Autorepairability of ChatGPT and Gemini: A Comparative Study

Chutweeraya Sriwilailak*, Yoshiki Higo†, Pongpop Lapvikai*, Chaiyong Ragkhitwetsagul* and Morakot Choetkiertikul*
*Faculty of Information and Communication Technology (ICT), Mahidol University, Nakhon Pathom, Thailand
†Graduate School of Information Science and Technology, Osaka University, Osaka, Japan

***Abstract*—In recent years, Automated Program Repair (APR), which focuses on automatically fixing source code without human intervention, has become a hot topic in the field of software engineering, leading to the proposal of various automatic repair techniques. Additionally, Lapvikai et al. introduced a new software quality metric called "Autorepairability." Autorepairability is a metric that indicates how easily bugs in the target source code can be fixed using APR techniques. By utilizing Autorepairability, it becomes possible to pre-check whether the program repair techniques will work effectively on the target software and to perform refactoring to improve Autorepairability. However, in the past two to three years, program repair using large language models (LLMs) has become more prevalent, and several studies have revealed that these models exhibit superior repair capabilities compared to traditional APR techniques. In this study, we applied Autorepairability to compare the performance of multiple APR techniques. Specifically, we measured and compared Autorepairability using ChatGPT and Gemini, which are representative large language models, as well as kGenProg, a traditional APR technique. The results demonstrated that Gemini exhibited higher repair capabilities compared to both ChatGPT and the traditional APR technique kGenProg. The five code functionalities that Gemini offers higher Autorepairability scores than ChatGPT include (1) geographic and mathematic operations, (2) validation, comparison, and searching operations, (3) data conversion operations, (4) data extraction and comparison operations, and (5) encoding operations.**

***Index Terms*—Automated program repair, Large language models, ChatGPT, Gemini**

I. Introduction

Large Language Models (LLMs) represent an advanced form of artificial intelligence (AI) that leverages machine learning techniques to emulate human language. These models are trained on vast datasets, enabling them to excel in tasks such as language translation, text prediction, and content generation. Unlike traditional Natural Language Processing (NLP) models, LLMs are characterized by their ability to process much larger datasets and utilize a significantly higher number of parameters. This increased complexity allows LLMs to produce language outputs that are more sophisticated and closely aligned with human communication. A recent survey by Fan et al. [1] shows that LLMs can be effectively applied to several software engineering tasks.

Automated program repair (in short, APR) is an automatic technique to fix bugs without human intervention. Traditional APR methods, often constrained by predefined patterns, struggle to adapt to the diverse and complex errors that arise in real-world software environments [2]. In contrast, LLMs, when utilized as APR tools, possess the capability to dynamically learn from and adjust to a wide array of programming errors. This enables them to generate more precise and contextually aware fixes, significantly improving the reliability of the repair process. Moreover, this approach minimizes the need for human intervention, making APR tasks more scalable and effective in addressing the intricate challenges posed by modern software development.

The previous study by Lapvikai et al. [3] introduces the concept of "Autorepairability", a new software quality characteristic that measures the effectiveness of techniques for specific code fragments, files, or projects (more detail in Section II-A). The methodology involved generating artificial bugs through mutation testing and applying any APR tools to repair them. Their experiment conducted on 1,282 functionally equivalent Java method pairs using kGenProg [4], an APR tool based on genetic algorithm, found that the tool offers a decent Autorepairability score of 0.45 on average. They also identified four key code structures that significantly impact Autorepairability at the syntactic level including curly braces surrounding code block, usage of the ternary operator, combined logical expressions in a conditional statement, and type of conditional statements.

In the past two to three years, program repair using large language models has become increasingly prevalent, and several studies have demonstrated that their repair capabilities surpass those of traditional APR techniques [5]–[7]. Nonetheless, there are no studies regarding the Autorepairability scores of LLMs in the literature. This study aims to fill the gap by performing a preliminary study of applying two LLMs, ChatGPT and Gemini, as APR tools and measuring their Autorepairability. The experiment focuses on comparing the performance of two popular and accessible LLMs: ChatGPT-3.5 Turbo, the most cost-effective paid option, and the Gemini-1.5-Flash, the free version of Gemini, across various functionalities. Understanding which model excels in different functionalities is essential for enabling users to make informed decisions about which LLM to use, balancing performance with budgetary constraints. This comparison seeks to provide valuable insights into the trade-offs between model capabilities and financial considerations, ultimately guiding the selection of the most suitable LLM for widespread deployment in cost-sensitive environments.

The results from our study show that Gemini offers higher Autorepairability scores than ChatGPT and kGenProg, showing its promising future as an APR tool. Additionally, we study the code functionalities that have high Autorepairability scores by LLMs, which offers insights into the code semantics that can be effectively repaired by LLMs. By closely examining how these models handle different types of programming errors and semantics, this research highlights the practical implications of their use, offering recommendations for optimizing their deployment in real-world software development environments. We believe that this study paves the way for future studies in using LLMs for automated program repairs.

II. Background and related work

A. Measuring Autorepairability

Autorepairability is a software quality characteristic that assesses how well APR techniques can fix bugs in a specific project or codebase. It reflects the ease with which APR tools can identify and correct errors, making it a metric for determining whether a software system is suitable for automated repairs. By measuring Autorepairability, developers can gauge the effectiveness of applying APR techniques to their projects, helping to ensure more efficient and reliable software maintenance.

According to Lapvikai et al. [3], the process starts with generating artificial bugs for the given project and measuring how well an APR technique can fix such artificial bugs. Mutation testing techniques are employed to generate artificial bugs [8], [9]. The process inputs include (1) source code S_P and test cases T_P of a target project P, (2) a mutation testing technique MU, and (3) an APR tool A.

To measure Autorepairability, the following three steps need to be performed. **Step-1:** One generates mutants of source code S_P, that come with a set of unit test cases T_P, by using a mutation testing technique MU. Each mutant $\mu \in M$ is a source code that is slightly different from the original source code. **Step-2:** The technique applies an APR tool being assessed A to each of the generated mutants. Each mutant μ includes a different bug similar to mutation testing. The generated fix, i.e., a patch, from A that passes all the tests in T_P is called a solution s. Set S represents all the solutions that A can generate over all the mutants in M. A is not applied to the mutant if a mutant passes all test cases in T_P. **Step-3:** One calculates the ratio of the number of generated solutions $|S|$ per the number of mutants $|M|$, i.e., the Autorepairability score. The calculation of Autorepairability is shown in Equation II-A below.

$$\text{Autorepairability} = \frac{|S|}{|M|} \tag{1}$$

One can use this score to compare how easily different programs are to be repaired by multiple APR tools.

B. Related work

There are a few recent studies on using LLMs for APR. Sobania et al. evaluated ChatGPT's bug-fixing capabilities using the QuixBugs benchmark, which includes challenging Python programs [6]. They repeatedly used the same prompt for each problem to assess the correctness of ChatGPT's fixes, comparing its performance to traditional APR methods and deep learning models like Codex and CoCoNut. The study also explored how providing contextual hints in dialogue could improve ChatGPT's success, demonstrating its potential as an automated program repair tool.

A study by Xia et al. [7] explores the use of large language models (LLMs), like Codex, to enhance automated program repair. The study evaluates how effectively LLMs can generate, repair, and refine code snippets based on natural language descriptions and existing code structures. Results show that LLMs can produce a variety of useful patches, though they sometimes struggle with complex contexts and perfect syntax. Despite these limitations, the research demonstrates the potential of LLMs in improving early-stage code repair, offering valuable suggestions for developers to refine.

Although these studies evaluate and give early insights into the effectiveness of using LLMs for APR, they do not compare different LLMs on the APR tasks.

III. Methodology

In this study, we aim to answer two research questions.

RQ1: What are the Autorepairability scores of ChatGPT and Gemini? We aim to understand the Autorepairability of Java programs using LLMs. So, we employed the two widely used LLMs, ChatGPT and Gemini for automated program repair tasks and measured the Autorepairability score. We also compare the Autorepairability of the two LLMs.

RQ2: What are the functionalities that affect the Autorepairability of the two LLMs? To understand the functionalities that are suitable for repairs by each LLM. We manually classified the code pairs that had strong differences in Autorepairability scores between ChatGPT and Gemini. This will provide more insights into the differences at the semantic level that affect using LLMs as APR tools.

A. Dataset

Similar to the study by Lapvikai et al. [3], the dataset used in this research is based on Higo et al.'s dataset of functionally equivalent Java methods [10]. We used 1,282 Java method pairs from this dataset. Each pair, containing two Java methods, performs the same functional task but differs in implementation style, reflecting various approaches or methodologies to achieve the same outcome. Additionally, the dataset includes multiple variants, or mutants, of each method, with each mutant containing different bugs. Lapvikai et al. used PIT [9] mutation testing techniques to create artificial bugs, which we refer to as mutants. We reused them. Test case files accompanying these mutants which come with the Higo et al.'s dataset were generated by EvoSuite [11]. The tool automatically generates test cases with assertions

for classes written, which are used to evaluate the software's performance, verify its correct operation, and identify any errors that need to be addressed.

These components in the dataset are used to assess the automated program repair capabilities of the Large Language Models (LLMs) in the study.

B. Experimental Procedure

In this experiment, two advanced LLMs were selected for evaluation: Gemini-1.5-Flash (free version) and ChatGPT-3.5 Turbo (the most cost-effective paid version). We decided to use these versions of the two LLMs because of two reasons. First, it enables the reproducibility of the study. Second, it is also possibly the versions that the developers are mostly affordable. The overall framework is illustrated in Figure 1. The experiment involves four main steps to evaluate the Autorepairability performance of the two LLMs. We explain each step in detail below.

1) Step 1: Prompt Engineering: We issue prompts based on the same prompt template across both models to assess their performance. These interactions were facilitated via the models' API connections, ensuring a consistent environment for comparing the models' outputs in terms of response quality, accuracy, and overall efficiency. The prompt template is identical for each model, except the code to be repaired and the test cases, to ensure consistency in the comparison. The template of the prompt that we used is shown in Figure 2. The prompt includes Java code that has failed certain test cases, along with its corresponding unit test cases. The models are instructed to fix the provided Java code to ensure it passes all the test cases. The LLMs were instructed to provide only the repaired Java code back as their output without any explanations for the ease of result analysis.

2) Step 2: Checking Correctness of the Repaired Code: In this step, the repaired code generated by the LLM models was executed against the provided test cases to evaluate the effectiveness of the fixes. The primary objective was to determine whether the modifications made by the models successfully addressed the issues present in the original code. The repaired code was run using the same set of test cases that initially exposed the bugs. If the repaired code passed all the test cases, it indicated that the models had resolved the issues, thereby successfully repairing the code, i.e., creating a solution. Conversely, if any of the test cases still failed, it suggested that the code had not been fully repaired, and the models were unable to completely fix the underlying problems.

3) Step 3: Autorepairability Score Calculation: In this step, the Autorepairability score was calculated based on the Equation II-A. According to this methodology, the Autorepairability score is determined by dividing the number of successfully generated solutions, where the repaired code passes all test cases, by the total number of mutants in the dataset. This approach provides a quantitative measure of how effectively the models can generate correct solutions in response to the given prompts, offering a clear indicator of the models' capability to repair code automatically.

TABLE I
COMPARISON OF MAXIMUM, MINIMUM, AND AVERAGE VALUES OF AUTOREPAIRABILITY

Value	KGenProg	ChatGPT	Gemini
Max	1.00	1.00	1.00
Min	0.00	0.00	0.00
Median	-	0.43	0.77
Standard Deviation	-	0.30	0.31
Average	0.45	0.44	0.69

4) Step 4: Manual Inspection of Java Methods: we performed a manual investigation of the original Java methods used to generate the mutants. To do the manual investigation, the first author, who has 4 years of experience in writing Java programs, carefully read the Java method to derive their functionality at a semantic level to identify which code functionalities are successfully repaired by LLM-based methods. We limited our manual analysis to only the code pairs that had large differences in terms of the Autorepairability scores between ChatGPT and Gemini, meaning that one LLM is better than the other in repairing such functionality. We filtered the methods to be looked at by finding the differences in Autorepairability scores of each method between Gemini and ChatGPT, i.e., $|\text{Autorepairability}_{\text{Gemini}} - \text{Autorepairability}_{\text{ChatGPT}}|$, were computed and ranked. Then, the methods with a score difference greater than 0.5 were selected, indicating a significant discrepancy in Autorepairability between Gemini and ChatGPT. Moreover, only those methods having more than 10 mutants were included in the analysis to make sure that the model sufficiently fixed the bugs in the code pairs.

IV. RESULTS

A. Answering RQ1

After completing the analysis of both LLMs' repaired code snippets, we calculated the Autorepairability scores of the two models as summarized in Table 1. We also include the results from Lapvikai et al.'s study [3] using kGenProg for a more comprehensive comparison with a traditional APR tool. The highest Autorepairability scores for both ChatGPT and Gemini are 1.00, indicating that in some cases, the models were able to fully repair the code, while the lowest scores are zero, showing instances where the models failed to repair the code at all. This is also similar to the results from kGenProg.

The average Autorepairability score for ChatGPT is 0.44, meaning it successfully resolved about slightly half of the mutants. Interestingly, the average score for Gemini is around 0.69, indicating that it was able to fix more bugs than ChatGPT. We can also see that ChatGPT did not outperform kGenProg by having a slightly lower Autorepairability score of 0.44 (ChatGPT) compared to 0.45 (kGenProg) respectively. The median scores also show the same observation with similar standard deviations of the Autorepairability scores of the two LLMs.

We also looked at the differences in Autorepairability scores of the two methods that form a pair in the dataset. The two

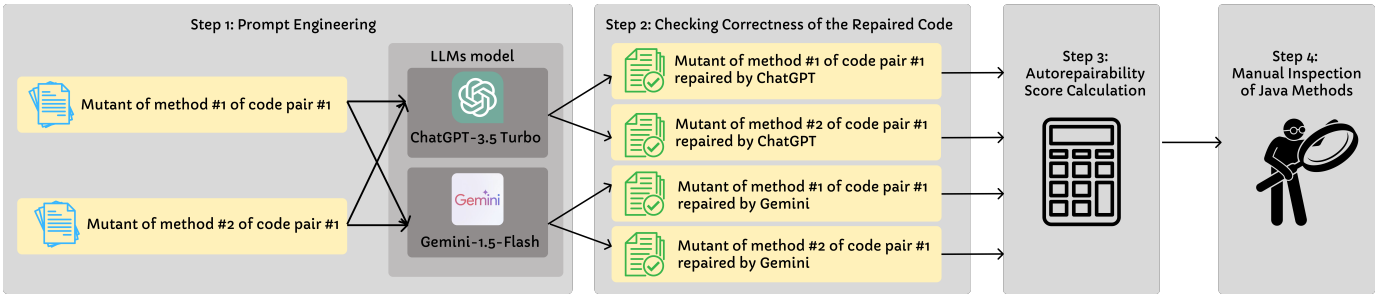

Fig. 1. The Experiment Framework

[Code of the Java method to be repaired]
[Unit test cases of the method]

From the Java code above, this code fail on some test case. Please update the code to make it run pass all the test case. Respond only with the updated Java code (do not include the test code) in this format:

```java
Repaired code
```

Fig. 2. The prompt template used in this study

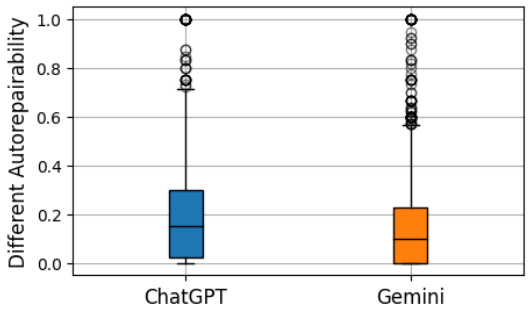

Fig. 3. Differences of Autorepairability Scores of Method Pairs

methods in the same pair should provide the same functionality. Thus, we can see how effective are the two LLMs at fixing bugs in the code snippets with the same functionality but different implementations. In this case, the lower the difference, the better. The result from 1,282 method pairs is shown as a boxplot in Figure 3. We can see that Gemini has lower differences in Autorepairability scores compared to ChatGPT, i.e., median scores of 0.10 compared to 0.15 respectively. This means it can fix a higher number of bugs in code with the same functionality but different implementations.

Therefore, to answer the first research question, Our study demonstrates that, using the dataset of functionally equivalent Java methods, Gemini outperforms ChatGPT and the traditional APR tool kGenProg in terms of Autorepairability with an average score of 0.69 compared to 0.44 by ChatGPT and 0.45 by kGenProg. Moreover, Gemini also outperforms ChatGPT in fixing bugs in the code with the same functionality but different implementations.

TABLE II
AUTOREPAIRABILITY SCORE OF EACH FUNCTIONALITY

Functionality	#Methods	Gemini	ChatGPT
Validation, Comparison, & Searching	66	0.92	0.22
Data Extraction & Comparison	34	0.89	0.25
Data Conversion	20	0.90	0.25
Encoding	7	0.89	0.16
Geographic & Math	3	0.85	0.27

B. Answering RQ2

Our manual analysis revealed that Gemini had higher Autorepairability scores than ChatGPT in 130 methods, while ChatGPT had a higher score in only 2 methods (i.e., Autorepairability score difference of the two LLMs on the same method is more than 0.5). After reading and understanding each method at the semantic level, we discovered five common coding functionalities that create large differences in the Autorepairability scores of ChatGPT and Gemini.

1. Geographic and Mathematical Operations: This functionality group encompasses methods primarily focused on geographic calculations and mathematical operations. These methods are often utilized in tasks requiring precision in mapping or spatial analysis, such as determining the correct zone or calculating differences between geographical points.

2. Validation, Comparison, and Searching Operations: This functionality ensures data integrity by validating formats, comparing values (e.g., greater than, less than, equal to), and searching data structures for specific elements. It is essential to verify data before processing to ensure accuracy.

3. Data Conversion Operations: This functionality focuses on converting data between formats, such as transforming bytes into integers. It is crucial in environments where binary data needs to be interpreted or processed in human-readable formats, enabling advanced data manipulation.

4. Data Extraction and Comparison Operations: This functionality involves extracting key information from data structures and comparing elements. It is vital for workflows requiring precise data manipulation and integrity checks.

5. Encoding Operations: This functionality handles data encoding by converting numerical values or bit sequences into

TABLE III
EXAMPLE JAVA METHODS WITH A LARGE DIFFERENCE IN AUTOREPAIRABILITY SCORES BY CHATGPT (C) AND GEMINI (G). CA AND GA STAND FOR CHATGPT'S AND GEMINI'S AUTOREPAIRABILITY SCORES RESPECTIVELY

Method Name	Mutant	Gemini's Solutions	ChatGPT's Solutions	GA	CA	Difference (GA − CA)
getUTMLatitudeZoneLetter	99	97	41	0.98	0.41	0.57
isFormatValid	12	10	1	0.83	0.08	0.75
byte2int	23	23	10	1.00	0.43	0.57
getKeyValues	11	9	1	0.82	0.09	0.73
getBase64Char	12	10	3	0.83	0.25	0.58

character representations using Base64 encoding. It is critical for data serialization, secure transmission, and storage.

Table II shows the average Autorepairability scores of ChatGPT and Gemini across the five functionalities. We can see that the largest group of methods is *Validation, Comparison, and Searching operations* with 66 methods, followed by *Data Extraction and Comparison operations* (34), and *Data Conversion operations* (20). Among these five functionalities, the average Autorepairability scores of Gemini are higher than ChatGPT ranging from 0.85 to 0.92, compared to 0.16 to 0.27.

Additionally, we show examples of the manual analysis results in Table III. We can see that for the 5 methods that perform the five identified functionalities, `getUTMLatitudeZoneLetter` (Geographic and Mathematical Operations), `isFormatValid` (Validation, Comparison, and Searching Operations), `byte2int` (Data Conversion Operations), `getKeyValues` (Data Extraction and Comparison), `getBase64Char` (Encoding Operations), Gemini offers a large gap of Autorepairability scores compared to ChatGPT in these five methods. On the other hand, ChatGPT demonstrated superior Autorepairability in only two methods with Data Conversion operations involving bytes with slight variations. The methods where ChatGPT excelled involved converting a string, particularly an IP address, into bytes.

To answer RQ2, based on the investigation, we found five functionalities that are better repaired using Gemini compared to ChatGPT. These functionalities include Geographic and Mathematical Operations, Validation, Comparison, and Searching Operations, and Data Conversion Operations.

V. THREATS TO VALIDITY

Internal Validity: The manual investigation of functionality, conducted by a single individual, introduces potential sources of human error, experiences, and biases which can affect the consistency and reliability of the findings. Additionally, the prompts used to interact with LLMs play a critical role in shaping the responses and may affect the validity of the results. **External Validity:** The results presented in this paper are only limited to the Higo et al.'s dataset [12] and may not be generalized to other datasets. Second, LLMs like ChatGPT and Gemini exist in different versions, each with its own set of capabilities. Consequently, the results obtained in this study may not be entirely representative of other LLMs or other versions of Gemini and ChatGPT.

VI. CONCLUSION AND FUTURE WORK

In this paper, we study the Autorepairability of two widely used LLMs, ChatGPT and Gemini, on a dataset of 1,282 functionality-equivalent Java methods. We found that Gemini outperforms ChatGPT in fixing bugs based on the generated mutants. We also identified five functionalities for which Gemini largely outperforms ChatGPT. For future work, we plan to strengthen the study by (1) including more LLMs such as Llama, Claude, and other open-source LLMs, and (2) repeating the experiment with more datasets, especially real-world software projects, to increase the generalizability of the findings and applicability of the Autorepairability metric in practical scenarios, (3) performing a comparative analysis of Autorepairabitliy scores across different datasets which could reveal trends and insights that are not apparent in a single dataset. We believe that the early results shown in this paper are beneficial for future studies on using LLMs as APR tools.

REFERENCES

[1] A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta, S. Yoo, and J. M. Zhang, "Large language models for software engineering: Survey and open problems," in *ICSE-FoSE'23*, 2023, pp. 31–53.

[2] L. Gazzola, D. Micucci, and L. Mariani, "Automatic software repair: A survey," *IEEE Transactions on Software Engineering*, vol. 45, no. 1, pp. 34–67, 2019.

[3] P. Lapvikai, C. Ragkhitwetsagul, M. Choetkiertikul, and Y. Higo, "Autorepairability: A new software quality characteristic," in *SANER'24*, 3 2024, pp. 787–791.

[4] Y. Higo, S. Matsumoto, R. Arima, A. Tanikado, K. Naitou, J. Matsumoto, Y. Tomida, and S. Kusumoto, "kGenProg: A High-Performance, High-Extensibility and High-Portability APR System," in *APSEC'18*, 2018, pp. 697–698.

[5] M. Jin, S. Shahriar, M. Tufano, X. Shi, S. Lu, N. Sundaresan, and A. Svyatkovskiy, "Inferfix: End-to-end program repair with llms," in *ESEC/FSE'23*, 2023, pp. 1646—-1656.

[6] D. Sobania, M. Briesch, C. Hanna, and J. Petke, "An analysis of the automatic bug fixing performance of ChatGPT," in *APR'23*. IEEE, 2023, pp. 23–30.

[7] C. S. Xia, Y. Wei, and L. Zhang, "Automated program repair in the era of large pre-trained language models," in *ICSE'23*, 2023, pp. 1482–1494.

[8] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, 2011.

[9] "Available mutators and groups in PIT," https://pitest.org/quickstart/mutators/.

[10] Y. Higo, "Dataset of functionally equivalent java methods and its application to evaluating clone detection tools," *IEICE Transactions on Information and Systems*, vol. E107.D, no. 6, pp. 751–760, 2024.

[11] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *ESEC/FSE'11*, 2011, pp. 416–419.

[12] Y. Higo, S. Matsumoto, S. Kusumoto, and K. Yasuda, "Constructing dataset of functionally equivalent java methods using automated test generation techniques," in *MSR'22*, ser. MSR '22, 2022, p. 682–686.