UTILIZING LARGE LANGUAGE MODELS FOR AUTOMATED BUG RESOLUTION

Ishwarbhai S. Desai [1], Dr. Ushaben L. Barad [2]

[1]PG Scholar, 2Assistant Professor

Computer Engineering Department, Hansaba College of Engineering & Technology

Gokul Global University, Siddhpur, Gujarat, India

***Abstract*—Bug fixing, which is known as Automatic Program Repair (APR), is a significant area of research in the software engineering field. It aims to develop techniques and algorithms to automatically fix bugs and generate fixing patches in the source code. Researchers focus on developing many APR algorithms to enhance software reliability and increase the productivity of developers. In this paper, a novel model for automated bug fixing has been developed leveraging large language models. The proposed model accepts the bug type and the buggy method as inputs and outputs the repaired version of the method. The model can localize the buggy lines, debug the source code, generate the correct patches, and insert them in the correct locations. To evaluate the proposed model, a new dataset which contains 53 Java source code files from four bug classes which are Program Anomaly, GUI, Test-Code and Performance has been presented. The proposed model successfully fixed 49 out of 53 codes using gpt-3.5-turbo and all 53 using gpt-4-0125-preview. The results are notable, with the model achieving accuracies of 92.45% and 100% with gpt-3.5-turbo and gpt-4-0125-preview, respectively. Additionally, the proposed model outperforms several state-of-the-art APR models as it fixes all 40 buggy programs in QuixBugs benchmark dataset.**

Keywords—Bug fixing; automated program repair; large language models; software debugging; software maintenance; machine learning

I. INTRODUCTION

Automated Program Repair (APR) is considered one of the most ideal tasks in automated software engineering, with the potential to reduce the costs associated with software development and maintenance [32]. APR refers to the automated fixing of bugs or defects in the source code by a software tool [31]. It aims to minimize human intervention in the debugging process through the development and implementation of intelligent algorithms capable of automatically detecting and fixing errors in the source code.

Historically, APR techniques have relied on a variety of approaches, ranging from genetic algorithms [12-14] to symbolic execution [37], each with its strengths and limitations. However, the advent of Large Language Models (LLMs) in artificial intelligence has opened new directions for research and application in software engineering tasks [18, 33, 34], offering promising prospects for enhancing the accuracy, efficiency, and scope of automated bug fixes.

This paper introduces a novel model for APR that leverages the capabilities of two state-of-the-art LLMs, gpt-3.5-turbo and gpt-4-0125-preview, to automate the bug-fixing process across various bug types. The proposed model distinguishes itself by not only localizing bugs with high accuracy but also generating and inserting the correct patches into the source code. Accordingly, it significantly reduces the time and resources traditionally required for bug fixing, thereby accelerating the software development lifecycle, and enhancing developer productivity.

To validate the effectiveness of the proposed model, a new dataset consisting of 53 Java source code files, categorized into four distinct bug classes: Program Anomaly, GUI, Test-Code, and Performance was constructed. The performance of the proposed model was evaluated against this dataset to reveal its capability in bugs repair, with results indicating an impressive accuracy rate. These findings not only support the potential of integrating LLMs into the APR process but also open new directions for future research in the field.

The proposed model distinguishes itself from other APR models due to its ability to successfully debug code, localize buggy lines, generate correct patches, and insert them in the appropriate locations. It achieves this by only requiring the buggy code and the bug type, without needing additional user input, information about test cases, or prior knowledge of patch attempts. This makes it a more efficient, practical, and novel model.

The main contributions of this paper are as follows:

- Presenting a novelty in the use of gpt-4-0125-preview for APR: To the best of our knowledge, we present the first-of-its-kind model that leverages gpt-4-0125-preview for automated program repair, enabling the repair of multi-hunk and multi-fault bugs simultaneously. This significantly extends the capabilities of current APR models.
- Presenting a self-contained repair mechanism: Our approach does not rely on external test cases, prior patch knowledge, or feedback loops. Instead, it operates only with the buggy method and bug type, making it more efficient and less resource-intensive than traditional APR techniques.

- Illustrating an advanced fault localization: The proposed model identifies and fix buggy lines of code without requiring identification of a statement-level bug localization, reducing the manual effort typically needed in bug fixing.
- Building a new dataset: A new dataset comprising 53 Java programs across four bug categories: Program Anomaly, GUI, Test-Code, and Performance has been generated in this research.
- Presenting a comparison with state-of-the-art models: This paper has demonstrated that the proposed model not only achieves a higher bug fix rate than state-of-the-art APR models, but also it achieved this with fewer dependencies on external tools and inputs, making it a more scalable and practical solution for real-world use.

The rest of this paper is structured as follows: Section II outlines the motivation behind this research; Section III reviews existing literature on automated bug fixing algorithms; Section IV describes the proposed model; Section V details the experimental study; Section VI presents the experimental results; Section VII discusses the implications of these results; and Section VIII concludes the paper.

II. MOTIVATION

To fix a bug in a software program, the location of the bug must first be identified. This includes the buggy class, the buggy method, and the buggy statement. Our previously proposed bug localization model [22] employs an information-retrieval-based approach to identify the buggy class and method within the class; however, it does not localize the buggy statement. On the other hand, spectrum-based fault localization (SBFL) can identify more precise locations, such as the buggy statement [19]. However, it requires a large number of passing and failing test cases with test oracles, which poses some limitations [19]. Furthermore, the performance of APR is influenced more by the quality of test cases than their quantity [19]. LLMs have shown significant improvements in software engineering, demonstrating exceptional performance in tasks such as code and document generation [6]. Consequently, this research proposes an automated bug fixing model that leverages a specific model of large language models, the gpt-4-0125-preview model. The main novelty in the proposed approach that it does not require test cases or oracles, nor does it require prior statement-level bug localization. It utilizes our previously proposed bug prediction model [23] to detect the bug type, identifies the buggy method via our localization model [22], and outputs the fixed version of the method, enhancing the accuracy of existing bug fixing models. The overall bug management system is presented in Fig. 1.

III. RELATED WORK

Automated bug fixing, also known as automated program repair (APR), is a prominent research topic that has attracted significant interest from many researchers in the field. The literature includes many existing studies that present algorithms for automated bug repair. As illustrated in Fig. 2, these studies can be classified into four main categories: bug reports-based APR, constraint-based APR, search-based APR, and learning-based APR. More details about each category will be presented in the following paragraphs.

A. *Bug reports-Based APR*

Liu et al. [1] proposed R2Fix model for automatically generating bug-fixing patches using bug reports. Their model uses machine learning techniques, semantic patch generation techniques, and past fix patterns to automatic bug fixing. They evaluated their model for three bug types which are buffer overflows, null pointer bugs, and memory leaks. In the evaluation, they used three projects, the Linux kernel, Mozilla, and Apache. Their model generated 57 correct patches.

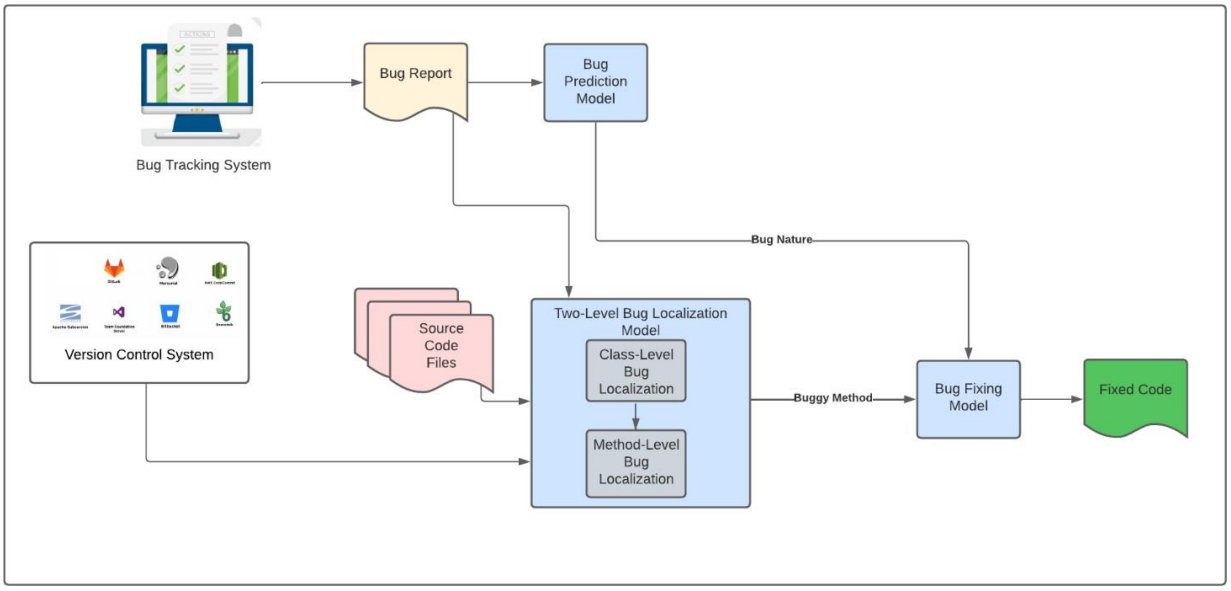

Fig. 1. Overall architecture of the bug management system.

Koyuncu et al. [2] proposed a novel model, namely iFixR, for bug localization and repair. In fact, their model is a bug repair system driven by bug reports. Their model uses bug reports as input. The main steps in their model are as follow: first, the bugs reports are fed to an information retrieval-based bug localizer; second, using fix pattern, patches are generated and validated by regression testing; third, patches are ordered by their priority for the developers. Their model did not have any assumption on the availability of test cases. To evaluate their model, they use and re-organize De-fects4J benchmark dataset and found that their proposed model can generate and recommend priority correct (and more plausible) patches for a wide range of issues reported by users.

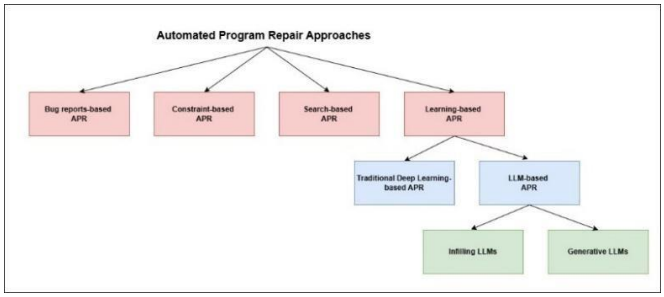

Fig. 2. Categories of existing APR approaches.

B. Constraint-Based APR

Constraint-based APR techniques use formal specifications and constraint solvers to transform the program repair problem into a constraint satisfaction problem. By focusing on expression-level variations and quickly pruning infeasible parts of the search space, these techniques can efficiently generate patches that satisfy the desired program behavior as specified by test cases or formal constraint [15].

Xuan et al. [3] proposed Nopol, a model for repairing faulty conditional statements, their model uses test cases as implicit program specifications. It generates patches by identifying expression-level changes that satisfy the constraints derived from the test cases.

Nguyen et al [4] introduced SemFix, a constraint-based method for APR, their method uses test cases to guide the patch synthesis process. The key features of SemFix include the use of symbolic execution and constraint solving to generate patches that ensure the program meets the desired behavior as specified by the test cases.

C. Search-Based APR

Goues et al. [14] presented GenProg, a generic and automated method for software repair, utilizing Genetic Programming to evolve and generate effective patches for a wide range of software defects. In the experiment, their method successfully fixed bugs in 16 programs written in C language which contain eight types of bugs. In their method, they presented three main novel ideas: Firstly, their method searches, in the program, at the statement level of the abstract syntax tree (AST). Secondly, to repair the bugs, their method does not introduce new code and uses only statements from the program. Lastly, genetic operators are localized to statements that are executed on the failing test case. However, their method does not support multi-hunk or multi-language program repair [15].

Yuan et al. [12] presented ARJA, a genetic programming-based approach for automated program repair in Java. ARJA suggests a new way of approaching automated program repair by treating it as a multi-objective optimization problem. They used a multi-objective optimization approach to minimize the weighted failure rate and patch size simultaneously, employing a multi-objective genetic algorithm (NSGA-II) to search for simpler repairs. The key innovation lies in designing a more detailed representation of patches, where the search spaces for likely-faulty locations, operation types, and ingredient statements are separated. This separation is intended to enhance the effectiveness of the genetic programming algorithm in finding appropriate solutions for fixing bugs in Java programs. The authors conducted a large-scale experimental study on both seeded and real-world bugs, demonstrating the effectiveness of ARJA in generating correct patches for a significant number of bugs compared to other repair approaches.

Yang et al. [13] presented a novel approach that uses similar bug fix information to automatic bug repair based on Genetic programming (GP). In their model, the candidate patches are generated by applying GP utilizing similar bug repair information. Then, the candidate patches are verified, by using a fitness function based on given test cases, if they are adoptable or not. Finally, the model generates the patch to fix the buggy code.

D. Learning-Based APR

Learning-based APR can be categorized further into two subcategories which are traditional deep learning-based APR, and Large Language Models-based (LLMs-based) APR.

1) Traditional deep learning-Based APR: Li et. al. [10] used deep learning algorithms in APR and proposed a two-level model, DLFix, which is based on deep learning algorithm that applies code transformation learning which learns from prior bug repairs and the surrounding code contexts of the fixes. The first tier contains an RNN model that is used to learn the context of bug fixes and the second tire uses the result from the previous tire as an additional weighting input learn the code transformations of the bug-fixing.

Lutellier et al. [16] presented a novel model, CoCoNuT, utilizes a new context-aware neural machine translation (NMT) architecture and an ensemble deep learning model which consists of combination of convolutional neural networks (CNNs) to automatically fix defects in multiple languages. The faulty source code and its surrounding context is separately represented using NMT. Their model utilizes CNNs in the hierarchical features extraction. However, their model cannot be used for multi-hunk bug fixing.

Huq et al. [17] proposed a novel sequence-to sequence model, Review4Repair, a deep learning-based approach that uses a neural machine translation (NMT) in APR. Their model uses the code review information to increase the performance of the automated bug fixing process.

DLFix, CoCoNuT, and Review4Repair all utilize context-aware strategies for patch generation and have the ability to fix multi-type bugs. However, all of them do not concern with the design of fault localization and work with the help of existing fault localization tools, and all do not have the capability of multi-hunk and multi-fault repair [15].

2) LLMs-Based APR: Recent advancements in large pre-trained LLMs offer a new direction for developing novel program repair models that do not rely on historical bug fixes [9]. Recently, some researchers proposed APR models based on LLMs. These models can be divided further into two sub-categories Infilling LLM-based APR, and Generative LLM-based APR.

Xia et al. [8] utilized CodeBERT [7], a pre-trained bimodal model designed for both programming languages (PL) and natural languages (NL), to propose a novel model for automated program repair (APR) called AlphaRepair. This model, which does not require retraining or fine-tuning on historical bug fixes datasets, represents the first cloze-style APR approach. Unlike traditional NMT tasks, AlphaRepair handles repair tasks as cloze tasks [15], aiming to predict the correct code based on its surrounding context [9]. Evaluation results showed that AlphaRepair can outperform state-of-the-art APR approaches. Mashhadi and H. Hemmati [20] presented a novel APR model which used CodeBERT [7] and fine-tuned it on ManySStuBs4J small and large datasets to generate fix patches for the buggy code. Evaluation results showed that their model can generate correct fixes in 19-72% of the cases.

Prenner & Robbes [21] presented a study which investigates the performance of Codex [11], a GPT language model that fine-tuned on GitHub code, in bug localization and fixing tasks. They used a dataset of 40 bugs in Java and Python and found that although Codex is not specifically trained for automated program repair task, it is effective for this task. Their observations also found that it is more effective at fixing Python bugs than Java bugs.

Xia and Zhang [5] presented ChatRepair, a conversational approach to automatically fixing bugs using ChatGPT. Their model takes as input relevant test failure information, enhancing bug-fixing capabilities of ChatGPT by learning from the failures and successes of previous patching attempts on the same bug. The model successfully fixed 162 out of 337 bugs from the Defects4j dataset.

Sobania et al. [18] presented an analysis of using ChatGPT for automated bug repair. In their study, they utilized 40 programs written in the Python programming language from the publicly available QuixBugs dataset to explore capabilities of ChatGPT in the bug fixing process. They found that providing ChatGPT with additional information (i.e., hints) about the bugs significantly improved its performance, resulting in a success rate that reached fixing 31 out of 40 bugs from the QuixBugs dataset. This performance outperforms several state-of-the-art APR models.

Despite the extensive development of various APR approaches, the field still faces significant challenges in improving the precision and generalizability of bug fixes across various programming environments. Existing models often rely heavily on extensive test suites, historical bug fixes, or detailed bug reports, which may not always be available or sufficiently comprehensive. Furthermore, many of these models struggle with complex bug fixes that require understanding detailed programming contexts or generating comprehensive patches. This paper aims to overcome some of current limitations by implementing a novel APR model that utilizes the advanced capabilities of LLMs such as gpt-3.5-trubo and gpt-4-0125-preview. Our approach reduces dependency on traditional inputs like test suites and historical fixes by directly interpreting the context of buggy code and generating appropriate fixes. By achieving this, the proposed model not only aims to enhance the accuracy and applicability of automated bug fixes but also to improve the repair process, making it more efficient and less reliant on extensive manual inputs. Thus, this paper aims to provide a foundation for future research in employing large language models to refine and expand automated repair algorithms.

IV. METHODOLOGY

This section illustrates the methodology that applied to develop the bug fixing model.

A. The Architecture of the Proposed Model

The architecture of the proposed model is illustrated in Fig. 3. This model takes the bug type and the buggy method as input and outputs the fixed version of the method. The bug types include Program Anomaly, GUI, Test-Code, and Performance. Program Anomaly bug refers to bugs in the source code files that occurs due to problems in the code [35] such as logical and syntax errors [28], GUI category refers to any bugs in the code that are related to the design of graphical user interface design or event handling [35], Test-Code bugs are occurs due to any problem which are related to the test code [35], while Performance bugs are related to the problem in the source code that are affect performance issues such as memory usage and memory leaks [35]. The two inputs are fed into the LLM using the prompt which is presented in the next subsection. In this research, two LLMs were tested to determine which one produces the highest accuracy in bug fixing for the generated dataset. These LLMs are gpt-3.5-turbo [24] and gpt-4-0125-preview [25]. As will be detailed in Section VI, gpt-4-0125-preview proved to be more accurate than gpt-3.5-turbo in terms of bug fixing and was therefore chosen for the proposed model.

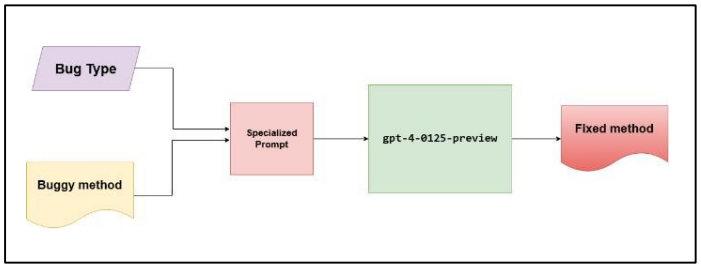

Fig. 3. High-level architecture of the proposed bug fixing model.

B. Large Language Models

LLMs are advanced deep learning algorithms capable of understanding, summarizing, translating, predicting, and generating content by leveraging large datasets [36]. In this research two LLMs from GPT family are used: the gpt-3.5-trubo model from GPT-3.5 and gpt-4-0125-preview model from GPT-4.

gpt-3.5-turbo [24]: is a variant of the GPT-3.5 language model. It is designed for enhanced performance and efficiency. This model is a group of models that improves upon GPT-3.5, enhances the ability to understand and generate both natural language and code.

gpt-4-0125-preview [25]: is a variant of the GPT-4 model. It is designed to give users a preview of the capabilities of the GPT-4 architecture. gpt-4-0125-preview performs tasks such as code generation more thoroughly than previous preview models. Additionally, this model is intended to reduce instances of "laziness" where the model does not fully complete a task [30].

C. Prompt Engineering

Models based on GPT utilize the paradigm of learning through prompts [26]. A prompt can be defined as a set of instructions that are given to the LLM, such as gpt-4-0125-preview, that program it by customizing it and/or improving or refining its capabilities [26]. Additionally, it is used to instruct the LLM to automate process or enforce rules [27]. Prompt engineering involves crafting clear, specific, and bias-mitigated prompts to guide the behaviour and output of AI models like GPT. It requires defining tasks, providing examples, and iterating on prompt design based on feedback to optimize results. Domain-specific knowledge may be necessary, and evaluation metrics help evaluate the effectiveness of the designed prompt.

The main contribution of this work lies in the application of LLMs for APR, and prompt engineering is an essential part of this process. The design of the prompt significantly influences the performance of the utilized LLM [6]. In the proposed model, the prompt is structured to instruct the LLM to accurately fix a buggy method. This prompt explicitly informs the LLM of the buggy method and categorizes the bug type. In the proposed model for automated bug fixing, a specific prompt template is utilized to guide LLMs, like gpt-3.5-turbo and gpt-4-0125-preview, to generate fixed versions of buggy Java methods. The template consists of several components: an objective statement, a placeholder for the incorrect Java function ({buggy_function}), and a detailed reason description placeholder ({reason_description}) that provides context about the type of bug, including categories such as GUI, Program Anomaly, Test-Code, and Performance. This context helps the model to better understand the issue and generate an appropriate fix. The template is structured as follows:

- **Objective Statement**: "Your goal is to correct the given Java function. The function will have a Javadoc that describes the function's purpose, parameters, and return value."
- **Function Placeholder**: {buggy_function}, which includes the buggy method which may has a Javadoc. The proposed model works effectively regardless of whether the method includes a Javadoc description.
- **Reason Description Placeholder:** {reason_description}, which explains the specific bug type, such as GUI issues related to layout problems or program anomaly involving logical errors.

The final prompt ends with "The correct Java function is:" prompting the model to generate the fixed function based on the provided details. This structured approach helps in automating the repair of code by leveraging the advanced capabilities of LLMs, thus enhancing the accuracy and efficiency of APR without requiring additional user input or test cases. This template plays a critical rule in achieving high accuracy rates in bug fixing, as demonstrated in the experiments conducted in this study.

V. Experimental Details

This section outlines the experimental setup and describes the implementation of the proposed bug fixing model. The experiments were conducted on a computer equipped with an x64 Intel® Core™ i7-10510U CPU @ 1.80GHz and 16.0 GB of RAM, running a 64-bit Windows Operating System. The model was developed using the Python programming language on Google Colab.

A. Dataset

To demonstrate the effectiveness of the proposed model, it is evaluated through two experiments. The first experiment involves evaluating of the proposed model using a new dataset. While in the second experiment, the proposed model is evaluated using a publicly available dataset to compare its performance against several state-of-the-art APR models.

1) Data collection and labeling: There are many reasons behind generating a new dataset in our research. First, there is no available dataset that contains source code files labeled in the same bug categories that are predicted by our bug prediction model. Secondly, we want to accurately evaluate our proposed model in a way that avoids information leakage from an existing dataset that might have already been seen by the LLM.

The presented dataset contains 53 Java source code files. These source code files were classified into four categories: Program Anomaly, GUI, Test-Code, and Performance. The Program Anomaly category contains bugs in the source code that occur due to logical and syntax errors. The GUI category refers to any bugs related to the design of graphical user interfaces or event handling. Test-Code bugs occur due to problems related to the test code. Performance bugs are related to issues in the source code that affect performance, such as memory usage, memory leaks, and missed performance improvements. The source code snippets used in this study were generated or collected from various sources available online. This involved identifying, collecting, and inspecting code snippets that contained bugs. The primary focus was on extracting examples that clearly demonstrated typical software

defects across different categories. These sources included open-source projects, coding forums, and educational resources. To ensure the quality and relevance of the dataset, each code snippet was carefully reviewed and categorized into one of the four main bug types mentioned above. This labeling process involved a detailed examination of each code snippet to identify the specific type of bug it represented. For instance, snippets involving inefficient coding practices or resource management issues were labeled as Performance Bugs, while those affecting the visual or functional aspects of the user interface were categorized as GUI Bugs. Similarly, errors within the test cases themselves were classified as Test-Code Bugs, and various coding errors leading to abnormal behavior or crashes were labeled as Program Anomaly Bugs.

This collaborative effort in data collection and labeling ensured a comprehensive and well-organized dataset, providing a solid foundation for analyzing common patterns and implications of different bug types in software engineering. Table I shows the number of Java source code files in each category. The Program Anomaly category contains 22 Java source code files, the GUI category contains 10 Java source code files, the Test-Code category includes 10 Java source code files, and the Performance category includes 11 Java source code files.

2) Analysis of the generated the dataset: The generated dataset consists of faulty source code files categorized into four main bug types: Performance, GUI, Test-Code, and Program Anomaly. Each category has distinct characteristics and common issues that are significant in understanding software bugs. A detailed analysis is presented for each bug type, their common patterns, and their effects in the following subsections.

a) Performance bugs: Performance bugs are mostly related to inefficient coding practices that reduce the runtime performance of applications. In the generated dataset, common issues include:

- **Inefficient String Operations:** Examples include using += in loops for string concatenation and creating new String objects unnecessarily.
- **Resource Management Issues**: Such as memory leaks from not clearing lists and inefficient use of wrapper classes leading to unnecessary boxing.
- **Control Flow Issues:** Infinite loops and concurrent modification exceptions which can cause applications to hang or crash.
- **Recursion Problems**: Recursive methods without proper termination can lead to stack overflow errors.
- **Other Issues:** Incorrect access patterns that can severely impact performance.

Effects: Performance bugs can cause significant degradation in application efficiency, leading to higher resource consumption and potential application failure. Identifying and optimizing these areas is crucial for enhancing software performance.

b) GUI Bugs: GUI bugs affect the usability and visual consistency of the application interface. Common GUI issues in the generated dataset include:

- **Visibility and Layout Problems:** Issues such as frames not being visible when they should be, incorrect component placements, and duplicate buttons in layouts.
- **Event Handling and Updates:** Bugs like missing initialization of components leading to null pointer exceptions, and incomplete status bar updates.
- **Drawing and Redrawing Inefficiencies**: Inefficient methods for redrawing components and missing drawing code.

Effects: GUI bugs can lead to poor user experience by making the interface confusing or non-functional. Proper testing and validation of the user interface components are essential to ensure smooth user interaction.

c) Test-Code Bugs

Test-Code bugs are errors within the test cases themselves which reduce the reliability of testing. Common issues in the generated dataset include:

- **Duplicate and Incorrect Test Methods:** Duplicate methods and logical errors in the methods being tested.
- **Uninitialized Variables:** Leading to compilation errors or incorrect test execution.
- **Incorrect Assertions:** Logical errors in assertions which lead to incorrect test outcomes.

Effects: Bugs in test code can lead to false positives or negatives, giving a misleading picture of the software quality. Ensuring the correctness of test code is as important as the application code itself.

d) Program anomaly bugs: Program Anomaly bugs involve a wide range of coding errors that result in abnormal behaviour or crashes. Common issues include:

- **Programming Errors:** Syntax errors and logical errors.
- **Control Flow Issues:** Infinite loops and incorrect loop conditions leading to unexpected behaviour.
- **Null Pointer and Type Safety Issues:** Potential null pointer exceptions and type mismatches.
- **Algorithmic and Recursive Errors:** Issues in algorithm implementation causing incorrect results or infinite recursion.

Effects: Program Anomaly bugs are critical as they often lead to crashes or incorrect program behaviour, significantly affecting the reliability and correctness of the software code. Comprehensive code review and rigorous testing are necessary to detect and fix these issues.

TABLE I. STATISTICS OF THE GENERATED DATASET

Bug Type Category	Total Number of Java Source Code Files in the category
GUI	10
Program Anomaly	22
Test-Code	10
Performance	11

3) QuixBugs benchmark dataset: The second experiment evaluates the proposed model using a publicly available benchmark dataset, QuixBugs[40], a collection of programs, each containing a specific bug, designed to test and evaluate the effectiveness of APR tools. The dataset includes 40 distinct algorithmic problems, such as sorting, graph traversal, and dynamic programming, implemented in both Java and Python. By providing a standardized set of challenges, QuixBugs allows researchers to consistently compare the performance of different APR methods. This dataset is widely used in academic research to advance the field of automated bug fixing. In the experiment, we use source code files written in Java only.

B. Evaluation Metrics

To evaluate the performance of the proposed model, the following evaluation metrics have been used.

- **Number of Repaired Defects**: It counts the defects which were successfully fixed by the repair algorithm. This metric is useful for demonstrating the capability of the algorithm. The diversity of defect classes in benchmark programs are crucial for the accuracy of this metric [29].
- **Repaired Defect Class:** Identifies specific classes of defects that the repair algorithm can successfully address. This helps in understanding the scope and specialization of the repair algorithm [29].
- **Success Rate (%):** The Success Rate of an automated program repair algorithm is quantitatively defined as the percentage of buggy programs that were successfully repaired by the algorithm out of the total number of buggy programs subjected to repair attempts. It is calculated using the Formula (1).

$$Success\ Rate\ (\%) = \left(\frac{Total\ Number\ of\ Successfully\ Fixed\ Programs}{Total\ Number\ of\ Buggy\ Programs}\right) \times 100 \quad (1)$$

VI. RESULTS

This section shows the results of the two experiments of the proposed bug fixing model on the generated dataset and on the benchmark, QuixBugs, dataset.

A. Results of the Proposed Model on the Generated Dataset

In the first experiment, the investigation of the achieved results by the proposed model explored the effectiveness of leveraging advanced language models, gpt-3.5- turbo and gpt-4-0125-preview, in automatically identifying and fixing bugs in software code. The analysis was structured around four main categories of bugs: GUI, Program Anomaly, Test-Code, and Performance. Table II shows the effectiveness of the proposed model in accurately repairing code from the used dataset.

The performance of the proposed model was evaluated based on its ability to correctly identify and fix these bugs. The proposed model can effectively localize the buggy lines, generate the correct patches, and insert them in the correct locations. The total accuracy of the proposed models (Success Rate) was calculated as follows: For the gpt-3.5-turbo model, the accuracy was 92.45%, while the gpt-4-0125-preview model achieved an accuracy of 100%. Fig. 4 shows an overview of the bug fixing results achieved by the proposed model.

TABLE II. RESULTS OF THE PROPOSED BUG FIXING MODEL

Bug Type	Correct Fixes by the Proposed Model (leveraging gpt-3.5-turbo)	Correct Fixes by the Proposed Model (leveraging gpt-4-0125-preview)
GUI	10 out of 10 (100%)	10 out of 10 (100%)
Program Anomaly	21 out of 22 (95.45%)	22 out of 22 (100%)
Test-Code	10 out of 10 (100%)	10 out of 10 (100%)
Performance	8 out of 11 (72.73%)	11 out of 11 (100%)
Total Fixes (Success Rate)	49 out of 53 (92.45%)	53 Out of 53 (100%)

B. Results of the Proposed Model on the QuixBugs Dataset

The proposed model that leverages gpt-4-0125-preview model has been evaluated using QuixBugs dataset. In more detail, the model achieved significant success, effectively repairing all 40 programs. This assessment covered a range of algorithmic problems, such as sorting, graph traversal, and dynamic programming. Utilizing advanced language model, gpt-4-0125-preview, our proposed APR model accurately detected and fixed bugs, demonstrating its robustness and efficiency.

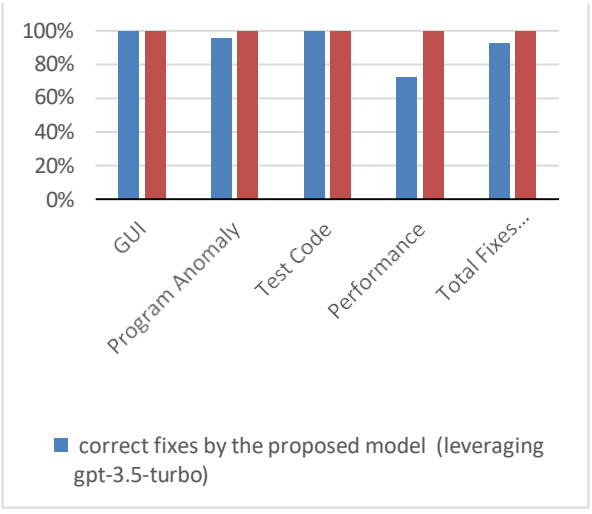

Fig. 4. An overview of the bug-fixing results achieved by the proposed model on the generated dataset.

VII. Analysis and Discussion

This section presents a comprehensive analysis and discussion of the results achieved by the proposed bug-fixing model in both experiments. Additionally, in this section, we answer the following research questions that address the improvements made by the proposed model in APR field.

- **Research Question 1**: What is the effect of different LLMs on the proposed bug fixing model on fixing different categories of programming bugs?
- **Research Question 2:** How does the proposed bug fixing model compare to several state-of-the-art APR models?
- **Research Question 3**: What are the practical improvements of integrating LLMs into the software development lifecycle for bug fixing?

A. *Discussion of the First Experiment*

To answer the first research question, we analyze the results of the first experiment.

Research Question 1: What is the effect of different LLMs on the proposed bug fixing model on fixing different categories of programming bugs?

Observations: Overall, the proposed model which leverages gpt-4-0125-preview demonstrates a 100% success rate, correctly fixing all 53 buggy source code files in the dataset. In contrast, the proposed model which leverages gpt-3.5-turbo achieves a 92.45% accuracy, successfully repairing 49 out of 53 buggy source code files. In the Program Anomaly category, the proposed model that leverages gpt-3.5-turbo fixed 21 out of 22 buggy source code files, while the proposed model that leverages gpt-4-0125-preview fixed all 22. The slight improvement with the latter model might be attributed to enhanced understanding or processing capabilities of gpt-4-0125-preview, possibly due to larger training data or improved algorithms. The most notable difference is observed in the performance category, where the proposed model that leverages gpt-3.5-turbo fixed 8 out of 11 bugs, whereas the proposed model that leverages gpt-4-0125-preview fixed all 11. This improvement may reflect advancements in the ability of gpt-4-0125-preview model to understand and optimize code for performance, a complex task that often requires deep under-standing and nuanced changes. Both models perform equally well in fixing GUI and Test-Code bugs, achieving a 100% success rate. This indicates robust capabilities in addressing issues within these specific categories, suggesting that the models have effectively learned patterns or solutions relevant to these types of bugs. Fig. 5 shows a buggy method in the dataset which is from GUI category. Fig. 6 and Fig. 7 show the fixed versions of this method using the proposed model (leveraging gpt-3.5-turbo model and gpt-4-0125-Preview model, respectively).

From this example, it is clear that both proposed models aim to address the issues found in the original code of the buggy method, but they do so with different ways. The buggy method has a bug in the loop that results in displaying incorrect keypad layout (i.e. it displays numbers from 1 to 12 instead of numbers from 1 to 9).

```java
/**
 * This method  for GUI that resembles a telephone keypad.
 *
 *
 */

public void telephone() {
   JFrame frame = new JFrame();
   frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
   frame.setSize(new Dimension(250, 200));
   frame.setTitle("Telephone");

   frame.setLayout(new BorderLayout());

   // main phone buttons
   JPanel centerPanel = new JPanel(new GridLayout(4, 3));
   for (int i = 1; i <= 12; i++) {
   centerPanel.add(new JButton("" + i));
   }
   centerPanel.add(new JButton("0"));
   centerPanel.add(new JButton("0"));
   centerPanel.add(new JButton("0"));
   frame.add(centerPanel, BorderLayout.CENTER);

   // south status panel
   JPanel southPanel = new JPanel(new FlowLayout());
   southPanel.add(new JLabel("Number to dial: "));
   southPanel.add(new JTextField(10));
   frame.add(southPanel, BorderLayout.SOUTH);
  frame.setVisible(true);
   }
```

Fig. 5. Buggy method from GUI category.

```java
/**
 * This method creates a GUI that resembles a telephone keypad.
 *
 * @param none
 * @return void
 */
public void createTelephoneGUI() {
    JFrame frame = new JFrame();
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setSize(new Dimension(250, 200));
    frame.setTitle("Telephone");

    frame.setLayout(new BorderLayout());

    // main phone buttons
    JPanel centerPanel = new JPanel(new GridLayout(4, 3));
    for (int i = 1; i <= 9; i++) {
        centerPanel.add(new JButton("" + i));
    }
    centerPanel.add(new JButton("0"));
    centerPanel.add(new JButton("*"));
    centerPanel.add(new JButton("#"));
    frame.add(centerPanel, BorderLayout.CENTER);

    // south status panel
    JPanel southPanel = new JPanel(new FlowLayout());
    southPanel.add(new JLabel("Number to dial: "));
    southPanel.add(new JTextField(10));
    frame.add(southPanel, BorderLayout.SOUTH);

    frame.setVisible(true);
}
```

Fig. 6. Fixed method from GUI using the proposed model (leveraging gpt-3.5-turbo model).

```java
import javax.swing.*;
import java.awt.*;

/**
 * Creates a GUI that resembles a telephone keypad. This interface includes a grid of number buttons
 * from 1 to 9, followed by '*', '0', and '#', alongside a display for the number to dial.
 */
public void telephone() {
    // Necessary import statements for JFrame, JPanel, JButton, etc.
    JFrame frame = new JFrame();
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setSize(new Dimension(250, 200));
    frame.setTitle("Telephone");

    frame.setLayout(new BorderLayout());

    // Main phone buttons
    JPanel centerPanel = new JPanel(new GridLayout(4, 3));
    for (int i = 1; i <= 9; i++) {
        centerPanel.add(new JButton("" + i));
    }
    // Correcting the loop to add "*", "0", "#"
    centerPanel.add(new JButton("*"));
    centerPanel.add(new JButton("0"));
    centerPanel.add(new JButton("#"));
    frame.add(centerPanel, BorderLayout.CENTER);

    // South status panel with a label and text field for entered number
    JPanel southPanel = new JPanel(new FlowLayout());
    southPanel.add(new JLabel("Number to dial: "));
    JTextField textField = new JTextField(10);
    southPanel.add(textField);
    frame.add(southPanel, BorderLayout.SOUTH);
    frame.setVisible(true);
}
```

Fig. 7. Fixed method from GUI using the proposed model (leveraging gpt-4-0125-preview model).

Additionally, it incorrectly adds three "0" buttons instead of the conventional "*", "0", and "#" buttons found on a telephone keypad. Fig. 6 shows that the proposed model that leverages gpt-3.5-turbo model modifies the name of method to create TelephoneGUI, aligning with Java naming conventions by clearly describing the action performed by the method. This change emphasizes the importance of readability and maintainability in code. Additionally, it fixes the keypad layout to include "1" through "9", followed by "0", "*", and "#". Although the bug in the telephone keypad GUI has been successfully resolved, the arrangement of the buttons on the bottom row remains incorrect. The expected order of the buttons is * 0 #; however, the current implementation places them in the order 0 * #. Additionally, this model lacks the inclusion of import statements necessary for a standalone application, assuming prior context or placement within a larger source code. This approach addresses the discovered bugs and improves the naming of the method for better clarity. However, it fails to make the code independently usable by omitting necessary import statements. Fig. 7 shows the fixing code of the same buggy method using the proposed model that leverages gpt-4-0125-preview model which offers a comprehensive repairing by not only fixing the keypad layout but also adding essential import statements and a javadoc comment at the beginning of the method. This fixing assumes no prior context, aiming to make the code snippet independently compliable and understandable. The fixed version of the code follows best practices by including a detailed method description and fixing the keypad issue, which matches the standard layout of a telephone keypad. This model provides a more thorough fixing by ensuring that the code is both correct and self-sufficient. It addresses not only the initial logical issues but also enhances the readability and reusability of the code by adding documentation and necessary technical details for compilation.

Another example of bug fixing ability of the proposed model can be illustrated by Fig. 8-10. Fig. 8 presents a buggy code from the Test-Code category. The original buggy code presents a simple divide method intended to perform division operations while handling a potential division-by-zero error through an exception. However, the accompanying test cases contain several issues: incorrect assertions that do not match the expected outcomes of the division operation, duplication of test method names (which is not allowed in Java), and lack of a test case to explicitly check for the division by zero scenario.

Fig. 9 and Fig. 10 present the fixed versions of the buggy code by the proposed model in the two experiments, the first one when proposed fixing model leveraging gpt-3.5-turbo model and the second one when the proposed fixing model leveraging gpt-4-0125-preview model. Both models address the core issues in the original code, but they do so with varying levels of detail and adherence to best practices in software testing and documentation. Fig. 9 shows that leveraging gpt-3.5-turbo model in the proposed model fixes the logical bugs in the test cases and suggests a structured approach to handle different test scenarios, including dividing by zero. It provides clear, separate test methods with descriptive names that reflect their purpose: one for a successful division operation and another for testing division by zero. Improves readability and maintenance of the test suite by clearly separating test cases. Additionally, it correctly identifies the need for and implements a test case for division by zero, enhancing the robustness of the tests. However, in comparison with Fig. 10, the fixing is somewhat basic and does not explicitly address best practices in exception handling within test methods (e.g., using the expected annotation attribute).

```java
/**
 * Calculates the division of two given numbers.
 *
 * @param a and b  The numbers for which the division is calculated.
 * @return The division of the input numbers.
 */
public int divide(int a, int b) {
    if (b == 0) {
        throw new IllegalArgumentException("Cannot divide by zero");
    }
    return a / b;
}


@Test
public void testDivide() {
    MathOperations mathOperations = new MathOperations();
    int result = mathOperations.divide(8, 2);
    assertEquals(8, result);
}

@Test
public void testDivide() {
    MathOperations mathOperations = new MathOperations();
    int result = mathOperations.divide(8, 2);
    assertEquals(16, result);
}
```

Fig. 8. Buggy code from Test-Code category.

Fig. 10 shows that leveraging gpt-4-0125-preview model in the proposed model offers a comprehensive fixing by not only repairing the test cases but also refining the testing approach to align with best practices. It fixes the bugs,

addresses method name duplication, and includes a detailed approach to testing exceptions using the expected attribute of the *@Test* annotation. This enhances the readability and maintainability of the test suite by demonstrating advanced testing techniques. However, the fixing assumes familiarity with the expected attribute for testing exceptions, which may require additional context for less experienced Java developers.

```java
/**
 * Calculates the division of two given numbers.
 *
 * @param a The numerator of the division.
 * @param b The denominator of the division.
 * @return The division of the input numbers.
 * @throws IllegalArgumentException if the denominator is zero.
 */
public int divide(int a, int b) {
    if (b == 0) {
        throw new IllegalArgumentException("Cannot divide by zero");
    }
    return a / b;
}

@Test
public void testDivide() {
    MathOperations mathOperations = new MathOperations();
    int result = mathOperations.divide(8, 2);
    assertEquals(4, result);
}

@Test
public void testDivideByZero() {
    MathOperations mathOperations = new MathOperations();
    assertThrows(IllegalArgumentException.class, () -> {
        mathOperations.divide(8, 0);
    });
}
```

Fig. 9. Fixed code from Test-Code category using the proposed model (leveraging gpt-3.5-turbo model).

```java
import static org.junit.Assert.assertEquals;
import org.junit.Test;

public class MathOperationsTest {

    /**
     * Calculates the division of two given numbers.
     *
     * @param a The dividend.
     * @param b The divisor.
     * @return The quotient from dividing a by b.
     */
    public int divide(int a, int b) {
        if (b == 0) {
            throw new IllegalArgumentException("Cannot divide by zero");
        }
        return a / b;
    }

    @Test
    public void testDivide_SuccessfulDivision() {
        MathOperations mathOperations = new MathOperations();
        int result = mathOperations.divide(8, 2);
        assertEquals(4, result); // Correcting the expected result to match 8 divided by 2.
    }

    // Example of how to test for division by zero scenario, if needed.
    @Test(expected = IllegalArgumentException.class)
    public void testDivisionByZero_ThrowsException() {
        MathOperations mathOperations = new MathOperations();
        mathOperations.divide(8, 0); // This should throw an IllegalArgumentException.
    }
}
```

Fig. 10. Fixed code from Test-Code category using the proposed model (leveraging gpt-4-0125-preview model).

The overall evaluation results provide a high-level view of the effectiveness of each model in the experiment in automated program repair. The best score achieved by the proposed model that leverages gpt-4-0125-preview is particularly significant, suggesting a notable advancement in the capability of the proposed model in understanding and fixing a wide range of software bugs.

The difference in accuracy between the two models (92.45% for the model leveraging gpt-3.5-turbo vs. 100% for the model leveraging gpt-4-0125-preview) could be attributed to several factors inherent to the evolution of generative pre-trained transformer models, such as increased model size, more diverse and extensive training data, or refined training techniques that improve understanding and generation capabilities.

B. Discussion of the Second Experiment

To answer the second research question, we analyze the results of the second experiment.

Research Question 2: How does the proposed bug fixing model compare to several state-of-the-art APR models?

Observations: To indicate the effectiveness of the proposed model, a comparison with the performance of several state-of-the-art APR models using the same dataset is crucial. Therefore, the achieved results of the proposed model on the QuixBugs dataset are compared to four studies which are:

- The results of the study presented by authors in the paper [39], involve an empirical study on automated bug repair using QuixBugs benchmark dataset.
- AlphaRepair [8] represents the first cloze-style APR approach, and it handles repair tasks as cloze tasks to predict the correct code based on its surrounding context.
- CoCoNut [16] which utilizes a new context-aware neural machine translation architecture and an ensemble deep learning model to fix buggy code.
- CURE [38] which is a novel APR tool that focuses on resolving software bugs through a sophisticated, context-aware neural machine translation (NMT) approach. Its main goal is to enhance the accuracy and effectiveness of bug fixes by utilizing detailed contextual information and robust neural network models. This allows CURE to generate more accurate patches by understanding not only the buggy code but also the surrounding context, significantly improving the quality and reliability of the fixes.

Table III illustrates the total number of QuixBugs programs out of 40 that are correctly fixed by the mentioned state of the art APR models and our proposed bug fixing model which leverages gpt-4-0125-preview model.

TABLE III. COMPARISON BETWEEN SEVERAL APR ON QUIXBUGS DATASET

Reference	Total Number of Correctly Fixed Programs	Success Rate
Ye et al. [39]	16 out of 40	40%
AlphaRepair [8]	28 out of 40	70%
CoCoNut [16]	13 out of 40	32.5%
CURE [38]	26 out of 40	65%
The Proposed Model	**40 out of 40**	**100%**

As observed from this table, our proposed model achieves the best results, successfully fixing all buggy programs in the dataset. Additionally, our proposed model does not need additional information about test cases, previous knowledge of patch attempts, nor additional follow up conversations which make it more efficient, practical and has its own novelty among other APR models as it only needs the buggy code and the bug type to debug the code, localized the buggy lines, generate the correct patches, and insert them in the right locations. Therefore, our proposed model outperforms all these state-of-the art APR models.

Furthermore, it was observed from this experiment that the proposed model not only fixed the identified bug, but also effectively enhanced the code by addressing the edge cases, aligning with Java naming conventions, and following better programming practices. In more details, Fig. 11 and Fig. 12 show the buggy version of TO_BASE.java program of QuixBugs dataset and its fixed version by the proposed model. Fig. 11 shows the buggy version of this program. The purpose of this Java program is to convert a given integer (num) into a string representation of that number in a specified base (b). The base can range from 2 to 36, allowing for conversion to binary, octal, decimal, hexadecimal, and other bases up to base 36, which includes digits 0-9 and letters A-Z. The main bug in this program is related to how the result string is constructed. In more detail, the code appends each new digit to the end of the result string, which results in the digits being in reverse order. Furthermore, there is no handling for the edge case where the input number is zero. From Fig. 12, the improvements made by the proposed model can be observed, which are:

- It fixes the bug; the fixed method reverses the resulting string ensuring the digits are in the correct order.
- It adds base validation; the fixed method includes a validation check to ensure the base is between 2 and 36. If the base is invalid, it returns an empty string.
- It adds zero handling; the fixed method correctly handles the case when num is zero, returning "0".
- It uses StringBuilder for efficiency; the fixed method uses StringBuilder for string concatenation, which is more efficient than repeatedly creating new string objects.
- It increases the clarity of the code; the fixed method is well-documented, explaining the purpose, parameters, and return value, making it easier to understand and maintain.

Therefore, the proposed model has significantly improved the original buggy method by adding necessary validations, handling edge cases, and improving performance through efficient string manipulation. These changes enhance the robustness of the method, efficiency, and readability, ensuring it works correctly across a wide range of inputs.

Furthermore, it was observed that the proposed model has the ability of multi-hunk and multi-fault repair. In more detail, the proposed model shows an ability for multi-hunk and multi-fault repair in software code. It can navigate and rectify issues spread across different parts of the source code (multi-hunk) and address several distinct types of faults within a single execution (multi-fault).

```java
import java.util.*;

/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

/**
 *
 * @author derricklin
 */
public class TO_BASE {
    public static String to_base(int num, int b) {
        String result = "";
        String alphabet = "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ";
        int i;
        while (num > 0) {
            i = num % b;
            num = num / b; // floor division?
            result = String.valueOf(alphabet.charAt(i))+result;
        }

        return result;
    }
}
```

Fig. 11. Buggy TO_BASE.java from QuixBugs.

```java
/**
 * Converts an integer to its representation in a specified base.
 *
 * @param num The number to convert.
 * @param base The base of the target number system (must be between 2 and 36 inclusive).
 * @return A string representation of the number in the specified base, or an empty string if the base is out of the valid range.
 */
public class TO_BASE {
    public static String to_base(int num, int base) {
        // Validate the base
        if(base < 2 || base > 36) {
            return ""; // Return an empty string for invalid base values
        }

        // Handle the case when num is 0
        if (num == 0) {
            return "0";
        }

        StringBuilder result = new StringBuilder();
        String alphabet = "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ";

        while (num > 0) {
            int remainder = num % base;
            num = num / base;
            result.append(alphabet.charAt(remainder));
        }

        // Since the conversion process builds the result string in reverse order,
        // we need to reverse it before returning.
        return result.reverse().toString();
    }
}
```

Fig. 12. Fixed TO_BASE.java by the proposed model.

The proposed model that leverages gpt-4-0125-preview model, in particular, demonstrates a higher level of sophistication in applying best practices, suggesting an advanced understanding and capability in handling complex repair scenarios efficiently. This analysis supports the

utilization of LLMs in automating comprehensive code repair tasks, highlighting their role in supporting and enhancing software development and maintenance processes.

Research Question 3: What are the practical improvements of integrating LLMs into the software development lifecycle for bug-fixing?

Observations: Integrating LLMs into the software development lifecycle for bug fixing has several practical improvements include:

- **Increased Productivity:** Developers can focus on higher-level tasks, as LLMs handle routine bug fixing, leading to increased productivity and efficiency.
- **Improved Code Quality:** Automated bug fixing can lead to more consistent and reliable code quality, as LLMs can systematically apply best practices and coding standards.
- **Reduced Time-to-Market:** Faster bug detection and fixing reduce the overall development cycle time, enabling faster releases and update**s.**
- **Enhanced Collaboration:** LLMs can help in bridging the gap between different team members (e.g., developers and testers) by providing clear and actionable bug debugging and fixes.

VIII. CONCLUSION

This paper introduced a novel approach to APR, utilizing LLMs to automate the bug-fixing process. Through the leveraging of gpt-3.5-turbo and gpt-4-0125-preview models, a significant leap forward in the field of software engineering has been demonstrated, particularly in enhancing software reliability and developer productivity. The proposed model presents an ability to accurately localize bugs across various code segments, debug the source code, generate correct patches, and integrate these fixes into the appropriate locations within the source code. The evaluation of the proposed model, conducted on a diverse dataset comprising 53 Java source code files categorized into four distinct bug categories, confirms the efficiency of the proposed model. The results show that the gpt-3.5-turbo model achieved an impressive success rate, successfully repairing 49 out of 53 source code files, equivalent to an accuracy of 92.45%. In contrast, the gpt-4-0125-preview model exhibited an exceptional performance, achieving 100% success rate in bug fixing. Additionally, the proposed model was evaluated using the QuixBugs benchmark dataset, and it can correctly fix all its Java buggy programs. The proposed model was compared to several state-of-the-art APR models and outperformed them. Such outcomes not only highlight the robustness of the proposed model in handling many types of bugs but also, reflect the advancements in large language-based program repair techniques. Furthermore, the comparative analysis offers valuable insights into the evolution of AI capabilities, particularly in the context of software debugging and maintenance. The superior performance of the gpt-4-0125-preview model, characterized by its ability to execute multi-hunk and multi-fault repairs with a higher degree of accuracy, points towards a promising future where the boundaries of automated software engineering can be expanded significantly. As the field continues to evolve, future research can uncover more advanced models and methodologies, further enhancing the scope and accuracy of automated bug fixing.

ACKNOWLEDGMENT

Deanship of Scientific Research (DSR) at King Abdulaziz University (KAU), Jeddah, Saudi Arabia, funded this project under grant no. (KEP-PhD-102-611-1443). The authors, therefore, acknowledge DSR for the financial support.

REFERENCES

[1] C. Liu, J. Yang, L. Tan, and M. Hafiz, "R2Fix: Automatically Generating Bug Fixes from Bug Reports," in *Proc. 2013 IEEE Sixth Int. Conf. Softw. Testing, Verification and Validation*, 2013, doi: 10.1109/ICST.2013.24.

[2] A. Koyuncu et al., "IFIXR: Bug Report Driven Program Repair," in *Proc. 2019 27th ACM Joint Meeting on European Softw. Eng. Conf. and Symp. on the Foundations of Softw. Eng.*, Aug. 2019, doi: 10.1145/3338906.3338935.

[3] J. Xuan, M. Martinez, F. Demarco, M. Clement, S. L. Marcote, T. Durieux, D. Le Berre, and M. Monperrus, "Nopol: Automatic repair of conditional statement bugs in java programs," IEEE Transactions on Software Engineering, vol. 43, no. 1, pp. 34–55, 2016.

[4] D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis," in *2013 35th International Conference on Software Engineering (ICSE)*, pp. 772–781, IEEE, 2013.

[5] H. S. Xia and L. Zhang, "Keep the Conversation Going: Fixing 162 out of 337 Bugs for $0.42 Each Using ChatGPT," *arXiv preprint arXiv:2304.00385*, 2023. [Online]. Available: https://arxiv.org/pdf/2304.00385.pdf [Accessed: Feb. 8, 2024].

[6] W. Ma et al., "LLMs: Understanding Code Syntax and Semantics for Code Analysis," 2024, *arXiv:2305.12138*. [Online]. Available: https://arxiv.org/pdf/2305.12138.pdf [Accessed: Mar. 25, 2024].

[7] Z. Feng et al., "Codebert: A Pre-Trained Model for Programming and Natural Languages," 2020, *arXiv:2002.08155*. [Online]. Available: https://arxiv.org/abs/2002.08155 [Accessed: Jan. 29, 2024].

[8] C. S. Xia and L. Zhang, "Less Training, More Repairing Please: Revisiting Automated Program Repair Via Zero-Shot Learning," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* , pp. 959–971, 2022.

[9] C. S. Xia, Y. Wei, and L. Zhang, "Automated Program Repair in the Era of Large Pre-Trained Language Models," in *Proceedings of the 45th International Conference on Software Engineering (ICSE 2023). Association for Computing Machinery*, May 2023, doi: 10.1109/ICSE48619.2023.00129.

[10] Y. Li, S. Wang, and T. N. Nguyen, "DLFix: Context-Based Code Transformation Learning for Automated Program Repair," in *Proceedings of the 42nd ACM/IEEE International Conference on Software Engineering (ICSE'20). 602 614.*, pp. 602–614, 2020.

[11] M. Chen et al., "Evaluating Large Language Models Trained on Code," 2021. [Online]. Available: http://arxiv.org/abs/2107.03374 [Accessed: Mar. 20, 2024].

[12] Y. Yuan and W. Banzhaf, "ARJA: Automated Repair of Java Programs via Multi-Objective Genetic Programming," *IEEE Trans. Softw. Eng.*, vol. 46, pp. 1040–1067, 2020, doi: 10.1109/TSE.2018.2874648.

[13] G. Yang, Y. Jeong, K. Min, J. Lee, and B. Lee, "Applying Genetic Programming with Similar Bug Fix Information to Automatic Fault Repair," *Symmetry*, vol. 10, p. 92, 2018, doi: 10.3390/sym10040092.

[14] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "GenProg: A Generic Method for Automatic Software Repair," *IEEE Trans. Softw. Eng.*, vol. 38, pp. 54–72, 2012.

[15] K. Huang et al., "A Survey on Automated Program Repair Techniques," 2023, *arXiv:2303.1*. [Online]. Available: https://arxiv.org/abs/2303.1 [Accessed: Mar. 17, 2024].

[16] T. Lutellier et al., "Coconut: Combining Context-Aware Neural Translation Models Using Ensemble for Program Repair," in *Proceedings of the 29th ACMSIGSOFT International Symposium on Software Testing and Analysis*, pp. 101–114, Jul. 2020, doi: 10.1145/3395363.3397369.

[17] F. Huq, M. Hasan, M. M. A. Haque, S. Mahbub, A. Iqbal, and T. Ahmed, "Review4Repair: Code Review Aided Automatic Program Repairing," *Inf. Softw. Technol.*, vol. 143, p. 106765, 2022.

[18] D. Sobania, M. Briesch, C. Hanna, and J. Petke, "An Analysis of the Automatic Bug Fixing Performance of ChatGPT," in *Proc. 2023 IEEE/ACM Int. Workshop on Automated Program Repair (APR)*, Melbourne, Australia, 2023, pp. 23-30, doi: 10.1109/APR59189.2023.00012.

[19] J. Zhang et al., "Revisiting Test Cases to Boost Generate-and-Validate Program Repair," in *Proc. 2021 IEEE Int. Conf. Softw. Maintenance and Evolution (ICSME)*, Sep. 2021, doi: 10.1109/ICSME52107.2021.00010.

[20] E. Mashhadi and H. Hemmati, "Applying Codebert for Automated Program Repair of Java Simple Bugs," in *Proc. 2021 IEEE/ACM 18th Int. Conf. Mining Softw. Repositories (MSR)*, Mar. 2021, doi: 10.1109/MSR52588.2021.00063.

[21] J. A. Prenner and R. Robbes, "Automatic Program Repair with OpenAI's Codex: Evaluating QuixBugs," 2021, *arXiv preprint arXiv:2111.03922*. [Online]. Available: https://arxiv.org/abs/2111.03922 [Accessed: Jan. 31, 2024].

[22] S. Alsaedi, A. A. Gad-Elrab, A. Noaman, and F. Eassa, "Two-Level Information-Retrieval-Based Model for Bug Localization Based on Bug Reports," *Electronics*, vol. 13, p. 321, 2024.

[23] S. A. Alsaedi, A. Y. Noaman, A. A. Gad-Elrab, and F. E. Eassa, "Nature-based prediction model of bug reports based on Ensemble Machine Learning Model," *IEEE Access*, vol. 11, pp. 63916–63931, 2023.

[24] OpenAI, "GPT-3.5 Turbo Model Documentation," OpenAI Platform, 2024. [Online]. Available: https://platform.openai.com/docs/models/gpt-3-5-turbo [Accessed: Apr. 15, 2024].

[25] OpenAI, "GPT-4 Turbo and GPT-4 Model Documentation," OpenAI Platform, 2024. [Online]. Available: https://platform.openai.com/docs/models/gpt-4-turbo-and-gpt-4 [Accessed: Apr. 15, 2024].

[26] P. Liu, W. Yuan, J. Fu, Z. Jiang, H. Hayashi, and G. Neubig, "Pretrain, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing," *ACM Comput. Surv.*, vol. 55, pp. 1–35, 2023.

[27] J. White et al., "A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT," 2023, *arXiv preprint arXiv:2302.11382*. [Online]. Available: https://arxiv.org/abs/2302.11382 [Accessed: Mar. 21, 2024].

[28] R. M. Karampatsis and C. Sutton, "How Often Do Single-Statement Bugs Occur? The ManySStuBs4J Dataset," in *Proc. 17th Int. Conf. Mining Softw. Repositories*, Jun. 2020, pp. 573–577, doi: 10.1145/3379597.3387491.

[29] Y. Qi, W. Liu, W. Zhang, and D. Yang, "How to Measure the Performance of Automated Program Repair," in *Proc. 2018 5th Int. Conf. Information Sci. and Control Eng. (ICISCE)*, Jul. 2018, doi: 10.1109/ICISCE.2018.00059.

[30] OpenAI, "New Embedding Models and API Updates," OpenAI Blog, 2024. [Online]. Available: https://openai.com/blog/new-embedding-models-and-api-updates [Accessed: Apr. 15, 2024].

[31] A. Zirak and H. Hemmati, "Improving Automated Program Repair with Domain Adaptation," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, pp. 1–43, 2024, doi: 10.1145/3631972.

[32] L. Gazzola, D. Micucci, and L. Mariani, "Automatic Software Repair: A Survey," *IEEE Trans. Softw. Eng.*, vol. 45, pp. 34–67, 2017.

[33] J. Wang et al., "Software Testing with Large Language Models: Survey, Landscape, and Vision," *IEEE Trans. Softw. Eng.*, doi: 10.1109/TSE.2024.3368208.

[34] X. Du et al., "Evaluating Large Language Models in Class-Level Code Generation," in *Proc. IEEE/ACM 46th Int. Conf. Softw. Eng.*, Apr. 2024, doi: 10.1145/3597503.3639219.

[35] H. A. Ahmed, N. Z. Bawany, and J. A. Shamsi, "CaPBug-A Framework for Automatic Bug Categorization and Prioritization Using NLP and Machine Learning Algorithms," *IEEE Access*, vol. 9, pp. 50496–50512, 2021, doi: 10.1109/ACCESS.2021.3069248.

[36] NVIDIA, "What Are Large Language Models?" NVIDIA Glossary. [Online]. Available: https://www.nvidia.com/en-us/glossary/large-language-models/ [Accessed: Apr. 20, 2024].

[37] T. Nazir and M. Pinzger, "SymDefFix—Sound Automatic Repair Using Symbolic Execution," 2022, *arXiv preprint arXiv:2209.03815*. [Online]. Available: https://arxiv.org/abs/2209.03815 [Accessed: Apr. 17, 2024].

[38] T. Jiang, T. Lutellier, and L. Tan, "Cure: Code-aware neural machine translation for automatic program repair," in *Proc. 2021 IEEE/ACM 43rd Int. Conf. Softw. Eng. (ICSE)*, Madrid, Spain, 22–30 May 2021, IEEE: New York, NY, USA, 2021, pp. 1161–1173.

[39] H. Ye, M. Martinez, T. Durieux, and M. Monperrus, "A comprehensive study of automatic program repair on the QuixBugs benchmark," *J. Syst. Softw.*, vol. 171, p. 110825, 2021.

[40] J. Koppel, "QuixBugs," [Online]. Available: https://github.com/jkoppel/QuixBugs/tree/master/ [Accessed: Jun. 2, 2024].