# An Empirical Comparison on ChatGPT, Gemini, and Mistral for Automated Program Repair in Real-World C/C++ Projects

Johnson Hu, Musheng He, Srikkanth Ramachandran, Rishika Garg, Harshil Bhandari
University of California, Davis
Davis, CA, USA

**GitHub Link:** See [6]

## 1 Introduction

Automated Program Repair (APR) represents a crucial stage in software development, aimed at minimizing manual intervention and enhancing productivity by automating the correction of code errors or bugs. Recently, the technically advanced class of Large Language Models (LLMs) significantly improved performance across a wide spectrum of NLP and coding tasks such as machine translation and defect detection, thereby exerting a profound influence on the APR domain within the Software Engineering (SE) community[18]. Additionally, emerging LLM-based APR works have also benefited from an increasing number of empirical studies to evaluate and probe the capabilities of LLMs. These empirical studies encompass a breadth of different pre-trained models[16], bug types[7], and programming languages [10], providing invaluable insights into future learning-based APR work.

However, existing empirical studies are predominantly centered on pre-trained models fine-tuned on a large-scale corpus of code files[16], which require additional model training and expensive computation resources, and lack accessibility to software developers lacking proficiency in the AI domain. Furthermore, a notable limitation of prior empirical investigation lies within their benchmarks. While many empirical studies evaluate LLMs on specific Java Benchmarks (e.g., Defects4J[9] and QuixBugs[14]) and benchmarks from class assignments and competitive programming submissions (e.g., DeepFix [5] and SPoC[11]), LLMs' capabilities of eliminating or mitigating real-world C/C++ bugs across various error categories are yet to be explored.

**Our Goal**. In this study, we aim to address these gaps by **conducting an empirical study on the program repair capabilities of 3 general-purpose LLMs for real-world C/C++ projects**. We have selected 3 generative text models that were not specifically fine-tuned for downstream code-related tasks: *GPT-3.5-Turbo*, *Gemini* and *Mistral*, and utilize a recent C/C++ benchmark collecting 200+ bugs from 22 open-source GitHub repositories: BugsC++[1], to evaluate the actual performance of these LLMs in fixing real-world C/C++ bugs.

### 1.1 Background

It is estimated that software bugs cost about $7 billion per year. Driven by this, researchers have long been striving to obtain practical automated tools that can identify and fix bugs (See for e.g., [15], [4]). The first approaches were ad-hoc search-based (SBSE) or heuristic-based (such as genetic programming) to compute plausible bug fixes. Since the emergence of machine learning models, researchers have looked into training models that can automatically provide bug fixes. Prior to the emergence of Large Language Models, such computation was generally very resource-intensive, typically requiring GPUs running for several hours- days, even, to train these models. Today, one can easily (and affordably) access Large Models trained on massively large datasets having impressive code-generation abilities. These models can provide solutions to well-known problems. E.g., they can readily provide code to compute the $n^{th}$ Fibonacci number (a quintessential recursive algorithm). Inspired by these possibilities, we seek to investigate their capabilities in APR, especially for real-world projects. In the study of APR tools, the most common metric is the number of *plausible* patches generated by the tool. A patch produced for a bug is deemed *plausible* if it passes all the corresponding test suites.

We select three models, namely (i) Mistral-Medium (ii) GPT 3.5-Turbo Instruct and (iii) Gemini-Pro. The costs of these models are measured per token of input given and output produced. A token is the most basic unit of data processed by these models. The precise number for a given text might differ for the various models, but generally one can estimate that one word equals approximately 1.3 tokens. The cost per token of the three models are $2.7, $1.5 and $0 per million input tokens and $8.1, $2 and $0 per million output tokens respectively for Mistral, GPT 3.5 and Gemini-Pro. Most notably, the model of Gemini we used offers a free API inference service. To the best of our knowledge, no other service offers a trial-free unlimited API service. The details of these pre-trained LLMs are not disclosed to the public, so we cannot confirm their sizes. It is, however, speculated that GPT 3.5-Turbo has a size between 20-175 billion parameters, and Mistral-Medium has a size between 70-196 billion parameters. The models of Mistral and Gemini used were last updated in February 2024. GPT 3.5 Turbo was last updated on November 6, 2023. The context window size (i.e., the largest piece of text from which the LLM can learn in one prompt) for the models are 32K, 16K, and 30K tokens for Mistral-Medium, GPT 3.5 and Gemini-Pro respectively. Higher context window enables the LLM to learn from more examples and provide better-quality output. Due to these different parameters and methods of training, the models can perform differently for the APR task.

## 1.2 Research Questions

Our first research question aims to extend prior work to real-world C/C++ programs.

**RQ1**. How effective are LLMs in identifying and fixing real-world C/C++ defects and does their effectiveness vary under different repair scenarios and using different prompting methods?

Our choice of dataset (BugsC++) enables us to evaluate the feasibility of LLMs against real-world C/C++ bugs. The BugsC++dataset provides a categorization of its bugs into different repair scenarios. Based on these available details, we designed four different prompts with varying levels of details embedded in them. We observed that additional information helped the LLMs to produce more compilable and plausible patches. Mistral-Medium generates the most plausible patches across all repair scenarios, demonstrating the best APR tool among the three LLMs.

The LLMs chosen have been known to exhibit significant capabilities in maintaining conversation, understanding context, and leveraging user feedback. In our next research question, we aim to gain a better understanding of these capabilities towards the APR task.

**RQ2**. Do LLMs perform significantly better with additional context information and extra rounds of interactions with the user?

We designed FixTalk, an automated mechanism to exchange a conversation with any LLM that provides an inference API. We find that FixTalk increases the number of compilable solutions provided by GPT 3.5 Turbo and Gemini, thereby suggesting that interaction does help make the model output robust patches.

**RQ3**. How competitive are LLMs against traditional APR tools?

We evaluate GenProg, a traditional state-of-art APR tool for C defects that utilizes evolutionary genetic programming and rests on the plastic surgery hypothesis[2], on the BugsC++ dataset. Compared to three LLMs, GenProg exhibits superior repair capabilities for C defects in most C projects, and Mistral-Medium is able to achieve a comparable performance to GenProg.

## 1.3 Our Contributions

Our empirical study delves into the APR abilities of general-purpose LLMs for real-world C/C++ defects. To sum up, our work has the following contributions.
⋆**Systematic Study**. We investigate the APR capabilities of three general-purpose LLMs on a new C/C++ defect benchmark in a systematic way. Our experiment design encompasses a wide range of buggy projects and various repair scenarios (single-line/multi-line bugs and more than 3 error types).
⋆**Prompting Techniques**. We explore various prompting designs that may enhance LLMs' repair capabilities. We develop fully automated construction functions for five distinct prompting methods, notably including a conversational style prompt that provides LLMs with instant feedback incorporating detailed failure information for effective APR.
⋆**Evaluation**. We evaluate the performance of three

LLMs using different prompting strategies against the current start-of-the-art traditional APR tool, GenProg. We are the first study assessing GenProg's repair capabilities on the BugsC++ benchmark. Our findings establish valuable benchmarks and baselines for future studies on LLMs' APR efficiency in the C/C++ domain.

## 2 Related Work

Here we discuss 3 papers most relevant to our study as the main sources for our topic.

*Xia, Wu and Zhang et al.* [16] evaluated the performance of LLMs for APR. They considered several models available on Hugging Face, by downloading the model parameters and locally generating prompts. Additionally, they also use an LLM called Codex using an inference API provided by OpenAI (which has since been either deprecated or merged with GPT). They showed that Codex could beat the then state-of-the-art APR tools on five datasets, Defects4J 1.2, Defects4J 2.0 [9], Quix-Bugs Java, Quix-Bugs Python [14] and ManyBugs [12]. Of these five data sets, only the ManyBugs dataset contains code written in C/C++. Their prompt design is of three types, (i) provide LLMs with a complete function and ask it for the fix, (ii) provide LLMs with the exact range of lines where the bug appears, and finally (iii) ask LLMs to change a given line of the code.

We extend their study with three general-purpose LLMs: Mistral, GPT-3.5-Turbo, and Gemini, which are not investigated in their work. Furthermore, we consider the BugsC++ benchmark [1], a real-world C/C++ benchmark that provides high usability through a similar user interface inspired by Defects4J, and ensures reproducibility by isolating each defect in Dockers.

*Xia, Wei and Zhang et al.*[17] introduced ChatRepair, a conversation-driven approach to Automated Program Repair (APR) with ChatGPT. The study improved APR using a conversational approach and evaluated ChatRepair against other solutions using standard datasets to test its effectiveness and adaptability. It questioned ChatRepair's performance against top methods, its effectiveness in various repair scenarios, and its components' contributions. Python and the get-3.5-turbo ChatGPT model were used for implementation, with a limit on repair attempts and the Defects4j and QuixBugs data sets for evaluation.

Our project expands the comparison of LLMs for APR beyond ChatRepair to include three general-purpose LLMs, aiming for a fully automated, conversation-driven approach. Unlike the paper focusing solely on ChatGPT, we compare multiple LLMs and specifically use the BugsC++benchmark for C/C++ program evaluation. Our comprehensive study also focuses on the competitiveness with traditional APR tools, considering metrics like plausible patches and response time.

*Huang, Meng, and Zhang et al.* [7] discovered that large language models of code (LLMCs) can repair programs using zero/few-shot learning, yet the potential of enhancing their repair abilities through fine-tuning is underexplored. This work investigated fine-tuned LLMCs' repair capabilities on multi-hunk defects, focusing on five LLMCs with different architectures: CodeBERT, GraphCodeBERT, PLBART, CodeT5, and UniXcoder. Their analysis spans bugs, vulnerabilities, and errors in both single-line and multi-line contexts, suggesting fine-tuning can outperform current APR tools with precise data.

However, the researchers did not examine the APR potential of general-purpose LLMs or the effectiveness of their prompts. Our study fills this gap by assessing the code repair capabilities of three general-purpose LLMs and optimal prompt design methodologies.

## 3 Data and Methods

### 3.1 Data

Our research employs *BugsC++* [1] as the benchmark to gauge the efficacy of three experimental LLMs in repairing practical C/C++ programs. It is a new benchmark that contains 215 real-world bugs collected from 22 open-source C/C++ projects containing pairs of buggy and patch versions of the source project. For classification, *BugsC++* applies 4 tag categories, including 14 bug types to distinguish different defects.

Our research centers on the two *Buggy Lines* tags: *single-line* and *multi-line*, and six *Error Types* tags: *invalid-condition*, *invalid-format-string*, *memory-error*, *logical-error*, *omission*, *division-by-zero*. We selected specific defects for each RQ to cater to their unique requirement. As illustrated in Figure 1, for RQ1, we selected 16 defects and divided them into 3 repair scenarios: RS1 (single-line, invalid-condition), RS2 (single-line, invalid-format-string), and RS3 (single-line, memory-error), to conduct experiments utilizing four different prompt designs; In our RQ2 study, additional 12 defects (berry 1~4, libtiff 2~5, libucl 1~4), involving multi-line bugs and three new error types (logical errors, omission, and division by zero), are included. We experimented with our conversational repair method on all 28 defects; For RQ3, we compared the efficacy of a traditional state-of-the-art APR tool against LLMs on 3 C projects.

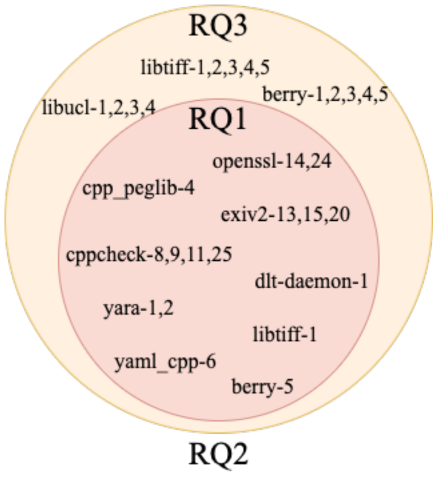


**Figure 1: Defects in each RQ**

In the project, we designed a data module as depicted in Figure 2, integrating the benchmark to download multiple buggy projects and implementing APIs necessary for automating our experiments. This module involves offering buggy code snippets for LLMs and testing the LLM-generated fixes to assess effectiveness.

The data module comprises three key components: *fetch*, *update*, and *test*. The *fetch* component is responsible for locating the file where the defect appears and extracting the complete function where the buggy lines reside. In addition, to implement prompt augmentation using bug localization, the *fetch* component also adds special labels to the returning function to facilitate LLMs locating the buggy lines.

After returning the buggy function via a specific interface for LLMs and receiving the repair code from LLMs, the *update* component will replace the original code snippets with the fixed code. Notably, the entire replacement process is risk-free, because this module will prepare a backup of the original buggy code in advance before code replacement. After the testing process is completed, the code will be restored as before.

The last *test* component executes benchmark-designed test cases for each defect and collects results that include compilation details, test pass rates, failure information, and code of failed tests to improve LLM prompts in interactive settings. Specifically, it compiles and runs LLM-generated repair code within an isolated docker container. The *BugsC++* does not directly provide the compilation output, but this component retrieves it from the container and uses a window mechanism to accurately and comprehensively capture error compilation information related to the repair code, facilitating LLMs in refining their repair code.

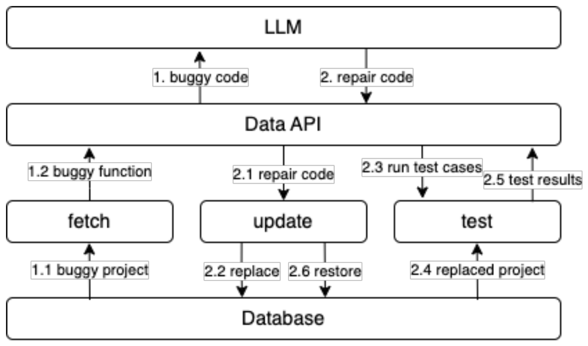


**Figure 2: Workflow of data module**

## 3.2 RQ1 Study

In this section, we describe our prompt design, validation procedures of candidate patches generated by LLMs, and metrics used in RQ1 study.

For our prompt design, we begin with a basic prompt that provides only a single function containing the bug, as done in prior work ([16]). Inspired by [8], we use two prompt augmentation techniques by providing bug type annotation and localization information based on our basic prompt. For facilitating downstream parsing of LLM's responses, we also structured the output of LLM by specifying a JSON template in our prompt with keys indicating the error-lines, error-types, explanations of each error, and their confidence.

We develop a series of Python scripts to automate the following 5 crucial steps to address RQ1: 1) extracting buggy code from a specific repository; 2) embedding the faulty code snippet within prompts for automatic generation; 3) parsing LLM responses to obtain the repaired function and the corresponding JSON file; 4) replacing the original buggy code with the generated "fixed" version, rebuilding the whole project, and running the entire test suites; 5) Obtain compilation and test information to evaluate the patch.

In our experiments, we find that all 3 LLM models can stably output the error information (error line number and error type) they identified, the explanations for each error, and their confidence score into the JSON format we required.

**P1** (Basic Prompt). We provided LLM a role (e.g., "You are an automated program repair tool.") and a single buggy function as input expecting them to output a corresponding fixed version.

**P2** (Bugs Type Annotations). In addition to **P1**, we provide the bug-type which could be one of invalid-condition, invalid-format-string, memory-error. These classifications are provided by BugsC++ and selected in our repair scenarios.

**P3**. (Bug Localization). In addition to **P1**, we annotate the buggy file with the exact line by surrounding the buggy region with special location markers <START_BUG> and <END_BUG>.

**P4**. For the last prompt, we combine **P2** and **P3**.

**Metrics.** To evaluate the APR performance of three LLMs, we utilize the following metrics in our RQ1 study:

- Compiled Patches: The number of patches that were successfully built.
- Plausible Patches: The number of patches that passed all test cases.
- Identify Accuracy: We asked the LLMs to identify the error-causing lines of code and compare them to the error lines located in the original buggy function. We calculate the fraction of the occurrences in which they are equal.
- Pass rate: The percentage of passed test suites of an LLM-generated patch. For each buggy function in the repository, we prompted LLMs to generate fixes and computed the pass rate after running the entire test suites.
- Confidence: We ask LLMs to include a score showing their confidence in outputting their repairs.
- Response Time: The time the LLM takes to respond once given the prompt.

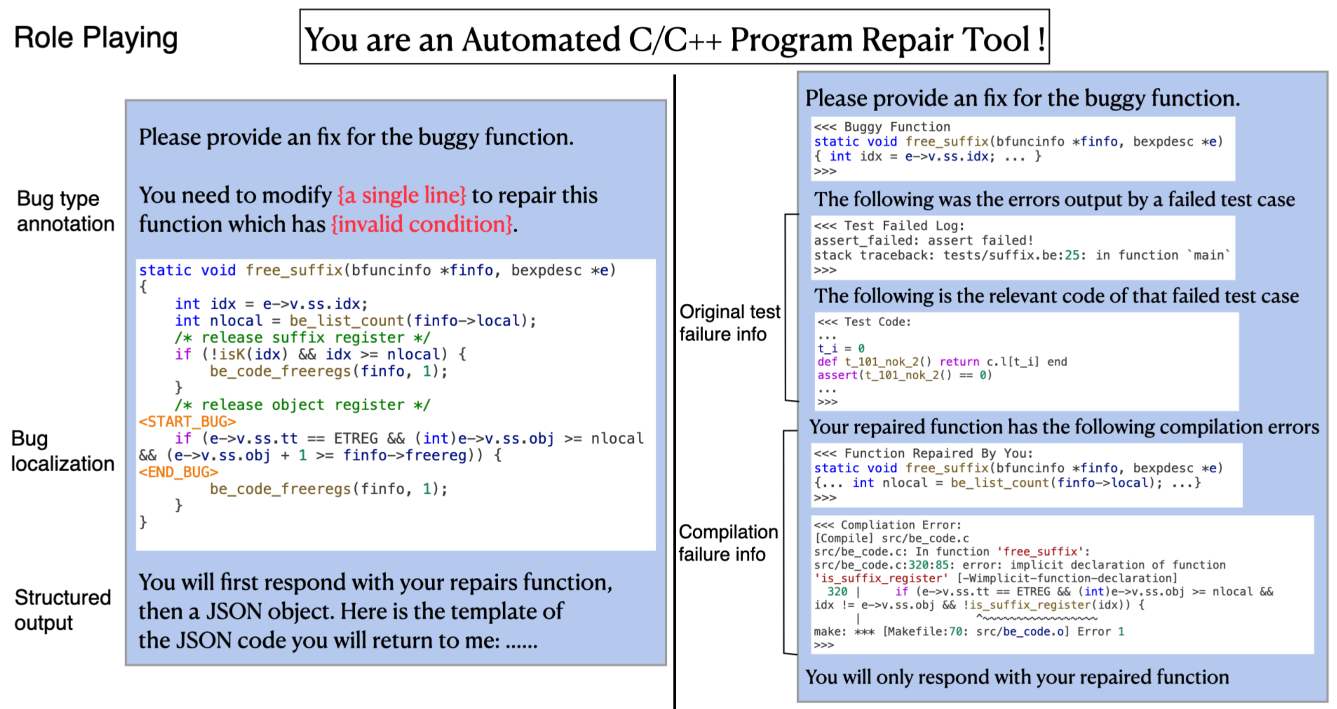
## 3.3 RQ2 Study: FixTalk

Our second research question (RQ2) investigates the potential of LLMs in repairing real-world defects through multiple rounds of conversations with users. In our RQ1 study, we utilize two prompt augmentation strategies: bug type annotation and bug localization, providing LLMs with specific information about the types and locations of bugs. However, in actual software development, developers often lack prior knowledge about the nature and location of program bugs. They typically rely on existing test suites or create new ones to verify the correct functionalities of programs. Upon encountering test failures, developers adopt the failure information to diagnose the symptoms and root causes of the bugs. The bug-fixing process is inherently iterative, involving repeated suspicious code modifications and test suite evaluations until a correct fix is identified.

Building on this insight, our hypothesis posits that LLMs can leverage compile and run-time error information to iteratively analyze code lines and make appropriate change to the original buggy function in a conversational manner. Inspired by Xia's work in 2023 [17], we introduce our method for RQ2 study as FixTalk, emphasizing the idea of the conversations between LLMs and us regarding the program-specific compilation and test information provided to aid LLMs in rectifying the buggy function. FixTalk applies to all defects within the BugsC++benchmark. Beyond the original 16 buggy repositories, which encompass only three single-line repair scenarios, we have incorporated an additional 12 defects (berry 1~4, libtiff 2~5, libucl 1~4) from BugsC++ that involves multi-line errors and three new error types: logical errors, vulnerabilities, and division by zero. We assessed the efficacy of FixTalk on these 28 defects, with detailed results to be discussed in Section 4.

In our FixTalk approach, The initial prompt provided to LLMs consists of a concatenation of the original buggy function and its relevant test failure information. This prompt serves as the initial input to the LLM, from which it generates a candidate patch. The patch is then verified using test suites through the interface of our data module built on top of BugsC++. Based on the test results, we iteratively update the input prompt and repeat the process until the maximum number of conversation iterations is reached. We describe each stage of our method in greater detail below.

**Initial Prompt.** At the outset, we give LLM a role as an automated program repair tool for C/C++ to prepare LLMs for the specific task. Next, our initial prompt showcases the original buggy function to the LLM, similar to our RQ1 study. However, in RQ2 we no longer



You are an Automated C/C++ Program Repair Tool !

Please provide an fix for the buggy function.

You need to modify {a single line} to repair this function which has {invalid condition}.

```
static void free_suffix(bfuncinfo *finfo, bexpdesc *e)
{
    int idx = e->v.ss.idx;
    int nlocal = be_list_count(finfo->local);
    /* release suffix register */
    if (!isK(idx) && idx >= nlocal) {
        be_code_freeregs(finfo, 1);
    }
    /* release object register */
<START_BUG>
    if (e->v.ss.tt == ETREG && (int)e->v.ss.obj >= nlocal
&& (e->v.ss.obj + 1 >= finfo->freereg)) {
<END_BUG>
        be_code_freeregs(finfo, 1);
    }
}
```

You will first respond with your repairs function, then a JSON object. Here is the template of the JSON code you will return to me: ......

Please provide an fix for the buggy function.

```
<<< Buggy Function
static void free_suffix(bfuncinfo *finfo, bexpdesc *e)
{ int idx = e->v.ss.idx; ... }
>>>
```

The following was the errors output by a failed test case

```
<<< Test Failed Log:
assert_failed: assert failed!
stack traceback: tests/suffix.be:25: in function `main`
>>>
```

The following is the relevant code of that failed test case

```
<<< Test Code:
...
t_i = 0
def t_101_nok_2() return c.l[t_i] end
assert(t_101_nok_2() == 0)
...
>>>
```

Your repaired function has the following compilation errors

```
<<< Function Repaired By You:
static void free_suffix(bfuncinfo *finfo, bexpdesc *e)
{... int nlocal = be_list_count(finfo->local); ...}
>>>
```

```
<<< Compliation Error:
[Compile] src/be_code.c
src/be_code.c: In function 'free_suffix':
src/be_code.c:320:85: error: implicit declaration of function
'is_suffix_register' [-Wimplicit-function-declaration]
  320 |     if (e->v.ss.tt == ETREG && (int)e->v.ss.obj >= nlocal &&
idx != e->v.ss.obj && !is_suffix_register(idx)) {
      |                      ^~~~~~~~~~~~~~~~~~
make: *** [Makefile:70: src/be_code.o] Error 1
>>>
```

You will only respond with your repaired function

a) Prompt 4 for the buggy function in berry-5 repo

b) Prompt of FixTalk after LLM outputs a uncompilable patch

**Figure 3: Example prompts constructed in RQ1 and RQ2**

augment our prompt using type annotation and localization. Instead, we incorporate two critical pieces of information from the original failing test: 1) the error message produced, 2) the relevant code lines triggering the test failure. As illustrated in Fig.3b), the error message from the terminal log and the relevant test code provide concrete details about the failing test: an assertion error happens when we compare the first element of the class `c.l`, which is linked to the buggy function. This combination of semantic information and hints from the failing test allows the LLM to better understand the correct usage of the function and the expected test output, thereby enabling it to generate a correctly repaired function. Finally, we instruct the LLM to respond solely with the repaired function.

**Conversation-Driven Repair.** Algorithm 1 outlines the repair process employed by our FixTalk method. Our method involves a conversational loop where the LLM is iteratively prompted to generate a candidate patch. The input prompt is then updated based on the patch's outcome: whether it is uncompilable, compilable but fails to pass all test suites, or a plausible patch as determined by test validation results. For patches that either fail to compile or pass the test, we augment the initial prompt with detailed feedback to construct a new input for the subsequent conversational round. This update includes the previous incorrect patch, allowing the LLM to learn from past mistakes and avoid generating the same incorrect patch again. Given the limited context window of LLMs, if a plausible patch is not generated, we base the prompt solely on feedback from the latest round, regardless of earlier errors. The first input prompt to FixTalk is initialized to the initial prompt as discussed above.

In each round of our conversation, the newly generated patch by the LLM is first tested against the original failing test. If the generated patch fails to compile, we concatenate the initial prompt, the generated patch, and the compilation errors together as the input for the next loop. Fig.3b) demonstrates an example prompt constructed following the identification of a patch as uncompilable (implicit declaration of function `'is_suffic_register'`). The prompt focuses on parts of the compilation log that highlight build errors to keep the input prompt succinct.

**Algorithm 1** Conversation-Driven Repair For RQ2

**Input:** LLM, maxTries, iPrompt (initial Prompt), oFailTest (original failing test suite), testSuite (complete test suites), Update(prompt update function)
**Output:** LLMPatches (all patches generated by LLMs), PlausiblePatches, avgResponseTime

```
1:  function FixTalk:
2:      currentTries ⟵ 0, totalTime ⟵ 0
3:      input ⟵ iPrompt
4:      LLMPatch ⟵ []                         ▹ all patch list
5:      pList ⟵ []                            ▹ plausible patch list
6:      while currentTries < maxTries do
7:          patch, responseTime ⟵ LLM(input)
8:          totalTime + = responseTime
9:          LLMPatch ⟵ [patch]
10:         otestResult ⟵ Test(patch, oFailTest)
11:         if otestResult is PASS then
12:             testResult ⟵ Validate(patch, testSuite)
13:             if testResult is PASS then
14:                 pList ⟵ [patch]
15:                 input ⟵ Update(iPrompt, pList)
16:             end if
17:         else if otestResult is BUILDERROR then
18:             info ⟵ otestResult.CompileInfo
19:             input ⟵ Update(iPrompt, patch, info)
20:         else
21:             info ⟵ "still doesn't fix original failure"
22:             input ⟵ Update(iPrompt, info)
23:         end if
24:         currentTries ⟵ currentTries + 1
25:     end while
26:     avgResponseTime = totalTime / maxTries
27:     return LLMPatch, pList, avgResponseTime
28: end function
```

If the patch builds successfully but fails to pass the original test case, the subsequent prompt is updated including the ineffective patch and a brief message indicating "It still does not fix the original test failure." Conversely, if the patch fixes the original issue, it then undergoes evaluation on the entire test suite. Failure in another test case leads to a reinitialization of the input prompt. That's because due to the time constraint of this project, for each defect we can only provide test failure information of the original failing test case. During implementation we noted that the test error messages of some defects do not directly point to any test code files, hence we hard code relevant test code only related to the original failing test case of each defect.

**Plausible Patches.** Following the plausible patch generation method utilized in ChatRepair[17], we compile a list of plausible patches. Each time a plausible patch is generated by the LLM, we add that patch to the list and construct the input prompt with this list based on the initial prompt, requesting LLMs to generate alternative variations in the next round. We hypothesize that LLMs can leverage useful components and patterns from this accumulation of plausible patches to create more varied plausible patches and potentially correct solutions.

Our methodology diverges from ChatRepair in two key ways. Firstly, we integrate the process of prompting LLMs to generate plausible patches in the conversational loop rather than treating it as a separate step. This adjustment is motivated by the observation that LLMs might still produce an incorrect fix even after being provided with a list of plausible patches for generating an alternative solution. However, such a fix might be close to meeting all testing criteria, as the LLM has assimilated the lessons from all previous plausible patches, guiding its repair attempts in a particular direction and pattern. Introducing relevant error information in this situation can assist the LLM in deriving a new plausible patch. Secondly, unlike ChatRepair, which bases its prompt construction on the precise location of the bug, FixTalk does not require predefined bug localization. Instead, it challenges LLMs to identify the erroneous code lines through the analysis of feedback, fostering a deeper level of reasoning and problem-solving.

## 3.4 RQ3 Study: Baseline Experiments

For RQ3 study, we compare 3 LLMs' performance under our RQ1 and RQ2 studies against the state-of-the-art traditional tool for C program repair, GenProg [13]. GenProg (GP) is an evolutionary program repair tool that uses the standard genetic algorithm to search for repairs of C defects. In the actual computation of GenProg, it represents candidate repairs as code edit sequences, evaluating each against test suites to identify higher fitness candidates, subjecting them to mutation and crossover processes.

GenProg is adaptable for C programs, capable of repairing defects without formal specifications, program annotations, or special coding practices [13]. To operate GP on any program, three inputs are essential: 1) pre-processed versions of the original buggy C code files; 2) a compilation script that instruct GP to compile the program after a new variant code is generated; 3) a test script that is callable by GP to test candidate repairs on the test suites and evaluate their fitness.

Our baseline comparison focuses on 14 defects from three C projects in BugsC++: berry, libtiff, and libucl. To assess GPT's efficacy, for each defect we prepare the corresponding pre-processed codes and scripts. Our compilation script follows the version used in the libtiff scenario of ManyBugs benchmark [12]. For the test script of each defect, we developed our custom version.

**Metrics.** For a comparable evaluation of 3 LLMs and GenProg, we adopt a subset of metrics from our RQ1 study: For RQ2 we assess the number of total patches, compilable patches, and plausible patches generated by each LLM after running FixTalk once for each defect. Additionally, we use the average response time to gauge the LLMs' efficiency in responding. In RQ3 for assessing GP's performance, we adhere to the metrics outlined in[12]: number of defects repaired per program and the average time to find a repair.

## 4 Results

### 4.1 RQ1: LLM Results on 3 Repair Scenarios

Our experiments tested 3 LLMs against 16 repositories each containing only a single-line error. For each of our 4 prompt variations, the LLMs were also queried 3 times to validate the robustness of LLMs' responses.

***Finding 1***. All three LLM models demonstrate significant capability in generating compilation patches across repair scenarios (RS).

Table 1 illustrates the repair outcomes for each LLM on various RSs, indicating the **plausible patch number/compilation patch number** and the distribution of plausible patches among repositories (#Patch and #Repo columns, respectively). Using prompt 1 for RS1, Mistral, Gemini, and GPT-3.5 achieve compilation rates of 46.2%, 42%, and 76.9%, respectively. However, the number of plausible patches within compilation patches remains low across most scenarios. This may be attributed to the absence of detailed syntactic information in our prompt design, such as class-level and method-level attributes, potentially leading LLMs to make conservative changes.

***Finding 2***. Mistral exhibits notable performance improvement when employing prompt augmentation approaches, particularly with the inclusion of bug type annotations and localization.

Table 2 demonstrates that these enhancements result in Mistral generating 14 more compilable patches and 6 more plausible patches under RS1. However, we failed to observe similar improvements for Gemini. For GPT3.5-Turbo, adding localization to the prompt appears to enhance its repair capability, particularly for a specific buggy repository, leading to a consistent generation of plausible patches.

***Finding 3***. Mistral, Gemini, and GPT3.5-Turbo perform differently not only under different RS but also when fixing different repositories in the same RS.

We observed an exceptional performance of Mistral when fixing memory-error bugs: only given the basic prompt Mistral can stably produce plausible patches for repositories containing memory errors. We acknowledged that this phenomenon needs to be further investigated to consider the entire set of memory-error repositories. Table 5 in the Appendix provides detailed statistics for selected repositories in RS1 showcasing Mistral's ability to generate plausible patches for 4 repositories using prompt 4, compared to 1 for GPT3.5 and none for Gemini. We observed discrepancies in the repositories where Mistral and GPT3.5 achieved a 100% average pass rate. Finally, in experiments, we found the identify accuracy metric does not truly reflect LLMs' ability to identify error codes due to mismatches in the error line reported by LLMs compared to the actual line LLM changed.

### 4.2 RQ2: Conversation-driven approach

***Finding 4***. FixTalk improves the compilation rate of the generated patches for all three LLMs.

Conversation-driven methodology can provide significantly better quality output in terms of the *compilation rate*, i.e., the fraction of solutions provided that compile successfully. See Figure 4 for a bar plot of the different compilation rates. For Mistral-Medium, the compilation rate is lower, whereas for GPT and Gemini, the compilation rates are significantly better.

**Table 1: Repair results for 3 LLMs on 3 repair scenarios**

| | #Total Repo | Mistral-Medium | | | | Gemini | | | | GPT-3.5-Turbo | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Prompt 1 | | Prompt 4 | | Prompt 1 | | Prompt 4 | | Prompt1 | | Prompt4 | |
| | | #Patch | #Repo | #Patch | #Repo | #Patch | #Repo | #Patch | #Repo | #Patch | #Repo | #Patch | #Repo |
| Repair Scenario 1 (single-line, invalid-condition) | 13 | 0/18 | 0 | **6/32** | **4** | 1/21 | 1 | 0/28 | 0 | 1/30 | 1 | 3/30 | 1 |
| Repair Scenario 2 (single-line, invalid-format-string) | 3 | **2/9** | **1** | **2/9** | **2** | 1/9 | 1 | 1/7 | 1 | 0/9 | 0 | 0/9 | 0 |
| Repair Scenario 3 (single-line, memory-error) | 2 | **5/6** | **2** | **4/6** | **2** | 0/2 | 0 | 0/1 | 0 | 0/6 | 0 | 0/6 | 0 |

**Table 2: Repair results for single-line and invalid-condition errors**

| | Mistral-Medium | | Gemini | | GPT-3.5-Turbo | |
|---|---|---|---|---|---|---|
| | #Patch | #Repo | #Patch | #Repo | #Patch | #Repo |
| Prompt 1 | 0/18 | 0 | 1/21 | 1 | 1/30 | 1 |
| Prompt 2 (*bug type*) | 8/31 | 3 | 1/25 | 1 | 0/31 | 0 |
| Prompt 3 (*localization*) | 3/31 | 2 | 0/29 | 0 | 3/30 | 1 |
| Prompt 4 (*bug type + localization*) | **6/32** | **4** | 0/28 | 0 | 3/30 | 1 |

**Table 3: Comparison of FixTalk and RQ1 prompts on different defects (X/Y: Plausible Patches/Compilable Patches)**

| Buggy Repos | Mistral-Medium | | | GPT-3.5-Turbo | | | Gemini | | |
|---|---|---|---|---|---|---|---|---|---|
| | P1 | P4 | FixTalk | P1 | P4 | FixTalk | P1 | P4 | FixTalk |
| cpp_peglib-4 | 0/0 | **1/3** | **4/4** | 0/3 | 0/3 | 0/5 | 0/3 | 0/3 | 0/0 |
| exiv2-15 | 0/3 | **1/3** | **2/3** | 1/3 | 3/3 | **5/5** | 1/3 | 0/3 | 0/5 |
| openssl-14 | 0/0 | 0/0 | **0/5** | 0/0 | 0/0 | 0/5 | 0/0 | 0/0 | 0/5 |
| openssl-24 | 0/0 | 0/0 | **0/5** | 0/0 | 0/0 | 0/5 | 0/0 | 0/0 | 0/0 |
| libtiff-1 | 0/3 | 3/3 | 0/3 | 0/3 | 0/3 | 0/4 | 0/2 | 0/1 | 0/2 |
| cppcheck-25 | 0/0 | 0/3 | 0/0 | 0/3 | 0/3 | 0/5 | 0/3 | 0/3 | 0/0 |
| yara-2 | 0/3 | 0/3 | 0/5 | 0/3 | 0/3 | 0/5 | 0/3 | 0/3 | **2/5** |

We hypothesize that the reason Mistral has a significantly lower compilation rate than prompt 4 is because the prompts constructed by our FixTalk algorithm have too much additional information that the model is not able to filter out the relevant necessary details. This is an unexpected anomaly in our experiments particularly because Mistral-Medium has the highest context window amongst the three models.

***Finding 5***. Conversation-driven methodology offers a slight improvement in terms of the number of *plausible patches*, i.e., the number of solutions that can pass all the test cases.

We find that Mistral-Medium is able to provide 4 *different* plausible patches for the cpp-peglib-4 repository, whereas it was able to provide only 1 when using prompts designed for RQ1. Similarly, Gemini can generate a plausible patch for the yara-2 bug, whereas it could not find one with any of our prompt designs for RQ1. GPT 3.5 generates the *same* number of plausible patches when using FixTalk and Prompt 4 from RQ1. Table 3 demonstrates our observations.

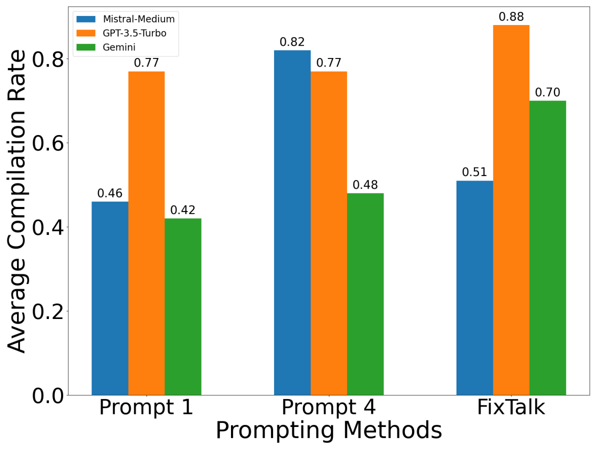


**Figure 4: Bar plot showing compilation rates of different LLMs using P1, P4 and FixTalk prompts**

## 4.3 RQ3: Comparison with GenProg

***Finding 6***. GenProg demonstrates stronger effectiveness in repairing defects across most C projects compared to the three evaluated LLMs.

Table.4 compares the number of plausible patches generated between LLMs and GenProg. For each LLM, we detail the number of plausible patches produced using three distinct prompting strategies: two from our

**Table 4: Plausible patches on three real-world C projects of the BugsC++benchmark**

| | # Defects | SLoC | Mistral-Medium | | | GPT-3.5-Turbo | | | Gemini | | | GenProg |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | P1 | P2 | FixTalk | P1 | P2 | FixTalk | P1 | P2 | FixTalk | |
| berry | 5 | 5,042 | 0 | **1** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| libtiff | 5 | 56,249 | 0 | **1** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **2** |
| libucl | 4 | 6,421 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **2** |

RQ1 study and the FixTalk method implemented in RQ2. For GenProg, we change the random seed number of each experiment starting from 1 to 10 until a repair is found. Based on the results, it is evident that GPT-3.5 and Gemini do not surpass GenProg in effectiveness across all the evaluated C projects. However, Mistral-Medium can exhibit comparable performances compared to GenProg when augmented by bug type annotation and fault localization. Notably, using prompt 4, Mistral successfully fixes a defect in the berry project where GenProg fails to provide any correct fix. Furthermore, our results show that Mistral provides one valid repair for libtiff, of which the source line of code across all bugs is beyond 50k, demonstrating its potential scalability in diagnosing and fixing bugs in large-scale projects.

We speculate that the reasons for these performance disparities are as follows: 1) GenProg can evolve candidate repairs. Its use of genetic programming allows it to explore a vast search space of potential fixes and iteratively refine each candidate based on fitness evaluation on test suites. LLMs inherently don't have the capabilities to validate the generated patch, marking a significant limitation; 2) LLMs face contextual constraints. While LLMs can be augmented with different techniques, they are limited by their contextual comprehension and the scope of their training data. In contrast, GenProg is tailored for repairing C defects, leveraging both syntactic and semantic insights into buggy C code, along with specialized treatments for common error patterns in C. This specialization results in a more accurate and effective search for repairs.

***Finding** 7*. LLMs exhibit significant advantages in terms of usability and versatility compared to GenProg.

GenProg faces notable usability limitations. The normal operation of GenProg depends on the compilation of the target project being compatible with the pre-processed version of the original buggy C code file using C Intermediate Language (CIL) tool. This tool is now abandonware. Consequently, for contemporary projects utilizing C, CIL may fail to produce a compilation-compatible pre-processed version of the buggy file. We observe the shortcomings of GenProg in Table.4: GenProg fails to generate any compilable candidates for any defects in berry even though the original buggy code is written purely in C. Furthermore, the final repaired code produced by GenProg remains in a pre-processed form, posing obstacles for developers to understand the bug and modify the code accordingly.

In addition, LLMs demonstrate substantial versatility advantages. LLMs can produce candidate patches and find potential correct repairs for projects written purely in C++ such as cpp_peglib, cppcheck as shown in Table.3. To the best of our knowledge, no traditional state-of-the-art tool currently exists for C++ repairs, underscoring the unique value of LLMs in this space.

## 4.4 Threats to Validity

**Internal.** In evaluating the validity of our study, we recognize some limitations. Our use of the BugsC++ benchmark does not cover all types of C/C++ bugs, as we mainly focused on single-line errors. This focus might make LLMs like GPT-3.5-Turbo, Gemini, and Mistral appear more effective in fixing bugs than they would be under more complicated repair scenarios. Another internal threats come from the potential data leakage of patched code files in corresponding fixed commits of repositories in BugsC++being part of the training data of the three LLMs.

**External.** Additionally, the rapid evolution in LLM technologies and the specificity of our prompt design could affect the external validity and applicability of our results. However, our findings may still not generalize to other C/C++ datasets.

## 5 Discussion and Conclusion

In this work, we conduct a comprehensive study on the APR capabilities of three general-purpose LLMs, Mistral-Medium, GPT-3.5-Turbo, and Gemini, across real-world C/C++ defects. Our key findings addressing our three research questions, reveal that: 1) While all three LLMs are capable of generating compilable patches for a majority of the projects in the BugsC++benchmark, they often fall short in producing a sufficient number of plausible patches. Mistral-Medium is the best model among the three and exhibits significant performance using prompt augmentation; 2) Our dialogue-based algorithm FixTalk demonstrates that the benefit of providing LLMs with compilation error feedback, thereby enhancing the rate of compilable patch generation and encouraging the production of varied plausible patches once an initial plausible patch is identified; 3) GenProg maintains superior effectiveness in repairing real-world C program defects but LLMs have better usabilities and adaptabilites in C/C++ APR along with substantial further improvement room.

APR for C/C++ remains a longstanding issue in Software Engineering (SE). Our study evaluates the three general-purpose LLMs on a C/C++ defect benchmark using various prompting strategies and compares LLMs' performance against a state-of-the-art traditional tool, presenting practical insights into optimizing prompt designs for leveraging LLMs' repair capabilities. Furthermore, our results on BugsC++benchmark can serve as a baseline for future evaluation of other LLMs'APR performance on this benchmark.

Our study, however, is limited by its focus on only a subset of defects within the BugsC++benchmark, leaving the repair potential of the LLMs under various scenarios unexplored. So one future work is to continue experimenting with the remaining defects in BugsC++to obtain a comprehensive understanding of the repair capabilities of three LLMs. Meanwhile, we aim to explore more advanced prompting methods to improve LLM performance for C/C++ APR, such as incorporating few-shot examples, syntactic hierarchies information [3], project-specific knowledge [2]. etc. Another limitation is that our research has not addressed the verification of correct patches, leaving the true accuracy of the plausible patches generated by LLMs for C/C++ defects unknown. The investigation of the overfitting problem in plausible patches generated by LLMs can be another interesting direction for future work.

## 6 Team Membership and Attestation

Johnson Hu led the research and team organization, implementing the Mistral model's calling and interaction logic and creating a unified testing script for various models. He also took part in designing the prompts, conducting the RQ1 and RQ2 experiments with Mistral, assessing GenProg's performance in RQ3, compiling statistical data for evaluation. He wrote several sections in this final report focusing on RQ2 and RQ3 (Sections 3.3 3.4, 4.3, 5) and proofread it.

Musheng He integrated the *BugsC++* benchmark into the project and developed the Data module to provide multiple APIs for LLMs to access buggy code snippets, validate LLM's repair code, and provide detailed experimental data. He participated in the RQ1 experiment using Mistral and significantly contributed to both the *Data* and *Related Work* sections of the report.

Srikkanth researched on the Gemini model and actively participated in designing prompts with extra context. He also carried out his portion of the RQ1 and RQ2 experiments on Gemini and documented the results. He also helped with writing sections of the report (Sections 1.1-1.3, RQ1/2 study) and proofreading.

Harshil Bhandari facilitated the necessary exploration of LLMs. He then set up the calling logic for the GPT-3.5-Turbo model and conducted RQ1 experiments, adding his findings to the report and assisting in editing. He updated the unified scripts for GPT-3.5-Turbo and ran experiments for RQ2 and additional ones for RQ1. He also researched GPT's maintenance of chat sessions.

Rishika Garg assisted actively with experiment design and research on LLMs, aiding in the final model selection. She also helped conduct experiments for RQ1 with GPT-3.5-Turbo and contributed in making the progress report. For the final report, she completed additional RQ1 experiments using Mistral and supported with research on LLMs' context-retention abilities along with an analysis of Mistral's different endpoints.

## References

[1] An, G., Kwon, M., Choi, K., Yi, J., and Yoo, S. Bugsc++: A highly usable real world defect benchmark for c/c++. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2023), pp. 2034–2037.

[2] Barr, E. T., Brun, Y., Devanbu, P., Harman, M., and Sarro, F. The plastic surgery hypothesis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (New York, NY, USA, 2014), FSE 2014, Association for Computing Machinery, p. 306–317.

[3] Clement, C. B., Lu, S., Liu, X., Tufano, M., Drain, D., Duan, N., Sundaresan, N., and Svyatkovskiy, A. Long-range modeling of source code files with ewash: Extended window access by syntax hierarchy. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021* (2021), M. Moens, X. Huang, L. Specia, and S. W. Yih, Eds., Association for Computational Linguistics, pp. 4713–4722.

[4] Durieux, T., Madeiral, F., Martinez, M., and Abreu, R. Empirical review of java program repair tools: A large-scale experiment on 2,141 bugs and 23,551 repair attempts. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2019), pp. 302–313.

[5] Gupta, R., Pal, S., Kanade, A., and Shevade, S. Deepfix: Fixing common c language errors by deep learning. *Proceedings of the AAAI Conference on Artificial Intelligence 31*, 1 (Feb. 2017).

[6] Hu, J., He, M., Ramachandran, S., Garg, R., and Bhandari, H. An evalutation of llms as apr tools for c/c++ programs. https://github.com/ECS260-WQ2024-Group3/llmAPRCpp, 2024.

[7] Huang, K., Meng, X., Zhang, J., Liu, Y., Wang, W., Li, S., and Zhang, Y. An empirical study on fine-tuning large language models of code for automated program repair. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2023), pp. 1162–1174.

[8] Jin, M., Shahriar, S., Tufano, M., Shi, X., Lu, S., Sundaresan, N., and Svyatkovskiy, A. Inferfix: End-to-end program repair with llms. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (New York, NY, USA, 2023), ESEC/FSE 2023, Association for Computing Machinery, p. 1646–1656.

[9] Just, R., Jalali, D., and Ernst, M. D. 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* (New York, NY, USA, 2014), ISSTA 2014, Association for Computing Machinery, p. 437–440.

[10] Kim, M., Kim, Y., Jeong, H., Heo, J., Kim, S., Chung, H., and Lee, E. An empirical study of deep transfer learning-based program repair for kotlin projects. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (New York, NY, USA, 2022), ESEC/FSE 2022, Association for Computing Machinery, p. 1441–1452.

[11] Kulal, S., Pasupat, P., Chandra, K., Lee, M., Padon, O., Aiken, A., and Liang, P. S. Spoc: Search-based pseudocode to code. In *Advances in Neural Information Processing Systems* (2019), H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32, Curran Associates, Inc.

[12] Le Goues, C., Holtschulte, N., Smith, E. K., Brun, Y., Devanbu, P., Forrest, S., and Weimer, W. The manybugs and introclass benchmarks for automated repair of c programs. *IEEE Transactions on Software Engineering 41*, 12 (2015), 1236–1256.

[13] Le Goues, C., Nguyen, T., Forrest, S., and Weimer, W. Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering 38*, 1 (2012), 54–72.

[14] Lin, D., Koppel, J., Chen, A., and Solar-Lezama, A. Quixbugs: a multi-lingual program repair benchmark set based on the quixey challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity* (New York, NY, USA, 2017), SPLASH Companion 2017, Association for Computing Machinery, p. 55–56.

[15] Monperrus, M. Automatic software repair: A bibliography. *ACM Computing Surveys (CSUR) 51*, 1 (2018), 1–24.

[16] Xia, C. S., Wei, Y., and Zhang, L. Automated program repair in the era of large pre-trained language models. In *Proceedings of the 45th International Conference on Software Engineering* (2023), ICSE '23, IEEE Press, p. 1482–1494.

[17] Xia, C. S., and Zhang, L. Keep the conversation going: Fixing 162 out of 337 bugs for $0.42 each using chatgpt. *CoRR abs/2304.00385* (2023).

[18] Zhang, Q., Fang, C., Ma, Y., Sun, W., and Chen, Z. A survey of learning-based automated program repair. *ACM Trans. Softw. Eng. Methodol. 33*, 2 (dec 2023).

# Appendix

## Table 5: Comparsion of 3 LLM's statistics using Prompt 4 on repairing buggy repositories in RS1

| | #Test | Org. PR% | Mistral Medium | | | | | Gemini | | | | | GPT-3.5-Turbo | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | #Patch | Avg. PR% | Identify% | Avg. Confidence | Response(s) | #Patch | Avg. PR% | Identify% | Confidence | Response(s) | #Patch | Avg. PR% | Identify% | Confidence | Response(s) |
| berry-5 | 21 | 95.2 | 0/2 | 61.9 | 33.3 | 0.93 | 19.56 | 0/2 | 49.2 | 0 | 0.9 | 6.17 | 0/0 | 0 | 0 | 0 | 2.79 |
| cpp_peglib-4 | 94 | 98.9 | **1/3** | **99.3** | 33.3 | 0.95 | 14.08 | 0/3 | 33.0 | 66.7 | 0.93 | 5.58 | 0/3 | 98.9 | 33.3 | 0.92 | 3.81 |
| cppcheck-8 | 84 | 98.8 | 0/3 | 65.5 | 0 | 0.93 | 15.43 | 0/3 | 63.1 | 0 | 0.93 | 9.68 | 0/3 | 32.9 | 0 | 0.32 | 4.76 |
| exiv2-15 | 4 | 75.0 | 1/3 | 33.3 | 0 | 0.93 | 71.75 | 0/3 | 25.0 | 0 | 0.8 | 22.00 | **3/3** | **100** | 0 | 0.88 | 18.98 |
| libtiff-1 | 82 | 98.8 | **3/3** | **100** | 0 | 0.93 | 52.53 | 1/0 | 24.4 | 0 | 1 | 27.24 | 0/3 | 98.8 | 0 | 0.82 | 29.07 |
| yara-1 | 201 | 99.5 | 1/3 | 71.5 | 0 | 0.93 | 29.75 | 0/3 | 99.5 | 0 | 0.93 | 13.58 | 0/3 | 99.5 | 0 | 0.9 | 14.79 |