Can Large Language Models Mitigate Challenges in Reproducing Issues Reported in Stack Overflow Questions? An Exploratory Study

Daniel Ibekwe
University of Saskatchewan, Canada
qie647@usask.ca

Kristian Manaloto
University of Saskatchewan, Canada
kcm662@usask.ca

Ali Udi
University of Saskatchewan, Canada
vky134@usask.ca

ABSTRACT

Reproduction is important in being able to solve reported issues on Stack Overflow. The ability to reproduce issues is strongly related with receiving an accepted solution. However, reproducing some issues is not easy, becoming a significant time investment for developers. This study addresses the challenge of unreproducible issues, whether due to incomplete code, missing context, or unclear requirements. This problem leads to inefficient resolution of programming issues, and can detract from community engagement. This study aims to investigate the power of Large Language Models to help reproduce these issues. Specifically we will look at (1) to what extent LLMs can generate missing parts of code to make snippets executable and reproduce issues, and (2) how effectively LLMs can identify and recreate the required environment for reproducing issues. We first collected a data set published by Mondal and Roy [4]. that contains 87 Java and 66 Python related issues that could not be reproduced. We then conduct our experiments by first, removing parts of the issue that do not relate to the issue the user is reporting. After, we craft a prompt, specifically asking for the issue to be reproduced and not solved, and we send the prompt along with the issue to the LLM. After receiving the response from the LLM, we manually test to see if the issue is reported. Our initial findings found that 50% of the issues could be reproduced. The remaining issues involved third-party APIs or unprovided context. This study suggests that LLMs hold promise for enhancing the reproducibility of issues on Stack Overflow by automating the generation of necessary code and identifying environmental dependencies. This approach can lead to faster problem resolution and community engagement by reducing the time and effort required to reproduce issues.

CCS CONCEPTS

• **Software and its engineering** → *Software repository mining*; *Software maintenance and evolution*; *Practitioners' perspective*; *Software maintenance tools.*

CMPT 470, April, 2025, Saskatoon, Canada

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

KEYWORDS

Stack Overflow, issue reproducibility, code segments, reproducibility challenges, LLMs, prompt engineering

ACM Reference Format:
Daniel Ibekwe, Kristian Manaloto, and Ali Udi. 2025. Can Large Language Models Mitigate Challenges in Reproducing Issues Reported in Stack Overflow Questions? An Exploratory Study. In *Advanced Software Engineering (Winter 2025), April 2025, Saskatoon, Canada.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Stack Overflow is one of the most popular platforms for users to post their programming problems. Other users can read and attempt to answer the mentioned issues. To solve issues, users will usually try to reproduce these issues so that they can pinpoint exactly where the problem could be (Mondal et al. [3]). This is where the problem lies. Often, code segments included with the Stack Overflow questions may not easily be reproducible, and the developer may spend a lot of time trying to reproduce the issue instead of solving the issue.

Previous studies (Mondal and Roy [4]) have shown the effect reproducible issues can have on receiving an accepted solution. When an issue can be reproduced, it is three times more likely (74.4% vs 20.7%) for Java questions to receive an accepted answer and two times (58. 1% vs 25. 7%) for Python. This study used a data set of 400 Java questions and 400 Python questions as its data set. In this study, however, issue reproduction was done manually by a team of developers. The team was unable to reproduce 87 (21.8%) of Java questions and 74 (18. 5%) of Python questions and marked them 'irreproducible'.

Traditionally, users often need to request additional code, context, or clarification from the original poster. Although this method can eventually lead to a more complete and reproducible code snippet, it introduces delays in obtaining a working solution. Back-and-forth communication can hinder rapid problem solving and may discourage users from seeking immediate help.

In this study, we are looking to take advantage of the power of Large Language Models to see if they can enhance the code snippets and help reproduce these 'irreproducible' issues. We are using the same dataset from the mentioned study. For each irreproducible issue, we will follow these three Steps. First, we will highlight the issue of the question by reading the text and the provided code segment. We want to remove the 'fluff' from a question. By removing unnecessary or conversational content from a question, such as greetings ("Hello"), statements of skill level ("I am new to Python"), or general requests ("can someone provide a fix?"), to

minimize potential biases in the LLM's responses. Second, once we have the issue highlighted, we will prompt the LLM to specifically reproduce the issue, along with the highlighted issue and the code segment . Third, we will try to run the code given to us by the LLM in our own environment and see if the problem was successfully reproduced. Additionally, if the issue was not reproduced but we think we can push the LLM in the right direction, we will do so to an extent. Our study looks to encourage use of automated tools to improve the code segments on Stack Overflow. This approach could cut the process of reproducing the issue and decrease the time it takes to get a quality solution. In this study, we made two major contributions by answering two research questions as follows.

RQ1 To what extent can LLMs generate missing parts of code (e.g., classes, methods) to make code snippets executable and reproduce issues? ***This research question is important because developers often encounter incomplete bug reports or code examples. Understanding how well LLMs can fill in missing elements helps assess their reliability in debugging and code recovery tasks.***

RQ2 How effectively can LLMs identify and recreate the required environment (e.g., libraries or configurations) for reproducing issues? ***Recreating the exact environment is critical for reproducing and fixing software bugs. Investigating LLMs' ability to infer dependencies and configurations can reveal their utility in streamlining issue resolution and minimizing manual setup errors.***

2 SIGNIFICANCE OF THE TOPIC

Reproducibility is essential for verifying that solutions work as intended. When an issue can be reproduced, it encourages contributors to provide more detailed and accurate responses. This, in turn, enhances the quality and trustworthiness of the advice on Stack Overflow.

Improving reproducibility has the potential to significantly boost community engagement on Stack Overflow. By reducing the time developers spend trying to recreate issues, they can focus on solving problems more quickly and effectively. This can lead to faster, higher-quality responses and a more active and robust community, ultimately driving more traffic to the site.

The motivation behind this study is that when an issue can be reproduced, it's much more likely to get solved. We want to make every problem easy to reproduce so that solutions can be found more quickly. By using AI to help recreate issues from code snippets—especially those that humans have struggled with—we can make the process of reproducing problems much less frustrating and more effective for everyone.

This research has the potential to drive meaningful advancements across multiple domains. In online developer communities, such as Stack Overflow, it could enhance tools that automatically detect and improve reproducibility issues, ensuring higher-quality contributions and more reliable solutions. In the area educational resources, the findings could help provide more dependable code examples and tutorials, leading to improved comprehension and skill development for learners. Additionally, in software development tools, integrating reproducibility checks within integrated development environments (IDEs) or version control systems could streamline the debugging process and foster more efficient collaboration. By addressing these areas, the research contributes to the broader goal of improving software quality and developer productivity.

3 METHODOLOGY

3.1 Methods & Techniques

This study adopts a multi-step approach to assess how large language models (LLMs) can mitigate challenges in reproducing reported issues:

(1) **Data Cleaning:** Extraneous or "fluff" content is removed from the questions to ensure a clear focus on the reported issue.
(2) **Prompt Engineering:** We develop and refine prompts that instruct the LLM to reproduce the issue accurately. The prompts incorporate both the refined question and its associated code snippet.
(3) **Reproduction Testing:** The output generated by the LLM is executed within integrated development environments (IDEs) such as IntellJ and PyCharm to test its ability to recreate the reported issue.
(4) **Outcome Recording:** Each experimental outcome is systematically documented for subsequent analysis.

These techniques are particularly suitable for our research objectives as they combine qualitative refinements with quantitative evaluation, ensuring that the LLM's performance is both measurable and reproducible.

3.2 Data set Collection

Our study expands on the previous work done on repducibility on Stack Overflow by Mondal et al.[3] [4]. This previous research on Stack Overflow reproducibility, involved manually attempts to reproduce over 400 Java and 400 Python issues reported on Stack Overflow. Mondal et al.'s investigation resulted in a classification of these issues into "Irreproducible" or "Reproducible" with reproducible issues being further divided into "Major modification", "Minor modification.

(1) **Minor Changes**: Issues that require small modifications to be resolved.
(2) **Major Changes**: Issues that need significant alterations for a working solution.
(3) **Irreproducible**: Issues that cannot be reliably recreated or diagnosed based on the provided information.

Our current research is focused on the issues labeled as "Irreproducible" by this study. These issues posed the greatest challenge for human developers seeking to understand and resolve the reported problem.

3.3 Prompt Design and Testing for Code and Environment Completion

A big challenge behind this was to create a prompt that we could use across all LLMs to ensure a structured experiment. We initially started with a basic prompt and continued to develop as fit.

I am going to send a stack overflow issue, I want you to give me the java/python files necessary that recreates the issue, the

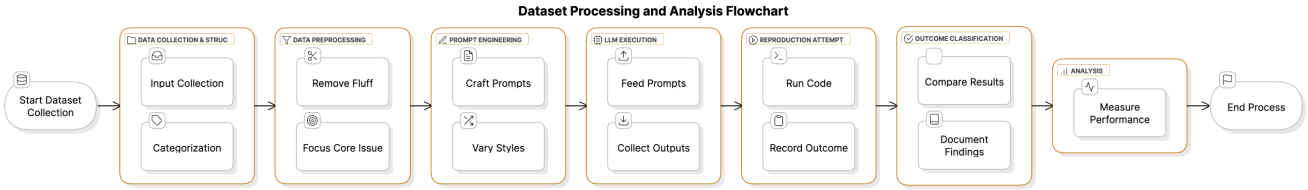

Figure 1: Methodology workflow: LLM reproduction of software issues

code files you give should be able to be executed. Below is the issue:

As we tested this prompt to verify it works, we ran into issues where the LLMs would still try to solve the issue. We had to specify that we want the issue recreated and not solved:

I am going to send a Stack Overflow issue. I want you to give me the Java/Python files necessary to recreate the issue specifically (I don't want the code to be solved). The code files you give should be able to be executed. Below is the issue:

To verify our prompt, we tested our LLM prompts using four different Stack Overflow for both Java and Python, including two different minor modifications (such as small code and syntax issues) and two major modifications (involving more complex logic, more code to add) to ensure coverage of issues. We used these issues for verification because they had already been solved by developers before. Additionally we tested issues that used libraries, multiple files, and external methods to assess the ability of LLM's to handle real world coding problems.

As we test with more questions, the prompts would also evolve over time. We would take the same prompt and problem and give it the same problem and would see if we got the desired result from the three LLMs we tested it on. Any evolution of the prompt will be be the new prompt going forward, ensuring a controlled experiment.

3.4 LLM-Based Code Generation and Environment Reconstruction

We begin with a Stack Overflow issue that includes a code snippet, and we remove everything except the important parts of the question. A more concise prompt will take away any factors that can throw the LLM off. Also, some users tend to ask for a fix within the post itself, which could lead to the LLM providing one.

It is important that we clean the issue, to minimize unnecessary information that could distract the LLM. Research done by Shi et al. [7] investigates the distractibility of LLM's, finding that there is a decrease in performance when adding small amounts of irrelevant information.

We then combine this trimmed issue with our prompt, and send it to the LLM. After prompting the LLM, we then follow its steps to recreate the issue. This is Deepseek's reply, but GPT4o and CoPilot followed similar formats of providing the files along with the steps to reproduce the issue.

Java: Resetting all values in the program

Asked 15 years, 11 months ago Modified 15 years, 11 months ago Viewed 3k times

1

I am working on this program where at the end of the game I ask the user if they want to play again. If they say yes, I need to start a new game. I made a restart() method:

```
public void restart(){
    Game g = new Game();
    g.playGame();
}
```

However when I call this method some of the values in my program stay at what they were during the previous game.

Is there a game to just clear everything and create an new instance of the game with all the default values?

Figure 2: Example of issue from our database

DeepSeek's attempt at recreating this issue.

```
public class Game {
    private static int score = 0;  // Problem:
        persists between games
    private int health = 100;      // Works
        correctly: resets each game

    public void playGame() {
        // Simulate gameplay
        score += 20;
        health -= 30;

        System.out.println("Game ended - Score: "
            + score + ", Health: " + health);
    }

    public void restart() {
        System.out.println("\nRestarting game...")
            ;
        Game newGame = new Game();
        newGame.playGame();
    }

    public static void main(String[] args) {
        // Initial game
        System.out.println("--- First Game ---");
        Game game = new Game();
        game.playGame();  // Score: 20, Health: 70

        // First restart
        game.restart();    // Score: 40, Health:
            70

        // Second restart
        game.restart();    // Score: 60, Health:
            70
    }
}
```

In this case we were given a Game class, where a static int score is the cause of a persistent value, along with a non persistent value int health for comparison. For every time playGame() is called, score will increase by 20, and health decreases by 30. and a similar restart function as provided in the original post.

Experiments are structured around a cycle of manual testing and iterative prompt refinement. The evaluation metrics include:

(1) **Reproducibility Rate:** The frequency with which the LLM-generated solution successfully recreates the reported issue.
(2) **Code Quality:** The robustness and efficiency of the code output generated by the LLM.

This design facilitates continuous improvement by providing systematic feedback to fine-tune the LLM prompts.

The study utilizes advanced large language models, including ChatGPT, Deepseek, and CoPilot (built on GPT4), to support its methodology. Additionally, it leverages industry-standard integrated development environments (IDEs) such as IntelliJ and PyCharm, ensuring a robust and efficient development workflow. Version control is managed through Git, a widely adopted tool known for its reliability in both academic research and industry practice. These technologies have been carefully selected for their proven effectiveness and widespread use, enhancing the study's methodological rigor.

3.5 Execution and Reproducibility Validation

It is crucial to recognize this that this step directly addresses the core concept of issue reproducibility. This step of our methodology mirrors [4] manual analysis where they attempt to reproduce the reported issue in an IDE after understanding the problem. We are aiming for "complete agreement between the reported issues and the investigated issues". We directly compare the output of any error messages from the LLM's code with the original posts description. Upon success or failure, we will mark the experiment with a yes or no.

In this example case, the output was.

Java Example

```
--- First Game ---
Game ended - Score: 20, Health: 70

Restarting game...
Game ended - Score: 40, Health: 70

Restarting game...
Game ended - Score: 60, Health: 70
```

We can see that after playing the game once, and restarting twice, the problematic persistent value score is never reset to 0, allowing it to reach a value of 60, while the properly resetting health stays consistent. Matching the users reported error of:

"However when I call this method some of the values in my program stay at what they were during the previous game."

We would mark this issue as a success and repeat it for the rest of the issues in our database.

3.6 Outcomes, Hypothesis, & Implications

We anticipate that our framework will enhance the reproducibility of programming issues on Stack Overflow by reducing the iterative clarifications typically required. The approach is expected to lead to more precise issue replication and, consequently, higher-quality code generation.

Our initial hypothesis posits that the integration of LLMs, when guided by refined prompts and systematic testing will improve the accuracy and efficiency of reproducing reported issues. This method is expected to outperform traditional approaches that rely on heuristic methods or ad hoc user interactions.

The findings from this study could have significant implications for the field by improving the efficiency and accuracy of problem resolution on platforms like Stack Overflow, providing a scalable approach to automate issue reproduction and code validation, and informing the development of advanced moderation and support tools that leverage large language model capabilities. These contributions have the potential to streamline community-driven support systems and enhance the overall quality of software development practices.

4 FINDINGS

Having expiremented with over 150 Stack Overflow issues marked as "irreprodicible" by human developers in the previously mentioned studies [4]. The data set was split into 87 Java issues and 66 Python issues. We will now present the findings of our experiments using the three LLMs in this section and how they relate to our research questions.

4.1 RQ1: LLMs' Ability to Complete Code Snippets

To assess whether LLM's can generate missing parts of code to make snippets executable, we measured the percentage of issues where each model successfully recreated the reported problem. Each model was given the same prompt for each issue, and responses were classified simply as "Yes" or "No." This binary approach provided a clearer and more direct assessment of the LLMs' ability to reproduce the issue, eliminating ambiguity from subjective labels like "close" or "almost."

Table 1: Executable Code Counts Across LLMs

Model	Java %	Python %	Average %
CoPilot	25.7%	43.75%	34.73%
GPT4o	65.33%	81.8%	73.57%
DeepSeek	52.95%	57.8%	55.37%

Java	Python	Average
49.68%	61.12%	54.56%

Table 2: Average success rate of all experiments

Our findings show there is a 54.56 percent average across all 450+ experiments done. Note the 13% increased rate in the LLMs

being able to recreate Python issues over Java issues. GPT 4o having the highest success rate across both languages at 73.57%. And the lowest success rate was CoPilot at 25.7%.

4.2 RQ2: Recreating Environments and Dependencies

To be able to better answer this question, we had to first classify what an environment related issue was. Establishing this classification was important for staying consistent in how we identified the issues. We considered an issue as enviroment related if it involved factors beyond the code itself, specifically issues that required certain configurations, tools, or files structures to reproduce. Our classifications includes:

- **Multiple files:** Issues that need more than one file to reproduce.
- **Environment or path variables:** Problems directly related to environment settings or the correct specification of file paths, potentially requiring the inference of the original user's file structure.
- **Needing dependencies:** Issues that cannot be reproduced without the installation or configuration of specific software dependencies (e.g., libraries, packages). This aligns with the challenges of "External library not found" identified by [4] and the emphasis on dependencies in the study by [2]
- **Requiring specific build tools:** Issues that necessitate the use of particular build automation tools or specific versions of compilers or interpreters to manifest, like Maven and Gradle for Java.

Using this classification, we looked what proportion of our dataset that these issues represented. Our analysis of the 87 irreproducible Java issues revealed that 33 of them (38%) were classified as environment-related. Similarly, within the 66 irreproducible Python issues, we identified 40 (60%) that fell under our environment-related classification, as summarized in Table 3.

Language	Non-Env	Env	Total
Java	87	33	38%
Python	66	40	60%

Table 3: Counting Environment Related Issues

By isolating these environment-related issues it allows us to look at the reproducibility rate of just those issues. We found that the reproducibility success rate goes down when we specifically look at environment related issues. CoPilot had a 14% decrease it its overall success rate when looking at these types of issues. Similarlly GPT4o and Deepseek saw decreaseses of 18% and 19% respectively.

There is a 37% average across the 180+ experiments on reproducing environment related issues. It is worth pointing out the difference in average success rate across the two languages, with a significant 25% higher rate of solving environment related Python issues over Java issues. These results suggest that environment related issues pose a considerable challenge to automated issue reproduction, leading to a reduction in the overall success rate when we isolated them to analyze.

Table 4: Reproducibility Rates on Environment Related Issues

Model	Java %	Python %	Average %
CoPilot	7.5%	33.33%	20.3%
GPT4o	50.74%	72.5%	56.62%
DeepSeek	27.27%	45%	36.10%

Java	Python	Average
25.15%	50.28%	37.70%

Table 5: Average success rate of all environment related experiments

4.3 Other Findings

Since we isolated environment specific issues, we were also able to look at issues that strictly only involved missing code and/or faulty logic. These issues contributed a significant portion of our dataset, representing 63% of the Java issues and 40% of the Python issues that were initially deemed irreproducible. Our findings indicate a notable improvement in the reproducibility success rates achieved by the LLMs, proving a greater proficiency in addressing code generation and logical errors.

Table 6: Reproducibility Rates on Environment Related Issues

Model	Java %	Python %	Average %
CoPilot	42.1%	57.14%	49.6%
GPT4o	79%	90%	84.58%
DeepSeek	69%	82%	75.91%

Java	Python	Average
63.5%	76.58%	70%

Table 7: Average Success rate of all non environment related Experiments

The LLMs were much more successful in being able to generate missing code and faulty logic that resulted in reproducing the reported issues. We found that there was a 64.5% average across the LLMs for Java issues, and a 76.6% average for Python issues. Notably, GPT4o was able to reproduce 84% of the these types of issues. CoPilot and DeepSeek saw increases in reprodicbility rate as well, going up 30% and 40% respectively. We found an almost 33% increase in reproducibility rate between environment and non environment related issues.

5 DISCUSSION

5.1 Key Findings

After analyzing over 150 Stack Overflow issues with three different Large Langauge Models, CoPilot, GPT4o, and DeepSeek, previously

marked as "irreproducible", we observed the following key insights regarding Large Language Models' capabilities in reproducing software issues.

Effectiveness of Code Generation:

- Across all experiments the average success rate of issue reproduction by the three LLMs was 54.56%. GPT4o demonstrated the highest overall proficiency at 73.57% significantly out performing CoPilot (34.73%) and DeepSeek (55.37%)
- Python issues were generally more reproducible that Java issues, with a 13% higher succcess rate.
- Isolating non environmental issues (missing code and logical errors), reproducibility significantly improved. The LLMs achieved a 70% average success rate, with GPT4o having the highest success rate at 84% for these issues.
- Non environment issues showed a notable increase in reproducibility, approximately 33% higher than environmental related issues. Proving a strong capability of LLMs reproducing issues that only require generating missing code and logical errors, compared to environment specif configurations

Environment Reconstruction:

- Environment related issues represented a substantial portion of the irreproducible dataset: 38% for Java and 60% for Python. These issues proved to be the most challenging, causing significant decreases in reproducibility rates, they lead to a average 37% decrease compared to the full dataset.
- Specifically for environment related issues, Python issues had a 25% higher success rate than Java issues. GPT4o again having the highest performance rate, in what we found to be the most challenging issue category.

Failure Cases and Limitations:

In several cases, the post included a description of the issue or a code snippet, but failed to provide the specific runtime error, exception trace, or compiler message. This omission significantly impacts the reproducibility of the issue, as error messages often serve as crucial signals for diagnosing problems and determining appropriate environment configurations. Without this information, LLMs are forced to rely on incomplete contextual cues and may incorrectly infer the root cause or propose irrelevant solutions.

Another trend we have noticed is there are many issues that were presented that had many dependencies attached to them, for example, custom libraries. While LLM's had better success reproducing issues that involved standard libraries. The LLMs' would struggle the most with reproducing issues involving these custom / 3rd party libraries.

5.2 Implications

Now that we have presented our findings of reproducibility regarding generating missing code and environment related issues from Stack Overflow using Large Language Models. Our analysis reveals distinct challenges in reproducibility rate depending on the nature of the reported issue, particularly how the LLMs had a difficult time reproducing environment specific issues. This section will go over the broader significance of these findings, and discuss the potential impact the have on areas such as the development of automated debugging tools the quality of questions on developer forums and future directions for research in software engineering and the application of LLMs as a tool in code related tasks.

For developers, this research suggests that if environmental factors were known, and could be explained to the LLM, they can be used to locate bugs. Rahman et al. [6] suggests that tools can complement incomplete bug reports by leveraging historical information or similar reports. LLMs can identify missing information or environment details. With the help of humans prompting AI with more context, it could lead to a higher rate of reproducible bugs.

In education, LLMs have the potential to reduce frustration for students learning to program. This study has showed that LLMs have a very high success rate in generating code and faulty logic, when successfully reproducing the issue it is because it knows the cause of the issue, therefore knowing how to solve the issue. LLMs can provide students with more immediate feedback and lessen frustrations with debugging as most assignments in high school or university level coding classes don't have as complex environment requirements. However this can raise concern about academic integrity. With how powerful LLMs are in generating code and fixing errors, students can skip the learning process and hand in AI generated work without understanding the coding lessons. Moving forward, educators will need to address how to handle this, and design more appropriate ways to assess their students. An interesting example from the University of Saskatchewan, is that students in the introductory computer science class now need to pass the final with a 65%+ in order to pass the class, a requirement much harsher than some higher level classes in computer science.

As for Stack Overflow, this research strongly suggests that LLM's are a powerful tool in being able to reproduce issues. The study done by Mondal et al. [3] show that reproducible issues lead to a three times higher chance to receive an accepted answer over irreproducible issues. Reproducible issues encourage more answers, By making issues posted easier to reproduce, Stack Overflow could see:

- More efficient knowledge sharing: Developers are more likely to receive faster answers and accurate solutions when their issues are easily understood and reproduced.
- Higher Quality Content: Reproducible questions lead to higher quality answers as the answerers can directly address the problem without wasting any time struggling to reproduce.
- A more engaged community: When questions are easier to answer, more users are likely to participate, helping the community grow.

5.3 Future Work

When starting this project we set out with the goal of experimenting on the 150 issues marked irreproducible by Mondal and Roy [4]. Their data set includes other issues that were marked "reproducible" by human developers. Categorized as issues requiring "major" or "minor" changes, this could lead to interesting research to see if there is code that humans can reproduce but AI can not. An interesting question that we did not really look our for while doing our experiments is how high quality is the code generated by the LLMs. There are concerns about other potential errors that the LLMs may

have generated, whether it affects the issue reproducibility rate or not. Another concern we would like to investigate is the significant different between CoPilot and GPT4o. CoPilot was running GPT4 (not GPT4o), so we can investigate what the underlying issues are that caused such a discrepancy in the success rate between the two LLMs.

6 THREATS TO VALIDITY

In this section, we discuss potential threats to the validity of our findings. Addressing these concerns is crucial for accurately interpreting our experimental results and for future studies aiming to replicate or extend our work.

6.1 Internal Validity

Internal validity refers to whether our methodology and measurements accurately capture what we claim. One potential threat is *prompt design bias*. Even minor changes in how we present Stack Overflow issues to the Large Language Models (LLMs) can influence the responses they generate. Another factor is the possibility of *subjective judgment* when labeling an outcome as "reproduced" versus "not reproduced." Though we designed a clear binary measure, edge cases sometimes required subjective calls.

6.2 External Validity

External validity concerns the generalizability of our findings. Our dataset primarily involves Java and Python code snippets previously marked as irreproducible. While these two languages are popular, LLM performance may differ for other languages or frameworks. Additionally, real-world issues on Stack Overflow can vary widely in complexity, which may influence the reproducibility success rate in practice.

6.3 Construct Validity

Construct validity examines whether our chosen metrics genuinely represent "issue reproducibility." We defined success as generating code that, when run, produces the same error or behavior described in the original post. This is a practical measure, but some nuances—like partial matches or environment-specific warnings—may not be captured.

6.4 Data Limitations

Our dataset originates from a prior study that manually classified these issues as irreproducible. Some questions may lack explicit error messages or be tied to outdated library versions. LLMs trained on more recent data might guess dependencies incorrectly, or fail to replicate older environments accurately.

7 RELATED WORK

Previous research has explored various facets of programming-related issue reproducibility and user behavior on online forums. Notably, two published studies by our teaching assistants have provided foundational insights in this domain. For instance, Mondal, Rahman, Roy, and Schneider (2022) investigated the reproducibility of programming-related issues on Stack Overflow, offering empirical evidence that underpins our work.

Traditional approaches in the literature typically involve soliciting additional code samples or relying on heuristic-based guesses to address user issues. In contrast, our approach emphasizes the systematic reproduction of reported issues, which is designed to minimize the iterative back-and-forth commonly observed between users. This method not only streamlines the resolution process but also aims to provide more reliable outcomes.

The primary innovation of our study lies in its departure from solely relying on large language models (LLMs) to solve user issues. By focusing on reproducing the issues within their original context, our approach increases the likelihood of generating human-curated, high-quality code. This shift addresses existing gaps in automated problem-solving techniques and represents a significant contribution to the field.

Our proposed methodology is anticipated to enhance the efficiency of platforms such as Stack Overflow. By reducing the need for extensive clarifications and iterative code requests, we expect to achieve higher accuracy in problem resolution and improve overall user satisfaction. These improvements could lead to more effective moderation and faster turnaround times in community-driven support environments.

Yang et al. [9] assess the usability of approximately 914K Java code snippets extracted from accepted Stack Overflow answers, focusing on parsability (e.g., syntax errors) and compilability (e.g., incompatible types). Using automated tools such as Eclipse JDT and ASTParser, they identify challenges that prevent these snippets from parsing and compiling. Horton and Parnin [2] examine the executability of Python code from GitHub Gist. They categorize execution failures, including import, syntax, and indentation errors. However, successful execution does not always guarantee issue reproducibility. While some compilability and executability challenges may overlap with reproducibility challenges, the latter requires additional manual testing and debugging, which Horton and Parnin did not conduct.

Mondal et al. [3] go beyond by manually investigating the reproducibility of issues in 400 Java questions using their corresponding code snippets. They catalog challenges that hinder reproducibility, such as missing essential code components. This study explores whether LLMs can help mitigate these challenges and improve issue reproduction.

Several studies examine the challenges of reproducing software bugs and security vulnerabilities [1, 5, 6]. Joorabchi et al.[6] investigate 576 non-reproducible bug reports from Firefox and Eclipse, uncovering 11 challenges such as bug duplication, missing information, and ambiguous specifications. Their survey of 13 developers reveals that developers either close non-reproducible bugs or request additional details. Mu et al. [5] analyze 368 security vulnerability reports and find that many lack essential information (e.g., system configurations, OS details) needed for reproduction. Their survey of security experts suggests that, aside from internet-scale crowdsourcing and heuristic-based approaches, manual debugging remains the primary method for retrieving missing information. Terragni et al. [8] develop CSnippEx, which automatically transforms Java code snippets into compilable Java source code files. Our study investigates the capability of LLMs to enhance code snippets for issue reproduction in Stack Overflow questions, addressing key reproducibility challenges.

8 CONCLUSION

In this paper, we explored how Large Language Models can assist in reproducing software issues originally deemed irreproducible on Stack Overflow. Our results show that LLMs can fill in missing code segments and infer dependencies for a significant subset of issues, particularly when standard libraries or relatively simple logic are involved. However, environment-specific complexities, such as custom libraries or tool configurations, often remain a stumbling block.

Drawing on our previous discussion regarding the validity of our approach, it is important to acknowledge potential limitations. As we noted, factors such as prompt design bias and the subjectivity in judging reproducibility could influence our findings. That is why we structured our expirements as so, each giving the same prompt to the LLMs to help reduce this effect.

By focusing on reproducibility rather than merely providing fixes, our approach can help foster faster and more efficient debugging on community-driven platforms. In doing so, we highlight a promising avenue for future development of automated tools and methodologies aimed at making issue reproduction more scalable. Ultimately, improving reproducibility can lead to better collaboration, higher-quality answers, and a more robust knowledge base for developers worldwide.

REFERENCES

[1] Mona Erfani Joorabchi, Mehdi Mirzaaghaei, and Ali Mesbah. 2014. Works for me! characterizing non-reproducible bug reports. In *Proceedings of the 11th working conference on mining software repositories*. 62–71.

[2] Eric Horton and Chris Parnin. 2018. Gistable: Evaluating the executability of python code snippets on github. In *International Conference on Software Maintenance and Evolution (ICSME)*. 217–227.

[3] Saikat Mondal, Mohammad Masudur Rahman, and Chanchal K Roy. 2019. Can issues reported at stack overflow questions be reproduced? an exploratory study. In *16th International Conference on Mining Software Repositories (MSR)*. 479–489.

[4] Saikat Mondal and Banani Roy. 2022. Reproducibility Challenges and Their Impacts on Technical Q&A Websites: The Practitioners' Perspectives. In *15th Innovations in Software Engineering Conference*. 1–11.

[5] Dongliang Mu, Alejandro Cuevas, Limin Yang, Hang Hu, Xinyu Xing, Bing Mao, and Gang Wang. 2018. Understanding the reproducibility of crowd-reported security vulnerabilities. In *27th USENIX Security Symposium (USENIX Security 18)*. 919–936.

[6] Mohammad Masudur Rahman, Foutse Khomh, and Marco Castelluccio. 2020. Why are Some Bugs Non-Reproducible?:–An Empirical Investigation using Data Fusion–. In *International Conference on Software Maintenance and Evolution (ICSME)*. 605–616.

[7] Freda Shi, Eric Zhang, Dorottya Demszky, Tatsunori B. Hashimoto, and Christopher D. Manning. 2023. Large Language Models Can Be Easily Distracted by Irrelevant Context. In *International Conference on Machine Learning (ICML) (Proceedings of Machine Learning Research)*. PMLR. https://arxiv.org/abs/2302.00093

[8] Valerio Terragni, Yepang Liu, and Shing-Chi Cheung. 2016. CSNIPPEX: automated synthesis of compilable code snippets from Q&A sites. In *Proceedings of the 25th international symposium on software testing and analysis*. 118–129.

[9] Di Yang, Aftab Hussain, and Cristina Videira Lopes. 2016. From query to usable code: an analysis of stack overflow code snippets. In *Proceedings of the 13th International Conference on Mining Software Repositories*. 391–402.