

Context Matters: Investigating Its Impact on ChatGPT's Bug Fixing Performance

Xianshan Qu[†], Fei Zuo[†], Xiaopeng Li[‡], Junghwan Rhee[†]

[†]University of Central Oklahoma, Edmond, Oklahoma, 73034, USA

{xqu1, fzu0, jrhee2}@uco.edu

[‡]Microsoft, Redmond, Washington, 98052, USA

xli.researcher@gmail.com

Abstract—In this study, we explore the role of contextual information in enhancing ChatGPT's capabilities in bug fixing. Our focus is specifically on the “Wrong Answer” problem, where a program executes without error but fails to produce the correct output. Our approach draws inspiration from human debugging practices, which heavily rely on understanding both the intended task of the program and the specific scenarios in which it fails, such as unit test cases. We evaluate ChatGPT's performance with various types and levels of contextual data. The results reveal three key insights. First, providing the model with a mix of correct and incorrect test cases sharpens its debugging skills. Second, giving ChatGPT detailed descriptions of the problems substantially enhances its ability to identify and resolve errors. Third, merging detailed problem descriptions with various test cases leads to a synergistic outcome. This combined approach significantly elevates the efficiency of the bug-fixing process compared to employing each type of contextual information individually. Our paper presents a thorough analysis based on these findings. It offers an extensive exploration of why and how contextual information can be strategically utilized to enhance ChatGPT's debugging effectiveness. Furthermore, this investigation enriches our comprehension of the underlying mechanisms by which contextual cues amplify the model's capacity for solving problems.

Index Terms—Bug Fixing, Contextual Information, Large Language Models, Problem Descriptions, Test Cases

I. INTRODUCTION

Debugging involves pinpointing and rectifying bugs in code to ensure it functions as intended. Automating this process of debugging and repair is a longstanding research challenge in the field of software engineering. Solutions to bug fixing can generally be divided into two main categories: traditional and learning-based. Traditional approaches either look for solutions in a space of candidates [1]–[3], or adopt a constraint solver to extract semantics [4]–[6]. These methods often require trade-offs between accuracy and scalability, making it challenging to improve both aspects simultaneously. By contrast, the learning-based approaches [7]–[9] leverage machine learning and artificial intelligence models to understand code and its functionality for the purpose of identifying and fixing bugs. This category of methods rely on well-maintained datasets of code to train models, potentially offering more dynamic solutions by adapting to the vast and evolving nature of programming languages and bugs. Consequently, the patterns that indicate errors can be identified, and corrections can be further suggested.

Recently, the emergence of large language models (LLMs) is undoubtedly one of the most impactful breakthroughs of AI. A large language model, as the name suggests, is a language model trained on massive amounts of text data and have billions of parameters. As a widely recognized instance of an LLM, the Generative Pre-trained Transformer (GPT) [10], [11] attracts increasing attention, because of its powerful ability to generate human-like responses in conversational interaction. In 2022, OpenAI launched ChatGPT to the public as an implementation of GPT. Beyond its initial acclaim as a chatbot, ChatGPT has shown remarkable adeptness in tackling a variety of software engineering problems, such as code synthesis, code refinement, and even program repair—a testament to its versatility. However, previous studies often concentrate on relatively simple bugs, usually those that can be resolved with a single line of code, such as those found in QuixBugs [12]. Moreover, despite the intricate nature of bug fixing in software engineering, current research has not extensively explored the impact of contextual information on ChatGPT's ability to resolve bugs.

With the understanding that ChatGPT holds the potential to navigate more complex debugging scenarios, especially with the aid of contextual information, our study seeks to explore this domain further. We pose the question: *How can the impact of contextual information be evaluated, and how can strategies for its use be optimized in ChatGPT's bug-fixing process?* This inquiry seeks to uncover effective methodologies for incorporating contextual data, thereby enhancing ChatGPT's debugging capabilities for more sophisticated and varied software issues.

Motivated by the human approach to debugging, where understanding the program's objective and identifying the specific error are crucial, we aim to assess whether supplying various types and levels of contextual information can improve ChatGPT's ability to fix bugs. We focus on evaluating the impact of individual types of contextual information and the cumulative effect of their integration, aiming to understand how each contributes and interacts to improve the model's debugging capabilities. Our findings suggest that including the problem description and test cases can individually enhance the model's performance. Moreover, when these two types of contextual information are integrated, they produce a synergistic effect, significantly boosting ChatGPT's debugging

capabilities. This underscores the importance of comprehensive context in facilitating more accurate and efficient bug resolution.

II. METHODOLOGY

To investigate ChatGPT's proficiency in bug fixing across varying contexts, we enhance the prompts given to ChatGPT with diverse types of contextual information. The question then arises: What specific contextual information should be included? Our methodology is influenced by the strategies humans use for debugging, which typically involve two key actions: ① they require understanding the problem that a code snippet is designed to solve; ② they involve analyzing the test cases that the code should pass but fails to.

- **Baseline Evaluation:** What is ChatGPT's bug-fixing ability with minimal context? Initially, we assess ChatGPT's debugging capability using a straightforward prompt that requests a bug fix for a given code snippet without providing additional context.
- **Problem Description Context:** How does ChatGPT's bug-fixing ability change with detailed information on the problem the code snippet addresses? To observe if understanding the problem's context enhances ChatGPT's debugging efficiency, we augment the prompt with a comprehensive description of the problem intended to be solved by the code,
- **Test Case Context:** Given the reliance of human debuggers on test cases to pinpoint and fix coding errors, we delve into how ChatGPT performs when equipped with detailed test case data. Our study aims at assessing the impact of various extents of contextual information on the model's debugging efficiency. We investigate two distinct setups: one approach involves providing only the failing test cases, while the other offers a comprehensive view by including both passing and failing test cases. Through this comparative analysis, we seek to reveal how different levels of test case information contribute to enhancing ChatGPT's precision in identifying and amending coding mistakes.
- **Integrating Problem Description and Test Case Context:** In our final analysis, we seek to evaluate the combined influence of providing both the problem description and test cases on the model's debugging performance. Our goal is to determine whether integrating in-depth understanding of the problem with detailed test case results yields a synergistic effect, enhancing the model's ability to efficiently debug issues.

III. EXPERIMENTAL DESIGN

A. Dataset and Evaluation Metrics

We employ the FIXEVAL [13] dataset, which is collected from CodeNet [14]. CodeNet dataset compiles programs in over 50 languages designed by participants for competitive programming on AtCoder and Aizu Online Judge platforms. FIXEVAL includes open-source unit tests for these problems, making it a valuable dataset for our study.

The choice of FIXEVAL offers two main benefits. First, its large size allows for targeted sampling or filtering to suit our research needs. Second, it features complex bugs, in contrast to datasets like QuixBugs [12], which typically contain simpler, single-line bugs. FIXEVAL's bugs often involve logic errors requiring multiple lines of code to be corrected. Preliminary testing with QuixBugs showed ChatGPT's effectiveness in solving simpler problems, leading us to choose a dataset that presents a more challenging environment to assess ChatGPT's debugging skills more rigorously. These complex cases are more representative of real-world scenarios in the software industry, making our research especially relevant.

The FIXEVAL dataset includes various bug types, such as Wrong Answer, Time Limit Exceeded, and Memory Limit Exceeded. Our research focuses on scenarios where code executes but fails specific tests due to logical errors. We investigate whether additional context can enhance ChatGPT's logical reasoning and debugging performance. We specifically select 110 instances with the Wrong Answer classification for analysis.

To evaluate the repaired program, we utilize the dataset's unit tests to check the functional correctness of the repaired codes. If the modified code passes all tests, we consider the debugging successful, indicating that the original bugs have been resolved.

B. ChatGPT Configurations

In this study, we employ GPT-4, a state-of-the-art multimodal model that accepts both text and image inputs to generate text outputs. GPT-4 stands out for its ability to tackle complex problems with unprecedented accuracy, surpassing previous models through its expansive general knowledge base and enhanced reasoning skills. It features a maximum context window of 128,000 tokens, providing a broad canvas for analysis and generation. Additionally, GPT-4 has been trained on data up to April 2023, ensuring its responses are informed by the most recent and relevant information available.

C. Experimental Settings

When interacting with GPT-4 via OpenAI APIs, we select "gpt-4-turbo-preview" as the model. The request messages are structured to include two types of content: one for the "system" role and another for the "user" role. The "system" role content outlines general instructions, detailing the input required from the user and the expected output from GPT-4. Meanwhile, the "user" role content provides specific details about the buggy code and its context.

D. Experimental Prompts

To investigate the various types of contextual information mentioned earlier, we designed experiments using specific prompts:

1) **Basic Prompt (BP):** We tested two foundational prompts. The first prompt (BP 1) provides minimal information regarding the presence or type of bugs within the program. It instructs as follows:

Examine the provided program code carefully. If the program contains any bugs, please output 'yes' followed by the corrected version of the program that fixes the bug(s). If the program does not contain any bugs, simply output 'no'.

Program [A program that has bugs]

Given that programmers often recognize the presence of bugs when outputs deviate from expectations, particularly in cases classified as "Wrong Answer" due to failure in some unit tests, we incorporate this scenario into our prompts, reflecting a common real-world context. The second prompt (BP 2) is structured as follows:

Review the provided program, which is known to produce incorrect outputs for certain test cases due to bugs. Your task is to identify and fix these bugs. Please return the corrected version of the program.

Buggy Program [A program that has bugs]

2) **Basic Prompt with Problem Description (BPP)**: Along with basic instructions, we add more information about the code being tested. We focus on explaining the code's purpose, specifically the problem it aims to solve. Therefore, we include a detailed problem description alongside the input/output format specifications. An illustration of such a prompt is shown below.

You will receive a program that is currently producing incorrect answers for certain test cases. Along with the buggy program, a detailed description of the problem it is intended to solve, as well as the specific constraints and formats for the input and output, will be provided. Your task is to identify the bugs in the program and correct them. Please pay close attention to the following sections:

Buggy Program [A program that has bugs]

Problem Statement [Description of the problem that the program is designed to solve]

3) **Basic Prompt with the Error Test Cases (BPE)**: This prompt focuses on presenting error test cases where the code snippets do not perform as expected. An example of such a prompt is illustrated below.

You will receive a program that is currently producing incorrect answers for certain test cases. Along with the buggy program, the examples of these test cases will be provided. Your task is to identify the bugs in the program and correct them. Please pay close attention to the following sections:

Buggy Program [A program that has bugs]

Error Test Cases: Here, you will find the test cases where the program fails, including:

Input 1 [The first input where the program fails]
Expected Output 1 [The correct output for input 1]

Input 2 [The second input where the program fails]
Expected Output 2 [The correct output for input 2]

4) **Basic prompt with Correct and Error Test Cases (BPCE)**: To address the concern that fixing code based solely on error test cases might result in a solution that only works for those specific scenarios and fails on previously successful ones, we explore the impact of including both correct and incorrect test cases on improving model performance. This approach is illustrated below.

You will receive a program that inconsistently produces results: it generates incorrect answers for certain test cases, while correctly solving others. Along with the buggy program, you will find examples of both the test cases where the program fails and where it succeeds. Your task is to identify the bugs in the program and correct them. Please pay close attention to the following sections:

Buggy Program [A program that has bugs]

Error Test Cases: Here, you will see where the program does not work as expected, including:

Input 1 [The input where the program fails]

Expected Output 1 [The correct output for input 1]

Correct Test Cases: These are test cases where the program performs correctly, featuring:

Input 2 [The input where the program succeeds]

Output 2 [The correct output for input 2]

5) **Basic Prompt with Problem Description and Test Cases (ALL)**: The final prompt we explore includes complete context information: the problem description along with all test cases, both functioning and malfunctioning. This comprehensive approach aims to give the model a full understanding of the program's objectives and the specific issues it faces. By presenting a detailed problem statement and examples of both successful and failing test cases, we facilitate the model's ability to diagnose and resolve issues, ensuring it can adapt the code to work across all scenarios. An example of this prompt type is displayed below.

You will receive a program that is currently producing incorrect answers for certain test cases. In addition, a description of what it's supposed to do, and details about how it should handle inputs and outputs will be given. We'll also provide you with examples where the program fails and where it works as expected. Your task is to identify the bugs in the program and correct them. Please pay close attention to the following sections:

Buggy Program [A program that has bugs]

Problem Statement [Description of the problem that the program is designed to solve]

Error Test Cases: Here, you will see where the program does not work as expected, including:

Input 1 [The input where the program fails]

Expected Output 1 [The correct output for input 1]

Correct Test Cases: These are test cases where the program performs correctly, featuring:

Input 2 [The input where the program succeeds]

Output 2 [The correct output for input 2]

TABLE I
PERCENTAGE OF CORRECTED BUGS BASED ON DIFFERENT PROMPTS

BP 1	BP 2	BPP	BPE	BPCE	ALL
16.7%	25.0%	75.0%	33.3%	58.3%	83.3%

IV. RESULTS AND ANALYSIS

If the corrected program passes all the test cases, we classify the bug as fixed. To calculate the accuracy, we use the ratio of corrected programs to the total number of programs in the dataset. Table I displays the accuracy for each type of prompt.

1) **Basic Prompt (BP)**: The accuracy results for the Basic Prompt (BP), in both version 1 (BP 1) and version 2 (BP 2), were relatively low, at 16.7% and 25%, respectively. These outcomes align with our initial expectations. For bugs categorized as “Wrong Answer” simply providing the buggy program without any context information makes it challenging for the model to pinpoint and accurately correct the bug. Our observations indicate minimal difference between the two versions of the Basic Prompt. Typically, the model requests additional context information, as illustrated in the following example:

The program logic seems correct for a specific purpose, but there's a potential for improvement in clarity and efficiency, depending on the intended use case. However, without further context about what this program is intended to calculate or achieve, it's challenging to suggest meaningful improvements or identify bugs beyond syntactical errors or logic inconsistencies.

We also observed instances where the model successfully fixed programs without additional context information. An example of such a case is listed below:

```
s = input()
if s[0] == s[-1]:
    print('Yes')
else:
    print('No0')
```

Even without context information, the model managed to generate the corrected program as follows:

```
s = input()
if s[0] == s[-1]:
    print('Yes')
else:
    print('No')
```

In such instances, it's understandable why the basic prompt proves effective. The code itself offers sufficient context information, enabling the model to deduce the required output format. Specifically, the model learns to produce the correct output format by changing the response from ‘No0’ to ‘No’, aligning it with the expected output format. This demonstrates the model's capability to adapt and correct based on the intrinsic clues provided within the code.

2) **Basic Prompt with Problem Description (BPP)**: The results show that including the problem description context significantly enhances the model's performance, even exceeding our expectations. This improvement may be attributed to

the simplicity of the problem's logic. With a clear understanding of the goal, the model can deduce much of the coding logic required. Interestingly, we observed that with the problem description, the model could correctly fix some programs that it was unable to fix with only test cases. An example of such a problem description and the buggy program are provided below:

Problem Statement: *A string of length 6 consisting of lowercase English letters is said to be coffee-like if and only if its 3-rd and 4-th characters are equal and its 5-th and 6-th characters are also equal. Given a string S, determine whether it is coffee-like.*

Constraints *S is a string of length 6 consisting of lowercase English letters.*

Input *Input is given from Standard Input in the following format: S*

Output *If S is coffee-like, print Yes; otherwise, print No.*

Buggy Program:

```
s = input()
if s[3:] == 'ffee':
    print('Yes')
else:
    print('No')
```

Based on the problem description, the model successfully identified and fixed the bug, providing explanations for its analysis. The analysis from ChatGPT is detailed as follows:

To correct the buggy program according to the problem statement, we need to adjust the logic to check whether the 3rd and 4th characters, as well as the 5th and 6th characters, are equal. The given program mistakenly checks if the substring from the 4th character onwards matches 'ffee', which is not aligned with the problem statement.

However, when only test cases are provided without accompanying problem descriptions, the model is required to deduce the program's intended functionalities based solely on the provided input/output pairs. This requirement hampers the model's ability to generalize effectively from the specific examples given, complicating the task of effectively repairing the program. The infer by the model is as follows:

The buggy program is intended to check if a given string ends with 'ffee'. However, it erroneously produces “Yes” for several inputs that do not end with 'ffee', indicating a misunderstanding in the logic of string comparison. The program mistakenly checks if the substring starting from the fourth character matches 'ffee', which is not an accurate way to determine if a string ends with a specific sequence. This leads to incorrect results for various test cases.

Based on the test cases, the model concludes that the task involves checking if a given string ends with “ffee”. While this deduction aligns well with the provided test cases, it does not accurately reflect the actual task the program is intended to perform.

When the problem description and input/output requirements are directly provided, the process becomes more straightforward for the model. It eliminates the need for the

model to deduce the program's requirements, thereby simplifying the debugging process. This direct approach enables the model to more easily identify and resolve the bug.

Table I reveals that the Basic Prompt with Problem Description (BPP) yields higher accuracy (75%) compared to the Basic Prompt with Error Test Cases (BPE) at 33.3% and the Basic Prompt with Correct and Error Test Cases (BPCE) at 58.3%. This demonstrates the superiority of utilizing a detailed problem description over relying solely on test cases, accounting for BPP's enhanced performance against both BPE and BPCE.

3) Basic Prompt with the Error Test Cases (BPE):

The results indicate that providing only error test cases does not significantly improve the accuracy of the model's bug-fixing ability. For instances that were not properly corrected, we analyzed the accuracy rates before and after attempts at fixing. We discovered that, some data points maintained the same accuracy percentage as the buggy program, while others exhibited even lower accuracy than the original buggy code.

Several issues arise in this context: First, error test cases alone do not adequately convey the program's inherent requirements, leading the model to possibly introduce incorrect logic. Consequently, the amended program might still fail to address the error test cases. Furthermore, even when the corrected code resolves the specific error test cases, it might not perform correctly across the broader set of test cases, failing on those it previously succeeded. This challenge highlights the complexity of ensuring that fixes are comprehensive and do not inadvertently introduce new issues.

However, in certain instances, presenting corner test cases - where the program fails - enables the model not just to deduce the program's intended task but also to understand the reasons behind the failure of these specific cases and subsequently repair them. An illustrative example of this capability is provided below, along with the problem description and the buggy program.

Problem Statement *Takahashi, Aoki and Snuke love cookies. They have A, B and C cookies, respectively. Now, they will exchange those cookies by repeating the action below: Each person simultaneously divides his cookies in half and gives one half to each of the other two persons. This action will be repeated until there is a person with odd number of cookies in hand. How many times will they repeat this action? Note that the answer may not be finite.*

```
a,b,c = map(int,input().split())

if a==b==c:
    print(-1)
else:
    ans = 0
    while True:
        if a % 2 == 1 or b % 2 == 1 or c % 2 == 1:
            break
        a,b,c = (b/2+c/2), (a/2+c/2), (a/2+b/2)
        ans += 1
    print(ans)
```

This above program fails to handle a specific test case shown as following:

Input: 1 1 1

Expected Output: 0

Based on the buggy program and the provided error test case, the model's analysis is as follows:

The buggy program you've shared is intended to find the number of operations required to make at least one of the three numbers odd by repeatedly replacing each number with the half-sum of the other two numbers. It seems like the logic to handle the case when all three numbers are odd from the beginning is incorrect.

In the provided program, when a, b, and c are all equal (and implicitly odd as per the error case provided), it prints -1 instead of 0, which is the expected output according to the error cases you've provided. This is incorrect for the case when all three numbers are odd because it's already in a state where the condition to break the loop (if a % 2 == 1 or b % 2 == 1 or c % 2 == 1) is met. Therefore, the loop does not need to execute, and the program should immediately print 0 because no operations are needed to make any of the numbers odd—they already are.

Key Corrections:

1. The initial condition if a==b==c: was removed because it's not just about all numbers being equal but whether any of them is odd to avoid performing any operations.

2. The condition to print 0 was updated to check if any of the numbers a, b, or c is odd before entering the while loop. This directly addresses the error case where all inputs are odd (e.g., 1 1 1), ensuring the expected output is 0. Inside the loop, the calculation for a, b, and c is preserved as in your original code. This part did not have a logical error concerning the problem described.

Based on the analysis, the model provides the corrected program below:

```
a,b,c = map(int,input().split())

# Correctly handle the case when all three
# numbers are odd from the beginning
if a % 2 == 1 or b % 2 == 1 or c % 2 == 1:
    print(0)
else:
    ans = 0
    while True:
        a,b,c = (b+c)/2, (a+c)/2, (a+b)/2
        ans += 1
        if a % 2 == 1 or b % 2 == 1 or c % 2 == 1:
            break
    print(ans)
```

From the model's analysis, two key insights emerge. First, the model successfully identifies the program's objective: to determine the number of operations needed to make at least one of the three numbers odd by continuously replacing each number with the half-sum of the other two. Second, it pinpoints the reason the given test case fails: the program erroneously outputs -1 instead of the correct 0 when the variables a, b, and c are all equal and odd. Consequently, the model proposes an accurate fix for this issue.

From the given example, it's evident that in certain scenarios, supplying only the error test case and the buggy program

Programs Fixed Using BPCE # Programs Fixed Using BPP

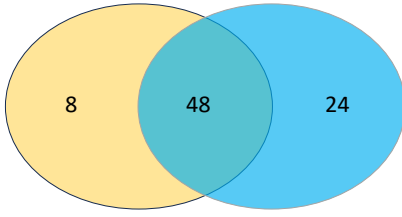


Fig. 1. Venn Diagram of Programs Fixed Using Basic Prompt with Correct and Error Test Cases (BPCE) Versus Basic Prompt with Problem Description (BPP).

enables the model to precisely identify and rectify the bug. Moreover, this approach allows for a more efficient use of input tokens by limiting the number of test cases presented, as opposed to including both correct and error test cases.

4) **Basic Prompt with Correct and Error Test Cases (BPCE)**: Including both correct and error test cases is expected to enhance the model's performance compared to only using error test cases (BPE). By presenting both types of test cases, the model achieves better generalization for certain problems, indicating an improvement (from 33.3% to 58.3%) in its ability to diagnose and fix issues accurately. However, as previously discussed, when relying solely on test cases, the model must infer the program's requirements, which can be a challenging extra step. Consequently, the model's performance with BPCE does not match the effectiveness seen with BPP (Basic Prompt with Problem Description). With BPCE the accuracy is 58.3% while with BPP the accuracy is 75%.

5) **Basic Prompt with Problem Description and Test Cases (ALL)**: While both the BPP (Basic Prompt with Problem Description) and BPCE (Basic Prompt with Correct and Error Test Cases) approaches successfully address some of the same bugs, each also uniquely resolves certain issues not fixed by the other. Figure 1 illustrates the overlap in bugs fixed by BPP and BPCE, alongside the distinct bugs remedied by each prompt independently. By incorporating both problem description and test cases, the model effectively addresses a combined set of bugs identified by BPP and BPCE prompts. Consequently, this comprehensive approach results in the highest performance among all tested contexts. The findings are consistent with our expectations, demonstrating the value of providing complete context information for optimal bug-fixing efficiency.

V. RELATED WORK

Debugging plays a crucial role in software security, thus there has been a large body of research on this topic. Our discussion is not intended to be exhaustive, so the techniques based on binary analysis [15], [16] and code repositories mining [17] are beyond the scope. Instead, we mainly review the previous work regarding GPT-based code review.

First, the bug fixing capability of ChatGPT are preliminarily investigated by existing work [18]–[20] using the benchmark dataset QuixBugs [12]. Through comparative study, [19] and

[18] both show that ChatGPT's bug fixing performance surpasses the baseline approaches, i.e., a standard automated program repair (APR) system [21] and Codex [22]. In addition, the findings of [18] indicate the bug fixing performance of ChatGPT is competitive to CoCoNuT [8], a deep learning-based counterpart. However, the size of QuixBugs is relatively small, which merely consists of 40 buggy programs in Python and Java. Furthermore, the programs in QuixBugs are implementations of classical computer science algorithms, thus being far away from real cases in practical development. More importantly, there are indications suggesting that ChatGPT already has knowledge about QuixBugs, which leads aforementioned research to be less convincing [23].

Second, how well ChatGPT performs in reviewing and refining code has been explored by prior work [23]–[26]. Fu et al. [24] experimented with ChatGPT's software vulnerability detection capabilities on C/C++ functions. They found that ChatGPT failed to accurately discover vulnerabilities in source code, and also could not generate correct repair patches for the vulnerable functions. Ribeiro et al. [25] focus on type errors in OCaml programs, and leverage GPT to automatically debug. Guo et al. [26] empirically investigated the potential of ChatGPT in code refinement tasks and examined the impacts of different prompts. Their evaluation result demonstrates that ChatGPT outperforms CodeReviewer [9].

Lastly, the potential applicability of GPT-based code review is examined in the scenario of education. Crandall et al. [27] implement an automatic code review application based on GPT to inspect programming assignments submitted by students in entry-level courses. Participants in this study generally consider the auto-generated feedback to be helpful. In addition, Jalil et al. [28] investigate the effectiveness of ChatGPT in answering typical questions within a popular software testing textbook. Consequently, adopting GPT for discovering defects in code from college students and assisting in the training of qualified software developers shows promise.

VI. CONCLUSION AND FUTURE WORK

In our research, we explore how different types of contextual information influence ChatGPT's bug-fixing capabilities. The initial findings reveal that without any context, the model's performance is notably low, at 16.7% and 25%. Drawing inspiration from the human debugging process, where understanding the program's objective and analyzing unit test cases are crucial, we examine the influence of these two types of contextual information and their combined effect. The study demonstrates that incorporating the problem description significantly boosts the model's accuracy to 75%. Similarly, including test cases raises the accuracy to 58.3%. Moreover, integrating both problem description and test cases results in the highest accuracy, reaching 83.3%. These outcomes align with our expectations, confirming that contextual information is instrumental in enabling the model to accurately identify and rectify bugs.

In this study, we specifically focused on the "Wrong Answer" bug type. Moving forward, we aim to broaden our

research to assess how contextual information influences the model's ability to address a variety of bug types, including "Time Limit Exceeded" among others. Additionally, we plan to explore the bug-fixing capabilities of other large language models, such as Gemini, to gain a comprehensive understanding of the strengths and limitations across different AI-driven debugging models. This expansion of our research scope will provide deeper insights into the effectiveness of contextual information and the comparative performance of various LLMs in software debugging scenarios.

REFERENCES

- [1] A. Arcuri and X. Yao, "A novel co-evolutionary approach to automatic software bug fixing," in *IEEE Congress on Evolutionary Computation*, 2008, pp. 162–168.
- [2] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "GenProg: A generic method for automatic software repair," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 54–72, 2011.
- [3] A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, J. Klein, M. Monperrus, and Y. Le Traon, "Fixminer: Mining relevant fix patterns for automated program repair," *Empirical Software Engineering*, vol. 25, pp. 1980–2024, 2020.
- [4] H. Samimi, M. Schäfer, S. Artzi, T. Millstein, F. Tip, and L. Hendren, "Automated repair of html generation errors in php applications using string constraint solving," in *34th International Conference on Software Engineering (ICSE)*, 2012, pp. 277–287.
- [5] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis," in *35th International Conference on Software Engineering (ICSE)*, 2013, pp. 772–781.
- [6] The LLVM Project, "Clang static analyzer," <https://clang-analyzer.llvm.org/>, accessed: 2024-02-01.
- [7] J. Bader, A. Scott, M. Pradel, and S. Chandra, "Getafix: Learning to fix bugs automatically," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–27, 2019.
- [8] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, "Coconut: combining context-aware neural translation models using ensemble for program repair," in *29th ACM SIGSOFT international symposium on software testing and analysis*, 2020, pp. 101–114.
- [9] Z. Li, S. Lu, D. Guo, N. Duan, S. Jannu, G. Jenks *et al.*, "Automating code review activities by large-scale pre-training," in *30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 1035–1047.
- [10] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, "Improving language understanding by generative pre-training," OpenAI, Technical Report, 2018.
- [11] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," OpenAI, Technical Report, 2019.
- [12] D. Lin, J. Koppel, A. Chen, and A. Solar-Lezama, "QuixBugs: A multilingual program repair benchmark set based on the quixey challenge," in *Proceedings Companion of the 2017 ACM SIGPLAN international conference on systems, programming, languages, and applications: software for humanity*, 2017, pp. 55–56.
- [13] M. M. A. Haque, W. U. Ahmad, I. Lourentzou, and C. Brown, "Fixeval: Execution-based evaluation of program fixes for programming problems," in *IEEE/ACM International Workshop on Automated Program Repair (APR)*, 2023, pp. 11–18.
- [14] R. Puri, D. S. Kung, G. Janssen, W. Zhang, G. Domeniconi, V. Zolotov, J. Dolby, J. Chen, M. Choudhury, L. Decker, V. Thost, L. Buratti, S. Pujar, S. Ramji, U. Finkler, S. Malaika, and F. Reiss, "Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks," 2021.
- [15] L. Luo, Q. Zeng, B. Yang, F. Zuo, and J. Wang, "Westworld: Fuzzing-assisted remote dynamic symbolic execution of smart apps on IoT cloud platforms," in *Proceedings of the 37th Annual Computer Security Applications Conference*, 2021, pp. 982–995.
- [16] F. Zuo, X. Li, P. Young, L. Luo, Q. Zeng, and Z. Zhang, "Neural machine translation inspired binary code similarity comparison beyond function pairs," in *Proceedings of the 26th Network and Distributed Systems Security (NDSS) Symposium*, 2019.
- [17] F. Zuo and J. Rhee, "Vulnerability discovery based on source code patch commit mining: a systematic literature review," *International Journal of Information Security*, vol. 23, pp. 1513–1526, 2024.
- [18] D. Sobania, M. Briesch, C. Hanna, and J. Petke, "An analysis of the automatic bug fixing performance of ChatGPT," in *IEEE/ACM International Workshop on Automated Program Repair (APR)*, 2023, pp. 23–30.
- [19] M. C. Wuisang, M. Kurniawan, K. A. W. Santosa, A. A. S. Gunawan, and K. E. Saputra, "An evaluation of the effectiveness of openai's chat-GPT for automated python program bug fixing using QuixBugs," in *2023 International Seminar on Application for Technology of Information and Communication (iSemantic)*, 2023, pp. 295–300.
- [20] T. Do Viet and K. Markov, "Using large language models for bug localization and fixing," in *12th International Conference on Awareness Science and Technology (ICAST)*, 2023, pp. 192–197.
- [21] H. Ye, M. Martinez, T. Durieux, and M. Monperrus, "A comprehensive study of automatic program repair on the QuixBugs benchmark," *Journal of Systems and Software*, vol. 171, p. 110825, 2021.
- [22] J. A. Prenner, H. Babii, and R. Robbes, "Can openai's codex fix bugs? an evaluation on QuixBugs," in *IEEE/ACM International Workshop on Automated Program Repair (APR)*, 2022, pp. 69–75.
- [23] Q. Zhang, T. Zhang, J. Zhai, C. Fang, B. Yu, W. Sun, and Z. Chen, "A critical review of large language model on software engineering: An example from ChatGPT and automated program repair," *arXiv preprint arXiv:2310.08879*, 2023.
- [24] M. Fu, C. Tantithamthavorn, V. Nguyen, and T. Le, "ChatGPT for vulnerability detection, classification, and repair: How far are we?" in *Asia-Pacific Software Engineering Conference (APSEC)*, 2023.
- [25] F. Ribeiro, J. N. C. de Macedo, K. Tsushima, R. Abreu, and J. Saraiva, "GPT-3-powered type error debugging: Investigating the use of large language models for code repair," in *Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering*, 2023, pp. 111–124.
- [26] Q. Guo, J. Cao, X. Xie, S. Liu, X. Li, B. Chen, and X. Peng, "Exploring the potential of ChatGPT in automated code refinement: An empirical study," in *46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–13.
- [27] A. S. Crandall, G. Sprint, and B. Fischer, "Generative pre-trained transformer (GPT) models as a code review feedback tool in computer science programs," *Journal of Computing Sciences in Colleges*, vol. 39, no. 1, pp. 38–47, 2023.
- [28] S. Jalil, S. Rafi, T. D. LaToza, K. Moran, and W. Lam, "ChatGPT and software testing education: Promises & perils," in *IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2023, pp. 4130–4137.