arXiv:2506.13182v1 [cs.SE] 16 Jun 2025

From Empirical Evaluation to Context-Aware Enhancement: Repairing Regression Errors with LLMs

ANH HO, The University of Melbourne, Australia
THANH LE-CONG, The University of Melbourne, Australia
BACH LE, The University of Melbourne, Australia
CHRISTINE RIZKALLAH, The University of Melbourne, Australia

Software systems constantly evolves to adapt to ever-changing customer demands and evolving markets. During the software evolution process, code changes across versions can unexpectedly break previously working functionalities, leading to so-called regression bugs. Manually repairing regression bugs is both challenging and time-consuming, as the process often requires developers to traverse the history of the software in order to understand how the bugs were introduced and, consequently, how to repair them. Despite the challenges of manual efforts, only a limited number of studies have investigated automated program repair (APR) for regression bugs, most of which date back several years and focus on traditional APR techniques. Since then, various APR approaches, especially those leveraging the power of large language models (LLMs), have been rapidly developed to fix general software bugs. Unfortunately, the effectiveness of these advanced techniques in the context of regression bugs remains largely unexplored. This gap motivates the need for an empirical study evaluating the effectiveness of modern APR techniques in fixing real-world regression bugs.

In this work, we conduct an empirical study of APR techniques on Java regression bugs. To facilitate our study, we introduce RegMiner4APR, a high-quality benchmark of Java regression bugs integrated into a framework designed to facilitate APR research. The current benchmark includes 99 regression bugs collected from 32 widely used real-world Java GitHub repositories. We begin by conducting an in-depth analysis of the benchmark, demonstrating its diversity and quality. Building on this foundation, we empirically evaluate the capabilities of APR to regression bugs by assessing both traditional APR tools and advanced LLM-based APR approaches. Our experimental results show that classical APR tools fail to repair any bugs, while LLM-based APR approaches exhibit promising potential. Motivated by these results, we investigate impact of incorporating bug-inducing change information into LLM-based APR approaches for fixing regression bugs. Our results highlight that this context-aware enhancement significantly improves the performance of LLM-based APR, yielding 1.8× more successful repairs compared to using LLM-based APR without such context.

CCS Concepts: • **Software and its engineering → Software testing and debugging**; • **General and reference → Empirical studies**.

Additional Key Words and Phrases: Regression Bugs, Regression Repair, Automated Program Repair, Benchmark, Large Language Models

ACM Reference Format:
Anh Ho, Thanh Le-Cong, Bach Le, and Christine Rizkallah. 2025. From Empirical Evaluation to Context-Aware Enhancement: Repairing Regression Errors with LLMs. 1, 1 (June 2025), 40 pages. https://doi.org/XXXXXXX.XXXXXXX

Authors' Contact Information: Anh Ho, The University of Melbourne, Australia, anh.ho1@student.unimelb.edu.au; Thanh Le-Cong, The University of Melbourne, Australia, congthanh.le@student.unimelb.edu.au; Bach Le, The University of Melbourne, Australia, bach.le@unimelb.edu.au; Christine Rizkallah, The University of Melbourne, Australia, christine.rizkallah@unimelb.edu.au.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
Manuscript submitted to ACM

1 Introduction

Software systems are continuously evolving to meet user demands and adapt to rapidly changing market conditions, resulting in increasing system complexity and frequent source code modifications. Each modification usually requires interactions with an ever-growing number of components within the code base such as calling to existing functionality or integrating with other parts of the system. However, these code changes can unintentionally introduce regression bugs, disrupting previously working features of software systems [56].

Regression bugs often stem from the unintended consequences of modifications to existing code. This unique characteristic makes regression bugs particularly challenging, as developers must identify the specific changes responsible, i.e., bug-inducing changes, and trace the effects of bug-inducing changes on previously functioning features [70]. In large-scale software projects, the challenges associated with regression bugs are intensive, as they often remain undetected for several years before being identified and resolved [11, 68]. Moreover, low-quality fixes might introduce new regressions, creating a recurring cycle of broken functionality and complicating the maintenance of software systems [10]. Consequently, regression bugs continue to be a persistent challenge in the software industry, affecting overall software quality and maintenance efforts [15, 45].

Automated program repair (APR) techniques were developed to reduce the manual effort required to fix bugs and improve software quality by automating the repair process. Hence, applying APR to regression bugs would be particularly valuable, given their complexity and the effort required to address them. To date, there has been extensive research on APR for general software bugs [17, 31, 41], including a growing interest in large language model (LLM)-based APR techniques in recent years [12, 19, 20, 32, 36, 49, 52, 63, 65, 67]. However, the most recent work specifically targeting regression bug repair dates back several years. Notably, Relifix, developed for C/C++ programs, focuses solely on traditional pattern-based and search-based APR techniques [56]. This naturally raises an important question:

How well do state-of-the-art APR techniques perform when applied specifically to regression bugs?

This motivates the need for an empirical study to revisit existing APR techniques to better understand their effectiveness on regression bugs, where they fall short and can be further improved.

One major challenge in conducting the empirical study lies in the lack of a high-quality and up-to-date benchmark that reflects both current software development practices and the ongoing progress in this research field. Popular datasets commonly used for evaluating APR techniques, such as Defects4J [23], Bugs.jar [50] and Bears [37], primarily target general software bugs and are associated with bug-triggering test cases that have been added/updated after the bug is reported [34]. These datasets are not constructed with regression-specific characteristics in mind. The bugs are not revealed through regression testing, making it unclear whether they truly represent regressions. Hence, these datasets lack the temporal and behavioral properties that define regression bugs. Existing datasets for regression bugs, such as CoREBench [11] and CIBugs [24], are outdated (bugs are before 2017) and difficult to reproduce, making them less suitable for studying regression repair.

To address these problems, we propose RegMiner4APR, a high-quality Java regression error benchmark integrated into a framework designed to facilitate APR research. The benchmark is built upon an automated tool that mines regression bugs from software evolution history, namely RegMiner [55]. By leveraging RegMiner, we automatically collect potential regression errors from code repositories and then validate the regressions. Our benchmark collection framework ensures the reproducibility and quality of each mined bug instance, while also supporting future benchmark extensions. Through this automated process, we have established RegMiner4APR, which includes 99 confirmed regression bugs from 32 widely used real-world Java repositories on GitHub.

We conduct an in-depth analysis of our benchmark RegMiner4APR to demonstrate desired properties of the benchmark, including: *reality*, where regression bugs are extracted from diverse real-world open-source software; *currency*, with the bugs spanning from June 2017 onward; *diversity*, encompassing a broad range of projects and diverse repair operators; *extensibility*, as our approach enables the dataset to expand over time with minimal human efforts by building upon RegMiner [55]; and *durable replicability*, with the framework offering easy access, extensibility, and snapshots of all projects and their dependencies, similar to that of Defects4J [23]. Prior study suggested that these properties are often required to enable fair evaluations [58].

Using our the diverse and high-quality benchmark, we conduct an empirical study to revisit APR techniques on Java regression errors. The evaluation is broadly categorized into two classes: *traditional* and *advanced LLM-based* techniques. For *traditional* techniques, we examine pattern-based and search-based tools, including GenProg [30], Kali [47], Cardumen [39], Arja [69], jMutRepair [38], and TBar [33]. For *advanced* techniques, we focus on LLM-based approaches, further divided into two groups: *fine-tuning-based* and *prompt-based* techniques. In the *fine-tuning* category, we evaluate full fine-tuning methods, including CodeGen models with 2B and 6B parameters and Incoder models with 1B and 6B parameters [60], as well as parameter-efficient fine-tuning methods, such as RepairLLaMA [52]. For *prompt-based* approaches, we explore two repair strategies, *zero-shot prompting* and *conversational interaction*, using advanced general-purpose LLMs: ChatGPT-3.5-Turbo and ChatGPT-4o. Our experiments show that traditional pattern-based and search-based APR tools fail to repair any regression bugs in our dataset. In contrast, LLM-based approaches demonstrate promising performance: RepairLLaMA, the top-performing fine-tuning-based technique, and ChatGPT-4o with conversational prompting, the most effective prompt-based method, each successfully repair 9 bugs. It is, however, noted that these LLM-based techniques are not inherently designed to tackle the nature of regression bugs, e.g., taking into account the bug-inducing changes to inform bug repair and thus leaving room for further improvements.

Finally, we further explore the impact of incorporating additional regression-specific context, e.g., bug-inducing change information, on improving the repair effectiveness of LLM-based APR. We do so by leveraging prompt-based approaches, explicitly integrating bug-inducing changes into the prompt design. This choice is driven by the inherent flexibility of prompt-based methods in terms of model usage and prompt construction, as well as their data efficiency compared to fine-tuning-based techniques. Our experiments show that incorporating this additional context significantly enhances the performance and precision of LLM-based techniques using prompting, enabling successful repairs for 16 regression bugs in the RegMiner4APR benchmark. This represents 1.8 times more correct repairs than the best-performing fine-tuning-based and prompt-based APR techniques evaluated in our study. Interestingly, our qualitative analysis further reveals that bug-inducing change information provides crucial context that helps models better identify the root cause of the regression and determine whether to fully or partially revert to previous correct statements. In addition, it helps narrow the fix scope, thereby improving fault localization and reducing unnecessary code modifications.

In summary, the key contributions of this work are as follows:

1) We introduce RegMiner4APR, a high-quality Java regression error benchmark integrated into a framework that supports reproducible studies in automated program repair and enables future extensions. We also provide a comprehensive analysis of the benchmark's characteristics.
2) We perform an empirical study evaluating the effectiveness of several APR techniques, including recent state-of-the-art LLM-based approaches, on regression bugs.

3) We investigate the impact of incorporating bug-inducing change information on the effectiveness of prompt-based repair techniques. Our study includes a qualitative analysis that reveals how this regression-specific context supports models in fixing regression errors.

Replication Packages: To support further research, we provide all artifacts associated with our dataset in a reproducible and easily accessible form. Moreover, the RegMiner4APR benchmark is designed for extensibility, allowing the database to grow over time with minimal human effort. All artifacts related to this work are publicly available at the following links [1–3, 5]. In addition, the implementation of the prompt-based APR techniques for regression bugs used in our experiments is also publicly available [4].

The remainder of this paper is organized as follows. Section 2 provides background information and discusses related work on regression bugs and automated program repair. Section 3 describes the study setup, including the research and experimental designs. Section 4, 5 and 6 reports the experimental results, followed by Section 7, which explores the implications of our findings and the threats to validity. Finally, Section 8 summarizes our conclusions and outlines directions for future work.

2 Background and Related Work

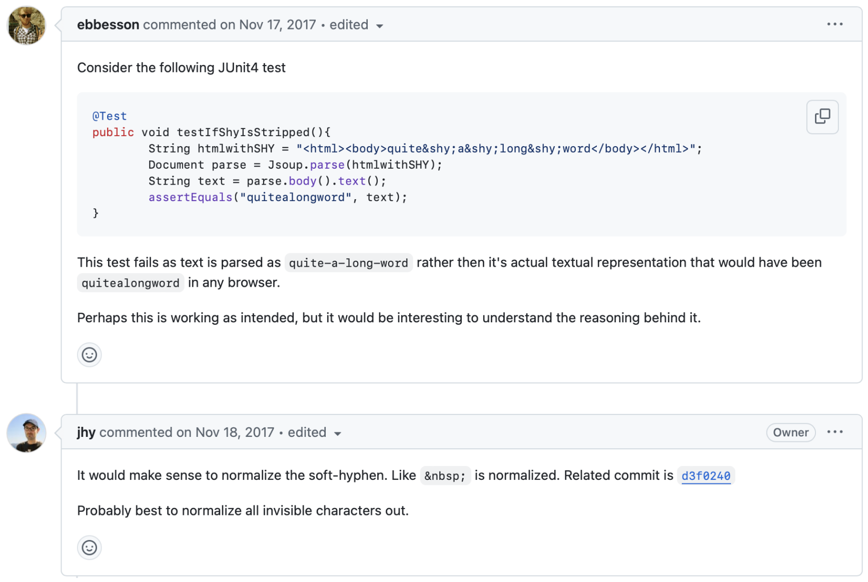
In this section, we first introduce the concept of regression bugs, illustrate their characteristics, and review existing benchmarks designed specifically on such bugs. We then discuss related work on program repair benchmarks and automated program repair (APR) techniques.

2.1 Regression Bug

Regression bug is a common class of bugs that occur when a feature that previously worked correctly stops functioning after certain changes, such as bug fixes, new feature additions, code refactoring, or other software modifications. A regression test is designed to ensure that existing functionality or intended behavior remains intact during software evolution. When such a test fails, indicating that a recent change has unintentionally reverted the software to a "less developed state", the associated bug is referred to as a *regression bug* [44].

To better understand the different causes of regression bugs, Tan et al. [56] categorize regression bugs into three types based on how the regression was introduced: *(i) Local:* where a code modification directly breaks existing functionality within the modified program element; *(ii) Unmask:* where a code modification exposes an existing bug that had previously not affected the behavior of some tests; and *(iii) Remote:* where a code modification introduces a bug in an unchanged program element, leading to failures elsewhere in the code base. Intuitively, if a functionality worked in a previous version, a *local* regression can often be fixed by reverting to the previous implementation. However, reverting may unintentionally undo newly intended updates associated with the modification. Conversely, for an *unmask* regression, where the earlier code effectively concealed an underlying bug, fixing the issue may involve restoring conditions that mask the problematic changes. In such cases, repair efforts focus on finding a scenario where the changes no longer impact test behavior, without simply reverting to the previous code, as doing so would re-hide the bug and fail to address the underlying problem.

2.1.1 An Illustrative Example. We present an example of a regression bug, illustrating how code changes unexpectedly create errors while introducing new features or bug fixes. This example, corresponding to *RegressionBug-42* in our benchmark, comes from the *jsoup* [1] project, a Java library designed to simplify working with real-world HTML and XML.



ebbesson commented on Nov 17, 2017 • edited

Consider the following JUnit4 test

```
@Test
public void testIfShyIsStripped(){
        String htmlwithSHY = "<html><body>quite&shy;a&shy;long&shy;word</body></html>";
        Document parse = Jsoup.parse(htmlwithSHY);
        String text = parse.body().text();
        assertEquals("quitealongword", text);
}
```

This test fails as text is parsed as `quite-a-long-word` rather then it's actual textual representation that would have been `quitealongword` in any browser.

Perhaps this is working as intended, but it would be interesting to understand the reasoning behind it.

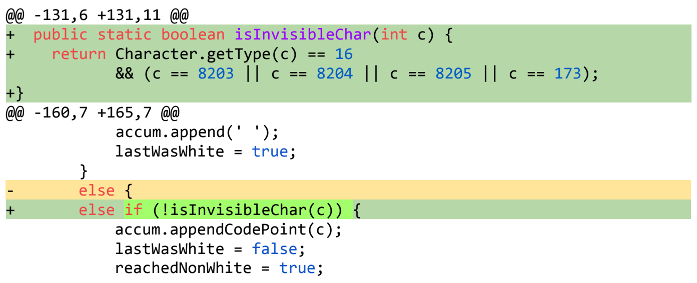
jhy commented on Nov 18, 2017 • edited Owner

It would make sense to normalize the soft-hyphen. Like `&nbsp;` is normalized. Related commit is d3f0240

Probably best to normalize all invisible characters out.

Fig. 1. An issue reported requiring developers to resolve; however, after fixing it, another bug is inadvertently introduced. See the original issue at: *https://github.com/jhy/jsoup/issues/978*

Bug-inducing changes – This regression bug occurred when developers attempted to fix an issue raised by the user as shown in Figure 1. In this issue, users pointed out that the current version requires an enhancement of text rendering and formatting features as the *&shy* character was displaying as a hyphen (-). To resolve this, developers constructed the `isInvisibleChar` method to normalize text by omitting invisible characters, including *zero-width space (ZWSP)*, *zero-width non-joiner (ZWNJ)*, *zero-width joiner (ZWJ)*, and *soft hyphen (SHY)*, as illustrated in Figure 2. Unfortunately, removing the *ZWNJ* and *ZWJ* caused a change in the emoji's length after applying this function; an unintended behavior that was only identified in other issues after several years (see Figure 3).

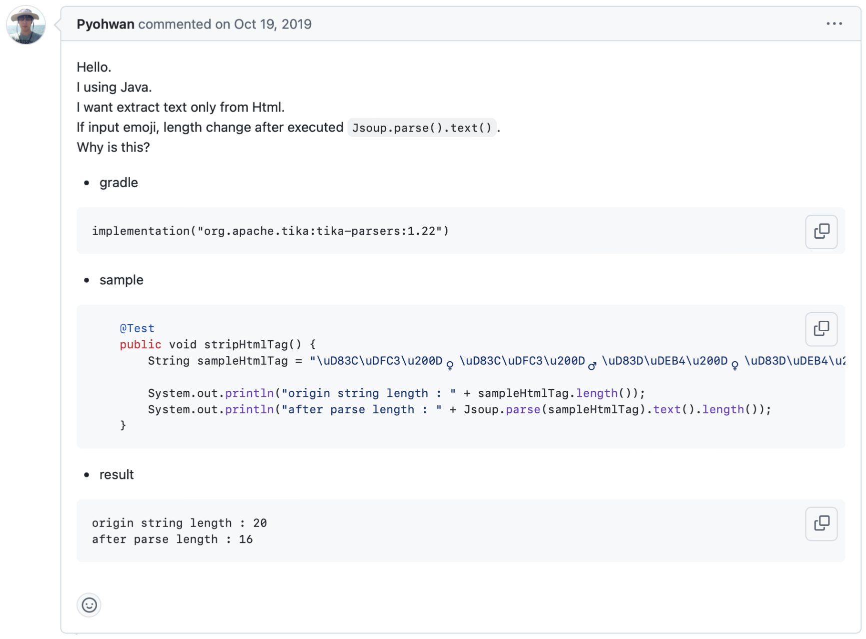


```
@@ -131,6 +131,11 @@
+  public static boolean isInvisibleChar(int c) {
+    return Character.getType(c) == 16
            && (c == 8203 || c == 8204 || c == 8205 || c == 173);
+}
@@ -160,7 +165,7 @@
            accum.append(' ');
            lastWasWhite = true;
        }
-       else {
+       else if (!isInvisibleChar(c)) {
            accum.appendCodePoint(c);
            lastWasWhite = false;
            reachedNonWhite = true;
```

Fig. 2. Regression-inducing changes where the intention was to fix an issue, but an unintended regression was introduced

[1]Available at: *https://github.com/jhy/jsoup*

Bug-fixing changes - As illustrated in Figure 4, to fix the regression bug, developers modified the `isInvisibleChar` method to selectively only drop the characters requiring removal, specifically *SHY* and *ZWSP*, while preserving functional invisible characters such as *ZWJ* and *ZWNJ*. Although this fix is relatively straightforward, it necessitates a deeper understanding of the underlying bugs and the initial purpose of bug-inducing changes. The root cause of the issue lies in the overly broad handling of invisible characters in the `isInvisibleChar` method. The original intent of the text normalizer was to remove unnecessary formatting characters (e.g., *ZWSP* and *SHY*) to facilitate cleaner text processing. However, the approach unexpectedly removed *ZWJ* and *ZWNJ* characters, which are essential for joining characters into meaningful sequences. In particular, in contexts such as emojis or in scripts such as Arabic. The bug fix, therefore, needs to ensure that *ZWJ* and *ZWNJ* are retained, while still allowing for the removal of *ZWSP* and *SHY*.



Pyohwan commented on Oct 19, 2019

Hello.
I using Java.
I want extract text only from Html.
If input emoji, length change after executed `Jsoup.parse().text()`.
Why is this?

- gradle

```
implementation("org.apache.tika:tika-parsers:1.22")
```

- sample

```
    @Test
    public void stripHtmlTag() {
        String sampleHtmlTag = "\uD83C\uDFC3\u200D♀ \uD83C\uDFC3\u200D♂ \uD83D\uDEB4\u200D♀ \uD83D\uDEB4\u2

        System.out.println("origin string length : " + sampleHtmlTag.length());
        System.out.println("after parse length : " + Jsoup.parse(sampleHtmlTag).text().length());
    }
```

- result

```
origin string length : 20
after parse length : 16
```

Fig. 3. An issue reported requiring developers to resolve. This issue is caused by the fix of a problem introduced two years prior (see Figure 1). See the original issue at: *https://github.com/jhy/jsoup/issues/1269*.

We can see that the generation of this fix required more than simply adjusting the buggy version to pass the test cases. For instance, reverting the code to the working commit would indeed pass the test case, but it would also undo the feature improvements made by the developers. Thus, careful consideration was needed to ensure the changes did not negatively impact other functions. This was accomplished by examining the bug-inducing changes and comparing the working version before the modifications with the failures introduced afterwards.

```
@@ -132,8 +132,8 @@
   public static boolean isInvisibleChar(int c) {
-     return Character.getType(c) == 16
             && (c == 8203 || c == 8204 || c == 8205 || c == 173);
+     return c == 8203 || c == 173;
   }
```

Fig. 4. Regression-fixing changes made by developers to resolve the regression bug.

2.1.2 Regression Bug Benchmark. To facilitate regression error research, two notable datasets have been developed for studying regression bugs: CoREBench [11], which was used by Relifix, and the more recent CIBugs [24]. CoREBench, introduced in 2014, comprises a collection of 70 regression errors extracted from four small to medium open-source C/C++ projects (less than 35KLOC): Make, Grep, Findutils, and Coreutils, which are publicly accessible through GNU homepage [2]. The dataset has not been updated since its release. Moreover, due to quality concerns, such as dissatisfaction with how regression errors were defined and challenges with reproducibility, Relifix ultimately utilized only a subset of the dataset in its evaluation [56]. The more recent benchmark, CIBugs, utilizes data from continuous integration (CI) systems. CIBug's strategy leverages the TravisTorrent database [9], which contains metadata from 2,640,825 Travis builds synthesized from Travis CI's API and build logs, last updated on February 8, 2017. The regression errors captured in CIBugs are identified through continuous integration testing, where existing tests (i.e., regression tests) that previously passed begin to fail after code changes. However, CIBugs lacks information on bug-inducing commits necessary for forming regression bugs, and identifying these commits remains a challenging task [7]. Furthermore, CIBugs requires strenuous human engineering efforts to manually curate the bugs, rendering it hard to scale.

2.1.3 RegMiner - A Regression Bug Mining Tool. While existing datasets rely heavily on manual curation, RegMiner offers an automated approach for identifying regression bugs from software evolution history [55]. Specifically, RegMiner constructs a regression bug by searching in code repositories for a bug-fixing commit (*bfc*), a bug-inducing commit (*bic*), and a test case (t) such that t passes on *bfc* and a version before *bic*, but fails on *bic* and a version before *bfc*.

To begin, RegMiner collects bug-fixing commits that introduce new test cases from version histories. A commit is confirmed as a bug-fixing commit if the added test case passes on the current version and fails on the version before it, which serves as a prerequisite to search the regression bug. RegMiner then applies a measurement heuristic to prioritize bug-fixing commits that are more likely to be regressions. Next, for each candidate *bfc*, RegMiner searches for a corresponding bug-inducing commit *bic*, where the test case fails on *bic* but passes on the version before *bic*. If such a commit is found, a regression instance is recorded as a tuple (*bfc*, *bic*, t).

This process encounters several technical challenges: *(i) Futile search on non-regression bug-fixing commits:* Searching through commit history is time-consuming, so RegMiner introduces a heuristic to quantify the likelihood of a commit being a regression-fixing commit; *(ii) Test dependency migration:* Verifying whether a test fails or passes on prior versions is complicated as code and dependency changes over time, especially when regressions were introduced years earlier and the project has since undergone significant refactoring or library updates; and *(iii) High overhead for validation:* Confirming a regression requires checking out, compiling, and running tests across potentially thousands of commits, which incurs significant computational cost.

To overcome these challenges, RegMiner introduces three algorithms: *(i)* a heuristic for estimating potential bug-fixing commits, *(ii)* a test migration algorithm, and *(iii)* a validation effort minimization algorithm. These strategies enable scalable mining. The *validation effort minimization* algorithm is a heuristic designed to reduce validation costs and to handle cases that suffer from incompatibility due to test migration. However, this approach can lead to incorrect identification of regression bugs. To address this problem, we develop a validation tool to ensure the reproducibility, correctness, and the overall quality of each mined regression instance.

2.2 Program Repair

The following is an overview of existing program repair benchmarks and of automated program repair methodologies.

[2]Availabel at: *http://savannah.gnu.org* and *http://debbugs.gnu.org*

2.2.1 Program Repair Benchmarks. High-quality datasets are critical for evaluating the effectiveness of APR techniques and facilitating rigorous comparisons across different approaches. Over time, researchers have introduced various benchmarks to meet these needs.

Defects4J [23], one of the most widely used APR benchmarks, contains 835 real-world bugs from Java projects. Bugs.jar [50], which includes 1,158 reproducible bugs collected from Apache projects, follows a similar construction process but expands the benchmark's size and covers different categories of Java applications. Bears [37] and BugSwarm [58] use automated processes to construct their datasets but differ in how bugs are reproduced. Bears includes 251 reproducible bugs, while BugSwarm collects unprocessed bugs from failed CI builds, yielding 3,091 pairs of failing and passing builds. However, a critical analysis [14] shows that only 50 Java and 62 Python bugs in BugSwarm are actually suitable for evaluating automated fault localization or program repair techniques. A recent study [34] indicated that approximately 90% of the bugs in Defects4J, Bugs.jar, and Bears are associated with test cases that were added or modified after the bugs were reported. These datasets are not constructed with regression-specific characteristics in mind. The bugs are not revealed through regression testing, making it unclear whether they truly represent regressions. Therefore, these datasets lack the temporal and behavioral properties that define regression bugs.

In addition to Java benchmarks, several datasets have been developed for other programming languages. For C/C++, ManyBugs [29] includes 185 bugs from nine large, popular open-source programs. IntroClass [29] contains 998 student-written programs with bugs, collected from small programming assignments in an introductory course. Codeflaws [57] includes 3,902 bugs extracted from Codeforces programming contests. Other notable benchmarks include BugsInPy [59] for Python and BugsJS [18] for JavaScript.

In contrast to these general software bug benchmarks, our proposed benchmark, RegMiner4APR, is specifically focused on regression bugs. It includes regression-revealing test cases, along with the corresponding bug-inducing and bug-fixing commits. Unlike non-regression benchmarks, where each bug is typically identified by one or more failing test cases, RegMiner4APR includes both a historical passing version and a subsequent failing version. This specificity distinguishes our benchmark and provides a valuable resource for understanding the characteristics of real-world regression bugs.

2.2.2 Automated Program Repair. Traditional APR techniques can be broadly categorized into three main groups: *search-based*, *template-based*, and *constraint-based* approaches. *Search-based* and *Pattern-based approaches* searches for program variants that retain the required functionality while addressing the identified bug. Notable examples leveraging genetic programming include GenProg [30], Kali [38], HDRepair [26], Arja [69], and jMutRepair [38]. Another common strategy leverages predefined fix templates, either manually defined or mined from code repositories. Notable examples are TBar [33] and Cardumen [39]. Relifix [56], applied to C/C++ regression errors, reduces the search space by applying code transformations based on syntactic change patterns observed across multiple software versions. *Constrain-based approaches* represent the space of patch candidates using symbolic constraints. By encoding the desired program behavior as a set of logical constraints, these approaches synthesize code that satisfies both the specification and the test suite. Notable techniques are SemFix [42], Nopol [64], Angelix [40], and S3 [25]. In our study, we do not evaluate semantic-based APR approaches as they often require the inclusion of manually-written models for system calls whose source codes are not available.

Shifting to *deep learning-based* techniques, these methods are typically trained from scratch on large bug-fix corpora to automatically learn repair patterns from human-written patches. Once trained, the models can be applied to generate patches for buggy programs. Recent state-of-the-art techniques are CURE [22], RewardRepair [66], Recoder [71], and

KNOD [21], NeverMore [6], among others. More recently, the rapid progress of large language models (LLMs) has given rise to a new line of *LLM-based APR techniques*, which have demonstrated substantial improvements over prior deep learning-based methods [20]. Due to their superior capabilities, our study places significant emphasis on LLM-based APR. These LLM-based methods typically utilize large pretrained model for code generation and can be grouped into two main categories: *fine-tuning-based* and *prompt-based* techniques. *Fine-tuning-based approaches* focus on refining the weights of code-specific LLMs, such as CodeLLama [48], Incoder [16] and CodeGen [43], to improve performance on bug-fixing tasks. Notably, Jiang et al. [20] demonstrated that full-parameter fine-tuning significantly enhances APR performance, achieving remarkable improvements over the base models. Silver et al. [52] investigated the effectiveness of parameter-efficient fine-tuning using QLoRA [13]. Their work evaluated various input-output formatting strategies and introduced RepairLLama, a fine-tuned version of CodeLLaMA tailored specifically for program repair. *Prompt-based approaches* aim to directly leverage LLMs without finetuning, mainly is prompting. They design and evaluate different prompting strategies to provide repair information to the model. Xia et al. [61] proposed ChatRepair, a conversation-driven APR approach that interacts with ChatGPT using contextual feedback such as test names, relevant test code, and error messages to iteratively guide the repair process. Yin et al. [67] applied chain-of-thought prompting and few-shot learning techniques to further enhance the performance of LLMs on APR tasks.

3 Study Setup

In this section, we describe the setup of our empirical study. We begin by outlining the research questions and presenting the overall workflow of the study. We then provide details of the benchmark construction process. Following that, we describe the selection criteria and experimental settings of the automated program repair (APR) techniques under our evaluation. Finally, we present the evaluation metrics used to assess the effectiveness of the repair methods.

3.1 Research Design

Our study is guided by the following research questions:

RQ1. *What is the degree of quality and diversity of the RegMiner4APR benchmark for APR studies?* This research question aims to examine the overall quality and characteristics of the RegMiner4APR benchmark, with the goal of assessing its suitability for supporting research in APR.

RQ2. *How effective are existing APR techniques at repairing regression bugs?* This research question aims to investigates the effectiveness of traditional and advanced APR techniques in repairing regression bugs.

RQ3. *What is the impact of providing additional context, particularly bug-inducing change information, on APR techniques?* This research question aim to explore whether providing additional context, specifically bug-inducing change information, can enhance the repair performance.

Our study begins with the construction of the RegMiner4APR benchmark, which includes 99 regression bugs collected from 32 widely used real-world Java GitHub repositories. Since data availability has long posed a challenge for APR research [23, 24, 37, 58], existing regression bug datasets are outdated, and difficult to reproduce. In addressing RQ1 (Section 4), we analyze our benchmark construction process and conduct an in-depth analysis of human-written patches to demonstrate the quality and diversity of the dataset. Building on this high-quality benchmark, we address RQ2 (Section 5) by empirically evaluating several APR techniques, including both traditional and advanced LLM-based approaches, on their ability to repair regression bugs. These evaluations are performed without considering the regression-inducing context, aligning with the conventional APR workflow, that typically take as input the buggy program and its associated test suite. Finally, in addressing RQ3 (Section 6), we focus on prompt-based APR techniques, motivated by their demonstrated effectiveness, inherent flexibility in model usage and prompt construction, and greater data efficiency compared to fine-tuning-based approaches. We investigate whether incorporating regression-specific context, specifically bug-inducing change information, into the prompt design can enhance repair effectiveness. Furthermore, we conduct a qualitative analysis of plausible patches to examine how models leverage this contextual information to repair regression bugs.

3.2 Benchmark Construction

In this section, we outline the process for constructing a benchmark of regression bugs. Figure 5 illustrates an overview of the benchmark construction.

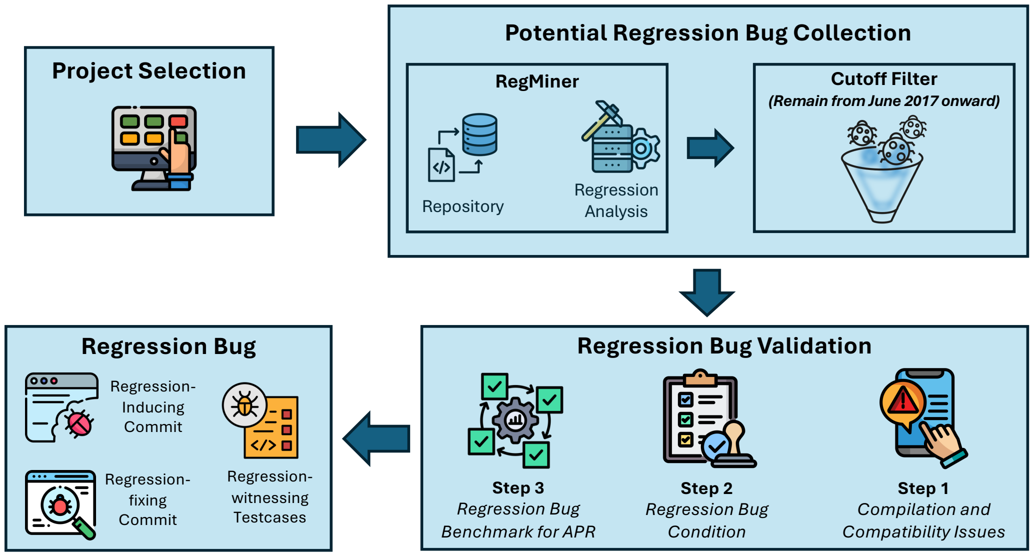

Project Selection
Potential Regression Bug Collection
RegMiner
Repository
Regression Analysis
Cutoff Filter
(Remain from June 2017 onward)
Regression Bug Validation
Step 3
Regression Bug Benchmark for APR
Step 2
Regression Bug Condition
Step 1
Compilation and Compatibility Issues
Regression Bug
Regression-Inducing Commit
Regression-witnessing Testcases
Regression-fixing Commit

Fig. 5. Overview of the RegMiner4APR benchmark construction.

3.2.1 Project Selection. In this study, we focus on Java regression bugs for two reasons. First, Java remains one of the most widely used programming languages today, as consistently ranked in the TIOBE Index [3]. Second, while there has been a substantial body of research on regression errors in C/C++ projects [11, 44, 56, 68, 70], most of it dates back several years, leaving a gap in up-to-date studies on Java regression errors. Fortunately, the RegMiner tool [55] provides support for mining regression bugs from Java repositories which enables us to efficiently construct our benchmark.

Accordingly, we select well-known libraries and tools hosted on GitHub, implemented in Java, that meet the following criteria: *(i)* The project has more than 500 commits, used as a proxy to ensure sufficient development activity and code evolution; *(ii)* The project uses Maven or Gradle as the build system and JUnit test cases, ensuring compatibility with RegMiner tool and test execution environment; *(iii)* The project is not related to mobile or distributed platforms (e.g., Android), which often require specialized environments (e.g., emulators or network configurations) that complicate reproducibility; and *(iv)* The project is not structured as a multi-module build, as multi-module builds introduce

[3] Available at: https://www.tiobe.com/tiobe-index/

complexity in dependency resolution and test isolation, which can hinder accurate mining, compiling, and test execution during regression analysis. Following these criteria, we sampled and selected 60 Java repositories[4] hosted on GitHub. This sample size was chosen due to the time-intensive nature of the bug mining tool, RegMiner, as well as the manual effort required for subsequent steps such as patch minimization and bug analysis.

3.2.2 Potential Regression Bug Collection. Then, we leverage RegMiner [55], a well-known regression mining tool to identify potential regression bugs from the selected GitHub repositories. In total, we identified 1274 potential regression bug candidates from these repositories. To ensure that our dataset is both timely and up-to-date, we filtered this set to retain only those whose bug-fixing commits were introduced from June 2017 onward. We chose this cutoff because the most recent existing dataset on Java regression errors, CIBugs [24], primarily includes bugs from before mid-2017, and we aim to provide a more up-to-date benchmark. After applying this filter, we retained 563 potential regression bugs for further validation.

3.2.3 Regression Bug Validation. However, it is possible that not all extracted candidates represent true regressions since RegMiner employs heuristics in its mining process. Therefore, after collecting the set of potential regression bugs, we developed an automated validation tool to identify true regression bugs. This tool applies a series of filtering criteria to evaluate the relevance and accuracy of each case as follows:

- *Step 1: Compilation and Compatibility Issues (Executability).* This step is necessary because RegMiner uses test cases extracted from the bug-fixing commit, which need to be migrated to earlier versions using rule-based heuristics. Our validation tool takes as input the potential bug-inducing commit, bug-fixing commit, and the corresponding test case that exposes the regression. It then attempts to reconstruct the four relevant commit versions with migrated test cases. However, 179 of the 563 potential regression bugs encountered compilation or compatibility failures during this step. After filtering out these cases, 384 potential regression bugs remained.
- *Step 2: Regression Bug Conditions (Validity).* This step ensures the behavioral conditions of a true regression bug are satisfied, as RegMiner may yield false positives. Specifically, our validation tool verifies that the test case passes on both the bug-fixing commit and the version prior to the bug-inducing commit, and fails on both the bug-inducing commit and the version prior to the bug-fixing commit. Bugs that do not satisfy this condition are discarded. After this step, 161 potential regression bugs remained.
- *Step 3: Regression Bug Benchmark for APR (Utility).* This step ensures that the regression bugs included in the benchmark are suitable for automated program repair (APR). We integrated an additional condition into our validation tool to check that the buggy version fails only the test cases that exposes the regression, and that the bug-fixing commit passes all test cases. Although this may exclude some complex but valid regressions (e.g., those affecting multiple test cases beyond the ones exposing the regression), it helps construct a clean and regression-focused benchmark tailored for APR research.

Ultimately, through this process, we curated a final dataset of 99 confirmed regression bugs, drawn from 32 distinct GitHub repositories.

3.2.4 Data Storage. To ensure extensibility and durable reproducibility, we store the validated regression bugs in a structured database hosted on GitHub. This centralized storage facilitates easy access and management of the regression bug dataset. In addition, we have developed an interface on top of this database to streamline usage for researchers,

[4]Available at: https://github.com/brojackvn/RegMiner4APR-Framework/blob/main/GITHUB_REPO_MINING.MD

support reproducible studies in program repair, and enable future extensions without requiring environment-specific setup for each individual bug. The key characteristics of our storage design include:

- *Durable reproducibility:* To guard against future dependency issues (e.g., broken builds due to removed or updated dependencies), we clone each required commit version along with all necessary resources and dependencies. For Maven-based projects, we use the built-in command `mvn dependency:copy-dependencies`; for Gradle-based projects, we implemented a custom function to extract and preserve all required dependencies.
- *Extensibility:* The database is structured to support future expansions of the benchmark. Each regression bug is represented by four key snapshots: the *bug-inducing* commit and its immediate predecessor, and the *bug-fixing* commit and its immediate predecessor. Each regression instance is stored within a branch in our GitHub repository, allowing new cases to be added easily.
- *Easy of Use:* To support ongoing research, we integrated the benchmark into a framework that provides essential commands to interact with the benchmark without extensive configuration. This framework offers uniform access to build and execution tasks by abstracting the underlying build systems. Supported commands include `info`, `env`, `checkout`, `compile`, and `test`.

At this stage, we finalize the construction and storage of the RegMiner4APR benchmark. To enable further analysis and better support APR studies, we extract both the bug-fixing changes (i.e., human-written patches) and the bug-inducing changes.

3.2.5 *Bug-inducing and Bug-fixing Change Extraction and Minimization.*

Since developers may introduce unrelated edits alongside bug fixes, isolating the critical changes associated with each regression error is essential for accurate benchmark analysis and for supporting APR studies (e.g., patch validation). To achieve this, we extract both the bug-inducing and bug-fixing changes by performing the following two steps, which aim to best capture the modifications relevant to each regression error.

Step 1: Test Execution and Coverage Collection. We begin by executing the bug-witnessing test cases on both the bug-inducing and bug-fixing commits to collect line-level code coverage. By comparing the coverage data with the corresponding code changes, we discard any lines not covered by the test cases and retain only those directly involved in triggering or fixing the bug.

Specifically, to isolate the bug-inducing and bug-fixing changes, we first collect the code coverage data from the relevant commits by executing the bug-witnessing test cases using JaCoCo[5], an industry-grade Java code coverage library. This process produces a `jacoco.exec` file, which contains the raw coverage information for the executed classes and methods. The coverage data is obtained by running the targeted test cases with the following command:

```
mvn test -Dtest="test_case_1;...;test_case_n"
-DargLine="-javaagent:.../jacocoagent.jar=destfile=jacoco.exec"
```

After execution, we generate the coverage report from the collected data using:

```
mvn org.jacoco:jacoco-maven-plugin:0.8.8:report -Djacoco.dataFile=jacoco.exec
```

From the resulting coverage report, we extract the set of lines covered by test cases. The overall coverage information is represented as: $\mathcal{G} = \{\text{file}_i \mapsto \{\text{cov}_{i1}, \text{cov}_{i2}, \dots\} \mid i = 1, \dots, n\}$, where n is the number of files covered by the test cases, and cov_{ij} denotes a line number covered in file_i. We then use a diff tool to obtain the code changes in both the bug-inducing and bug-fixing commits. These changes are computed as diffs against each commit's immediate

[5]Available at: https://www.jacoco.org/jacoco/

predecessor. The code changes are represented as: $\mathcal{H} = \{\text{file}_i \mapsto \{\text{chg}_{i1}, \text{chg}_{i2}, \dots\} \mid i = 1, \dots, m\}$, where m is the number of files modified, and chg_{ij} denotes a line number modified in file_i. Finally, for each file_i, the set of relevant changes covered by the test cases is computed as the intersection: $\mathcal{R}_i = \mathcal{G}(\text{file}_i) \cap \mathcal{H}(\text{file}_i)$, where $\mathcal{R}_i$ denotes the set of changed lines in file_i that are covered by the targeted test cases.

Each commit, bug-inducing or bug-fixing, has its own computed set $\mathcal{R}_i$, representing the code changes directly exercised by the bug-witnessing test cases. These refined sets are crucial for pinpointing the specific code modifications responsible for the observed regression behavior.

Step 2: Patch Minimization through Rules. To further enhance the clarity and effectiveness of our patch analysis, we manually identify and remove code changes that do not affect the bug-inducing or bug-fixing behavior. Following Defects4J [6], we define the following rules to guide our patch minimization efforts:

- *Simple Refactoring:* Code changes that do not alter the program's functionality may be excluded. This includes comments, unnecessary new lines, and easy seeing syntactically equivalent changes change code appearance without affecting its behavior.
- *Dead Code:* Code changes that are never executed or referenced can be safely removed. This encompasses declarations of unused variables, unused import statements, unused functions, and expressions that yield no side effects.
- *Unrelated Changes:* Code changes that do not directly contribute to fixing the bug should be removed. This includes new features unrelated to the bug and code style changes made solely for aesthetic reasons.

After applying our patch minimization process, we observe that the overall patch size (i.e., the total number of added, modified, and removed lines) across all analyzed bugs decreased from 2,568 to 1,796 lines, representing a reduction of approximately 30.05%.

3.3 APR Technique Selection and Experimental Settings

In this section, we present the APR techniques and their experimental settings evaluated in our empirical study. We revisit both traditional and LLM-based approaches to assess their applicability to regression bugs.

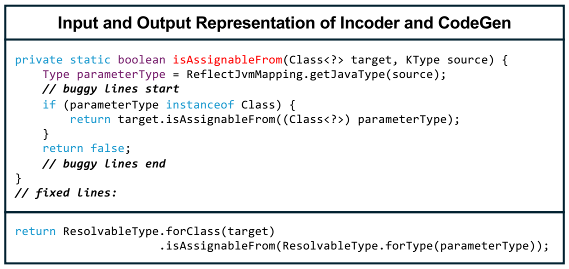
3.3.1 *Traditional APR Techniques.*

To examine whether traditional search-based and pattern-based approaches, utilizing mutation operators and fix patterns, can effectively repair regression errors, we selected a set of widely studied heuristic-based APR tools. Specifically, we include five search-based techniques: GenProg [30], Kali [47], Cardumen [39], Arja [69], and jMutRepair [38], following the selection made by Kabadi et al. [24]. For template-based APR, we include TBar [33], which systematically aggregates fixing patterns derived from prior template-based APR techniques and is widely regarded as a representative approach in the program repair literature. We do not evaluate semantic-based APR approaches in our study as they often requires the inclusion of manually-written models for system calls whose source codes are not available.

To conduct our experiments, we utilize Cerberus [51], a unified framework for running and evaluating traditional APR tools. All experiments for these tools are performed on the server equipped with an AMD CPU (16 cores) and 64GB of RAM. We retain the default configuration settings provided by Cerberus for each tool to ensure consistency and fairness.

[6]Available at: https://github.com/rjust/defects4j/blob/master/framework/bug-mining/Patch-Minimization-Guide.md

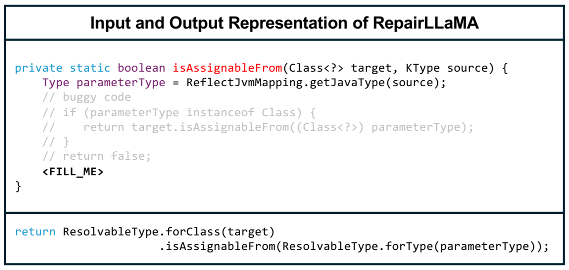
3.3.2 Fine-tuning-based APR Techniques. To investigate how well fine-tuning-based approaches perform when applied specifically to regression errors, we revisit several recently proposed fine-tuning-based APR methods. These techniques have demonstrated promising results compared to earlier deep learning-based methods [20]. Following the work of Jiang et al. [20], we include several full-parameter fine-tuned models that exhibit significant performance improvements over their original pretrained counterparts. Specifically, we evaluate Incoder-1B and Incoder-6B [16], as well as CodeGen-2B and CodeGen-6B [43]. In addition, we evaluate RepairLLaMA [52], a fine-tuned version of CodeLLaMA [48] trained using QLoRA [13], a parameter-efficient fine-tuning method. This technique is particularly interesting due to its reduced memory and computation requirements during training while still achieving strong repair performance.


Input and Output Representation of Incoder and CodeGen

```
private static boolean isAssignableFrom(Class<?> target, KType source) {
    Type parameterType = ReflectJvmMapping.getJavaType(source);
    // buggy lines start
    if (parameterType instanceof Class) {
        return target.isAssignableFrom((Class<?>) parameterType);
    }
    return false;
    // buggy lines end
}
// fixed lines:
```

```
return ResolvableType.forClass(target)
                     .isAssignableFrom(ResolvableType.forType(parameterType));
```

Fig. 6. The input and output representation of Incoder and CodeGen.


Input and Output Representation of RepairLLaMA

```
private static boolean isAssignableFrom(Class<?> target, KType source) {
    Type parameterType = ReflectJvmMapping.getJavaType(source);
    // buggy code
    // if (parameterType instanceof Class) {
    //    return target.isAssignableFrom((Class<?>) parameterType);
    // }
    // return false;
    <FILL_ME>
}
```

```
return ResolvableType.forClass(target)
                     .isAssignableFrom(ResolvableType.forType(parameterType));
```

Fig. 7. The input and output representation of RepairLLaMA.

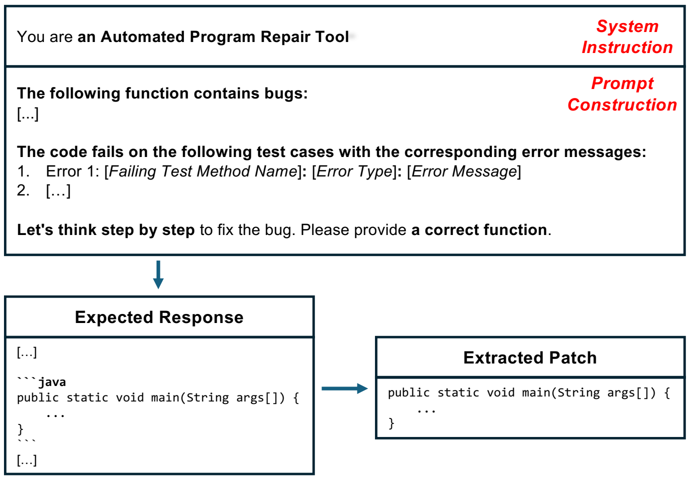
For input and output representation, we adopt model-specific configurations aligned with the respective fine-tuning procedures, as illustrated in Figure 6 and Figure 7. In particular, *Incoder* and *CodeGen* [20] require the entire buggy function along with annotations marking the buggy location. The buggy segment is highlighted using the markers *"// buggy lines start*" and *"// buggy lines end"*, followed by a placeholder line starting with *"// fixed lines:"*, which designates where the fix should be generated. The model is expected to output the corresponding fixed code chunk that replaces the buggy code. In contrast, *RepairLLaMA* [52] adopts a different input format tailored to its infilling-based approach. The input consists of a code prefix followed by a comment beginning with the marker *"// buggy code"*, which introduces the buggy lines retained in commented-out form. This is followed by the special token *"<FILL_ME>"*, which indicates the location where the model should generate the fix. The model is expected to output the corresponding fixed code chunk that replaces the buggy segment while preserving the original context.

All experiments are conducted using FLAMES [28], a unified framework for executing and configuring APR evaluations with large language models. We run all fine-tuning-based techniques on our highest-performance machine, which is equipped with a single NVIDIA A100 GPU (80GB VRAM), 250GB of system RAM, and a 32-core Intel Xeon CPU running at 2.90GHz. To maintain consistent generation quality across models, we set the temperature hyperparameter to 1.0 to control output diversity. Additionally, to prevent out-of-memory crashes on our setup, we limit the number of generated candidates using a beam size of 10.

3.3.3 Prompt-based APR Techniques. To investigate how well prompt-based approaches perform when applied specifically to regression errors, we explore techniques that leverage general-purpose large language models (LLMs), such as ChatGPT, without requiring any fine-tuning. These approaches rely solely on prompt engineering to guide the model in generating plausible patches. In this category, two common strategies have emerged: *zero-shot prompting APR* and *conversational APR.* Zero-shot prompting APR involves issuing a single prompt to the LLM to generate a patch, without relying on any interactive feedback or prior conversational context [62]. In contrast, conversational APR techniques incorporate multiple dimensions of feedback to iteratively query the model. These methods maintain a conversation history and leverage prior failed patching attempts to improve future generations through prompting [61, 67].

In our experiments, we select advanced models from the ChatGPT family to implement prompt-based repair. Following prior work by Xu et al. [63], we note that the next-token prediction objective used by decoder-only LLMs (e.g., GPT-4) is misaligned with the masked span prediction objective employed in current infilling-style methods. Since these models are used without any fine-tuning, the effectiveness of LLMs can be improved by aligning the repair task with their training objective-that is, allowing them to refine the entire program without explicitly providing buggy code hunks.

To this end, we input the entire buggy method along with relevant debugging information as part of the prompt. Following prior works [61, 62, 67], we manually examined a few alternative approaches using selected bugs through the web-based interface of ChatGPT[7]. Based on these insights, we construct our prompt as illustrated in Figure 8. More specifically, we initialize ChatGPT with the system instruction *"You are an Automated Program Repair Tool"* to explicitly define its role. We then construct the user prompt by providing various types of contextual information related to the buggy function. This includes the full body of the buggy function, followed by a list of failing test cases along with their associated error messages. Initially, only functional errors are present in the benchmark. In this case, our framework RegMiner4APR supports the extraction of key diagnostic information, including the name of the failing test case (which often serves as a concise summary of the function under test), the error type (e.g., *org.junit.ComparisonFailure*), and the detailed error message (e.g., *expected:<[6]> but was:<[1]>*), which provides concrete evidence of the failure. Finally, the prompt concludes with a chain-of-thought indicator, *"Let's think step by step to fix the bug"*, followed by the task description, *"Please provide a correct function"*. This prompt structure encourages the model to reason about the bug and generate a complete and syntactically correct replacement for the entire function, rather than returning an unformatted or incomplete patch.


You are **an Automated Program Repair Tool**

The following function contains bugs:
[...]

The code fails on the following test cases with the corresponding error messages:
1. Error 1: [*Failing Test Method Name*]: [*Error Type*]: [*Error Message*]
2. [...]

Let's think step by step to fix the bug. Please provide **a correct function**.

Expected Response

[...]
~~~
```java
public static void main(String args[]) {
    ...
}
```
[...]
~~~

Extracted Patch

```
public static void main(String args[]) {
    ...
}
```

Fig. 8. Structure of our prompt-based APR query. The code snippet and error messages are omitted for clarity. The expected response is formatted to allow reliable extraction of the repaired function from the LLM's output.

[7]Available at: https://chatgpt.com

In the case of zero-shot prompting APR, we use only the initial prompt to query ChatGPT and extract a candidate patch from the model's output. In contrast, conversational APR techniques begin with the same initial prompt but continue with an interleaved process of patch generation and validation-driven feedback. Each generated patch is immediately validated by compiling and running the program against the test suite. If the patch fails, we construct a feedback prompt that includes both the previously generated incorrect patch and the associated failure information, which is then used to prompt the next generation. We observe that there are four main cases in which feedback is provided to the model: *(i)* ***Compilation Error***, when the patch cannot be compiled; *(ii)* ***Functional Error***, when the patch compiles but fails one or more test cases; *(iii)* ***No Response Code***, when the model does not return a valid code function; and *(iv)* ***Timeout***, when the patch execution exceeds a predefined time limit, often indicating an infinite loop or inefficient implementation. For each case, we formulate a corresponding natural language feedback prompt that communicates the type of failure and requests a corrected function. These feedback templates are summarized in Table 1.

Table 1. Prompt Templates for Feedback Cases

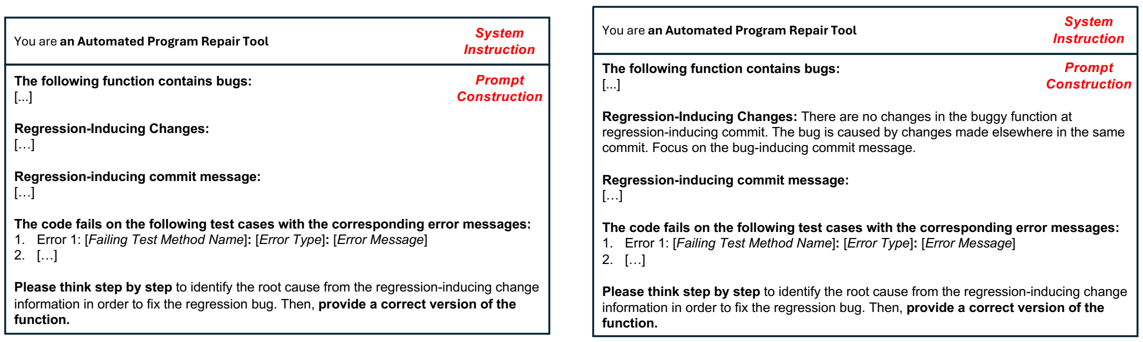
Case	**Prompt Template**
Compilation Error	The fixed version is not compilable. The code has the following compilation error: **{error_message}** Please provide the correct function along with any required imports.
Functional Error	The fixed version is still not correct. The code fails on the following test cases with the following error messages: **{error_message}** Please provide the correct function again.
No Response Code	The response does not provide the code function. Please provide the correct function again.
Timeout	The fixed version is still not correct and ran out of time. Please provide the correct function again.

The repair tool is developed in Python and communicates with the ChatGPT API endpoint[8]. We use *gpt-3.5-turbo-0125* and *gpt-4o-2024-08-06*, which were the most up-to-date and cost-effective models at the time of implementation,. Inference and evaluation scripts are executed on a server equipped with a 16-core AMD CPU and 64GB of RAM. To control a moderated randomness of text generation, we follow prior work and set the *temperature* hyperparameter to 1.0, which is also the default value provided by OpenAI. Additionally, we adopt a relatively small sampling size of 10 candidates per bug, given the significantly higher inference cost (in terms of both time and financial budget) of LLMs compared to other APR tools. This setting also ensures a fair comparison with the fine-tuning-based approaches evaluated in this study.

For the conversational strategy, since ChatGPT has a limited context window[9], meaning it cannot process arbitrarily long input sequences, we introduce a threshold on conversation length (i.e., the number of prompt-response exchanges in a continuous repair session). Once this maximum conversation length is reached, the repair process is restarted from the initial prompt. A maximum conversation length of 1 corresponds to zero-shot prompting APR. As the maximum conversation length increases, the model can incorporate more repair history (e.g., previous patches and corresponding feedback) into its prompts. In our experiments, we retain the sampling size of 10 and set the maximum conversation length to 5. This setting still ensures a fair comparison, as it assumes that each new prompt in the conversation builds upon the model's previous patch rather than generating a completely new patch.

[8]Available at: https://platform.openai.com/docs/overview
[9]Available at: https://platform.openai.com/docs/guides/chat

3.3.4 Prompt-based APR Techniques Augmented with Bug-Inducing Change Information. To explore the potential of leveraging regression information in automated program repair, we focus on prompt-based techniques, which are motivated by their effectiveness, inherent flexibility in model usage and prompt construction, and greater data efficiency compared to fine-tuning-based approaches. In particular, large language models, such as ChatGPT, have demonstrated remarkable generalization capabilities and an ability to understand and generate code from natural language instructions [35]. These strengths make LLMs especially well-suited for our investigation. We hypothesize that providing LLMs with additional contextual information, specifically bug-inducing change information, can significantly enhance their ability to fix regression bugs. Identifying where and how a regression was introduced offers valuable insight that can support the patch generation process.



You are **an Automated Program Repair Tool**

System Instruction
Prompt Construction

The following function contains bugs:
[...]

Regression-Inducing Changes:
[…]

Regression-inducing commit message:
[…]

The code fails on the following test cases with the corresponding error messages:
1. Error 1: [*Failing Test Method Name*]**:** [*Error Type*]**:** [*Error Message*]
2. […]

Please think step by step to identify the root cause from the regression-inducing change information in order to fix the regression bug. Then, **provide a correct version of the function.**

(a) Regression-inducing commit includes changes to the buggy function.

You are **an Automated Program Repair Tool**

The following function contains bugs:
[...]

Regression-Inducing Changes: There are no changes in the buggy function at regression-inducing commit. The bug is caused by changes made elsewhere in the same commit. Focus on the bug-inducing commit message.

Regression-inducing commit message:
[…]

The code fails on the following test cases with the corresponding error messages:
1. Error 1: [*Failing Test Method Name*]**:** [*Error Type*]**:** [*Error Message*]
2. […]

Please think step by step to identify the root cause from the regression-inducing change information in order to fix the regression bug. Then, **provide a correct version of the function.**

(b) Buggy function is unchanged, and the fault is introduced by other changes in the regression-inducing commit.

Fig. 9. Prompt templates designed for prompt-based APR with regression-inducing change information.

In our experiments, we extend existing prompt-based APR frameworks by adapting the prompt design to explicitly incorporate regression-specific context. While our implementation maintains the two repair strategies, *zero-shot prompting* and *conversational APR*, we design the prompt to align with our research objective of investigating how augmenting prompts with bug-inducing change information affects repair effectiveness. The overall interaction structure, inference process, and configuration settings remain consistent with the prompt-based APR techniques described earlier in this section.

The primary distinction in this approach lies in the design of the prompt. Figure 9 illustrates two prompt templates used in prompt-based APR for regression bugs, each corresponding to a different scenario based on how the regression was introduced. Specifically, we initialize ChatGPT with the system instruction *"You are an Automated Program Repair Tool"* to clearly define its role. The user prompt is then constructed by providing contextual information related to the buggy function, including its full body and a list of failing test cases along with their corresponding error messages. At the initial stage, our benchmark includes only functional errors. To support this process, we use RegMiner4APR, which extracts diagnostic information from the failing test cases, similar to the prompt-based APR setup described earlier. In the case where the regression-inducing commit includes changes to the buggy function, we extract the corresponding code diff and commit message to provide additional context, as shown in Figure (9a). This additional information helps guide the model toward understanding how the changes may have introduced the regression. In contrast, some regression bugs are caused by changes made elsewhere in the same regression-inducing commit, leaving the buggy function itself unchanged. In such cases, as illustrated in Figure (9b), we explicitly note that the buggy function was not modified and highlight the commit message as a key signal for localizing the root cause of the errors. Finally, each prompt concludes with a specialized chain-of-thought instruction tailored for regression scenarios: *"Let's think step by step to identify the root cause from the regression-inducing change information in order to fix the regression bug"*, followed by the task instruction: *"Then, provide a correct version of the function"*. This structure encourages the model to reason about the root cause of the regression and generate a complete and syntactically correct replacement for the function, rather than producing an unstructured or partial patch.

3.4 Evaluation Metrics

Following prior work, we adopt two widely used metrics for evaluating APR tools: *(i) the number of correct patches* and *(ii) the number of plausible patches*. A *plausible patch* is one that passes all test cases, while a *correct patch* is semantically or syntactically equivalent to the developer's reference patch. We follow common practice in APR and manually assess semantic equivalence to determine correct patches.

Specifically, we compute three derived metrics: plausible rate (PR), correct rate (CR), and precision (P). The correct rate and plausible rate indicate the overall effectiveness of APR tools, while precision reflects the reliability of the generated patches with respect to the overfitting problem [27, 53].

$$PR = \frac{\text{\#Plausible Patches}}{\text{\#Bugs in Dataset}} \qquad CR = \frac{\text{\#Correct Patches}}{\text{\#Bugs in Dataset}} \qquad P = \frac{\text{\#Correct Patches}}{\text{\#Plausible Patches}}$$

4 RQ1: Quality and Diversity of the RegMiner4APR Benchmark

To validate the quality and diversity of the RegMiner4APR benchmark, we conduct an analysis of its construction process and perform a detailed study of its overall characteristics. Our methodology consists of two parts: assessing the quality of the benchmark construction and characterizing the features of the benchmark.

4.1 Benchmark Quality

We systematically analyze each filtering and validation step applied during the benchmark construction process, as detailed in Section 3.2. At each step, we record the number and proportion of potential regression bugs filtered out, as presented in Figure 10. This allows us to quantitatively assess the effectiveness of each criterion and demonstrate the overall reliability and soundness of the resulting RegMiner4APR benchmark.

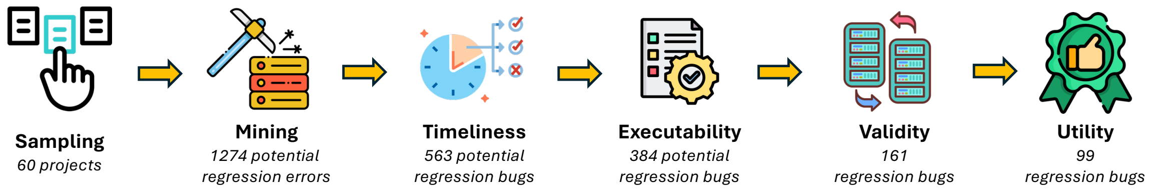


Fig. 10. Number of bugs filtered out during the regression validation process.

Overall, our validation process filters out 92.15% of the initially mined candidates, retaining only those that meet the criteria for true regression bugs. The resulting dataset not only preserves the defining behavioral characteristics of regression errors but is also specifically designed to support research on automated regression bug repair.

4.2 Benchmark Characterization

We characterize the features of RegMiner4APR benchmark by conducting an in-depth analysis of its key properties. Specifically, we examine overall benchmark statistics and provide a detailed study of human-written patches, focusing on their associated repair operators.

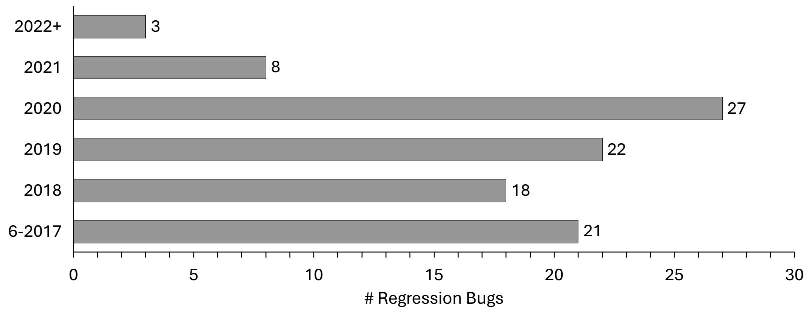


Fig. 11. Time distribution of the regression bugs in our benchmark

4.2.1 Subject Dataset: RegMiner4APR. Our benchmark consists of 99 regression bugs extracted from 32 open-source software projects hosted on GitHub. For each bug, RegMiner4APR provides four snapshots of the project versions: the regression-fixing commit, the regression-inducing commit, and their immediate predecessors. Additionally, RegMiner4APR bugs are characterized as follows: *(i)* they have been selected after June 2017, as shown in Figure 11; *(ii)* they are related to source code fixes, with exclusions for changes within the build system, configuration files, documentation, or tests; *(iii)* they are reproducible, as each bug contains at least one test case that satisfies the conditions for regression errors mentioned in Section 3.2; and *(iv)* they are isolated, ensuring that regression-inducing changes and regression-fixing patches do not include unrelated modifications, such as new features or refactoring.

The detailed information about each bug are included in our artifacts, as provided in Section 1. Each bug in RegMiner4APR is referenced using simple notation: *"RegressionBug-"* followed by a bug ID ranging from 1 to 100.

4.2.2 Data Statistics. We begin by analyzing basic properties of patches in our benchmark. In particular, we focus on patch size and spreading scope, which help us understand the complexity and challenges of bug fixes.

Table 2. Descriptive statistics for patch size and scope

	Min	25%	50%	75%	90%	95%	Max
# Added lines	0	0	1	8	18	27	159
# Removed lines	0	0	0	2	16	47	111
# Modified lines	0	0	1	2	5	14	55
# Patch size	1	1	4	13	33	92	256
# Chunks	1	1	1	3	10	31	93
# Modified files	1	1	1	1	3	7	24
# Modified classes	1	1	1	1	3	7	24
# Modified methods	1	1	1	2	5	15	66

Size of the RegMiner4APR patches. This is presented by the number of lines added, removed, and modified, as shown in Table 2.

- *Added lines:* 25% of patches contain no added lines, while half of the patches add at most one lines. Patches with more than 18 added lines are less frequent.
- *Removed lines:* Half of the patches contain no removed lines, and 90% of the patches have no more than 16 removed lines.
- *Modified lines:* 90% of patches include a maximum of 5 modified lines.
- *Patch size:* The total size of a patch, calculated as the sum of added, removed, and modified lines, involves at most 4 lines for half of the patches. For 90% of patches, no more than 33 lines are affected.

Scope of the RegMiner4APR patches. This is analyzed by the number of chunks, modified methods, modified classes, and modified files, as demonstrated in Table 2.

- *Chunks:* Defined as continuous lines of code changes, 75% of patches contain three or fewer chunks.
- *Modified methods:* 29 patches (29%) change more than one method.
- *Modified classes:* We observe that the number of modified classes is closely related to the number of modified files in each patch.
- *Modified files:* 20 patches require changes in at least two files to address the bug. Of these, 7 follow a unique regression regression-fixing pattern, namely *Revert to previous statement.*

Overall, the median patch size in RegMiner4APR is four lines, with approximately 10% of patches affecting more than 33 lines. Among the 100 patches, 51 contain a single chunk, 61 modify only one method, and 80 are confined to a single file. The occurrence of larger patches (7 in total) is attributed to a specific regression bug-fixing pattern, namely *Revert to previous statement*, as highlighted in our bug analysis. This patch size distribution implies that the benchmark is suitable for evaluating a wide range of APR techniques. For instance, traditional APR tools typically produce small patches, while LLM-based approaches can handle larger modifications. Moreover, the presence of a few much larger patches highlights the need for future research into techniques capable of addressing more substantial repairs.

4.2.3 Bug Categorization. We further justify the diversity of RegMiner4APR benchmark by characterizing the constituent bugs. We dissect bugs in the benchmark into several different operators, of which some are obtained from Java defects classes proposed by Pan et al. [46] and Sobreira et al. [54]. Each operator is characterized by the type of code change performed to fix the defect. Table 3 summarizes the groups of repair operators observed in the benchmark. We describe the details of each operator group below.

Table 3. Repair Operators

Acronym	Group	Action
asgn	Assignment	A/R/M
cnd	Condition	A/R/M
lp	Loop	A/R/M
mc	Method call	A/R/M
md	Method definition	A/R/M
obj	Object Instantiation	A/R/M
exp	Exception	A/R
ret	Return	A/R/M
var	Variable	A/R/M
rev	Revert to previous statement	N.A

$^{\text{abbreviation}}$A: Add; R: Remove; M: Modify.

- *Assignment:* Changes involving a simple assignment operator (=), unary increment/decrement operators, or compound assignment operators with arithmetic operations. Actions applicable to this group include *addition* and *removal* at the statement level, and *modification* at the expression level (e.g., modifying the assigned value).

- *Condition:* Changes related to conditional branches, such as `if-else`, `switch-case`, and the ternary operator. Actions applicable to this group include *addition* and *removal* at the statement level, and *modification* at the expression level (e.g., modifying, expanding, or reducing the conditional expression).
- *Loop:* Changes related to loop constructions, such as `for` and `while`. Actions applicable to this group include *addition*, *removal* at statement level, and *modification* at both statement and expression levels.
- *Method Call:* Changes related to method calls. Actions applicable to this group include *addition* and *removal* at the statement level, and *modification* at the expression level (e.g., adding, removing, swapping, and modifying parameters).
- *Method definition:* Changes related to method definitions and signatures. Actions applicable to this group include *addition* and *removal* at the statement level, and *modification* at the expression level (e.g., adding or removing parameters, modifying parameter types, changing the return type, and altering the method's scope).
- *Object Instantiation:* Changes related to the instantiation of objects are observed through the keyword `new`. Actions applicable to this group include *addition* and *removal* at the statement level, and *modification* at the expression level (e.g., modifying the parameters passed in objects).
- *Exception:* A statement related to exception handling and throwing. Actions applicable to this group include *addition* and *removal* at the statement level (e.g., adding or removing `try-catch` or `throw` blocks).
- *Return:* Changes related to `return` statements. Actions applicable to this group include *addition* and *removal* at the statement level (e.g., wrapping with a conditional `if-else`), and *modification* at the expression level (e.g., modifying the logic expression of the return).
- *Variable:* Changes related to variable declaration and usage. Actions applicable to this group include *addition* and *removal* at the statement level, and *modification* at the expression level (e.g., modifying the variable type, replacing it with another variable or method call, and altering the assigned value).
- *Revert to previous statement:* Changes related to rolling back at the statement level. The rationale behind this action is that developers may have made mistakes that should have been left intact, resulting in the regression bug at hand. This is particularly unique for regression bugs, as it involves observing multiple versions and reasoning through the changes.

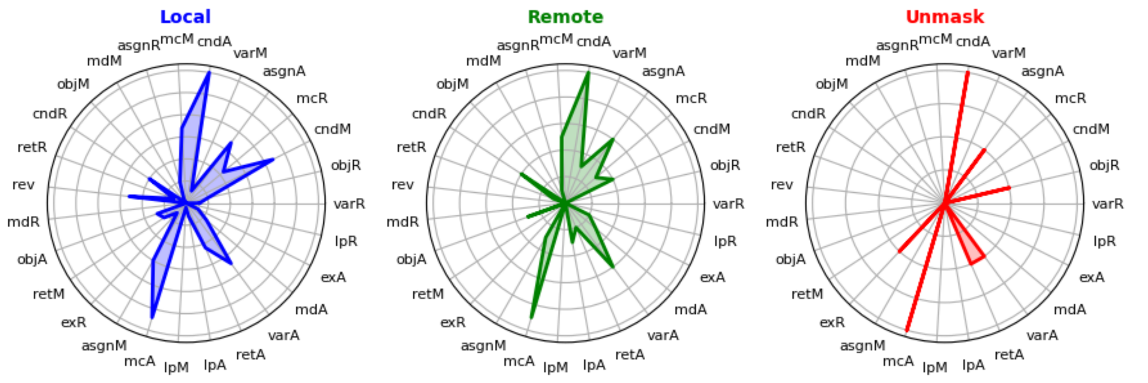


Fig. 12. Distribution of repair operators on different types of regression bug. Note that, we abbreviated the names of the repair action groups following Table 3. For example, "mcA" denotes *Method Call Addition*.

Following the regression categorization by Tan et al. [56], based on how the regression was introduced (as described in Section 2), our dataset contains 77 *local* bugs, 20 *remote* bugs, and 2 *unmask* bugs. Figure 12 illustrates the overall distribution of repair operators for each type of regression bug. The radial axis corresponds to the repair operators discussed in the previous section, indicating their occurrence across the different bug types. Common patterns, such as *Conditional Addition*, *Method Call Addition*, and *Assignment Addition*, are prevalent in all three types of regression bugs. Notably, the *Revert to Previous Statement* operator is more frequently observed in the *Local* bug category.

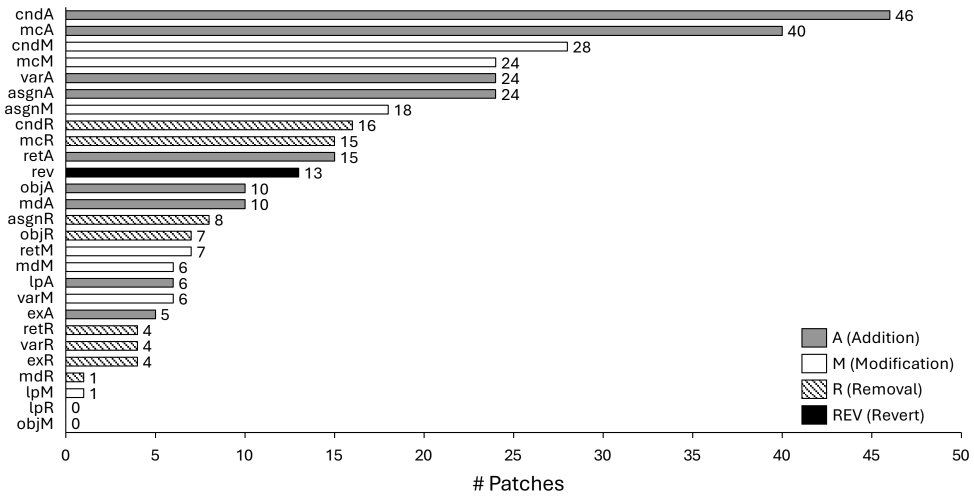


Fig. 13. Occurrence of repair operators in RegMiner4APR. Note that, we abbreviated the names of the repair action groups following Table 3. For example, "mcA" denotes *Method Call Addition*.

Figure 13 demonstrates the ranking of repair operators based on human patches, with the vertical axis representing the operators and the horizontal axis indicating the number of patches in which they are found. To enhance clarity, grey bars represent addition actions, diagonal striped bars indicate removal actions, white bars signify modification actions, and black bars denote revert actions. *Conditional Addition* is the most prevalent repair operator, appearing in 46 patches, followed by *Method Call Addition* with 40 patches and *Conditional Modification* with 28 patches. The *Revert to Previous Statement* operator is exclusive to *local* regression bugs, appearing in 13 patches. In all cases, addition exceeds removal and modification by 25.95% and 35.28%, respectively.

We further analyze the patches that utilize the *revert to previous statement* operator. In our dataset, there are 13 regression bugs that were patched by humans using only this repair operator. Additionally, we analyze these patches to identify the repair operators that humans would have employed if the revert operator had not been applied. Figure 14 illustrates all the repair operators used by humans to address these bugs. In total, 102 repair operators were employed to fix these 13 bugs, accounting for 29.74% of the total operators used across all bugs. This underscores the importance of recognizing this pattern in addressing regression bugs effectively.

Figure 15 presents the distribution of the number of repair operators per patch. The whiskers and outliers indicate that the data spans from around 1 (lower whisker) to 12 (outlier), with at least one patch having a significantly higher number of repair operators compared to the majority. The median (highlighted in red) appears to be around 3, while the box extends from approximately 1 to 5, indicating that the middle 50% of the data lies within this range. The distribution

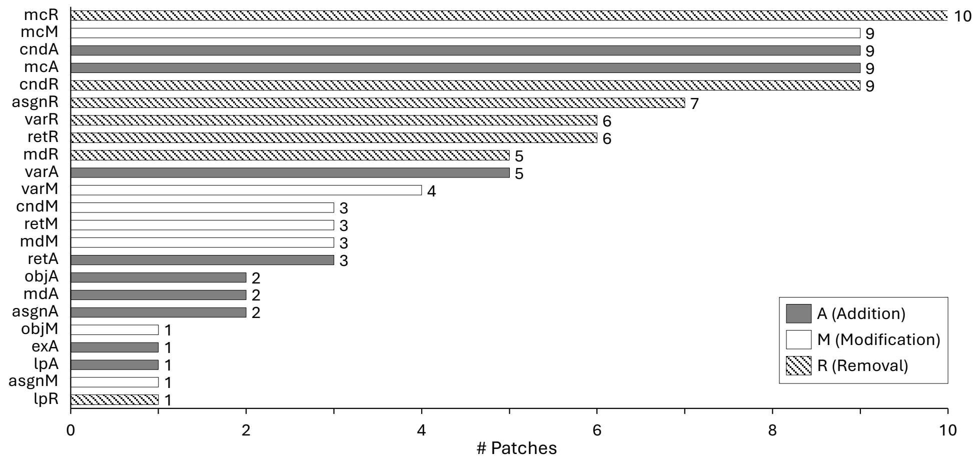


Fig. 14. Detailed examination of a repair operator, namely *revert to previous statement*. Note that, we abbreviated the names of the repair action groups following Table 3. For example, “mcA" denotes *Method Call Addition*.

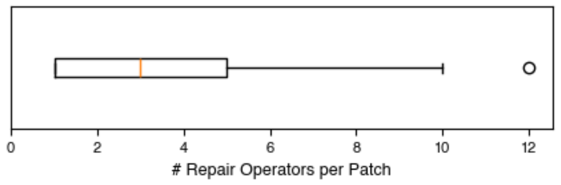


Fig. 15. Distribution of the number of repair operators per patch.

of the number of repair operators per patch is skewed to the right, with a few patches exhibiting a much higher number of operators than the majority.

Overall, the results highlight the diversity of repair operators required to fix bugs in our dataset. Compared with Defects4J [54], the *Loop Removal* and *Object Instantiation Modification* operators are absent from our benchmark. Notably, for *Local* regression bugs, the *Revert to Previous Statement* operator proves effective. Furthermore, a substantial portion of the patches involve the application of multiple repair operators, indicating the non-trivial nature of many fixes.

These results demonstrate that RegMiner4APR captures a wide range of patch sizes and scopes. With several patches exhibiting distinctive regression-fixing patterns, the dataset includes both representative and challenging edge cases that are rarely addressed by existing tools. This diversity makes RegMiner4APR a valuable benchmark for evaluating and advancing automated program repair techniques.

Answer to RQ1: The analysis demonstrates the high quality of the benchmark for APR research, highlighting key characteristics such as reality (sourced from real-world Java repositories), timeliness (extending the timeframe of the most recent and relevant benchmarks), and diversity (in terms of patch size, scope, and repair operators). In addition, our structured data storage process ensures durable reproducibility of all bugs.

Table 4. Performance Evaluation of APR Methods on the RegMiner4APR Benchmark

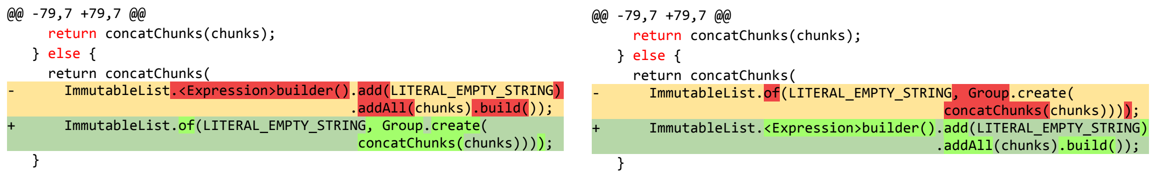
Method	#Plausible Patches (Plausible Rate %)	#Correct Patches (Correct Rate %)	Precision (%)
ChatGPT-4o + Conversation	33 *(33.33%)*	9 *(9.09%)*	27.27%
ChatGPT-4o + Zero-shot Prompting	20 *(20.20%)*	7 *(7.07%)*	35.00%
ChatGPT-Turbo-3.5 + Conversation	22 *(22.22%)*	5 *(5.05%)*	22.73%
ChatGPT-Turbo-3.5 + Zero-shot Prompting	15 *(15.15%)*	5 *(5.05%)*	33.33%
RepairLLaMA	19 *(19.19%)*	9 *(9.09%)*	47.37%
Incoder-6B	10 *(10.10%)*	3 *(3.03%)*	30.00%
Incoder-1B	9 *(9.09%)*	3 *(3.03%)*	33.33%
CodeGen-6B	6 *(6.06%)*	1 *(1.01%)*	16.67%
CodeGen-2B	6 *(6.06%)*	1 *(1.01%)*	16.67%
TBar	0 *(0%)*	0 *(0%)*	N/A
jGenProg	0 *(0%)*	0 *(0%)*	N/A
Cardumen	0 *(0%)*	0 *(0%)*	N/A
jKali	0 *(0%)*	0 *(0%)*	N/A
Arja	0 *(0%)*	0 *(0%)*	N/A
jMutRepair	0 *(0%)*	0 *(0%)*	N/A

5 RQ2: Effectiveness of APR Techniques on Regression Bugs

To investigate the effectiveness of existing APR techniques in the context of regression errors, we evaluate both traditional and LLM-based approaches in repairing buggy programs, without providing additional regression-specific context. This setup aligns with the its conventional workflow, which relies solely on the buggy program and its associated test suite for validation. The traditional APR tools include jGenProg [30], jKali [47], Cardumen [39], Arja [69], jMutRepair [38] and TBar [33]. The LLM-based approaches include Incoder-1B and Incoder-6B [16], CodeGen-2B and CodeGen-6B [43], and RepairLLaMA [52]. Additionally, we evaluate prompt-based techniques, including zero-shot prompting and conversational APR, using two advanced models: ChatGPT-3.5-Turbo and ChatGPT-4o. The selection, configuration, and implementation details of these tools are described in Section 3.3. Table 4 summarizes the overall performance of APR techniques on the RegMiner4APR benchmark, reporting the number of plausible patches, the number of correct patches among them, and the corresponding precision scores.

Our experiments indicate that *traditional APR techniques* were unable to repair any bugs within our dataset. A deeper analysis suggests the limitation stems from two main factors: *(i)* the unavailability of fixes within the current version of the program, as many fixes may only exist in earlier versions [8], and *(ii)* the inadequacy of existing bug-fixing patterns, which do not encompass many of the bugs present in our dataset. For instance, Figures (16a) and (16b) present the bug-inducing and bug-fixing changes for *RegressionBug-84* from the RegMiner4APR benchmark. In this case, the error emerged when developers sought to enhance system performance but inadvertently introduced implementation errors, resulting in subtle semantic bugs. The most straightforward resolution involved reverting the previous changes to eliminate the bug. However, if this bug is examined in isolation, without considering the associated regression-inducing commit, as is often the case with many APR tools, the bug-fixing pattern appears more complex, requiring numerous modifications. This complexity likely explains the limited applicability of such patterns in state-of-the-art

APR techniques. Conversely, if the regression-inducing commit is included in the input, a simple *"Revert to Previous Statement"* pattern could effectively resolve the issue.



```
@@ -79,7 +79,7 @@
     return concatChunks(chunks);
   } else {
     return concatChunks(
-      ImmutableList.<Expression>builder().add(LITERAL_EMPTY_STRING)
                                          .addAll(chunks).build());
+      ImmutableList.of(LITERAL_EMPTY_STRING, Group.create(
                                          concatChunks(chunks))));
   }
```

(a) Regression-inducing changes in RegressionBug-84.

```
@@ -79,7 +79,7 @@
     return concatChunks(chunks);
   } else {
     return concatChunks(
-      ImmutableList.of(LITERAL_EMPTY_STRING, Group.create(
                                          concatChunks(chunks))));
+      ImmutableList.<Expression>builder().add(LITERAL_EMPTY_STRING)
                                          .addAll(chunks).build());
   }
```

(b) Regression-fixing changes in RegressionBug-84.

Fig. 16. Illustration of the *Revert to Previous Statement* repair operator in *RegressionBug-84* from the RegMiner4APR benchmark.

Moving to more advanced approaches, APR techniques based on large language models demonstrate significantly improved performance over traditional tools. We categorize these techniques into two main families: *fine-tuning-based* and *prompt-based* methods. Our results show promising outcomes, some bugs can be fixed with minimal context, requiring only the buggy function and debugging information. However, other bugs necessitate richer contextual information to understand the cause of the regression and to reuse relevant variables or logic from earlier versions to construct the correct fix (e.g., reverting a previous change as in *RegressionBug-84*).

Fine-tuning-based APR tools, which employ LLMs further fine-tuned on large-scale bug-fix corpora, generate between 6 and 19 plausible patches, with 1 to 9 correct patches and precision rates ranging from 16.67% to 47.37%. Among them, RepairLLaMA achieves the highest performance, producing 9 correct patches and attaining the highest precision rate of 47.37%. Incoder outperforms CodeGen by correctly repairing 2 bugs more. Specifically, both Incoder-6B and Incoder-1B can generate three correct patches; however, Incoder-1B demonstrates slightly better precision (by 3.33%). Similarly, CodeGen-6B and CodeGen-2B yield the same number of correct patches (one each). Notably, it is able to fix every bug resolved by Incoder and CodeGen, with the exception of *RegressionBug-48*, for which it generates a plausible but incorrect patch.

On the other hand, *prompt-based APR techniques*, including both *zero-shot prompting* and *conversational* strategies, also demonstrate strong performance. These methods yield between 15 and 33 plausible patches, with 5 to 9 correct patches, and precision rates ranging from 22.73% to 35.00%. Among them, ChatGPT-4o with conversational interaction achieves the highest number of correct patches (9 in total) and is capable of repairing every bug fixed by the other tools in this category. Overall, ChatGPT-4o outperforms ChatGPT-3.5-Turbo in terms of correct patches. For both models, the conversational configuration generally generate more plausible patches than zero-shot prompting. In the case of ChatGPT-3.5-Turbo, however, both configurations produce the same number of correct patches. Interestingly, the zero-shot prompting configuration results in higher precision for both models, by approximately 6.06% to 12.27%, compared to their conversational counterparts. This discrepancy may be due to the tendency of conversational strategies to overfit to the given test cases, as they rely on iterative feedback from test executions to guide patch generation, resulting in a higher number of plausible patches. As a result, they often produce a higher number of plausible patches, but relatively few correct ones among them.

In sum, RepairLLaMA and ChatGPT-4o (Conversation) achieve the best performance in terms of the number of correct repairs, each successfully fixing 9 bugs. However, RepairLLaMA exhibits superior precision, making it the most reliable tool among all evaluated techniques. The union of both tools results in 14 unique bugs being fixed in the RegMiner4APR benchmark. Notably, RepairLLaMA is able to repair five bugs, *RegressionBug-17, 30, 37, 54, and 95*, that

ChatGPT-4o (Conversation) fails to fix. This may be attributed to ChatGPT-4o's conversational strategy, which tends to over-modify the code or overfit to specific failing test cases, likely due to over-inferring.

For example, in *RegressionBug-95*, the correct fix involves adjusting a numeric constant: *"IOUtils. getChars(nanos, 30 - (9 - nanoSize), buf);"* to *"IOUtils.getChars(nanos, 29 - (9 - nanoSize), buf);"*. This change precisely aligns the buffer offset with the expected location of the *nanos* value in timestamps formatting. Although ChatGPT-4o (Conversation) effectively utilizes the failure message to exactly pinpoint the faulty location, it generates an overfitting patch: *"IOUtils.getChars(nanos, buf.length, buf);"*. This version only passes the test case when *"nanoSize"* is *9*, making it a coincidental success rather than a semantically correct fix. Another example is *RegressionBug-30*, where ChatGPT-4o (Conversation) produces a plausible patch after five attempts and three rounds of interaction. The final patch is unnecessarily complex and lacks the core corrective logic expected in the human-written fix. As a result, it introduces semantic divergence from the intended behavior, despite passing the test suite.

Conversely, ChatGPT-4o (Conversation) is able to repair five bugs, including *RegressionBug-22, 40, 62, 78*, and *85*, that RepairLLaMA fails to fix. To illustrate this effectiveness, we present *RegressionBug-85* as a representative example. ChatGPT-4o successfully repaired this bug after four attempts and two rounds of interaction. In the first round, it produced a partial fix and responded with the following insight:

> *"... The error message indicates that the expected output should have the 'metadata' argument only in the 'if' condition and not in the 'else' condition. The current implementation mistakenly includes 'metadata' in both branches of the conditional expression..."*

Although this patch did not yet pass the test case, the response demonstrated its ability to reason about the structure of the bug. In the second round, ChatGPT-4o refined its fix with the following response:

> *"... To address the error you mentioned, it seems the issue lies in how the conditional expression is being constructed, particularly regarding the parameters passed to the 'construct' method for the 'SOY_VISUAL_ELEMENT'. The test case indicates that when 'GOOG_DEBUG' is 'true', the 'SOY_VISUAL_ELEMENT' should be constructed with both the ID and the name, while when 'GOOG_DEBUG' is 'false', it should be constructed with only the ID..."*

This revised fix passed the test case and demonstrates how ChatGPT-4o with conversation effectively leverages failing test case information and interaction history to guide its repair process. The example highlights the benefit of interactive, test-driven repair in addressing complex semantic errors.

Answer to RQ2: Our experimental results on 15 APR methods demonstrate that traditional techniques fail to repair any bugs in the dataset, largely due to the limitations of their underlying bug-fixing patterns. In contrast, advanced APR tools leveraging large language models show promising results. Notably, RepairLLaMA and ChatGPT-4o (Conversation) achieve the highest number of correct repairs, each successfully fixing 9 bugs. Among all evaluated techniques, RepairLLaMA attains the highest precision, making it the most reliable in terms of generating correct patches.

6 RQ3: Impact of Bug-Inducing Change Information on APR Performance

To explore whether providing additional context, specifically bug-inducing change information, can enhance regression error repair performance, we focus on prompt-based techniques and adapt the prompt design to explicitly incorporate regression-specific context. We choose this approach based on its demonstrated effectiveness (as shown in RQ2), its inherent flexibility in both model usage and prompt construction, and its greater data efficiency compared to fine-tuning-based methods. The selection, configuration, and implementation details of these techniques are provided in Section 3.3.

Table 5 summarizes the overall performance of prompt-based APR techniques on the RegMiner4APR benchmark, reporting the number of plausible patches, the number of correct patches among them, and the corresponding precision scores.

Table 5. Performance Evaluation of Prompt-based APR Approaches With and Without Bug-Inducing Change (BIC) Information

Base Model	**Method**	**#Plausible Patches (Plausible Rate %)**	**#Correct Patches (Correct Rate %)**	**Precision (%)**
ChatGPT-4o	***Zero-shot Prompting w/o BIC***	20 *(20.20%)*	7 *(7.07%)*	*35.00%*
	Conversation w/o BIC	33 *(33.33%)*	9 *(9.09%)*	*27.27%*
	Zero-shot Prompting with BIC	33 *(33.33%)*	12 *(12.12%)*	*36.36%*
	Conversation with BIC	43 *(43.43%)*	16 *(16.16%)*	*37.21%*
ChatGPT-Turbo-3.5	***Zero-shot Prompting w/o BIC***	15 *(15.15%)*	5 *(5.05%)*	*33.33%*
	Conversation w/o BIC	22 *(22.22%)*	5 *(5.05%)*	*22.73%*
	Zero-shot Prompting with BIC	20 *(20.20%)*	7 *(7.07%)*	*35.00%*
	Conversation with BIC	33 *(33.33%)*	8 *(8.08%)*	*24.24%*

6.1 Experimental Results

Our experimental results show that for both base models, including ChatGPT-4o and ChatGPT-Turbo-3.5, incorporating regression-specific context, specifically bug-inducing change (BIC) information, improves the plausible rate, correct rate and precision compared to their counterparts configured without this information. Additionally, when BIC context is included, the conversational configuration consistently outperforms the zero-shot prompting configuration in both base models.

For ChatGPT-4o, the conversational strategy with BIC successfully repairs 16 bugs (16.06%), compared to only 9 bugs (9.09%) without BIC, an increase of approximately 7.07%. The precision also increases by 9.94% (from 27.27% to 37.21%). Similarly, zero-shot prompting with BIC achieves 12 correct repairs, outperforming the version without BIC, which fixes only 7 bugs, with a precision gain of 1.36% (from 35.00% to 36.36%). A similar trend is observed for ChatGPT-Turbo-3.5. The conversational strategy with BIC fixes 8 bugs, compared to 5 without BIC, accompanied by a 1.51% increase in precision (from 22.73% to 24.24%). Likewise, zero-shot prompting with BIC results in 7 correct repairs, outperforming its counterpart without BIC, which fixes 5 bugs, with a corresponding precision gain of 1.67% (from 33.33% to 35.00%). When comparing configurations within each base model, zero-shot prompting with BIC consistently outperforms both the conversational and zero-shot configurations without BIC, further underscoring the value of incorporating regression-specific context. Lastly, it is important to note that ChatGPT-4o under conversation configuration augmented with BIC achieves the best overall performance across all evaluated configurations, in terms of both correct repairs and precision.

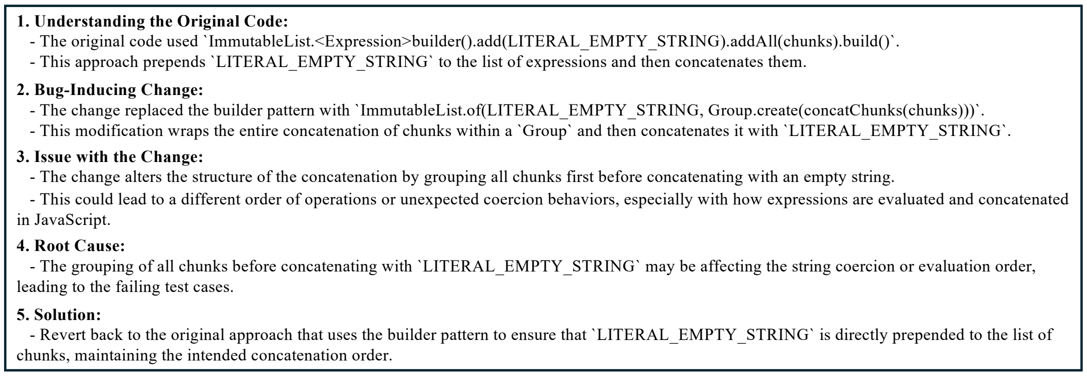
In total, the union of both base models enhanced with BIC results in 18 correct repairs (18.18%) in the RegMiner4APR benchmark, whereas the union of their counterparts without BIC repairs only 9 bugs, demonstrating a twofold improvement in repair effectiveness. Notably, all bugs fixed by techniques without BIC are also repaired by those with BIC. The additional bugs exclusively fixed when providing BIC information include *RegressionBug-3, 4, 30, 48, 59, 65, 69, 84*, and *99*.

6.2 Qualitative Analysis of Plausible Patches

To better understand how bug-inducing change (BIC) information influences the behavior of LLMs in automated program repair, we conduct a qualitative analysis of selected correct patches where the presence of bug-inducing change information support the repair process in different ways. Through manual patch validation, we selected cases in which regression bugs were successfully repaired only when BIC information was included, specifically *RegressionBug-3, 4, 30, 48, 59, 65, 69, 84*, and *99* (as reported in the Section 6.1). We organize our findings into representative case studies, each illustrating a distinct pattern in how the model leverages contextual and historical information to generate correct patches.

6.2.1 ***Observation 1:*** *Fully reverting to previous statements guided by bug-inducing change information.*

In some cases, APR techniques successfully repair the bug by reverting all modified statements to the version that was functioning correctly before the regression was introduced. Generating such patches correctly requires the model to understand the broader context of both the bug-inducing changes and the buggy program. For example, the model must be able to distinguish between intended updates, such as bug fixes or feature additions, and unintended changes that should be reverted. This behavior is observed in *RegressionBug-3* and *RegressionBug-84*. For instance, Figure 16 presents an illustration of the *Revert to Previous Statement* repair operator in *RegressionBug-84* from the *RegMiner4APR* benchmark. In this case, the developer had introduced a change that wrapped a + expression in parentheses (e.g., `"" + (a + b)`) to prevent premature string coercion. However, this modification was incorrect because the JavaScript + operator is not associative. The fix reverted this change by removing the parentheses and directly prepending `""` to the list of expressions (i.e., `"" + a + b + c`), allowing JavaScript to perform left-to-right type coercion and thereby preserving the intended semantics.


1. Understanding the Original Code:
- The original code used `ImmutableList.<Expression>builder().add(LITERAL_EMPTY_STRING).addAll(chunks).build()`.
- This approach prepends `LITERAL_EMPTY_STRING` to the list of expressions and then concatenates them.

2. Bug-Inducing Change:
- The change replaced the builder pattern with `ImmutableList.of(LITERAL_EMPTY_STRING, Group.create(concatChunks(chunks)))`.
- This modification wraps the entire concatenation of chunks within a `Group` and then concatenates it with `LITERAL_EMPTY_STRING`.

3. Issue with the Change:
- The change alters the structure of the concatenation by grouping all chunks first before concatenating with an empty string.
- This could lead to a different order of operations or unexpected coercion behaviors, especially with how expressions are evaluated and concatenated in JavaScript.

4. Root Cause:
- The grouping of all chunks before concatenating with `LITERAL_EMPTY_STRING` may be affecting the string coercion or evaluation order, leading to the failing test cases.

5. Solution:
- Revert back to the original approach that uses the builder pattern to ensure that `LITERAL_EMPTY_STRING` is directly prepended to the list of chunks, maintaining the intended concatenation order.

Fig. 17. Excerpt from ChatGPT-4o's response under the zero-shot prompting configuration augmented with bug-inducing change information for repairing *RegressionBug-84*.

Figure 17 presents an excerpt from ChatGPT-4o's reasoning under zero-shot prompting configuration with BIC context for *RegressionBug-84*. Specifically, the model first analyzes the original code, then interprets the bug-inducing change, identifies the semantic issue it introduced, and ultimately concludes that the regression stems from replacing the builder-style concatenation with a grouped expression. This modification unintentionally alters the evaluation order and type coercion behavior in JavaScript, resulting in a failing test case. ChatGPT-4o correctly identifies the root cause and proposes reverting to the original builder-based expression to restore the intended semantics, aligning with the developer's manual fix.

6.2.2 ***Observation 2:*** *Partially reverting edits guided by bug-inducing change information.* Beyond full reversion, partially reverting selected code fragments also proves effective in cases where only specific statements introduced in the bug-inducing commit need to be undone. This strategy allows the model to preserve valid modifications while selectively removing only the changes responsible for the regression. This behavior is observed in *RegressionBug-4, 48* and 99. For example, Figure 18 presents an illustration of the *Partially Revert to Previous Statement* repair operator in *RegressionBug-99*, as generated by ChatGPT-4o under the conversation configuration enhanced with bug-inducing change information. In this case, the developers changed the behavior of `copyFile` to copy symbolic links itself (i.e., the path to the target), rather than dereferencing the link and copying the actual target file. This change reversed the default behavior expected by many systems and broke existing code that relied on the original semantics. As discussed in the commit history[10], the developers chose to revert only the code responsible for the bug, while retaining the condition *"if (preserveFileDate && !isSymLink && !setTimes(srcFile, destFile)) {"*. The retained code does not change behavior, it simply avoids setting file timestamps on symbolic links to avoids errors related to modifying *symlink* metadata, which is often OS-dependent and error-prone. This careful partial reversion ensures correctness without discarding unrelated, valid improvements.

```
@@ -825,7 +835,7 @@
      * @see #copyFileToDirectory(File, File, boolean)
      * @since 2.8.0
      */
-    public static void copyFile(final File srcFile, final File destFile, final boolean
preserveFileDate, final CopyOption... copyOptions) throws IOException {
+    public static void copyFile(final File srcFile, final File destFile, final boolean
preserveFileDate, CopyOption... copyOptions) throws IOException {
         Objects.requireNonNull(destFile, "destination");
         checkFileExists(srcFile, "srcFile");
         requireCanonicalPathsNotEquals(srcFile, destFile);
@@ -834,10 +844,18 @@
            checkFileExists(destFile, "destFile");
            requireCanWrite(destFile, "destFile");
        }
+       final boolean isSymLink = Files.isSymbolicLink(srcFile.toPath());
+       if (isSymLink && !Arrays.asList(copyOptions).contains(LinkOption.NOFOLLOW_LINKS)) {
+           final List<CopyOption> list = new ArrayList<CopyOption>(Arrays.asList(copyOptions));
+           list.add(LinkOption.NOFOLLOW_LINKS);
+           copyOptions = list.toArray(new CopyOption[0]);
+       }
        Files.copy(srcFile.toPath(), destFile.toPath(), copyOptions);

        // On Windows, the last modified time is copied by default.
-       if (preserveFileDate && !setTimes(srcFile, destFile)) {
+       if (preserveFileDate && !isSymLink && !setTimes(srcFile, destFile)) {
            throw new IOException("Cannot set the file time.");
        }
    }
```

(a) Bug-inducing changes in RegressionBug-99.

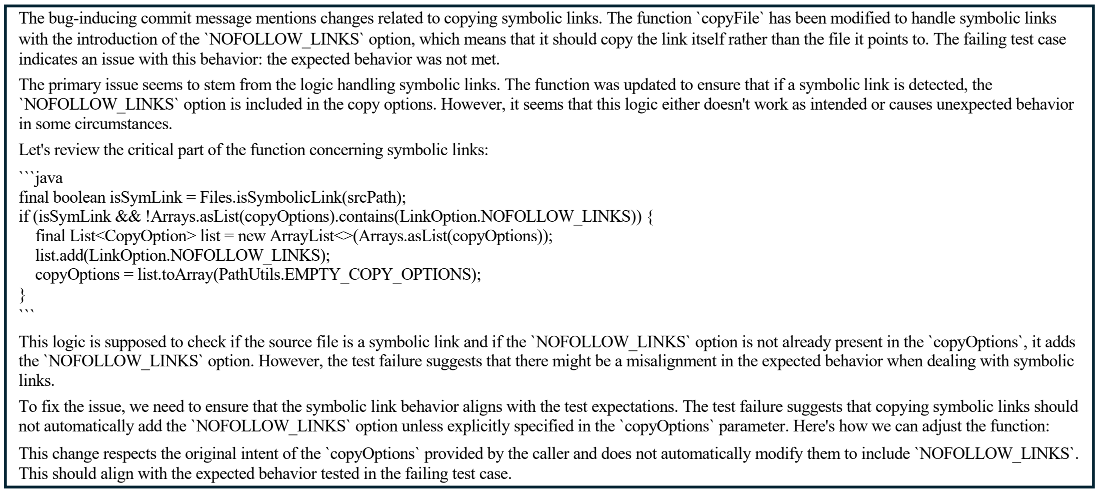
```
        }
        final Path srcPath = srcFile.toPath();
        final boolean isSymLink = Files.isSymbolicLink(srcPath);
-       if (isSymLink && !Arrays.asList(copyOptions).contains(LinkOption.NOFOLLOW_LINKS)) {
-           final List<CopyOption> list = new ArrayList<>(Arrays.asList(copyOptions));
-           list.add(LinkOption.NOFOLLOW_LINKS);
-           copyOptions = list.toArray(PathUtils.EMPTY_COPY_OPTIONS);
-       }
+       // Instead of automatically adding NOFOLLOW_LINKS, we respect the passed copyOptions
+       // and avoid any default addition of NOFOLLOW_LINKS unless explicitly intended by the caller.
        Files.copy(srcPath, destFile.toPath(), copyOptions);
        // On Windows, the last modified time is copied by default.
        if (preserveFileDate && !Files.isSymbolicLink(srcPath) && !setTimes(srcFile, destFile)) {
```

(b) Patch generated by ChatGPT-4o for RegressionBug-99.

Fig. 18. Illustration of the *Partially Revert to Previous Statement* repair operator in *RegressionBug-99*, as generated by ChatGPT-4o under the conversation configuration enhanced with bug-inducing change information.

Figure 19 presents an excerpt from ChatGPT-4o's reasoning under conversation configuration with BIC context for *RegressionBug-99*. Specifically, the model explains that the regression stems from logic that forcibly adds the `NOFOLLOW_LINKS` option when a symbolic link is detected, thereby overriding the caller's intended behavior. It correctly identifies that this automatic injection violates expectations, particularly in cases where callers rely on the default behavior of dereferencing symbolic links. Guided by the bug-inducing change information, the model proposes reverting the conditional block responsible for modifying the copy options, while preserving the surrounding logic, including the check that skips timestamp setting for symbolic links. This patch aligns with the human-written fix, as it reverts only the faulty logic introduced in the commit while retaining valid behavior.

[10]Available at: https://github.com/apache/commons-io/commit/ec4144b01b4107d6b39f5f4d784cf05217ea4dfa



The bug-inducing commit message mentions changes related to copying symbolic links. The function `copyFile` has been modified to handle symbolic links with the introduction of the `NOFOLLOW_LINKS` option, which means that it should copy the link itself rather than the file it points to. The failing test case indicates an issue with this behavior: the expected behavior was not met.

The primary issue seems to stem from the logic handling symbolic links. The function was updated to ensure that if a symbolic link is detected, the `NOFOLLOW_LINKS` option is included in the copy options. However, it seems that this logic either doesn't work as intended or causes unexpected behavior in some circumstances.

Let's review the critical part of the function concerning symbolic links:

```java
final boolean isSymLink = Files.isSymbolicLink(srcPath);
if (isSymLink && !Arrays.asList(copyOptions).contains(LinkOption.NOFOLLOW_LINKS)) {
    final List<CopyOption> list = new ArrayList<>(Arrays.asList(copyOptions));
    list.add(LinkOption.NOFOLLOW_LINKS);
    copyOptions = list.toArray(PathUtils.EMPTY_COPY_OPTIONS);
}
```

This logic is supposed to check if the source file is a symbolic link and if the `NOFOLLOW_LINKS` option is not already present in the `copyOptions`, it adds the `NOFOLLOW_LINKS` option. However, the test failure suggests that there might be a misalignment in the expected behavior when dealing with symbolic links.

To fix the issue, we need to ensure that the symbolic link behavior aligns with the test expectations. The test failure suggests that copying symbolic links should not automatically add the `NOFOLLOW_LINKS` option unless explicitly specified in the `copyOptions` parameter. Here's how we can adjust the function:

This change respects the original intent of the `copyOptions` provided by the caller and does not automatically modify them to include `NOFOLLOW_LINKS`. This should align with the expected behavior tested in the failing test case.

Fig. 19. Excerpt from ChatGPT-4o's response under the conversation configuration augmented with bug-inducing change information for repairing *RegressionBug-99*.

6.2.3 ***Observation 3:*** *Fully or Partially reverting edits could generate the plausible but incorrect patches.*

Apart from *Observation 1* and *2*, our patch validation process revealed recurring patterns, notably *fully or partially reverting to previously correct statements*, that lead to patches which pass the test suite but are semantically incorrect (i.e., plausible but incorrect patches). These patterns frequently appear in patches generated by techniques that leverage bug-inducing change information. Specifically, such behavior is observed in patches produced by ChatGPT-4o with Conversation (*RegressionBug-2, 34, 35, 37, 39, 47, 94, 98*), ChatGPT-4o with Zero-shot Prompting (*RegressionBug-2, 35, 37, 39, 47, 98*), ChatGPT-Turbo-3.5 with Conversation (*RegressionBug-35, 94*), and ChatGPT-Turbo-3.5 with Zero-shot Prompting (*RegressionBug-37, 94*). While these patterns often produce plausible patches that satisfy the test suite, they may reflect an over-reliance on syntactic rollback strategies and suggest that the model lacks a deeper understanding of the contextual semantics required to correctly fix the underlying bug.

```
@@ -300,9 +310,14 @@
   }

   private void readUntilEndOfEntry() throws IOException {
+    if (localFileHeader.getCompressedSize() == 0) {
+      return;
+    }
+
     if (endOfEntryBuffer == null) {
       endOfEntryBuffer = new byte[512];
     }
+
     while (read(endOfEntryBuffer) != -1);
   }
```

(a) Bug-inducing changes in RegressionBug-2.

```
@@ -303,7 +302,10 @@
   }

   private void readUntilEndOfEntry() throws IOException {
-    if (localFileHeader.isDirectory() ||
 localFileHeader.getCompressedSize() == 0) {
+    // If data descriptor exists, proceed to reading the
 entry, or else skip if the entry is directory
+    // or compressed size is 0
+    if (!localFileHeader.isDataDescriptorExists()
+        && (localFileHeader.isDirectory() ||
 localFileHeader.getCompressedSize() == 0)) {
      return;
     }

     if (endOfEntryBuffer == null) {
```

(b) Bug-fixing changes in RegressionBug-2.

Fig. 20. The bug-inducing and bug-fixing changes in *RegressionBug-2* from RegMiner4APR benchmark.

To identify the root cause of the regression bug, let's analyze the code and the changes made:

1. Original Functionality:
- The function `readUntilEndOfEntry()` was designed to skip over data until the end of a compressed entry in a file.

2. Bug-Inducing Change:
- The additional condition `if (localFileHeader.getCompressedSize() == 0) { return; }` was introduced. This means that if the compressed size of the entry is zero, the function will return immediately without reading any further.

3. Impact of the Change:
- The change causes the function to exit early for entries that have a compressed size of zero.
- While this seems logical at first (since there's no compressed data to read), it overlooks the fact that sometimes an entry might need to be processed in a specific way even if its compressed size is zero, especially when dealing with certain ZIP file structures or when metadata needs to be processed.

4. Failing Test Case:
- The test case `testReadingZipBySkippingDataCreatedWithJDKZipReadsAllEntries` expects to read all entries correctly, including those with a compressed size of zero. The error shows that only 1 entry was read instead of the expected 6, which indicates that the function is prematurely returning and not processing all entries correctly.

5. Solution:
- We need to ensure that entries with a compressed size of zero are still processed correctly. This means removing or modifying the condition that causes the function to return early.

Here's the corrected function without the problematic condition:

```java
private void readUntilEndOfEntry() throws IOException {
  if (localFileHeader.isDirectory()) {
    return;
  }
  if (endOfEntryBuffer == null) {
    endOfEntryBuffer = new byte[512];
  }
  while (read(endOfEntryBuffer) != -1);
}
```

By removing the condition `if (localFileHeader.getCompressedSize() == 0) { return; }`, the function will now correctly attempt to read through the end of each entry, even if the compressed size is zero. This should resolve the regression issue and allow all entries to be processed as expected.

Fig. 21. Excerpt from ChatGPT-4o's response under the conversation configuration augmented with bug-inducing change information for repairing *RegressionBug-2*.

To illustrate this behavior, we examine *RegressionBug-2* alongside the response from ChatGPT-4o under zero-shot prompting configuration augmented with BIC context, as shown in Figures 20 and 21. In this case, the regression-inducing changes were introduced to prevent read errors when handling JAR files using ZIP4J [11], but they unintentionally triggered a new bug. The issue occurred when only the first entry in a ZIP archive could be read, and subsequent entries were skipped. The regression was caused by logic that returned early if an entry's compressed size was zero (Figure 20a). However, according to the ZIP specification, this is valid behavior when the third bit of the general purpose flag is set. This bit indicates that the actual compressed size is not available in the local file header but will instead be provided later in a separate data descriptor. To fix this, the condition was refined to be more precise: entries are now skipped only if they do not use a data descriptor and are either directories or have a compressed size of zero (Figure 20b). This update preserves the original safeguard against invalid entries while allowing valid entries that rely on the data descriptor mechanism to be processed correctly.

The ChatGPT model correctly identified the root cause of the bug: the change caused valid ZIP entries to be skipped when their compressed size was zero, resulting in only one out of six entries being read. This occurred because the code returned early without properly handling ZIP structures that rely on data descriptors or contain essential metadata (as seen in parts 3 and 4 of the excerpt in Figure 21). However, the model proposed removing the condition entirely, which would allow entries to be read even if the compressed size is zero. While this may avoid the regression error, it overlooks the original intent of the change.

6.2.4 ***Observation 4:*** *Combining test case feedback with bug-inducing change information to narrow the fix scope.* In some cases, when the model correctly understands the context of bug-inducing changes, such as feature additions or prior bug fixes, it avoids fully or partially reverting to a previously working version. Instead, it analyzes the current

[11]Available at: https://github.com/srikanth-lingala/zip4j/issues/194

buggy function while simultaneously reasoning about the root cause of the regression. This combined reasoning enables more accurate fault localization, as the bugs happens after the relevant change is introduced, making the bug-inducing changes a strong signal for identifying the issue. This, in turn, supports the generation of targeted and minimal patches. This behavior is observed in *RegressionBug-30* and *RegressionBug-69*, where failing test cases provide useful information the bugs, while the bug-inducing changes provide complementary context that helps the model localize the root cause. By leveraging both sources of information, the model is able to narrow the fix scope and avoid unnecessary or overly broad modifications.

```
@@ -1284,12 +1284,59 @@

                 }
                 final FieldInfo fieldInfo = fieldDeser.fieldInfo;
+                Field field = fieldDeser.fieldInfo.field;
                 Type paramType = fieldInfo.fieldType;

+                if (paramType == boolean.class) {
+                    if (value == Boolean.FALSE) {
+                        field.setBoolean(object, false);
+                        continue;
+                    }
+                    if (value == Boolean.TRUE) {
+                        field.setBoolean(object, true);
+                        continue;
+                    }
+                } else if (paramType == int.class) {
+                    if (value instanceof Number) {
+                        field.setInt(object, ((Number) value).intValue());
+                        continue;
+                    }
+                } else if (paramType == long.class) {
+                    if (value instanceof Number) {
+                        field.setLong(object, ((Number) value).longValue());
+                        continue;
+                    }
+                } else if (paramType == float.class) {
+                    if (value instanceof Number) {
+                        field.setFloat(object, ((Number) value).floatValue());
+                        continue;
+                    }
+                } else if (paramType == double.class) {
+                    if (value instanceof Number) {
+                        field.setDouble(object, ((Number) value).doubleValue());
+                        continue;
+                    } else if (value instanceof String) {
+                        double doubleValue = Double.parseDouble((String) value);
+                        field.setDouble(object, doubleValue);
+                        continue;
+                    }
+                } else if (value != null && paramType == value.getClass()) {
+                    field.set(object, value);
+                    continue;
+                }
                 String format = fieldInfo.format;
                 if (format != null && paramType == java.util.Date.class) {
                     value = TypeUtils.castToDate(value, format);
                 } else {
-                    value = TypeUtils.cast(value, paramType, config);
+                    if (paramType instanceof ParameterizedType) {
+                        value = TypeUtils.cast(value, (ParameterizedType) paramType, config);
+                    } else {
+                        value = TypeUtils.cast(value, paramType, config);
+                    }
                 }
                 fieldDeser.setValue(object, value);
```

Fig. 22. Regression-inducing changes in *RegressionBug-69*

To illustrate this behavior, we present *RegressionBug-69*, where the developers attempted to improve the performance of `JSONObject.toJavaObject` by introducing optimizations [12], as shown in Figure 22. However, the modification introduced a regression due to a mismatch between the field type and the expected parameter type. Specifically, the checks used primitive types, while the actual `paramType` values were boxed class types. As a result, the new conditions failed to trigger as intended, leading to incorrect behavior during deserialization.

[12]Available at: https://github.com/alibaba/fastjson/commit/7b416faa1015ce04505e88d1f9b0575bcf13657f

To identify the root cause of the bug, let's analyze the changes and failing test cases step by step:

Changes Analysis

The changes made in the code are primarily focused on the process of setting field values in an object. The bug-inducing changes added direct handling for primitive types like `boolean`, `int`, `long`, `float`, and `double` using reflection. Additionally, it cast the value directly to the expected type using `TypeUtils`.

Error Analysis

The failing test case indicates an error when setting an `Integer` field with a value of type `int`. The error message specifically mentions that it cannot set an `Integer` field to an `int` value.

Root Cause

The root cause of the issue is that the code attempts to set a primitive type directly using the `Field`'s primitive setters (e.g., `setInt`, `setBoolean`) without considering the possibility that the field might be a boxed type (e.g., `Integer`, `Boolean`). This leads to a `JSONException` when the actual field type is a boxed type, but the value is primitive.

Solution

To fix the issue, we need to ensure that we handle both primitive and boxed types appropriately. We should use the boxed type's setter methods for fields that are of a boxed type. Here's the corrected version of the function:
…

(a)

The error message you're encountering suggests a type mismatch issue. Specifically, it seems like the code is trying to set an `int` value into a field of type `Integer`. This can happen if the `Field` object is expecting an `Integer` object rather than a primitive `int`.

To address this, you need to ensure the types match when setting field values. Here's a revised version of your function:
…

(b)

Fig. 23. Excerpts from ChatGPT-4o's response under the conversational configuration for repairing *RegressionBug-69*: (a) with bug-inducing change information and (b) without BIC information

```
@@ -1,7 +1,5 @@
-    public Object createInstance(Map<String, Object> map, ParserConfig config) //
-    throws IllegalArgumentException,
-    IllegalAccessException,
-    InvocationTargetException {
+    public Object createInstance(Map<String, Object> map, ParserConfig config)
+    throws IllegalArgumentException, IllegalAccessException, InvocationTargetException {
         Object object = null;
         if (beanInfo.creatorConstructor == null && beanInfo.factoryMethod == null) {
             object = createInstance(null, clazz);
@@ -16,28 +14,28 @@
                 Field field = fieldDeser.fieldInfo.field;
                 Type paramType = fieldInfo.fieldType;
                 if (field != null) {
-                    if (paramType == boolean.class) {
+                    if (paramType == boolean.class || paramType == Boolean.class) {
                         if (value == Boolean.FALSE) {
-                            field.setBoolean(object, false);
+                            field.set(object, false);
                             continue;
                         }
                         if (value == Boolean.TRUE) {
-                            field.setBoolean(object, true);
+                            field.set(object, true);
                             continue;
                         }
-                    } else if (paramType == int.class) {
+                    } else if (paramType == int.class || paramType == Integer.class) {
                         if (value instanceof Number) {
-                            field.setInt(object, ((Number) value).intValue());
+                            field.set(object, ((Number) value).intValue());
                             continue;
                         }
-                    } else if (paramType == long.class) {
+                    } else if (paramType == long.class || paramType == Long.class) {
                         if (value instanceof Number) {
-                            field.setLong(object, ((Number) value).longValue());
+                            field.set(object, ((Number) value).longValue());
                             continue;
                         }
-                    } else if (paramType == float.class) {
+                    } else if (paramType == float.class || paramType == Float.class) {
                         if (value instanceof Number) {
-                            field.setFloat(object, ((Number) value).floatValue());
+                            field.set(object, ((Number) value).floatValue());
                             continue;
                         } else if (value instanceof String) {
                                   String strVal = (String) value;
@@ -47,12 +45,12 @@

                                   } else {
                                       floatValue = Float.parseFloat(strVal);
                                   }
-                            field.setFloat(object, floatValue);
+                            field.set(object, floatValue);
                             continue;
                         }
-                    } else if (paramType == double.class) {
+                    } else if (paramType == double.class || paramType == Double.class) {
                         if (value instanceof Number) {
-                            field.setDouble(object, ((Number) value).doubleValue());
+                            field.set(object, ((Number) value).doubleValue());
                             continue;
                         } else if (value instanceof String) {
                             String strVal = (String) value;
@@ -62,7 +60,7 @@

                             } else {
                                 doubleValue = Double.parseDouble(strVal);
                             }
-                        field.setDouble(object, doubleValue);
+                        field.set(object, doubleValue);
                         continue;
                     }
                 } else if (value != null && paramType == value.getClass()) {
```

(a) Patch generated by ChatGPT-4o under the conversation configuration augmented ***with*** *bug-inducing change information.*

```
@@ -102,21 +100,21 @@
             Object param = map.get(fieldInfo.name);
             if (param == null) {
                 Class<?> fieldClass = fieldInfo.fieldClass;
-                if (fieldClass == int.class) {
+                if (fieldClass == int.class || fieldClass == Integer.class) {
                     param = 0;
-                } else if (fieldClass == long.class) {
+                } else if (fieldClass == long.class || fieldClass == Long.class) {
                     param = 0L;
-                } else if (fieldClass == short.class) {
+                } else if (fieldClass == short.class || fieldClass == Short.class) {
                     param = Short.valueOf((short) 0);
-                } else if (fieldClass == byte.class) {
+                } else if (fieldClass == byte.class || fieldClass == Byte.class) {
                     param = Byte.valueOf((byte) 0);
-                } else if (fieldClass == float.class) {
+                } else if (fieldClass == float.class || fieldClass == Float.class) {
                     param = Float.valueOf(0);
-                } else if (fieldClass == double.class) {
+                } else if (fieldClass == double.class || fieldClass == Double.class) {
                     param = Double.valueOf(0);
-                } else if (fieldClass == char.class) {
+                } else if (fieldClass == char.class || fieldClass == Character.class) {
                     param = '0';
-                } else if (fieldClass == boolean.class) {
+                } else if (fieldClass == boolean.class || fieldClass == Boolean.class) {
                     param = false;
                 }
                 if (missFields == null) {
```

(b) Patch generated by ChatGPT-4o under the conversation configuration ***without*** *bug-inducing change information. This patch includes additional edits not present in (a), while other parts remain similar to (a).*

Fig. 24. Patches generated by ChatGPT-4o for *RegressionBug-69* under different configurations.

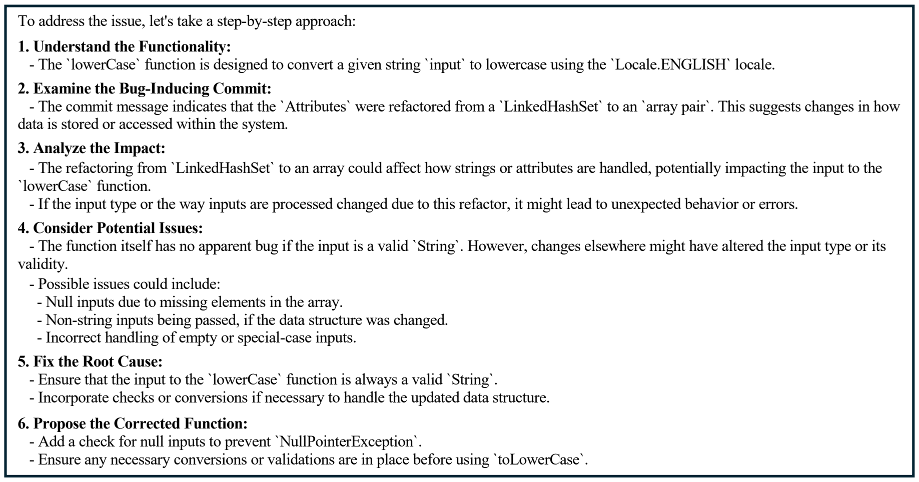
ChatGPT-4o, under both configurations, with and without bug-inducing change (BIC) information, was able to identify the type mismatch as the root cause of the failure by analyzing the test case's error message. However, the version augmented with BIC context demonstrated a deeper understanding of the problem and provided a clearer explanation, as shown in Figures 23a and 23b.

More notably, when comparing the patches generated under both settings in Figures 24a and 24b, both configurations produced a correct fix at the appropriate location. Nevertheless, the patch generated without BIC context included unnecessary edits to structurally similar but semantically unrelated code. Although the resulting patch passed the test suite, the added changes introduced redundancy. This illustrates how the BIC context helps the model focus on the minimal fix necessary to resolve the regression without introducing superfluous modifications.

6.2.5 ***Others:***

Apart from the specific cases discussed earlier, we also observed two cases, including *RegressionBug-59* and *RegressionBug-65*, in which the bug-inducing change information contributed to the patch generation or model reasoning to varying degrees. In these cases, the benefit of incorporating regression-specific context was less evident.

Specifically, in *RegressionBug-59*, the bug-inducing change introduced an entirely new method, `convexHull()`, intended to fix a prior issue. Unfortunately, the regression occurred within the newly added code itself. The accompanying commit message, *"Stolstov/issue 172 (#174)"*, provided little semantic guidance, making it difficult for the model to infer the intended behavior or reasoning behind the change. Under the conversational configuration, ChatGPT-Turbo-3.5 required 8 attempts and 3 rounds of interaction to produce a plausible patch. In the initial round, the model recognized the problem through the test failure message, which included a `java.lang.NullPointerException` and noted that the `convexHull()` method failed to handle certain geometry types correctly. Despite this, the bug-inducing change context offered limited additional support, and a valid patch was only produced after two additional rounds of interaction driven by test case feedback.


To address the issue, let's take a step-by-step approach:

1. Understand the Functionality:
  - The \`lowerCase\` function is designed to convert a given string \`input\` to lowercase using the \`Locale.ENGLISH\` locale.

2. Examine the Bug-Inducing Commit:
  - The commit message indicates that the \`Attributes\` were refactored from a \`LinkedHashSet\` to an \`array pair\`. This suggests changes in how data is stored or accessed within the system.

3. Analyze the Impact:
  - The refactoring from \`LinkedHashSet\` to an array could affect how strings or attributes are handled, potentially impacting the input to the \`lowerCase\` function.
  - If the input type or the way inputs are processed changed due to this refactor, it might lead to unexpected behavior or errors.

4. Consider Potential Issues:
  - The function itself has no apparent bug if the input is a valid \`String\`. However, changes elsewhere might have altered the input type or its validity.
  - Possible issues could include:
    - Null inputs due to missing elements in the array.
    - Non-string inputs being passed, if the data structure was changed.
    - Incorrect handling of empty or special-case inputs.

5. Fix the Root Cause:
  - Ensure that the input to the \`lowerCase\` function is always a valid \`String\`.
  - Incorporate checks or conversions if necessary to handle the updated data structure.

6. Propose the Corrected Function:
  - Add a check for null inputs to prevent \`NullPointerException\`.
  - Ensure any necessary conversions or validations are in place before using \`toLowerCase\`.

Fig. 25. Excerpt from ChatGPT-4o's response under zero-shot prompting configuration augmented with bug-inducing change information for repairing *RegressionBug-65*.

In *RegressionBug-65*, this case represents a *remote* regression bug, where code modification introduces a bug in an unchanged program element, leading to failures elsewhere in the code. The bug-inducing change information in this case includes only a vague commit message: *"Refactored Attributes to be an array pair vs LinkedHashSet. Also a couple perf* *(cpu / garbage) tweaks"* [13]. Based on this context, the model inferred that the `Attributes` class had been refactored from `LinkedHashSet` to an array-based structure, implying a change in how attribute data is stored or accessed (*Point 2 in Figure 25*). Despite the limited context, the model, guided by both the commit message and the test failure, hypothesized potential causes such as incorrect data types or null values handling (*Point 4 in Figure 25*). It then correctly diagnosed the issue and generated an appropriate fix. This case demonstrates that even minimal bug-inducing context, when interpreted in conjunction with test case feedback, can help the model understand the underlying cause and produce a correct patch.

Answer to RQ3: Our experimental results demonstrate that incorporating bug-inducing change (BIC) information significantly enhances the effectiveness of prompt-based APR techniques in repairing regression errors. Notably, ChatGPT-4o under the conversational configuration with BIC context achieves the highest number of correct repairs, successfully fixing 16 bugs.
Qualitative analysis further reveals that BIC context helps the model more effectively identify the root cause of regressions and decide whether a full or partial reversion to prior code is appropriate. However, such reversion must be applied judiciously, guided by a proper understanding of the bug context. Additionally, BIC context helps narrow the fix scope, enhancing fault localization and reducing unnecessary modifications.

7 Discussion

7.1 Lesson Learned

In this subsection, we highlight key lessons learned through our experiments and analysis that can drive future research in the field.

Enhanced APR Workflow with Bug-inducing change Information: Our empirical results highlight the significant impact of incorporating regression-inducing context on repair effectiveness. Conventionally, APR workflows operate by taking a buggy program and test suite, and attempting to generate a minimal change that pass all test cases. However, our findings suggest that enriching this workflow with additional context, specifically by detecting whether a bug is regression-related and extracting the corresponding bug-inducing commit, can substantially enhance repair outcomes. This extension is particularly beneficial in the context of continuous integration (CI) systems, where regression bugs emerge frequently and need to be addressed efficiently at scale.

APR Approach Selection for Regression Repair: Our empirical results demonstrate the promising potential of large language models (LLMs) in repairing regression errors. Furthermore, our investigation into the impact of regression-specific context on prompt-based techniques shows substantial improvements in repair effectiveness when such context is incorporated. These findings suggest that future APR approaches targeting regression errors could leverage specialized LLM-based methods tailored to the unique characteristics of such bugs. In addition, we observe strong performance from the fine-tuning-based APR techniques, such as RepairLLaMA, implying that future research may also benefit from fine-tuning LLMs on regression-specific datasets to further enhance repair performance.

Context-Aware Use of Reversion Operators: Our qualitative analysis reveals several cases where *fully or partially reverting to previously correct statements* operators result in plausible but incorrect patches. These findings suggest that while reversion-based repair operators can be effective patterns, their successful application requires a proper understanding of the bug context. Blindly applying such operators, without considering the underlying cause of the

[13]Available at: https://github.com/jhy/jsoup/commit/ea1fb65e9ff8eee82c4e379dc3236d09a5ab02e1

regression, can lead to overfitting and semantically incorrect fixes. This also suggest that template-based APR techniques must understand the context to determine whether reversion patterns are appropriate.

7.2 Experimental Cost Report

Running APR experiments using LLMs requires substantial computational resources. In our study, experiments involving RepairLLaMA, Incoder-1B/6B, and CodeGen-2B/6B were conducted using an NVIDIA A100 GPU with 80GB of VRAM. To prevent out-of-memory (OOM) crashes observed during generation, we limited the beam size to 10.

In contrast, experiments using ChatGPT models (ChatGPT-4o and ChatGPT-Turbo-3.5) did not require local computational resources, as these models were accessed via OpenAI's API. However, using these APIs incurs direct monetary costs, as users are charged based on the number of input and output tokens processed by the model. While the exact cost depends on usage, it is worth noting that these models still rely on significant backend computing infrastructure, even if this burden is offloaded from the researcher to the service provider. As of the time of our implementation, the cost for ChatGPT-4o was *$0.0025* per *1,000* input tokens and *$0.01* per *1,000* output tokens. For ChatGPT-Turbo-3.5, the cost was *$0.0005* per *1,000* input tokens and *$0.0015* per *1,000* output tokens.

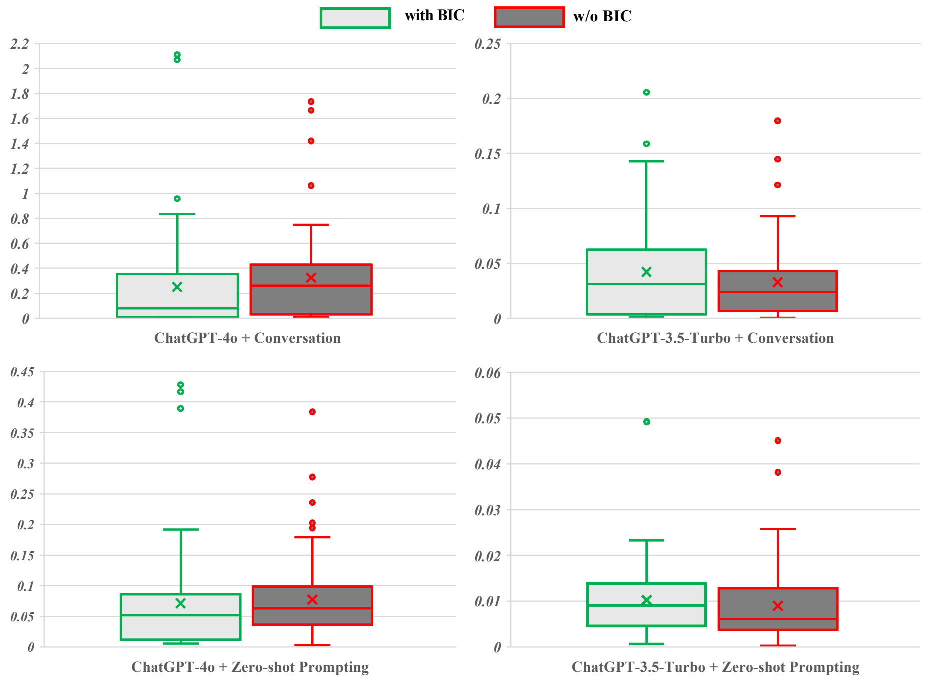


Fig. 26. Token cost of patch generation per bug across different methods.

Figure 26 presents the cost of patch generation per bug across different methods. For zero-shot prompting, the model generates up to 10 patches per bug and stops once a plausible patch is found. In the conversation strategy, the model performs up to 10 attempts per bug, with each attempt involving 5 rounds of interaction guided by test case feedback, terminating upon generation of a plausible patch.

Overall, the conversation-based strategy incurs a higher cost than zero-shot prompting due to the multiple rounds of interaction involved. For ChatGPT-Turbo-3.5, incorporating bug-inducing change (BIC) context leads to slightly higher costs compared to its counterpart without BIC, as longer prompts increase the number of input tokens and thus monetary cost. Interestingly, the opposite trend is observed with ChatGPT-4o: although incorporating BIC context increases prompt length, which would typically raise the cost, we observe a slight reduction in overall cost. This is because BIC helps the ChatGPT-4o model generate plausible patches more effectively, thereby reducing the number of attempts or rounds of interaction.

7.3 Threat to Validity

We anticipate that there are the following threats to the validity of our findings.

7.3.1 Internal Validity. An internal threat comes from the manual validation used to determine the correctness of the plausible patches. To mitigate this threat, we conduct a thorough examination each patch following prior works [63], along with careful double-checking of the results. Still, the assessment of patch correctness remains an active area of research, and further investigation is needed to refine assessment methodologies. Another internal threat concerns the manual bug categorization, which involves analyzing, dissecting and classifying developer-written patches according to established repair operators defined in prior work [46, 54]. To mitigate this risk, we involved an author with over three years of experience in Java programming to perform this task, which was then double-checked by another author. Since these repair operators represent fundamental concepts in the Java programming language, this categorization process can be considered reasonably reliable.

7.3.2 External Validity. One external threat relates to the potential exposure of our dataset to an LLM's training corpus. However, prior work [20] has shown that data leakage is less of a concern for APR compared to other code-related tasks. This is because LLM training corpora typically contain, at most, individual versions of a program, either buggy or fixed, but not aligned bug-fix pairs. Without such aligned data, LLMs are unlikely to learn APR tasks directly from training. Another external threat concerns the generalizability of our findings. In this study, we evaluated several APR techniques using 99 regression bugs collected from 32 different Java programs. While the dataset is not large, the findings could be further strengthened by expanding to a larger and more diverse set of programs with additional real-world bugs. Nevertheless, we mitigate this threat through the analysis presented in RQ1, which demonstrates substantial diversity in both fix scope and repair operators. Moreover, the benchmark has been designed with extensibility, allowing future expansion as more regression bugs are identified and validated.

7.3.3 Construct Validity. One construct validity threat concerns the experimental setup used in our empirical study. To ensure a fair comparison, we adopt the original configurations (e.g., architectures and parameters) provided by the authors of the respective techniques, preserving their internal structures. The only modification made was to the sampling size, which was adjusted to control computational cost while keeping the core techniques intact.

8 Conclusion

In this study, we conduct an empirical study of automated program repair (APR) techniques on regression errors, including both traditional and LLM-based approaches. The evaluation is performed using our high-quality benchmark, RegMiner4APR, which comprises 99 real-world regression bugs collected from 32 widely-used GitHub repositories. Experimental results reveal that traditional APR tools fail to repair any of the bugs, whereas LLM-based techniques demonstrate promising performance. Motivated by the effectiveness of LLM-based approaches, we focus on prompt-based techniques due to their flexibility and efficiency compared to fine-tuning-based methods. We specifically investigate whether incorporating regression-specific context, particularly bug-inducing change (BIC) information, can enhance repair effectiveness. Our results show substantial improvements in repair performance when BIC context is included. Additionally, our qualitative analysis reveals that BIC information provides crucial context that helps models better identify the root cause of regressions and decide whether to fully or partially revert to previously correct statements. It also narrows the fix scope, improving fault localization and reducing unnecessary code modifications

Reflecting on the findings, we draw two key insights for future work: improving regression bug repair and leveraging additional context in APR. First, our study suggests that exposing models to how past code changes introduced a regression provides valuable cues for understanding root causes and generating correct fixes. This highlights a new opportunity to guide general-purpose LLMs in identifying regression patterns and applying suitable repairs. Second, building on recent advances in LLMs, future APR research may benefit from systematically mining and integrating additional context, such as software evolution, into model training. In this direction, fine-tuning LLMs on regression-specific datasets presents a promising avenue for enhancing their ability to understand and fix regression errors.

References

[1] 2025. RegMiner4APR Benchmark. https://github.com/brojackvn/RegMiner4APR-Benchmark.

[2] 2025. RegMiner4APR Framework. https://github.com/brojackvn/RegMiner4APR-Framework.

[3] 2025. RegMiner4APR Homepage. https://brojackvn.github.io/RegMiner4APR-Homepage.

[4] 2025. RegRepair: Prompt-based Program Repair for Regression Bugs. https://github.com/brojackvn/Program-Repair-Framework.

[5] 2025. Regression validation tool. https://github.com/brojackvn/RegMiner4APR-Framework/blob/main/regression-validation.

[6] Abdulaziz Alhefdhi, Hoa Khanh Dam, Thanh Le-Cong, Bach Le, and Aditya Ghose. 2025. Adversarial patch generation for automated program repair. *Software Quality Journal* 33, 1 (Jan. 2025), 23 pages. doi:10.1007/s11219-025-09709-4

[7] Gabin An, Jingun Hong, Naryeong Kim, and Shin Yoo. 2023. Fonte: Finding bug inducing commits from failures. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 589–601.

[8] Earl T Barr, Yuriy Brun, Premkumar Devanbu, Mark Harman, and Federica Sarro. 2014. The plastic surgery hypothesis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 306–317.

[9] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2017. Travistorrent: Synthesizing travis ci and github for full-stack research on continuous integration. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 447–450.

[10] Marcel Böhme, Bruno C d S Oliveira, and Abhik Roychoudhury. 2013. Regression tests to expose change interaction errors. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. 334–344.

[11] Marcel Böhme and Abhik Roychoudhury. 2014. Corebench: Studying complexity of regression errors. In *Proceedings of the 2014 international symposium on software testing and analysis*. 105–115.

[12] Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. 2024. Repairagent: An autonomous, llm-based agent for program repair. *arXiv preprint arXiv:2403.17134* (2024).

[13] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. 2023. Qlora: Efficient finetuning of quantized llms. *Advances in neural information processing systems* 36 (2023), 10088–10115.

[14] Thomas Durieux and Rui Abreu. 2019. Critical review of bugswarm for fault localization and program repair. *arXiv preprint arXiv:1905.09375* (2019).

[15] Emelie Engström and Per Runeson. 2010. A qualitative survey of regression testing practices. In *Product-Focused Software Process Improvement: 11th International Conference, PROFES 2010, Limerick, Ireland, June 21-23, 2010. Proceedings 11*. Springer, 3–16.

[16] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999* (2022).

[17] Xiang Gao, Yannic Noller, and Abhik Roychoudhury. 2022. Program repair. *arXiv preprint arXiv:2211.12787* (2022).

[18] Péter Gyimesi, Béla Vancsics, Andrea Stocco, Davood Mazinanian, Arpád Beszédes, Rudolf Ferenc, and Ali Mesbah. 2019. Bugsjs: a benchmark of javascript bugs. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 90–101.

[19] Kai Huang, Jian Zhang, Xinlei Bao, Xu Wang, and Yang Liu. 2025. Comprehensive Fine-Tuning Large Language Models of Code for Automated Program Repair. *IEEE Transactions on Software Engineering* (2025).

[20] 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.

[21] Nan Jiang, Thibaud Lutellier, Yiling Lou, Lin Tan, Dan Goldwasser, and Xiangyu Zhang. 2023. Knod: Domain knowledge distilled tree decoder for automated program repair. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1251–1263.

[22] 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.

[23] 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.

[24] Vinay Kabadi, Dezhen Kong, Siyu Xie, Lingfeng Bao, Gede Artha Azriadi Prana, Tien-Duy B Le, Xuan-Bach D Le, and David Lo. 2023. The Future Can't Help Fix The Past: Assessing Program Repair In The Wild. In *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 50–61.

[25] Xuan-Bach D Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. S3: syntax-and semantic-guided repair synthesis via programming by examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 593–604.

[26] Xuan Bach D. Le, David Lo, and Claire Le Goues. 2016. History Driven Program Repair. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. 213–224. doi:10.1109/SANER.2016.76

[27] Xuan-Bach D Le, Ferdian Thung, David Lo, and Claire Le Goues. 2018. Overfitting in semantics-based automated program repair. In *Proceedings of the 40th international conference on software engineering*. 163–163.

[28] Thanh Le-Cong, Bach Le, and Toby Murray. 2024. Semantic-guided Search for Efficient Program Repair with Large Language Models. *arXiv preprint arXiv:2410.16655* (2024).

[29] Claire Le Goues, Neal Holtschulte, Edward K Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. 2015. The ManyBugs and IntroClass benchmarks for automated repair of C programs. *IEEE Transactions on Software Engineering* 41, 12 (2015), 1236–1256.

[30] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2011. Genprog: A generic method for automatic software repair. *Ieee transactions on software engineering* 38, 1 (2011), 54–72.

[31] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated program repair. *Commun. ACM* 62, 12 (2019), 56–65.

[32] 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).

[33] 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.

[34] Kui Liu, Li Li, Anil Koyuncu, Dongsun Kim, Zhe Liu, Jacques Klein, and Tegawendé F Bissyandé. 2021. A critical review on the evaluation of automated program repair systems. *Journal of Systems and Software* 171 (2021), 110817.

[35] Yue Liu, Thanh Le-Cong, Ratnadira Widyasari, Chakkrit Tantithamthavorn, Li Li, Xuan-Bach D Le, and David Lo. 2024. Refining chatgpt-generated code: Characterizing and mitigating code quality issues. *ACM Transactions on Software Engineering and Methodology* 33, 5 (2024), 1–26.

[36] Wenqiang Luo, Jacky Keung, Boyang Yang, He Ye, Claire Le Goues, Tegawendé F. Bissyandé, Haoye Tian, and Xuan Bach D. Le. 2025. When Fine-Tuning LLMs Meets Data Privacy: An Empirical Study of Federated Learning in LLM-Based Program Repair. *ACM Trans. Softw. Eng. Methodol.* (May 2025). doi:10.1145/3733599 Just Accepted.

[37] Fernanda Madeiral, Simon Urli, Marcelo Maia, and Martin Monperrus. 2019. Bears: An extensible java bug benchmark for automatic program repair studies. In *2019 IEEE 26th international conference on software analysis, evolution and reengineering (SANER)*. IEEE, 468–478.

[38] Matias Martinez and Martin Monperrus. 2016. Astor: A program repair library for java. In *Proceedings of the 25th international symposium on software testing and analysis*. 441–444.

[39] Matias Martinez and Martin Monperrus. 2018. Ultra-large repair search space with automatically mined templates: The cardumen mode of astor. In *Search-Based Software Engineering: 10th International Symposium, SSBSE 2018, Montpellier, France, September 8-9, 2018, Proceedings 10*. Springer, 65–86.

[40] 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.

[41] Martin Monperrus. 2018. *The living review on automated program repair*. Ph. D. Dissertation. HAL Archives Ouvertes.

[42] 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.

[43] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474* (2022).

[44] Dor Nir, Shmuel Tyszberowicz, and Amiram Yehudai. 2008. Locating regression bugs. In *Hardware and Software: Verification and Testing: Third International Haifa Verification Conference, HVC 2007, Haifa, Israel, October 23-25, 2007. Proceedings 3*. Springer, 218–234.

[45] Akira K Onoma, Wei-Tek Tsai, Mustafa Poonawala, and Hiroshi Suganuma. 1998. Regression testing in an industrial environment. *Commun. ACM* 41, 5 (1998), 81–86.

[46] Kai Pan, Sunghun Kim, and E James Whitehead. 2009. Toward an understanding of bug fix patterns. *Empirical Software Engineering* 14 (2009), 286–315.

[47] 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*. 24–36.

[48] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).

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

[50] Ripon K Saha, Yingjun Lyu, Wing Lam, Hiroaki Yoshida, and Mukul R Prasad. 2018. Bugs. jar: A large-scale, diverse dataset of real-world java bugs. In *Proceedings of the 15th international conference on mining software repositories*. 10–13.

[51] Ridwan Shariffdeen, Martin Mirchev, Yannic Noller, and Abhik Roychoudhury. 2023. Cerberus: a Program Repair Framework. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 73–77.

[52] André Silva, Sen Fang, and Martin Monperrus. 2023. Repairllama: Efficient representations and fine-tuned adapters for program repair. *arXiv preprint arXiv:2312.15698* (2023).

[53] Edward K Smith, Earl T Barr, Claire Le Goues, and Yuriy Brun. 2015. Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 532–543.

[54] 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.

[55] Xuezhi Song, Yun Lin, Siang Hwee Ng, Yijian Wu, Xin Peng, Jin Song Dong, and Hong Mei. 2022. Regminer: towards constructing a large regression dataset from code evolution history. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 314–326.

[56] Shin Hwei Tan and Abhik Roychoudhury. 2015. relifix: Automated repair of software regressions. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 471–482.

[57] Shin Hwei Tan, Jooyong Yi, Sergey Mechtaev, Abhik Roychoudhury, et al. 2017. Codeflaws: a programming competition benchmark for evaluating automated program repair tools. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 180–182.

[58] David A Tomassi, Naji Dmeiri, Yichen Wang, Antara Bhowmick, Yen-Chuan Liu, Premkumar T Devanbu, Bogdan Vasilescu, and Cindy Rubio-González. 2019. Bugswarm: Mining and continuously growing a dataset of reproducible failures and fixes. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 339–349.

[59] Ratnadira Widyasari, Sheng Qin Sim, Camellia Lok, Haodi Qi, Jack Phan, Qijin Tay, Constance Tan, Fiona Wee, Jodie Ethelda Tan, Yuheng Yieh, et al. 2020. Bugsinpy: a database of existing bugs in python programs to enable controlled testing and debugging studies. In *Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 1556–1560.

[60] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated program repair in the era of large pre-trained language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1482–1494.

[61] 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 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 819–831.

[62] Jiahong Xiang, Xiaoyang Xu, Fanchu Kong, Mingyuan Wu, Zizheng Zhang, Haotian Zhang, and Yuqun Zhang. 2024. How far can we go with practical function-level program repair? *arXiv preprint arXiv:2404.12833* (2024).

[63] 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).

[64] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clement, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2016. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering* 43, 1 (2016), 34–55.

[65] Boyang Yang, Haoye Tian, Jiadong Ren, Shunfu Jin, Yang Liu, Feng Liu, and Bach Le. 2025. Enhancing Repository-Level Software Repair via Repository-Aware Knowledge Graphs. arXiv:2503.21710 [cs.SE] https://arxiv.org/abs/2503.21710

[66] 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.

[67] Xin Yin, Chao Ni, Shaohua Wang, Zhenhao Li, Limin Zeng, and Xiaohu Yang. 2024. Thinkrepair: Self-directed automated program repair. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1274–1286.

[68] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi Bairavasundaram. 2011. How do fixes become bugs?. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 26–36.

[69] Yuan Yuan and Wolfgang Banzhaf. 2018. Arja: Automated repair of java programs via multi-objective genetic programming. *IEEE Transactions on software engineering* 46, 10 (2018), 1040–1067.

[70] Andreas Zeller. 1999. Yesterday, my program worked. Today, it does not. Why? *ACM SIGSOFT Software engineering notes* 24, 6 (1999), 253–267.

[71] 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.