLMs tend to perform best for different languages, which makes it hard to develop cross-platform, single-LLM repair techniques. 2) Combining models by pooling repairs adds value with respect to uniquely fixed bugs, so a committee of expert models should be considered. 3) Under realistic assumptions of imperfect FL, we observe significant drops in accuracy from the usual practice of using perfect FL. Our insights will help develop reliable and generalizable APR techniques and their evaluation on realistic and fair environments.

INDEX TERMS Automated program repair, empirical evaluation, large language models.

I. INTRODUCTION

Automated Program Repair (APR) [1], [2] has gained significant attention in the last few years. It attempts to repair software bugs automatically, and hence, promises great support for software practitioners in maintaining code. APR research in the last decade has focused on search-based, semantics-based, template-based, and learning-based methods. Search-based techniques [3] apply evolutionary code mutations to change the program to pass a test suite. Semantics-based techniques [4] use a test suite to extract specifications as constraints via, e.g., symbolic execution. These constraints then drive a program synthesis to generate a repair, e.g., expressions or whole statements.

The associate editor coordinating the review of this manuscript and approving it for publication was Mu-Yen Chen¹.

Template-based techniques [5] use pre-defined patch templates that are either manually crafted for a specific domain or mined from large code repositories. Finally, learning-based techniques [6] attempt to directly learn how to generate patches based on large sets of historic fixes. Despite these research achievements in APR and existing industrial deployments [7], [8], [9], there are still issues with generalizing and benchmark overfitting. The first international competition on APR [10] in 2024 using the Cerberus framework [11] has shown that current techniques are not yet able to reliably provide software patches, i.e., they failed to work in realistic settings and generalize beyond the community benchmarks. For example, in the *Functional Error* track for Java, the best tool was able to correctly fix only 5 out of 50 previously unseen bugs (i.e., 10% fix rate) and many techniques were unable to provide any correct patches. The benchmarks had

been curated with new and unforeseen repair tasks, and the tools had been evaluated in a uniform environment enforcing strict resource constraints.

Machine Learning (ML) techniques, specifically Large Language Models (LLM), play a significant role in the SE and APR community's most recent research efforts. Due to their pre-training on large amounts of open-source code and due to their enormous capacity, LLMs have demonstrated problem-solving skills over a wide range of tasks [12]. In APR, recent works [13], [14] have shown that LLMs can be used to generate repairs and build SE agents [15], [16], [17], which attempt to mimic the work of software engineers. Nevertheless, most of the current LLM-based APR approaches use LLMs in a rather ad-hoc strategy (e.g., [13], [18], [19], [20]), and while some existing works study the usage of LLMs in a more systematic way (e.g., [21], [22], [23]), they all fall short on at least one of the following points: (i) the investigation is based on one specific LLM only, (ii) no/limited exploration of various prompting strategies/designs is conducted, in particular regarding repair ingredients (like information about failing test cases, bug reports/descriptions, error messages, etc.), (iii) a perfect Fault Localization (FL) is assumed to be given, which represents an unrealistic assumption in a practical setting, and (iv) only one specific benchmark with one programming language is covered, e.g., Defects4J [24].

To address the above issues and to provide a comprehensive overview of LLMs' single-prompt, zero-shot APR capabilities, this work presents a systematic empirical evaluation with state-of-the-art *closed* and *open* LLMs on multiple benchmarks *across* programming languages. Hereby, we refer to *open* models as those with publicly available weights,¹ whereas *closed* LLMs are proprietary models whose weights, training data, and model architectures have not been publicly released. Particularly, we compare closed models like Claude 3.7 Sonnet, Gemini 1.5 Pro, OpenAI o3-mini, and OpenAI GPT-4o with open ones like Llama 3.3, CodeLlama, and DeepSeek Coder on benchmarks for Java (Defects4J [24]), JavaScript (BugsJS [25]), Python (BugsInPy [26]), and PHP (BugsPHP [27]). In total, we generated and evaluated $\approx 195,000$ patches in four programming languages and fixed 542 out of 712 bugs.

We study the following aspects: (1) The question of whether certain LLMs display language-agnostic repair capabilities. (2) The influence of different information sources in the prompt (specifically, localization and test errors). (3) The robustness of LLMs to the quality of Fault Localization (FL). To do so, we systematically evaluate LLMs' APR accuracy for line-level vs function-level localization, and when combined with a real-world spectrum-based automated FL. (4) The performance across different patch complexity levels (i.e., single-line, single-hunk, multi-hunk).

¹Note that, according to our definition, open models may not necessarily come with open-source licenses, and that – unlike their weights – their training code and/or data may remain undisclosed.

Based on our experiments, current LLMs do not generalize well across programming languages for the task of program repair. In particular, we observed that different LLMs perform well on different benchmarks. Similar to others [28], we observed that including information about the failing test case significantly boosted the repair performance for all models. We found LLMs to show good robustness when repairing bugs of higher complexity. However, notably, our comparison between perfect and automated FL techniques showed a drastic decline in repair capabilities if the buggy function cannot reliably be identified. Overall, our findings provide fundamental and comprehensive understanding of the current capabilities of LLMs with respect to the various aspects of APR, which is beneficial for our community (e.g., for designing future research efforts) and for practitioners (e.g., for selecting appropriate LLMs for their repair use case). In summary, we make the following core contributions:

- We provide an extensive evaluation of state-of-the-art (open and closed) LLMs' single-prompt, zero-shot program repair capabilities, including multiple models, programming languages, prompt variations, patch complexities, and fault localization methods.
- To the APR practitioner, our main take-away results are that (1) different LLMs are suited best for different languages, suggesting language-specific model committees, (2) test information appears to be a key signal for LLM-based APR, and (3) open models appear to be catching up to their closed counterparts.
- To the future APR researcher, our study shows that (1) fault localization is a potential bottleneck for APR pipelines, which suggests a strong focus on this aspect. (2) APR accuracy generalizes poorly, with different models reacting differently when varying programming language or prompt information; and (3) combining different LLMs improves results across languages and offers interesting directions for future work. We conclude that APR pipelines need to be benchmarked carefully and broadly. To support such a holistic evaluation of future progress in APR, we open-source our experimental setup and results for other researchers.

II. BACKGROUND AND RELATED WORK

A. LLM BASED PROGRAM REPAIR

Numerous APR techniques have been studied in the last decade, ranging from the traditional search-based repair [3], constraint-based repair [4], template-based repair [5], and learning-based repair [29]. More recently, with the advancement of LLMs for code generation, various techniques [6], [13], [14] have been explored by the research community to utilize LLMs for the program repair problem. LLM-based repair techniques transform the program repair problem into a code generation problem. These techniques can be broadly classified as model training [18], [22], zero-shot learning [13] and few-shot learning [8], [30], [31]. Silva et al. [22]

explored how different input and output representations yield optimized results using LLMs for program repair tasks. Xia et al. [14] studied the effect of test diagnostic feedback for LLM-driven program repair. Lin et al. [32] studied the impact of repair granularities encountered in program repair tasks. More specifically, they examined how expression-level and token-level repairs differ for different LLMs.

In our work, we incorporate these different findings and techniques into an extensive study with 13 state-of-the-art LLMs and evaluate the generalizability across different languages. We also evaluate the repair capability of the latest models by varying the information we provide (i.e., test diagnostic feedback) and varying the granularity of the task (function level vs statement level).

B. EMPIRICAL STUDIES OF PROGRAM REPAIR

Several empirical studies have been conducted to evaluate program repair techniques in their efficacy [33], performance [34], trust [35], and overfitting [36], [37] aspects. Ye et al. [37] conducted an empirical study using 10 pre-LLM program repair tools for Java programs and found overfitting to the respective test suites to be an issue. Similarly, Durieux et al. [36] conducted a large-scale evaluation using 11 APR tools to fix 2,141 Java programs in five different repair benchmarks. Their experiments showed that repair tools do not generalize across benchmarks, highlighting the problem of benchmark overfitting in repair techniques. Liu et al. [33] conducted another study on the efficiency of repair tools using 16 Java repair tools. Their study highlighted the performance inefficiencies in repair tools, which predominantly generated a large number of incorrect patches despite their overall repair capabilities. Noller et al. [35] performed a small-scale experiment using five program repair tools and 60 C programs. Their findings reported that different experiment configurations lead to different results, highlighting the importance of fair comparison under a uniform setup. A similar study was conducted by Zhong et al. [34] for learning-based repair tools, highlighting the importance of a unified experimental setup with respect to the training data and evaluation metrics used for comparisons.

More recently, Xia et al. [38] conducted an extensive evaluation using 9 LLMs and five repair benchmarks covering multiple languages under different repair configurations. Their findings highlighted the benefits of using LLMs for APR, with respect to the quality of the generated patches and the performance improvements compared to non-LLM-based APR tools. Several other studies [21], [23] investigated fine-tuned coding models for program repair using multiple benchmarks, showing they can outperform non-LLM-based repair tools. Ouyang et al. [39] studied 11 repair tools including traditional techniques, NMT-based, and LLM-based repair tools. Their findings further emphasized the benefits of using LLM-based repair tools, which showed less susceptibility to overfitting and better generalization

across benchmarks. Unlike existing work, this study performs an extensive analysis using the most recent 13 LLMs including the most advanced reasoning models across various experiment configurations. We focus on the generalizability of LLMs across different languages, repair configurations and patch complexities. Additionally, we revisit the impact of accurate fault localization on program repair with respect to the usage of the most advanced reasoning LLMs.

III. STUDY METHODOLOGY

The primary objective of this work is to provide a comprehensive empirical evaluation of current LLMs' program repair capabilities. This study focuses on single-prompt, zero-shot LLM-based APR as a controlled baseline. We deliberately exclude agentic, iterative, and multi-step repair workflows in order to isolate the fundamental repair capabilities of LLMs across languages and configurations – a necessary prerequisite for understanding and improving more complex systems. Our findings are thus intended as empirical groundwork for future agentic and pipeline-based APR research.

The rest of this section is structured as follows. We first discuss the four research questions addressed in our study (Section III-A), followed by the 13 LLMs we evaluate (Section III-B) and the 4 benchmark datasets in different programming languages (Section III-C). We outline further technical details – the evaluation protocol, technical setup and prompt design – in Sections III-D to III-F.

A. RESEARCH QUESTIONS

To assess the accuracy of LLM-based program repair across different influencing factors, we formulate the following research questions:

- RQ1* How well does LLM-based APR generalize across different programming languages?
- RQ2* How is LLM-based repair impacted by fault localization (FL), particularly when paired with a non-ML localization of faulty code segments?
- RQ3* How does LLM-based repair perform across varying levels of patch complexity?
- RQ4* How do open LLMs compare to closed ones in the context of APR?

1) RQ1—GENERALIZATION ACROSS PROGRAMMING LANGUAGES

To answer this question, we apply LLM-based APR on repair benchmarks for Java, Python, PHP, and JavaScript and assess the quality of the generated patches. As part of this question, we also investigate whether one model appears superior across all programming languages, or whether different models work best on specific languages (which would suggest a language-specific model selection). To investigate this question, we use two prompt designs (see Section III-F); (1) a basic prompt including only the code passage to be repaired, and (2) an extended prompt which also includes additional information from failing tests' error messages.

2) RQ2—IMPACT OF FAULT LOCALIZATION (FL)

Traditionally, APR pipelines feature a fault localization (FL) step (in which a faulty code subsegment is identified) and a repair step (in which the subsegment is fixed). LLMs could either follow this approach and take localization hints from external systems such as FLACOCO [40] using spectrum-based fault localization [41], or operate in an end-to-end fashion, fixing a given chunk of code without explicitly localizing faulty subsegments. So far, there are no conclusive research results as to what extent LLMs benefit from localization, with a lot of research assuming perfect localization to be given [13], [14], [20], [21], [22], [23], others focusing on end-to-end operation [42], and only few recent comparisons of both with inconclusive results so far [22], [43]. To this end, we investigate whether current LLMs benefit from localization signals at code line granularity.

3) RQ3—PATCH COMPLEXITIES

To better understand LLMs' repair capabilities, we explore our results with respect to various patch complexities: single-line, single-hunk, and multi-hunk bugs (i.e., those which span multiple, separate locations within the same function). While we can expect that LLMs can repair single-line bugs, it is still unclear how they perform on multi-hunk ones.

4) RQ4—OPEN VS CLOSED LLMs

In our last question, we focus on the differences in APR accuracy between open LLMs and closed LLMs. While closed models have been found to offer superior performance on the majority of natural language understanding (NLU) tasks in the past (e.g., [12]), open LLMs offer the benefit of being operational locally, thus allowing for better reproducibility of results and supporting technological sovereignty and democratization. Therefore, we track the progress of both categories of LLMs over time, to investigate whether open models are catching up with their closed counterparts.

B. MODELS

The goal of our study was to compare a broad range of current frontier models, i.e. models that perform exceptionally well on code-related tasks. To investigate the trade-off between cost and performance, we included both, closed or proprietary models and open-source/open-weight models. Model selection was based on recent code-focused leaderboards (Aider's polyglot leaderboard,² BigCodeBench,³ and RepairBench⁴). In total, we evaluated 13 code language models (CLMs).

Of the closed models, we included OpenAI's *GPT-4o*, a high-performance model, and *o3-mini*, a lightweight yet capable variant. From Anthropic, we selected *Claude 3.7 Sonnet* for its strong reasoning abilities, along with the smaller *Claude 3.5 Haiku*. Google's *Gemini 1.5 Pro* and the speed-optimised *Gemini 2.0 Flash* concluded the set

of proprietary models. For the open models, we included *CodeLlama 13B* and *70B* from Meta, both specifically trained for coding tasks. We further evaluated *Llama 3.3*, a more recent multi-purpose LLM with competitive performance. From DeepSeek, we included *DeepSeek Coder 33B* and *DeepSeek R1* in a distilled variant aligned with *Llama 70B*. We also evaluated Alibaba's *Qwen 2.5 Coder 33B* for its multilingual coding strengths, and *CodeGemma 7B*, Google's lightweight open coding model.

C. BENCHMARKS

Our cross-language study uses defect datasets in four widely used programming languages, selected based on the following criteria: First, the benchmarks should cover a diverse range of common programming languages. Second, they must consist of real bug fixes in real-world programs, excluding artificial data, toy programs, or competition entries. Third, we require code samples in the benchmarks to be executable, i.e. include a test suite with at least one relevant test case per bug, exposing the faulty behavior. Fourth, we only use benchmarks that provide a ground-truth patch written by a human developer. Finally, the benchmarks should offer a sufficient number of bugs that are still reproducible today. As Zhu et al. [44] have shown, reproducibility is often a challenge in executable software defects datasets, as legacy versions of software artifacts are prone to breakage due to missing/deprecated dependencies or external resources.

Similar to prior studies [13], [38], [39], we identified bugs of comparable complexity within the benchmarks to enable fair comparisons across programming languages. To do so, we evaluated single-*function* fixes within the benchmarks, meaning we considered only those bugs that could be resolved by modifications within a single function. Unlike many previous studies, however, we did not restrict our analysis to single-*line* [5], [45] or single-*hunk* fixes [13], that is, bugs that can be resolved by modifying a single contiguous piece of code.

Based on these criteria, we selected the following benchmarks in Java, Python, JavaScript, and PHP.

1) DEFECTS4J V3.0 [24]

This curated collection of reproducible bugs, supported by a robust infrastructure, has been widely adopted in software engineering research. Most prior studies on LLM-based APR were performed on Defects4J [24], thus enabling a direct comparison with prior work. It contains 854 bugs from 17 open-source Java projects. For our first research question, we used the 264 single-hunk bugs that could be reproduced.

2) BUGSINPY [26]

This benchmark dataset comprises 493 real bugs from 17 open-source Python projects across diverse domains, including machine learning, developer tools, scientific computing, and web frameworks. Unlike Defects4J, which distinguishes between active and deprecated bugs

²<https://aider.chat/docs/leaderboards/>

³<https://huggingface.co/spaces/bigcode/bigcodebench-leaderboard>

⁴<https://repairbench.github.io/>

(i.e. those no longer reproducible with current Java versions), BugsInPy does not track the reproducibility status of its bugs. Aguilar et al. [46] evaluated the reproducibility of the dataset three years after its initial release and found that only 67% of the expected results could be reproduced in the unmodified BugsInPy dataset. Of these 337 reproducible bugs, 82 were single-hunk bugs, which we used in our first experiment.

3) BUGSJS [25]

BugsJS is a collection of JavaScript bugs, collected from 10 widely used server-side JavaScript projects based on Node.js. BugsJS features 453 actual bugs that have been verified manually. Similar to BugsInPy, many of the BugsJS subjects suffer from reproducibility issues. Out of the original 139 single-hunk bugs, we identified 72 reproducible cases with at least one relevant test case.

4) BUGSPHP [27]

BugsPHP is a benchmark dataset of PHP bugs in real-world applications consisting of a training and test dataset. We use BugsPHP's test split, which contains 513 bug-fixing commits from 15 popular open-source PHP projects covering the time period from Jan 2020 to Mar 2023. Each commit is equipped with at least one relevant failing test case. For our initial experiment, we identified 129 reproducible single-hunk bugs.

5) BENCHMARK FILTERING AND EXPERIMENTAL SUBSETS

We use two different experimental setups, sharing a common pre-filtering step but differing in scope, for our experiments. First, we restrict all benchmarks to bugs whose fix is contained within a single function and verify reproducibility in the provided Docker environment: the buggy version must fail the relevant test cases and the developer patch must make them pass. Bugs failing this check are discarded.

RQ1 uses the *single-hunk* subset of the reproducible single-function bugs, matching the experimental setting used in most prior APR work and maximizing comparability across benchmarks. The resulting subsets – 264 (Defects4J), 82 (BugsInPy), 72 (BugsJS), and 129 (BugsPHP), totaling 547 samples – form the *RQ1* evaluation set.

RQs 2–4 use the broader pool of reproducible single-function bugs, which also includes multi-hunk bugs, to cover a wider range of patch complexities. To keep token costs manageable, we draw a stratified sample of 100 bugs per benchmark, preserving each benchmark's natural distribution of single-line, single-hunk, and multi-hunk bugs (Table 1), with the constraint that every repository from the original benchmark is represented. The resulting distributions are summarized in Table 5; the selected bug IDs are provided in the supplementary material [47].

D. EVALUATION PROTOCOL

For a meaningful comparison between models, we select metrics that capture key aspects of APR. The goal is to generate patches that fix a bug without introducing new

issues. A valid patch must be syntactically correct, parseable, and compilable, as well as functionally correct, meaning it resolves the original issue while preserving existing functionality. When it comes to evaluating these criteria, two issues arise:

1) TEST-OVERFITTING IN APR

A challenge in execution-based evaluation of APR is test overfitting [48]: A patch may pass all tests but still fail to implement the intended functionality. To address this issue, the literature distinguishes between *plausible* patches (those that pass the test suite) and *correct* patches (those that satisfy the intended requirements). Since a manual review of patches is infeasible at scale, various proxy metrics, such as Syntactic Equivalence with the developer patch (SYE) and Trivial Compiler Equivalence (TCE), where the patch compiles to equivalent bytecode, have been proposed to approximate patch correctness [39]. However, while these metrics display some correlation with correct patch quality, they have not outperformed plausibility consistently as predictors. Ouyang et al. [39] examined the relationship between manually verified correct patches and less expensive metrics (namely, plausibility, TCE, and SYE), and found that TCE and plausibility strongly correlate with patch correctness. A more recent study [49] analyzed test overfitting across several APR approaches and concluded that overfitting may be less problematic than previously assumed. Based on these findings, we adopt plausibility (i.e., the passing of test cases) as our primary evaluation criterion.

2) NON-DETERMINISM OF LLMs

Due to the inherent non-determinism of LLM-generated output, results may vary across runs, posing a challenge for achieving reliable, reproducible results [50], [51], [52]. To mitigate the problem, we sample multiple patches per bug from the same LLM, and report the percentage of bugs for which at least one generated patch passes all tests. More precisely, we adopt the approach by Chen et al. [53], generating $n \geq k$ samples per task and computing an unbiased estimator of $pass@k$ as:

$$pass@k := \mathbb{E}_{Problems} \left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \right] \quad (1)$$

where c is the number of plausible samples among n generations. We report $pass@k$ for $k = 1, 5$. Based on the standard deviation analysis of $pass@1$ for LLM-based APR [28], we use $n = 15$ as a reasonable, yet manageable number of generations. We distributed the generation across three independent runs, each generating five candidate patches using each respective model's standard setting with a temperature of 1.0.

3) SIGNIFICANCE TESTING

To assess whether observed differences in $pass@k$ between models are statistically meaningful, we apply the Wilcoxon

TABLE 1. Fraction of single-file, single-function, single-hunk and single-line bugs in the benchmarks.

Benchmark	all bugs	single-file	single-function	single-hunk	single-line
Defects4J	845	703	471	276	151
BugsInPy	493	280	189	132	52
BugsJS	513	272	207	139	69
BugsPHP	452	406	282	178	80

TABLE 2. Model identifiers and API access period.

Model	Provider	Identifier / Checkpoint	API access
Claude 3.7 Sonnet	Anthropic	claude-3-7-sonnet-20250219	Mar.–May 2025
Claude 3.5 Haiku	Anthropic	claude-3-5-haiku-20241022	Mar.–May 2025
Gemini 1.5 Pro	Google	gemini-1.5-pro-001	Jan.–May 2025
Gemini 2.0 Flash	Google	gemini-2.0-flash	Feb.–May 2025
GPT-4o	OpenAI	gpt-4o-2024-05-13	Jan.–May 2025
o3-mini	OpenAI	o3-mini-2025-01-31	Feb.–May 2025
DeepSeek R1 (dist.)	DeepSeek	DeepSeek-R1-Distill-Llama-70B	–
Meta Llama 3.3 70B	Meta	Llama-3.3-70B-Instruct	–
CodeLlama 70B	Meta	CodeLlama-70b-Instruct-hf	–
CodeLlama 13B	Meta	CodeLlama-13b-Instruct-hf	–
DeepSeek Coder 33B	DeepSeek	deepseek-coder-33b-instruct	–
Qwen 2.5 Coder 32B	Alibaba	Qwen2.5-Coder-32B-Instruct	–
CodeGemma 7B	Google	codegemma-7b-it	–

signed-rank test at a significance level of $\alpha = 0.05$. Results are considered significant if the null hypothesis of equal performance can be rejected. In tables comparing model performance, the best result per language is shown in bold, and all results that are not significantly different from the best are underlined.

E. TECHNICAL SETUP

1) MODEL VERSIONS AND API ACCESS

Closed-source models were queried via their respective vendor APIs. Table 2 lists model identifiers and the period during which the APIs were accessed. Open models were loaded from Hugging Face checkpoints and executed on one to four NVIDIA RTX 6000 48 GB GPUs.

2) DECODING PARAMETERS

We used provider and model-card recommended default decoding settings, without tuning. For closed-source models, defaults were used as provided by each API (temperature = 1.0 for all models; top_p = 0.95 for Gemini models and 1.0 for others). For open models, Hugging Face default configurations were used unless otherwise specified (e.g., temperature = 0.6 and top_p = 0.95 for DeepSeek R1 Distill and Llama 3.3 70B, and temperature = 1.0 and top_p = 1.0 for other models).

We generated 15 candidate patches per bug by requesting $n=5$ completions per call and repeating each experiment three times. For Anthropic models, which do not support batched sampling, we issued 15 independent API calls per bug. We used fixed random seeds where supported by the inference framework; for API-based models where seed control is not exposed, runs are inherently stochastic.

3) PATCH EXTRACTION AND PREPROCESSING

From each model response, we extracted the last fenced code block (surrounded by triple backticks) using a regular

expression. Any model-generated import statements were removed, and the resulting function body was reintegrated into the original source file for execution.

4) TEST EXECUTION ENVIRONMENT

Patch validation was performed inside the Cerberus framework [11], which isolates each benchmark subject into a Docker container. Only *relevant* test cases, i.e., those that fail on the buggy version and pass on the developer-supplied fix, were considered when computing $pass@k$. Pre-existing failures were excluded.

F. PROMPT DESIGN

The goal of this study is to evaluate the automated program repair capabilities of various large language models across different programming languages in a standardized comparison. Therefore, we use a consistent prompt design across all models and benchmarks. All prompts are zero-shot, i.e., they do not include repair examples that could bias performance, and non-iterative, meaning that each model processes a sample/query without follow-up interactions. This design isolates the effects of prompt information (e.g., test cases and fault localization) and ensures that differences in repair performance reflect the models themselves rather than more sophisticated repair workflows. Table 6 summarizes the four prompt configurations used in our experiments. The *Base* prompt contains only the buggy function. The *Test* prompt augments it with the failing test case(s) and error message. The *Perfect Line-Level FL* prompt augments the *Base* prompt with line-level fault localization hints. Finally, the *Automated FL* prompt replaces the known buggy function with the three most suspicious functions identified by a spectrum-based FL tool, representing a realistic repair scenario.

TABLE 3. Overview of the four prompt configurations.

Prompt	Code provided	Additional information
Base	Buggy function	–
Test	Buggy function	Failing tests + errors
Perfect FL (LL)	Buggy function	Line-level TODO hints
Automated FL	Top-3 suspicious functions	–

1) "BASE" PROMPT

The first prompt is used to evaluate LLMs' out-of-the-box APR capabilities. It consists of an initial system directive, followed by the code of the buggy method. An example is shown in Listing 1. While previous work has demonstrated that including the failing test case and the resulting error message can improve the bug-fixing performance of

```

System: You are an automatic program repair tool. Your
task is to fix the provided buggy code written in
java.

User: The following code snippet contains a bug:

```java
< FUNCTION CODE >
```

Please provide a fixed version of the buggy function, and
only that function without additional imports. Make
sure the code is between ``` and ```.

```

LISTING 1. Basic prompt template, example for Java.

LLMs [14], [28], we intentionally adhere to this minimal prompt format to simulate real-world APR scenarios where such test information may be unavailable. The specific prompt is largely adopted from prior work [28], with no prompt engineering or optimization applied to prevent model-specific bias.

2) "TEST" PROMPT

The second, more informative prompt incorporates additional information from the failed test cases. It extends the base prompt with the block shown in Listing 2. In addition to the buggy function, the prompt includes the names and source code of the failing test cases along with the corresponding error messages. Specifically, we adopt the prompt template from RepairBench [42] to ensure comparability.

```

The code fails the following tests:

Test <TEST FILE NAME>::<TEST NAME> :

```java
< TEST CODE >
```

The error message is:

```
< ERROR MESSAGE >
```

```

LISTING 2. Test prompt template (Java).

3) "PERFECT LINE-LEVEL LOCALIZATION" (PERFECT FL/LL) PROMPT

To investigate the impact of fault localization (FL) on APR performance, we simulated *perfect line-level localization*, by enriching the *base* prompt with explicit hints marking buggy code locations. In preliminary experiments, we evaluated different methods for integrating such line-level localization information into APR prompts, drawing from prior work [22], [38] and developer intuition. The most effective method was inserting a simple comment "TODO: Fix here:" directly before each buggy code snippet, as shown in Listing 3. The end of the buggy code segment was not marked.

4) "AUTOMATED (FUNCTION-LEVEL) LOCALIZATION" (AUTOMATED FL) PROMPT

We also investigate a more realistic setup by drawing the candidate locations from an automated fault localization tool, thus avoiding the assumption of perfect fault localization. In our experiments on Defects4J, we employed the spectrum-based tool FLACOCO [40]. For each defect, the top three suspicious functions identified by FLACOCO were included in the *base* prompt, as illustrated in Listing 4. If a candidate location could not be parsed into a valid function, it was replaced by the next highest-ranked valid function. This configuration mirrors practical scenarios where fault localization is imperfect yet still informative.

```

```java
// Program to fix:
<M extends Map<String, String>> M putIn(final M map)
{
 for (final Entry<String, Integer> entry : mapping
 .entrySet()) {
 // TODO: fix here
 final int col = entry.getValue().intValue();
 map.put(entry.getKey(), values[col]);
 }
 return map;
}
```

```

LISTING 3. Perfect line-level fault localization prompt template (Java).

```

System: You are an automatic program repair tool. Your
task is to fix the provided buggy code written in
java.

User: I have a buggy program that does not pass all the
tests.
Here are some candidate functions that might need fixing:

```java
< FUNCTION CODE >
```

```java
< FUNCTION CODE >
```

```java
< FUNCTION CODE >
```

Please identify the buggy function and provide a fixed
version. Return only the fixed function without
additional imports. Make sure the code is between
``` and ```.

```

**LISTING 4.** Automated fault localization prompt template (Java).

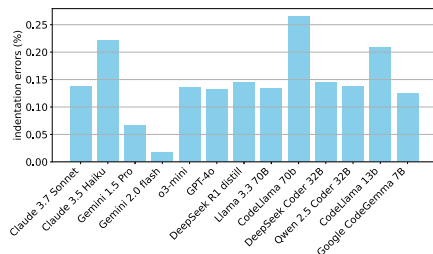
## IV. RESULTS AND ANALYSIS

### A. RQ1: APR PERFORMANCE OF CURRENT LLMS ACROSS DIFFERENT PROGRAMMING LANGUAGES

Our first research question aims at comparing LLMs' language- and prompt-specific APR capabilities. We first present an overall impression across all 13 models and four languages (Section IV-A1), then investigate the influence of adding test information to the prompt (Section IV-A2), and

**TABLE 4.** *pass@1* and *pass@5* results using a simple prompt that contains only the buggy function. For each language, the best result is shown in bold, and all results not significantly different from the best according to the Wilcoxon test ( $\alpha = 0.05$ ) are underlined.

| Model               |     | Java          |               | JavaScript    |               | PHP           |               | Python        |               | average       |               |
|---------------------|-----|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|
|                     |     | <i>pass@1</i> | <i>pass@5</i> | <i>pass@1</i> | <i>pass@5</i> | <i>pass@1</i> | <i>pass@5</i> | <i>pass@1</i> | <i>pass@5</i> | <i>p@1</i>    | <i>pass@5</i> |
| Claude 3.7 Sonnet   |     | <b>22.69%</b> | <b>34.90%</b> | 11.71%        | 24.23%        | 13.60%        | 23.28%        | 7.59%         | 13.64%        | 13.97%        | 24.06%        |
| Claude 3.5 Haiku    |     | 17.74%        | 26.50%        | <b>14.48%</b> | <b>24.14%</b> | 13.44%        | 19.64%        | <u>7.28%</u>  | <u>10.49%</u> | 13.24%        | 20.30%        |
| Gemini 1.5 Pro      |     | <u>20.45%</u> | <u>33.83%</u> | <u>11.14%</u> | <u>19.73%</u> | <u>17.44%</u> | <u>26.92%</u> | 6.36%         | <u>13.30%</u> | <b>14.04%</b> | <u>23.71%</u> |
| Gemini 2.0 Flash    |     | 18.83%        | 27.05%        | 10.34%        | 16.15%        | 14.39%        | 22.06%        | <b>12.51%</b> | <b>18.03%</b> | 14.21%        | 21.11%        |
| OpenAI o3-mini      |     | <u>19.95%</u> | 27.96%        | 9.20%         | <u>18.10%</u> | 12.63%        | 17.56%        | <u>8.92%</u>  | <u>11.50%</u> | <u>13.13%</u> | 19.36%        |
| OpenAI GPT-4o       |     | 16.33%        | 27.77%        | <u>10.57%</u> | <u>23.53%</u> | 14.55%        | 23.96%        | 3.94%         | 12.56%        | 11.41%        | <u>22.11%</u> |
| DeepSeek R1 (dist.) | 70B | 17.57%        | 31.35%        | 9.55%         | 21.66%        | <b>18.43%</b> | <b>33.43%</b> | 6.56%         | 15.63%        | 13.24%        | <b>25.85%</b> |
| Meta Llama 3.3      | 70B | 15.12%        | 21.84%        | 9.09%         | 14.96%        | 9.28%         | 16.48%        | 5.33%         | 9.29%         | 9.71%         | 15.85%        |
| Meta CodeLlama      | 70B | 1.52%         | 4.86%         | 4.87%         | 14.31%        | 2.41%         | 10.38%        | 1.13%         | 3.56%         | 1.69%         | 6.27%         |
| DeepSeek Coder      | 32B | 6.34%         | 15.61%        | 4.28%         | 12.03%        | 6.93%         | 17.28%        | 3.51%         | 9.19%         | 5.32%         | 13.47%        |
| Qwen 2.5 Coder      | 32B | 14.67%        | 22.30%        | 7.93%         | 15.71%        | 10.89%        | 14.95%        | 3.79%         | 3.79%         | 9.33%         | 14.88%        |
| Meta CodeLlama      | 13B | 4.27%         | 12.50%        | 6.43%         | 21.43%        | 4.28%         | 15.98%        | 1.95%         | 7.63%         | 4.25%         | 14.40%        |
| Google CodeGemma    | 7B  | 6.70%         | 16.32%        | 6.50%         | 19.88%        | 6.37%         | 18.40%        | 3.49%         | 9.48%         | 6.02%         | 16.71%        |



**FIGURE 1.** Frequency of indentation errors for the Python benchmark. Notably, most models struggle with indentation issues, with the Gemini models being the exception.

finally address the question whether LLMs tend to perform well on specific languages (Section IV-A3).

### 1) OVERALL RESULTS WITH BASE PROMPT

Table 4 summarizes the performance of all models across the four benchmarks, reporting *pass@1* (the success rate when generating a single fix) and *pass@5* (success among five generated candidates). Models were evaluated on all reproducible single-hunk bugs using the basic prompt, which contained only the buggy function. The upper section of the table lists closed models, while the lower section presents open models. Most models achieve the best performance on Defects4J, only for DeepSeek R1 (dist.) we see a slightly better performance on BugsPHP with +0.86% *pass@1* and +2.08% *pass@5*. However, performance varies across languages, and no model consistently outperforms others across all benchmarks. For Java, Claude 3.7 Sonnet performs best with 22.69% *pass@1*, and 34.90% *pass@5*. For Javascript, best results are achieved by Claude 3.5 Haiku with 14.48% *pass@1* and 24.14% *pass@5*. On BugsPHP, the distilled version of DeepSeek R1 (distilled to Llama 3 70B) achieves the best *pass@1* at 18.43% and *pass@5* at 33.43%, while for Python Gemini 2.0 Flash shows the best performance, reaching 12.51% *pass@1* and 18.03% *pass@5*.

A notable observation is the surprisingly poor performance of all models on the Python benchmark. Further examination

of the generated patches reveals that this is largely due to indentation issues in the generated patches. While other programming languages are more forgiving, Python's off-side syntax rule leads to execution failures when code is misaligned. Figure 1 illustrates the percentage of indentation errors in the generated patches. Notably, the Gemini models, particularly Gemini 2.0 Flash, produce significantly fewer indentation errors than other models. This is reflected in the comparatively good performance on the Python benchmark. This finding suggests that when applying APR with LLMs to Python, practitioners should consider integrating a linting or formatting step into their post-processing pipeline to mitigate syntax-related issues.

### 2) INFLUENCE OF TEST INFORMATION

Next, we compare the *base* prompt, which contains only the faulty method and a generic repair instruction, with the *test* prompt, which extends the base prompt to include the names, source code, and error messages of failing test cases. For this and all subsequent experiments, we used the stratified 100-bug sample per benchmark described in Section III-C, which broadens the scope from single-hunk to single-function bugs; the complexity breakdown is shown in Table 5.

Table 6 includes a comparison of both prompts for the most promising models from Table 4. These include the four best-performing closed-source models (averaged across all languages) and, among the open models, the best large model and the best medium-sized model. Please note that the figures for the base prompt differ from Table 4, as the evaluation data is different. We observe that adding test information (rows *Test*) yields substantial improvements compared to the base prompt, reaching up to almost +47% *pass@1* (DeepSeek R1 (dist.) on Python). This confirms results of prior work [14], [28] for the most recent models, and also across multiple programming languages. While all models pick up on the additional test information, the effect appears to be stronger for certain models (particularly for OpenAI's o3-mini). Notably, the improvements by test information

show less variation between the different *languages*: Across all four benchmarks, test information consistently improves results, with Python (+34.7% *pass@1*, averaged over all models) showing the strongest improvements.

**TABLE 5. Distribution of multi-hunk, single-hunk and single-line bugs in the evaluation subset. Categories are exclusive.**

| Benchmark | all | multi-hunk | single-hunk | single-line |
|-----------|-----|------------|-------------|-------------|
| Defects4J | 100 | 41         | 26          | 33          |
| BugsInPy  | 100 | 30         | 42          | 28          |
| BugsJS    | 100 | 32         | 34          | 34          |
| BugsPHP   | 100 | 36         | 35          | 29          |

### 3) LANGUAGE SPECIALIZATION

Table 4 has already suggested that model accuracy varies between programming languages, with four different LLMs performing best on the four language-specific benchmarks. Since model complementarity and language specialization are important issues for practitioners, we investigate these issues further. First, the Venn diagrams in Figure 2 displays what model combinations perform well on which subsets of cases. Here, all cases from all benchmarks were joined, and a model is defined to “solve” an issue if at least one of its  $n = 15$  generations is plausible. For each model, we find cases that can only be fixed by this particular model. This effect holds for all three prompt types “basic”, “test”, and “line-level localization (LL)” (see Section III-F). To further investigate how different models complement each other, we evaluate, how *pass@k* changes when the  $k$  candidates are derived from two models rather than just one. To this end, we calculate  $p_{\text{combined}}@k$  when we draw a fraction of  $\alpha \in [0, 1]$  of the  $n = 15$  generations from one model and the remaining  $(1-\alpha) \cdot n$  generations from a second model:

$$p_{\text{combined}}@k := \mathbb{E}_{\text{Problems, Simulations}} \left[ 1 - \frac{\binom{n-c_{\text{combined}}}{k}}{\binom{n}{k}} \right] \quad (2)$$

where  $c_{\text{combined}}$  represents the number of correct patches among the  $n$  combined generations. To avoid rounding errors, we approximate this process by simulation: for each  $\alpha$ , we repeatedly (100 times in our experiments) sample  $\alpha \cdot n$  generations from the first model and  $(1-\alpha) \cdot n$  generations from the second model, and determine  $c_{\text{combined}}$  as the number of plausible patches in the combined set. We then estimate  $p_{\text{combined}}@k$  as the expectation over all simulation runs. Figure 3 shows  $p_{\text{combined}}@k$  for ensembles that pair the best-performing model in each benchmark with one of the next four best models, evaluated across different mixing fractions  $\alpha$ , using the *test* prompt. In most cases, combining two models improved performance over the best individual model, with more pronounced gains at higher values of  $k$ .

For instance, with the *test* prompt *pass@5* improved in 14 out of 16 experiments. Note that, since  $\alpha$  corresponds to the models’ mixing ratio, the rightmost point in each plot corresponds to sampling only from the best model,

and the leftmost point to sampling only from the second model. We found *pass@5* to improve for (1) JavaScript from 68.00% (o3-mini) to 71.68% (o3-mini & DeepSeek R1 (dist.)), (2) Java from 62.60% (o3-mini) to 65.73% (o3-mini & Claude 3.7), (3) PHP from 58.44% (o3-mini) to 63.12% (o3-mini & DeepSeek R1 (dist.)), and (4) Python from 63.87% (DeepSeek R1 (dist.)) to 65.48% (Claude 3.7 & DeepSeek R1 (dist.)).

Table 7 summarizes the results, reporting for each benchmark and prompt the best single model and the best two-model ensemble, constructed by combining two-thirds of the generations from the top-performing model with one-third from a complementary model. Overall, the results suggest there are substantial differences in LLM repair performance between languages, and demonstrate that LLMs can effectively complement each other on repair tasks.

### B. RQ2: IMPACT OF FAULT LOCALIZATION ON LLM-BASED APR

Current work in APR lacks a standardized approach regarding assumptions about fault localization [54]. While end-to-end repair –including fault localization (FL), patch generation, and prioritization – is inherently more challenging than patch generation in isolation, most empirical evaluations of neural program repair assume *perfect fault localization* as suggested by Liu et al. [55], usually derived directly from developer fixes. This assumption enables objective evaluation but also introduces a disconnect from realistic use cases where fault localization is noisy and imprecise. The impact of this assumption remains largely underexplored, raising concerns about localization bias [55], [56], which may lead to an overestimation of model performance under idealized conditions. Therefore, we investigate two aspects: the impact of FL granularity and the effect of noisy, tool-generated FL on patch generation.

#### 1) GRANULARITY OF FAULT LOCALIZATION

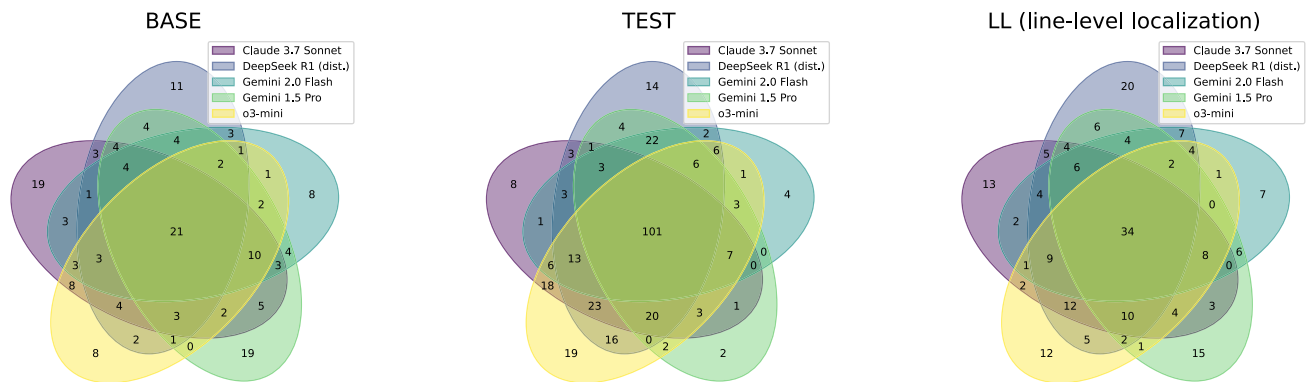
We compare the base prompt – which includes (only) the faulty method and thus simulates perfect method-level FL – with the Line-level Localization (LL) prompt, which includes a hint about the particular code lines to fix. Table 6 displays results for both prompts across the most promising models from RQ1. We see that line-level localization – while improving results – appears to add less value compared to test errors for (almost) all languages-model combinations. Also, we observe that the improvements with line-level localization appear to be inconsistent: Particularly, in the PHP benchmark, the accuracy deteriorates for 4 out of 6 LLMs when introducing line-level localization. This suggests that future research on localization in APR should evaluate across multiple programming languages.

#### 2) AUTOMATED FAULT LOCALIZATION

Assuming perfect (method- or line-level) localization may be a questionable assumption in practice, where the scope

**TABLE 6.** *Pass@1* and *pass@5* with varying prompt ingredients: the *Base* prompt includes only the buggy function; *Test* includes failing test case information; *LL* adds line-level fault localization via comments.

|                   | Prompt | Java   |        | JavaScript |        | PHP    |        | Python |        |
|-------------------|--------|--------|--------|------------|--------|--------|--------|--------|--------|
|                   |        | pass@1 | pass@5 | pass@1     | pass@5 | pass@1 | pass@5 | pass@1 | pass@5 |
| Claude 3.7 Sonnet | Base   | 19.02% | 32.04% | 8.39%      | 19.12% | 14.08% | 21.65% | 5.91%  | 10.71% |
|                   | Test   | 45.33% | 59.16% | 42.39%     | 56.34% | 39.06% | 50.25% | 29.17% | 36.76% |
|                   | LL     | 27.78% | 38.13% | 20.90%     | 29.03% | 14.91% | 22.52% | 8.89%  | 15.18% |
| Gemini 1.5 Pro    | Base   | 19.36% | 28.78% | 8.39%      | 14.70% | 21.11% | 29.13% | 6.67%  | 12.12% |
|                   | Test   | 33.86% | 41.75% | 26.39%     | 32.41% | 31.76% | 41.78% | 38.89% | 47.41% |
|                   | LL     | 20.36% | 26.24% | 20.35%     | 27.48% | 14.36% | 21.86% | 9.36%  | 16.74% |
| Gemini 2.0 Flash  | Base   | 10.27% | 15.84% | 9.10%      | 13.90% | 17.87% | 24.39% | 11.35% | 16.27% |
|                   | Test   | 33.56% | 41.59% | 28.40%     | 39.05% | 27.96% | 37.52% | 39.79% | 48.14% |
|                   | LL     | 14.25% | 20.87% | 18.53%     | 23.31% | 13.75% | 17.92% | 13.62% | 21.43% |
| o3-mini           | Base   | 15.02% | 23.61% | 6.32%      | 13.01% | 13.33% | 16.49% | 6.24%  | 8.04%  |
|                   | Test   | 48.50% | 62.60% | 51.44%     | 68.00% | 45.04% | 58.44% | 35.69% | 43.42% |
|                   | LL     | 22.81% | 33.21% | 18.92%     | 28.89% | 16.01% | 22.68% | 9.12%  | 12.60% |
| DeepSeek R1 dist. | Base   | 17.16% | 29.82% | 7.71%      | 18.95% | 17.50% | 29.33% | 6.27%  | 14.94% |
|                   | Test   | 32.81% | 47.70% | 34.79%     | 56.71% | 38.97% | 57.35% | 40.07% | 63.87% |
|                   | LL     | 18.00% | 29.67% | 12.58%     | 29.88% | 17.39% | 28.67% | 6.46%  | 14.91% |
| Qwen 2.5 Coder    | Base   | 7.41%  | 12.84% | 5.11%      | 10.73% | 12.78% | 16.04% | 2.72%  | 4.85%  |
|                   | Test   | 16.13% | 28.86% | 24.00%     | 34.86% | 28.24% | 40.00% | 17.01% | 28.98% |
|                   | LL     | 11.81% | 17.96% | 8.96%      | 16.12% | 10.82% | 17.52% | 3.58%  | 5.95%  |



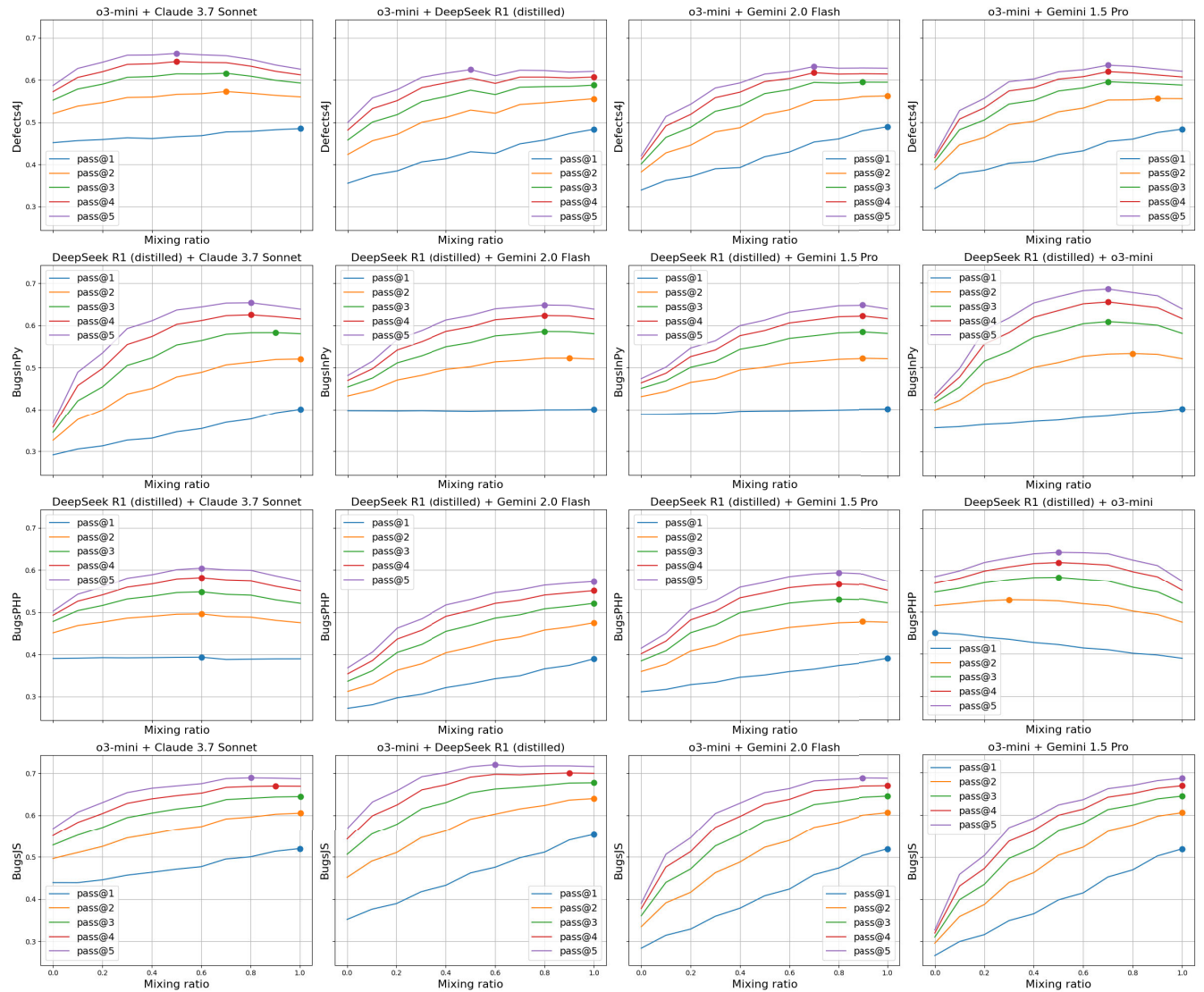
**FIGURE 2.** Bug fixes per model, determined for 400 single-function bugs in four programming languages.

**TABLE 7.** Comparison between the strongest standalone models and most effective model combinations ( $\frac{2}{3}$  of the generations from the top model and  $\frac{1}{3}$  from a complementary model).

|            | Prompt | Best single model   | pass@5 | Best ensemble                        | pass@5 |
|------------|--------|---------------------|--------|--------------------------------------|--------|
| Java       | Base   | Gemini 1.5 Pro      | 28.78% | Gemini 1.5 Pro & Claude 3.7          | 35.30% |
|            | Test   | o3-mini             | 62.60% | o3-mini & Claude 3.7                 | 65.73% |
|            | LL     | Claude 3.7          | 38.13% | Claude 3.7 & DeepSeek R1 (dist.)     | 43.30% |
| JavaScript | Base   | DeepSeek R1 (dist.) | 18.95% | DeepSeek R1 (dist.) & Claude 3.7     | 21.34% |
|            | Test   | o3-mini             | 68.00% | o3-mini & DeepSeek R1 (dist.)        | 71.68% |
|            | LL     | Claude 3.7          | 19.12% | Claude 3.7 & DeepSeek R1 (dist.)     | 35.76% |
| Python     | Base   | Gemini 2.0          | 16.27% | Gemini 2.0 & Claude 3.7              | 18.39% |
|            | Test   | DeepSeek R1 (dist.) | 63.87% | Claude 3.7 & DeepSeek R1 (dist.)     | 65.48% |
|            | LL     | Gemini 2.0          | 21.43% | Gemini 2.0 & o3-mini                 | 25.81% |
| PHP        | Base   | Gemini 1.5 Pro      | 29.13% | Gemini 1.5 Pro & DeepSeek R1 (dist.) | 32.61% |
|            | Test   | o3-mini             | 58.44% | o3-mini & DeepSeek R1 (dist.)        | 63.12% |
|            | LL     | DeepSeek R1 (dist.) | 28.67% | DeepSeek R1 (dist.) & Gemini 1.5 Pro | 30.93% |

of bugs may have to be narrowed down within large software projects [55]. To study the relationship between

fault localization and neural patch generation in a realistic setting, we use FLACOCO [40], a recent spectrum-based



**FIGURE 3.** Combined  $pass@k$  (using the test prompt) for ensembles of the best performing model per benchmark with a second model.

**TABLE 8.**  $pass@1$  results on Defects4J, using perfect function-/line-level and automated (top-3 spectrum-based) Fault Localization.

|                     | Perfect FL |        | Automated FL |
|---------------------|------------|--------|--------------|
|                     | Function   | Line   | Function     |
| Claude 3.7 Sonnet   | 19.02%     | 27.78% | 2.76%        |
| Gemini 1.5 Pro      | 19.36%     | 20.36% | 3.33%        |
| Gemini 2.0 Flash    | 10.27%     | 14.25% | 2.40%        |
| o3-mini             | 15.02%     | 22.81% | 3.64%        |
| DeepSeek R1 (dist.) | 17.16%     | 18.00% | 0.31%        |
| Qwen 2.5 Coder      | 7.41%      | 11.81% | 1.61%        |

localization tool, to identify the three most suspicious functions as potential locations for the fix. We then prompted the model using these candidate functions. As the tool currently supports only Java, this experiment was conducted on Defects4J (using the subselection of 100 samples). The results are presented in Table 8. The first two columns show the  $pass@1$  results using perfect fault localization.

In comparison,  $pass@1$  using automated fault localization in the third column shows a significant drop. The performance drop ranges from  $-7.83\%$  for Gemini 2.0 Flash to  $-16.20\%$  for Claude 3.7 Sonnet compared to the base prompt. This can be attributed to two main factors. First, the accuracy of the automated fault localization tool is lower than that of perfect fault localization. In our setup, the actual fix location appeared among the top-3 candidates for only 28 out of 100 faults. For the remaining cases, the prompt included only irrelevant code locations. We discovered only one such case for which two LLMs produced plausible patches, and even these could be attributed to test overfitting. Also, even when the correct function was identified, we observed a degradation in performance: on average,  $pass@k$  decreased by  $5.54\%$ , and in  $9.8\%$  of those cases, no plausible patch could be generated – despite the base prompt yielding plausible solutions within the 15 generations per model. Since automated line-level localization can be expected to be even

**TABLE 9.** Comparison of different patch complexities using failing test cases in the prompt. Best results per language and complexity are shown in bold, and results not significantly different from the best (Wilcoxon test,  $\alpha = 0.05$ ) are underlined.

|           |                     | Single-Line   |               | Single-Hunk   |               | Multi-Hunk    |               |
|-----------|---------------------|---------------|---------------|---------------|---------------|---------------|---------------|
|           |                     | pass@1        | pass@5        | pass@1        | pass@5        | pass@1        | pass@5        |
| Defects4J | Claude 3.7 Sonnet   | 56.54%        | 68.49%        | 44.24%        | 52.90%        | 39.28%        | 56.68%        |
|           | Gemini 1.5 Pro      | 42.93%        | 53.88%        | 28.67%        | 34.96%        | 31.75%        | 38.76%        |
|           | Gemini 2.0 Flash    | 44.62%        | 47.44%        | 25.71%        | 32.04%        | 31.38%        | 43.78%        |
|           | o3-mini             | <b>58.93%</b> | <b>69.37%</b> | <b>48.67%</b> | <b>66.15%</b> | <b>42.75%</b> | <b>57.37%</b> |
|           | DeepSeek R1 (dist.) | 50.26%        | 67.00%        | 25.15%        | 37.30%        | 26.03%        | 41.21%        |
|           | Qwen 2.5 Coder      | 19.75%        | 40.84%        | 13.64%        | 26.96%        | 15.50%        | 22.99%        |
| BugsJS    | Claude 3.7 Sonnet   | 52.59%        | 64.04%        | 41.33%        | 60.43%        | 35.96%        | 47.64%        |
|           | Gemini 1.5 Pro      | 50.37%        | 54.32%        | 23.01%        | 29.72%        | 12.11%        | 19.04%        |
|           | Gemini 2.0 Flash    | 43.70%        | 55.94%        | 22.80%        | 35.56%        | 22.11%        | 29.89%        |
|           | o3-mini             | <b>66.91%</b> | <b>82.70%</b> | <b>49.33%</b> | <b>68.56%</b> | <b>42.11%</b> | <b>57.10%</b> |
|           | DeepSeek R1 (dist.) | 54.78%        | 77.68%        | 27.59%        | 52.85%        | 27.62%        | 46.13%        |
|           | Qwen 2.5 Coder      | 44.69%        | 58.38%        | 14.44%        | 26.84%        | 16.84%        | 24.49%        |
| BugsInPy  | Claude 3.7 Sonnet   | 24.76%        | 32.93%        | 34.63%        | 44.03%        | 25.29%        | 30.37%        |
|           | Gemini 1.5 Pro      | <b>63.81%</b> | 71.67%        | 35.61%        | 45.00%        | 27.45%        | 35.32%        |
|           | Gemini 2.0 Flash    | 44.44%        | 51.47%        | <b>45.20%</b> | 53.21%        | 30.39%        | 39.97%        |
|           | o3-mini             | 34.29%        | 43.18%        | 40.00%        | 46.29%        | <b>31.37%</b> | 40.11%        |
|           | DeepSeek R1 (dist.) | 47.94%        | <b>73.00%</b> | <b>45.20%</b> | <b>71.08%</b> | 28.69%        | <b>49.11%</b> |
|           | Qwen 2.5 Coder      | 20.63%        | 34.67%        | 22.28%        | 36.16%        | 8.43%         | 16.80%        |
| BugsPHP   | Claude 3.7 Sonnet   | 43.14%        | 59.64%        | <b>45.42%</b> | 53.04%        | 25.90%        | 34.56%        |
|           | Gemini 1.5 Pro      | 37.71%        | 53.05%        | 30.21%        | 39.75%        | 25.64%        | 29.12%        |
|           | Gemini 2.0 Flash    | 36.19%        | 44.25%        | 31.87%        | 43.84%        | 12.05%        | 20.70%        |
|           | o3-mini             | <b>58.59%</b> | <b>76.00%</b> | 44.30%        | 57.26%        | <b>28.72%</b> | 37.55%        |
|           | DeepSeek R1 (dist.) | 43.92%        | 60.76%        | 42.15%        | <b>59.09%</b> | <b>28.72%</b> | <b>50.83%</b> |
|           | Qwen 2.5 Coder      | 28.57%        | 45.73%        | 35.21%        | 42.44%        | 19.23%        | 29.29%        |
| Average   |                     | 45.06%        | 57.06%        | 34.37%        | 45.98%        | 27.50%        | 38.11%        |

harder (and due to the mixed results with (perfect) line-level localization in Table 6), we did not explore automated line-level localization further.

Overall, our results for RQ2 suggest that practical, automated localization has a significant impact on APR results. Future APR research might consider including practice-oriented evaluations with localization tools, or even put an even stronger focus on integrated approaches of repair and localization, such as [57].

### C. RQ3: PATCH COMPLEXITY

We also test how repair accuracy varies for different levels of patch complexity. To do so, Table 9 reports pass@1 and pass@5 for all benchmarks (using the *test* prompt) separately for single-line, single-hunk, and multi-hunk issues. While we expect complexity to increase with these categories significantly, the expected performance deterioration was not as high as we would have expected. For pass@1, we observed a drop from 45.06% (single-line) over 34.37% (single-hunk) to 27.50% (multi-hunk), i.e., LLMs cope still relatively well with increasing patch complexity. We even observe 9 of 48 performance figures to improve from single-line to single-hunk (particularly for Python), and 10 of 48 from single-hunk to multi-hunk (particularly for Java). Note that the experiment focused on single-function cases. However, these promising results suggest to even investigate multi-hunk, multi-function APR further.

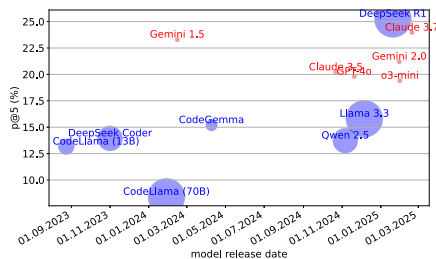
### D. RQ4: OPEN VS CLOSED MODELS

Unsurprisingly, the accuracy in most experiments is significantly higher on average for closed models, which are likely trained on larger and higher-quality datasets, are typically larger in parameter count, and may have been subject to more extensive fine-tuning on failure cases compared to open models. To get an impression of how open models and closed models progress over time, we plot the quality of program repair against the models' release dates. As a quality measure, we use the pass@5 rate (see Table 4), averaged over all datasets. For OpenAI's models, which undergo continuous development, we use the exact date of the version used in our experiments (for GPT-4o: Nov 11 2024, for o3-mini: Jan 31 2025). Figure 4 shows the resulting graph: While a certain stagnation for closed models can be observed, there is a positive trend for open models, with DeepSeek R1 (dist.) in particular showing strong improvements compared to its predecessors. These findings do not only underline that reasoning models are a particularly interesting future direction for program repair, but also indicate that open models are becoming a viable alternative for software development practitioners, allowing them to avoid vendor lock-in and increasing digital sovereignty.

## V. DISCUSSION AND THREATS TO VALIDITY

### A. IMPLICATIONS FOR AGENTIC APR WORKFLOWS

Recent research has moved toward agentic repair systems [15], [16], [17], [58] in which LLMs iteratively



**FIGURE 4.** The accuracy of program repair (pass@5, base prompt, macro-averaged over all 4 benchmarks), plotted against models' release dates. For open models (blue), circle size indicates the respective model's size. Open models are catching up to closed models (red), and have even surpassed them with DeepSeek R1 (dist.)'s release.

localize faults, generate patches, and validate results using external tools. While our study deliberately focuses on single-prompt setups to isolate fundamental LLM capabilities, our findings directly inform the design of such systems. First, our result that no single model dominates across programming languages (RQ1) suggests that multi-language agentic systems should consider language-specific model selection or model-committees rather than relying on a single backbone LLM. Second, we replicate and extend prior findings [14], [28] on the strong positive effect of test information across a broader set of models and languages, further corroborating the value of incorporating test execution feedback into iterative repair loops. Third, and most critically, our finding that automated fault localization results in dramatic performance drops ranging from  $-7.8\%$  to  $-16.2\%$  pass@1 (RQ2) directly identifies fault localization as a main bottleneck for agentic pipelines operating in realistic, repository-scale settings. Finally, our finding that multi-hunk repair performance degrades more gracefully than expected (RQ3) is encouraging for agentic systems that must handle complex, multi-location bugs.

## B. EXPERIMENTS WITH LLMs

Sallou et al. [51] discussed several evaluation aspects when using LLMs in software engineering research, including the problem of output variability, data leakage, and closed-source models. To address the output variability of LLMs, we generated not only one but 15 patches (see details in Section III-D). To evaluate the generated patches, we used the established *pass@k* metric [53] representing an estimator for the model's effectiveness. Assessing plausibility (i.e., passing test cases) rather than semantic correctness (i.e., semantical equivalence with a ground truth like a developer patch) represents a threat to construct validity. While manual inspection is the gold standard correctness criterion in LLM-based APR, it is infeasible at our scale ( $\approx 195,000$  patches across four benchmarks). Existing automated correctness proxies such as Syntactic Equivalence (SYE) and Trivial Compiler Equivalence (TCE), studied by Ouyang et al. [39], are specific to Java and rely on JVM bytecode comparison, making them inapplicable to BugsJS, BugsInPy, and BugsPHP. Consequently, plausibility is used as a uniform

cross-benchmark metric, consistent with prior large-scale multilingual APR studies [28], [42]. Ouyang et al. [39] further report a strong correlation between plausibility and manually validated correctness, supporting its use as a proxy measure. The data leakage issue is more challenging to address because we cannot control the training data of state-of-the-art LLMs. Related works, e.g., RepairBench [42] used a dataset with more recent data, whose time period is now also included in the cut-off dates of the latest models. At this point, we have to assume that some of the benchmark data may have been included in the training corpora of some of the LLMs we used in our experiments, which represents a threat to internal validity. In fact, Zhou et al. [59] recently investigated 83 SE benchmarks regarding their potential data leakage. According to their analysis Defects4J (in its 2.0 version) has a leaked ratio of 0.41% and BugsInPy 11.0%. Both numbers are relatively low. Interestingly, although BugsInPy obviously has some data leakage issues, our experiments showed it as the most challenging benchmark for the tested LLMs. Ramos et al. [60] further studied the impact of benchmark leakage across several open-source models and found that, although some models show substantial evidence of memorization on widely used benchmarks such as Defects4J, newer models, trained on larger datasets such as Llama 3.1 show only limited signs of leakage. As part of our empirical evaluation, we support a comparative analysis of models by selecting a wide range of open-source/weight and closed models. Our broad selection of models and benchmarks reduces the threat to external validity; however, we cannot claim generality beyond our experiments.

## C. PROMPT OPTIMIZATION

As described in Section III-F, we used zero-shot prompts with different repair ingredients in our experiments. Those prompts were designed based on and in consistency with prior work [22], [28], [38], [42], and have been tested in preliminary experiments. Additional optimization and tuning of the prompts could lead to better repair performances on our benchmarks; however, we did intentionally not perform further optimizations to prevent introducing model-specific biases and further benchmark overfitting.

## D. REPAIR TIME

We noticed that, e.g., due to its reasoning strategy, the DeepSeek R1 (dist.) model produces more iterative output, requiring significantly more time to conclude with the final repair than the other models. Other studies [35] showed that developers could accept a timeout of up to one hour for integrating APR technology in their workflows. This timeout is still much higher than LLMs would require; in our experiments spanning between seconds and several minutes. However, it is currently unknown what an acceptable timeout for LLM-based APR would be so that, e.g., it can effectively integrate into agentic workflows. Different resource demands

across the models, in particular with respect to timing, can lead to unfair comparison representing a threat to internal validity. Our study methodology did not limit the time for model inference or patch validation because we wanted to focus on the actual repair capabilities, and we leave a more rigorous analysis regarding repair time for future work.

## VI. CONCLUSION

This work demonstrates a comprehensive exploration of the current capabilities of state-of-the-art LLMs to perform generalizable repair, i.e., across multiple languages and patch complexities. After comparing open models with closed models on benchmarks for Java, JavaScript, PHP, and Python, we conclude that no single model consistently outperforms the others on all benchmarks. Instead, it becomes necessary to combine various models to realize cross-language repair capabilities. In our experiments with various prompting strategies, we observed that LLM-based APR with automated/imperfect fault localization leads to significantly lower repair performance. Therefore, we advocate for more realistic evaluations in the APR community. In the future, we plan to extend our study to more aspects, such as repair time, particularly regarding reasoning models. Further, we want to understand closer the challenges in repairing across language barriers and how to detect and leverage model uncertainty in patch generation, both as a signal for patch selection and as a stopping criterion for iterative repair. A promising future direction is to investigate how our single-prompt findings translate to multi-round and agentic repair pipelines, and to what extent the model-level insights reported here carry over to more sophisticated orchestration frameworks. We believe that the insights of this study are important for further advancing the integration of APR technologies in software development practice.

## VII. DATA AVAILABILITY

The data that support the findings of this study, including all our results and the generated patches, are openly available in our supplemental material: <https://doi.org/10.6084/m9.figshare.29183990>.

## ACKNOWLEDGMENT

The views and conclusions expressed in this article are those of the authors alone and do not represent the official policies or endorsements of Sonar or any of its subsidiaries or affiliates. Furthermore, the findings presented herein are independent and should not be interpreted as an evaluation of the quality of Sonar's products.

## REFERENCES

- [1] C. Le Goues, M. Pradel, A. Roychoudhury, and S. Chandra, "Automatic program repair," *IEEE Softw.*, vol. 38, no. 4, pp. 22–27, Jul. 2021.
- [2] X. Gao, Y. Noller, and A. Roychoudhury, "Program repair," 2022, *arXiv:2211.12787*.
- [3] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "GenProg: A generic method for automatic software repair," *IEEE Trans. Softw. Eng.*, vol. 38, no. 1, pp. 54–72, Jan. 2012.
- [4] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "SemFix: Program repair via semantic analysis," in *Proc. 35th Int. Conf. Softw. Eng. (ICSE)*, May 2013, pp. 772–781.
- [5] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "TBar: Revisiting template-based automated program repair," in *Proc. 28th ACM SIGSOFT Int. Symp. Softw. Test. Anal.* New York, NY, USA: Association for Computing Machinery, Jul. 2019, pp. 31–42, doi: [10.1145/3293882.3330577](https://doi.org/10.1145/3293882.3330577).
- [6] H. Ye and M. Monperrus, "ITER: Iterative neural repair for multi-location patches," in *Proc. IEEE/ACM 46th Int. Conf. Softw. Eng.* New York, NY, USA: Association for Computing Machinery, Feb. 2024, pp. 1–13, doi: [10.1145/3597503.3623337](https://doi.org/10.1145/3597503.3623337).
- [7] D. Williams, J. Callan, S. Kirbas, S. Mechtaev, J. Petke, T. Prideaux-Ghee, and F. Sarro, "User-centric deployment of automated program repair at bloomberg," in *Proc. 46th Int. Conf. Softw. Eng., Softw. Eng. Pract.* New York, NY, USA: Association for Computing Machinery, Apr. 2024, pp. 81–91, doi: [10.1145/3639477.3639756](https://doi.org/10.1145/3639477.3639756).
- [8] M. Jin, S. Shahriar, M. Tufano, X. Shi, S. Lu, N. Sundaresan, and A. Svyatkovskiy, "InferFix: End-to-end program repair with LLMs," in *Proc. 31st ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.* New York, NY, USA: Association for Computing Machinery, Nov. 2023, pp. 1646–1656, doi: [10.1145/3611643.3613892](https://doi.org/10.1145/3611643.3613892).
- [9] A. Marginean, J. Bader, S. Chandra, M. Harman, Y. Jia, K. Mao, A. Mols, and A. Scott, "SapFix: Automated end-to-end repair at scale," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng., Softw. Eng. Pract. (ICSE-SEIP)*, May 2019, pp. 269–278.
- [10] R. Shariffdeen, Y. Noller, M. Mirchev, H. Ruan, G. Xiang, A. Costa, G. J. Duck, and A. Roychoudhury, "APR competition 2024," in *Proc. 5th ACM/IEEE Int. Workshop Automated Program Repair*. New York, NY, USA: Association for Computing Machinery, Apr. 2024, pp. 46–49, doi: [10.1145/3643788.3648015](https://doi.org/10.1145/3643788.3648015).
- [11] R. Shariffdeen, M. Mirchev, Y. Noller, and A. Roychoudhury, "Cerberus: A program repair framework," in *IEEE/ACM 45th Int. Conf. Softw. Engineering: Companion Proc. (ICSE-Companion)*, Melbourne, VIC, Australia, May 2023, pp. 73–77, doi: [10.1109/ICSE-Companion58688.2023.00028](https://doi.org/10.1109/ICSE-Companion58688.2023.00028).
- [12] W. X. Zhao et al., "A survey of large language models," 2023, *arXiv:2303.18223*.
- [13] C. S. Xia and L. Zhang, "Less training, more repairing please: Revisiting automated program repair via zero-shot learning," in *Proc. 30th ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.* New York, NY, USA: Association for Computing Machinery, Nov. 2022, pp. 959–971, doi: [10.1145/3540250.3549101](https://doi.org/10.1145/3540250.3549101).
- [14] C. S. Xia and L. Zhang, "Automated program repair via conversation: Fixing 162 out of 337 bugs for \$0.42 each using ChatGPT," in *Proc. 33rd ACM SIGSOFT Int. Symp. Softw. Test. Anal.* New York, NY, USA: Association for Computing Machinery, Sep. 2024, pp. 819–831, doi: [10.1145/3650212.3680323](https://doi.org/10.1145/3650212.3680323).
- [15] Y. Zhang, H. Ruan, Z. Fan, and A. Roychoudhury, "AutoCodeRover: Autonomous program improvement," in *Proc. 33rd ACM SIGSOFT Int. Symp. Softw. Test. Anal.* New York, NY, USA: Association for Computing Machinery, Sep. 2024, pp. 1592–1604, doi: [10.1145/3650212.3680384](https://doi.org/10.1145/3650212.3680384).
- [16] H. Ruan, Y. Zhang, and A. Roychoudhury, "SpecRover: Code intent extraction via LLMs," 2024, *arXiv:2408.02232*.
- [17] J. Yang, C. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press, "SWE-agent: Agent-computer interfaces enable automated software engineering," in *Proc. Adv. Neural Inf. Process. Syst.*, 2024, pp. 50528–50652.
- [18] C. S. Xia, Y. Ding, and L. Zhang, "The plastic surgery hypothesis in the era of large language models," in *Proc. 38th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Sep. 2023, pp. 522–534, doi: [10.1109/ase56229.2023.00047](https://doi.org/10.1109/ase56229.2023.00047).
- [19] Q. Zhu, Z. Sun, W. Zhang, Y. Xiong, and L. Zhang, "Tare: Type-aware neural program repair," in *Proc. IEEE/ACM 45th Int. Conf. Softw. Eng. (ICSE)*, May 2023, pp. 1443–1455.
- [20] S. B. Hossain, N. Jiang, Q. Zhou, X. Li, W.-H. Chiang, Y. Lyu, H. Nguyen, and O. Tripp, "A deep dive into large language models for automated bug localization and repair," *Proc. ACM Softw. Eng.*, vol. 1, pp. 1471–1493, Jul. 2024, doi: [10.1145/3660773](https://doi.org/10.1145/3660773).
- [21] K. Huang, X. Meng, J. Zhang, Y. Liu, W. Wang, S. Li, and Y. Zhang, "An empirical study on fine-tuning large language models of code for automated program repair," in *Proc. 38th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Sep. 2023, pp. 1162–1174.

- [22] A. Silva, S. Fang, and M. Monperrus, "RepairLLaMA: Efficient representations and fine-tuned adapters for program repair," 2023, *arXiv:2312.15698*.
- [23] N. Jiang, K. Liu, T. Lutellier, and L. Tan, "Impact of code language models on automated program repair," in *Proc. IEEE/ACM 45th Int. Conf. Softw. Eng. (ICSE)*, May 2023, pp. 1430–1442.
- [24] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A database of existing faults to enable controlled testing studies for Java programs," in *Proc. Int. Symp. Softw. Test. Anal.*, Jul. 2014, pp. 437–440, doi: [10.1145/2610384.2628055](https://doi.org/10.1145/2610384.2628055).
- [25] P. Gyimesi, B. Vancsics, A. Stocco, D. Mazinanian, Á. Beszédes, R. Ferenc, and A. Mesbah, "BUGSJS: A benchmark and taxonomy of JavaScript bugs," *Softw. Test., Verification Rel.*, vol. 31, no. 4, pp. 90–101, Jun. 2021, doi: [10.1002/stvr.1751](https://doi.org/10.1002/stvr.1751).
- [26] R. Widyasari, S. Q. Sim, C. Lok, H. Qi, J. Phan, Q. Tay, C. Tan, F. Wee, J. E. Tan, Y. Yieh, B. Goh, F. Thung, H. J. Kang, T. Hoang, D. Lo, and E. L. Ouh, "BugsInPy: A database of existing bugs in Python programs to enable controlled testing and debugging studies," in *Proc. 28th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, Nov. 2020, pp. 1556–1560, doi: [10.1145/3368089.3417943](https://doi.org/10.1145/3368089.3417943).
- [27] D. Pramod, T. De Silva, U. Thabrew, R. Shariffdeen, and S. Wickramanayake, "BugsPHP: A dataset for automated program repair in PHP," in *Proc. 21st Int. Conf. Mining Softw. Repositories*, Lisbon, Portugal, Apr. 2024, pp. 128–132, doi: [10.1145/3643991.3644878](https://doi.org/10.1145/3643991.3644878).
- [28] N. Parasaram, H. Yan, B. Yang, Z. Flahy, A. Qudsi, D. Ziaber, E. Barr, and S. Mehtaev, "The fact selection problem in LLM-based program repair," 2024, *arXiv:2404.05520*.
- [29] F. Long and M. Rinard, "Automatic patch generation by learning correct code," in *Proc. 43rd Annu. ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang.*, New York, NY, USA: Association for Computing Machinery, Jan. 2016, pp. 298–312, doi: [10.1145/2837614.2837617](https://doi.org/10.1145/2837614.2837617).
- [30] Z. Tang, J. Ge, S. Liu, T. Zhu, T. Xu, L. Huang, and B. Luo, "Domain adaptive code completion via language models and decoupled domain databases," in *Proc. 38th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Sep. 2023, pp. 421–433, doi: [10.1109/ase56229.2023.00076](https://doi.org/10.1109/ase56229.2023.00076).
- [31] X. Yin, C. Ni, S. Wang, Z. Li, L. Zeng, and X. Yang, "ThinkRepair: Self-directed automated program repair," in *Proc. 33rd ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, New York, NY, USA: Association for Computing Machinery, Sep. 2024, pp. 1274–1286, doi: [10.1145/3650212.3680359](https://doi.org/10.1145/3650212.3680359).
- [32] B. Lin, S. Wang, M. Wen, L. Chen, and X. Mao, "One size does not fit all: Multi-granularity patch generation for better automated program repair," in *Proc. 33rd ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, New York, NY, USA: Association for Computing Machinery, Sep. 2024, pp. 1554–1566, doi: [10.1145/3650212.3680381](https://doi.org/10.1145/3650212.3680381).
- [33] K. Liu, S. Wang, A. Koyuncu, K. Kim, T. F. Bissyandé, D. Kim, P. Wu, J. Klein, X. Mao, and Y. L. Traon, "On the efficiency of test suite based program repair: A systematic assessment of 16 automated repair systems for Java programs," in *Proc. ACM/IEEE 42nd Int. Conf. Softw. Eng.*, New York, NY, USA: Association for Computing Machinery, Jun. 2020, pp. 615–627, doi: [10.1145/3377811.3380338](https://doi.org/10.1145/3377811.3380338).
- [34] W. Zhong, H. Ge, H. Ai, C. Li, K. Liu, J. Ge, and B. Luo, "StandUp4NPR: Standardizing SetUp for empirically comparing neural program repair systems," in *Proc. 37th IEEE/ACM Int. Conf. Automated Softw. Eng.*, Cincinnati, OH, USA: Standard, Oct. 2022, pp. 1–13, doi: [10.1145/3551349.3556943](https://doi.org/10.1145/3551349.3556943).
- [35] Y. Noller, R. Shariffdeen, X. Gao, and A. Roychoudhury, "Trust enhancement issues in program repair," in *Proc. IEEE/ACM 44th Int. Conf. Softw. Eng. (ICSE)*, New York, NY, USA: Association for Computing Machinery, May 2022, pp. 2228–2240, doi: [10.1145/3510003.3510040](https://doi.org/10.1145/3510003.3510040).
- [36] T. Durieux, F. Madeiral, M. Martinez, and R. Abreu, "Empirical review of Java program repair tools: A large-scale experiment on 2,141 bugs and 23,551 repair attempts," in *Proc. 27th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, New York, NY, USA: Association for Computing Machinery, Aug. 2019, pp. 302–313, doi: [10.1145/3338906.3338911](https://doi.org/10.1145/3338906.3338911).
- [37] H. Ye, M. Martinez, T. Durieux, and M. Monperrus, "A comprehensive study of automatic program repair on the QuixBugs benchmark," in *Proc. IEEE 1st Int. Workshop Intell. Bug Fixing (IBF)*, Feb. 2019, pp. 1–10.
- [38] C. S. Xia, Y. Wei, and L. Zhang, "Automated program repair in the era of large pre-trained language models," in *Proc. IEEE/ACM 45th Int. Conf. Softw. Eng. (ICSE)*, Melbourne, VIC, Australia, May 2023, pp. 1482–1494, doi: [10.1109/ICSE48619.2023.00129](https://doi.org/10.1109/ICSE48619.2023.00129).
- [39] Y. Ouyang, J. Yang, and L. Zhang, "Benchmarking automated program repair: An extensive study on both real-world and artificial bugs," in *Proc. 33rd ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, Vienna, Austria, Sep. 2024, pp. 440–452, doi: [10.1145/3650212.3652140](https://doi.org/10.1145/3650212.3652140).
- [40] A. Silva, M. Martinez, B. Danglot, D. Ginelli, and M. Monperrus, "FLA-COCO: Fault localization for Java based on industry-grade coverage," 2021, *arXiv:2111.12513*.
- [41] H. A. de Souza, M. L. Chaim, and F. Kon, "Spectrum-based software fault localization: A survey of techniques, advances, and challenges," 2016, *arXiv:1607.04347*.
- [42] A. Silva and M. Monperrus, "RepairBench: Leaderboard of frontier models for program repair," 2024, *arXiv:2409.18952*.
- [43] J. Xu, Y. Fu, S. Hwei Tan, and P. He, "Aligning the objective of LLM-based program repair," 2024, *arXiv:2404.08877*.
- [44] H.-N. Zhu and C. Rubio-González, "On the reproducibility of software defect datasets," in *Proc. IEEE/ACM 45th Int. Conf. Softw. Eng. (ICSE)*, Melbourne, VIC, Australia, May 2023, pp. 2324–2335, doi: [10.1109/ICSE48619.2023.00195](https://doi.org/10.1109/ICSE48619.2023.00195).
- [45] S. D. Kolak, R. Martins, C. Le Goues, and V. J. Hellendoorn, "Patch generation with language models: Feasibility and scaling behavior," in *Proc. Deep Learn. Code Workshop*, 2022. [Online]. Available: [https://openreview.net/forum?id=RHlZJh\\_b1-5](https://openreview.net/forum?id=RHlZJh_b1-5)
- [46] F. Aguilar, S. Grayson, and D. Marinov, "Reproducing and improving the BugsInPy dataset," in *Proc. IEEE 23rd Int. Work. Conf. Source Code Anal. Manipulation (SCAM)*, Oct. 2023, pp. 260–264, doi: [10.1109/scam59687.2023.00036](https://doi.org/10.1109/scam59687.2023.00036).
- [47] V. Campos, R. Shariffdeen, A. Ulges, and Y. Noller. (2026). *Replication package for Exploring Generalizable Automated Program Repair With Large Language Models*. Figshare. [Online]. Available: <https://figshare.com/s/947fd7030f10a67a1c9f?file=62060827>
- [48] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun, "Is the cure worse than the disease? Overfitting in automated program repair," in *Proc. 10th Joint Meeting Found. Softw. Eng.*, New York, NY, USA: Association for Computing Machinery, Aug. 2015, pp. 532–543, doi: [10.1145/2786805.2786825](https://doi.org/10.1145/2786805.2786825).
- [49] J. Petke, M. Martinez, M. Kechagia, A. Aleti, and F. Sarro, "The patch overfitting problem in automated program repair: Practical magnitude and a baseline for realistic benchmarking," in *Companion Proc. 32nd ACM Int. Conf. Found. Softw. Eng.*, New York, NY, USA: Association for Computing Machinery, Jul. 2024, pp. 452–456, doi: [10.1145/3663529.3663776](https://doi.org/10.1145/3663529.3663776).
- [50] S. Ouyang, J. M. Zhang, M. Harman, and M. Wang, "LLM is like a box of chocolates: The non-determinism of chatgpt in code generation," 2023, *arXiv:2308.02828*.
- [51] J. Sallou, T. Durieux, and A. Panichella, "Breaking the silence: The threats of using LLMs in software engineering," in *Proc. ACM/IEEE 44th Int. Conf. Softw. Eng., New Ideas Emerg. Results*, Lisbon, Portugal, Apr. 2024, pp. 102–106, doi: [10.1145/3639476.3639764](https://doi.org/10.1145/3639476.3639764).
- [52] S. Ouyang, J. M. Zhang, M. Harman, and M. Wang, "An empirical study of the non-determinism of ChatGPT in code generation," *ACM Trans. Softw. Eng. Methodol.*, vol. 34, no. 2, pp. 1–28, Jan. 2025, doi: [10.1145/3697010](https://doi.org/10.1145/3697010).
- [53] M. Chen et al., "Evaluating large language models trained on code," 2021, *arXiv:2107.03374*.
- [54] J. Renzullo, P. Reiter, W. Weimer, and S. Forrest, "Automated program repair: Emerging trends pose and expose problems for benchmarks," *ACM Comput. Surv.*, vol. 57, no. 8, pp. 1–18, Aug. 2025.
- [55] K. Liu, A. Koyuncu, T. F. Bissyandé, D. Kim, J. Klein, and Y. Le Traon, "You cannot fix what you cannot find! An investigation of fault localization bias in benchmarking automated program repair systems," in *Proc. 12th IEEE Conf. Softw. Test., Validation Verification (ICST)*, Apr. 2019, pp. 102–113, doi: [10.1109/ICST.2019.00020](https://doi.org/10.1109/ICST.2019.00020).
- [56] D. Yang, Y. Qi, and X. Mao, "An empirical study on the usage of fault localization in automated program repair," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, Shanghai, China, Sep. 2017, pp. 504–508, doi: [10.1109/ICSME.2017.37](https://doi.org/10.1109/ICSME.2017.37).
- [57] R. Shariffdeen, C. S. Timperley, Y. Noller, C. Le Goues, and A. Roychoudhury, "Vulnerability repair via concolic execution and code mutations," *ACM Trans. Softw. Eng. Methodol.*, vol. 34, no. 4, pp. 1–27, Apr. 2025, doi: [10.1145/3707454](https://doi.org/10.1145/3707454).
- [58] J. He, C. Treude, and D. Lo, "LLM-based multi-agent systems for software engineering: Literature review, vision, and the road ahead," *ACM Trans. Softw. Eng. Methodol.*, vol. 34, no. 5, pp. 1–30, Jun. 2025, doi: [10.1145/3712003](https://doi.org/10.1145/3712003).

- [59] X. Zhou, M. Weyssow, R. Widyasari, T. Zhang, J. He, Y. Lyu, J. Chang, B. Zhang, D. Huang, and D. Lo, "LessLeak-bench: A first investigation of data leakage in LLMs across 83 software engineering benchmarks," 2025, *arXiv:2502.06215*.
- [60] D. Ramos, C. Mamede, K. Jain, P. Canelas, C. Gamboa, and C. Le Goues, "Are large language models memorizing bug benchmarks?" in *Proc. IEEE/ACM Int. Workshop Large Lang. Models Code (LLM4Code)*, Ottawa, ON, Canada, May 3, 2025, pp. 1–8, doi: [10.1109/LLM4CODE66737.2025.00005](https://doi.org/10.1109/LLM4CODE66737.2025.00005).



**VIOLA CAMPOS** received the B.S. and M.S. degrees in computer science from the RheinMain University of Applied Sciences, Wiesbaden, Germany, in 2021. She was a Software Engineer with the Federal Statistical Office of Germany, from 2015 to 2018, and an Adjunct Lecturer with the RheinMain University of Applied Sciences, from 2018 to 2022. Since 2022, she has been a Ph.D. Researcher with the RheinMain University of Applied Sciences. Her research interests include

the application of large language models (LLMs) to software quality improvement, with a particular focus on uncertainty estimation and correctness estimation for LLM-generated code.



**RIDWAN SHARIFFDEEN** (Senior Member, IEEE) received the Bachelor of Science (B.Sc.) degree (Hons.) from the Computer Science Engineering Department, University of Moratuwa, Sri Lanka, and the Ph.D. degree in computer science from the National University of Singapore. From 2016 to 2018, he worked in the industry as a Senior Software Engineer for System Security and Automation Technology. From 2022 to 2025, he was a Research Fellow with the National

University of Singapore. He is currently a Principal Research Scientist with Sonar. His current research is in automated software engineering, focusing on improving developer productivity. He is a Professional Member of ACM.



**ADRIAN ULGES** received the Diploma and Ph.D. degrees in computer science from TU Kaiserslautern, in 2005 and 2009, respectively. He has worked with the German Research Center for Artificial Intelligence, from 2005 to 2012, and Google Inc., as a Visiting Scientist, Mountain View, in 2005, and Zürich, in 2011. Since 2013, he has been a Professor of computer science with the RheinMain University of Applied Sciences, Wiesbaden, Germany. His publication record includes more than 70 peer-reviewed articles. His research interests include machine learning, mostly focused on language understanding, and often involving interactive human-in-the-loop systems with scarce labels. He is a member of the Hessian Graduate Center for Applied Computer Science (PZAI), has served as a committee member for multiple conferences and workshops, and was awarded a Google Research Award, in 2010.



**YANNIC NOLLER** (Member, IEEE) received the B.Sc. and M.Sc. degrees in software engineering from the University of Stuttgart, Germany, in 2013 and 2016, respectively, and the Ph.D. degree in computer science from Humboldt-Universität zu Berlin, Germany, in 2020. From 2020 to 2021, he was a Research Fellow, and from 2021 to 2023, a Research Assistant Professor with the National University of Singapore. From 2023 to 2024, he has been a tenure-track Assistant Professor with

Singapore University of Technology and Design. Since 2024, he has been a tenure-track Associate Professor with Ruhr University Bochum, Germany, where he leads the software quality research group. His research focuses on how software quality can be maintained and improved with automated testing and repair technologies. He is a member of the Association for Computing Machinery (ACM), the ACM Special Interest Group on Software Engineering, and the German Informatics Society (GI).

...