International Journal of
INTELLIGENT SYSTEMS AND APPLICATIONS IN ENGINEERING
ISSN:2147-6799 www.ijisae.org Original Research Paper

Automatic Program Repair: A Comparative Study of LLMs on QuixBugs

Poonam Ponde[1], Manisha Bharambe[2] , Vinaya Keskar[3] , Harshita Vachhani[4]

Submitted: 02/11/2024 **Revised:** 16/12/2024 **Accepted:** 25/12/2024

Abstract*:* Software bugs are errors or flaws in a program's code that can lead to incorrect or unexpected behavior, making their detection and resolution crucial for reliable and secure software development. Debugging is a human-centric, time-consuming and resource-intensive process, making it one of the most expensive phases in software development. Automatic Program Repair (APR) is an emerging area of research that aims to automatically fix software bugs with minimal human intervention. Traditional APR tools use search-based or learning-based techniques to find software bugs based on test suites and bug patterns, thereby having heavy reliance on test cases. AI-driven APR tools are trained on large-scale codebases, open-source bug-fix histories, and benchmarks like QuixBugs. They can analyze buggy code, fix bugs and generate code patches that are syntactically and semantically correct. This reduces the debugging time and improves software reliability The QuixBugs benchmark has 40 programs from the Quixey Challenge in two languages: Python and Java. Each program contains a one-line defect and failing testcases. This paper presents a comparative study of APR techniques on the QuixBugs benchmark, which includes 40 buggy programs in both Python and Java. This study evaluates and compares the automatic bug fixing capability of LLMs such as ChatGPT and Google Gemini on the QuixBugs benchmark, thereby contributing to the understanding of LLMs' role in automatic program repair.

Keywords: Bugs, Debugging, Automatic Program Repair, ChatGPT, Gemini.

1. Introduction

Bugs are very common in the software development process. They are flaws or unintended behaviors in a program that can cause incorrect or unpredictable results. It is crucial to identify and resolve them to ensure that the software performs reliably and as expected. Testing is a crucial stage in the software development lifecycle. Debuggers help developers identify and fix bugs by code inspection, execution and program state investigation. It is however a time consuming and intensive task, often incurring huge amount of money. Inspite of rigorous testing, software bugs are inevitable and can lead to significant costs and even failures if not addressed in time (Ye et al., 2020). Manual debugging is labor-intensive, which has driven researchers to develop Automated Program Repair (APR) techniques that can automatically detect and fix bugs. This makes Automatic Program Repair (APR) an important area of software development. APR aims to fix bugs with little or no human intervention. It analyzes buggy code and generates patches to resolve errors. Traditional APR tools such as GenProg and SemFix use search-based or learning-based techniques to find software bugs based on test suites and bug patterns, thereby having heavy reliance on test cases as specifications (Le Goues et al., 2012; Nguyen et al., 2013, Zhang et al., 2023). Moreover, another limitation of traditional approaches is that they do not generalize well across various benchmarks and programming languages.

Recent advancements in Artificial Intelligence (AI) and Large Language Models (LLMs) present new opportunities for automatic bug identification and repair without relying solely on test cases (Xia et al., 2023). These Natural Language Processing (NLP) models have demonstrated promising performance in code generation, human-like code suggestions and bug fixes (Prenner et al., 2022; Wuisang et al., 2023). Conversational AI models like OpenAI's ChatGPT and Google's Gemini are have automated debugging capability. However, a detailed investigation of their automatic bug detection and fixing efficiency and reliability is necessary to assess their effectiveness and applicability. This study evaluates and compares the effectiveness of two LLM tools ChatGPT and Google Gemini in APR using the QuixBugs benchmark (Lin et al., 2017). By analyzing both, performance and limitations of these models, this study contributes to the understanding of LLMs' role in automatic program repair.

This study is significant for several reasons:

1 Associate Professor , Department of Computer Science, Nowrosjee Wadia College, Pune, India

2 Associate Professor,Department of Computer Science, Abasaheb Garware College, Pune, India

3 ATSS College of Business Studies and Computer Application Chinchwad Pune, India

4 Pratibha College of Commerce and Computer Studies, Pune, India

Email ID : 1 poonamponde@nowrosjeewadiacollege.edu.in, 2mgb.agc@mespune.in 3 vasanti.Keskar@gmail.com 4Profharshita@gmail.com

** Corresponding Author: Poonam Ponde*

E-mail Id: poonamponde@nowrosjeewadiacollege.edu.in

1. Benchmarking AI systems: Provides a comparative analysis of state-of-the-art AI systems in the context of automated program repair.
2. Improvement insights: Identifies strengths and weaknesses of and ChatGPT, offering insights for future improvements.
3. Practical applications: Helps in understanding the practical applicability of these AI systems in real-world debugging scenarios.

1.1 Benchmarks

Benchmarks in Automatic Program Repair (APR) are curated datasets of real-world or synthetic bugs used to evaluate and compare repair tools. They help researchers test how well APR systems fix bugs across different languages, domains, and complexity levels. Popular benchmarks like Defects4J (Just et al., 2014) and ManyBugs (Le Goues et al., 2015), are very useful but suffer from overuse and limited scope (Ye et al., 2020). To overcome this, Lin et al. (2017) introduced QuixBugs, a multilingual benchmark featuring 40 algorithmic problems in both Java and Python. It includes a collection of small, buggy programs, each accompanied by a correct version. The dataset is implemented in both Python and Java, making it suitable for evaluating tools across different programming languages. The bugs in QuixBugs cover a wide range of common programming errors, including: Logical Bugs, Syntax bugs, Semantic Bugs and Structural Bugs. It is open-source and available on platforms like GitHub. QuixBugs is significant because it provides a standardized way to evaluate and compare automated program repair tools.

2. Related Work

Early APR techniques like GenProg (Le Goues et al., 2012), SemFix (Nguyen et al., 2013), and Angelix (Mechtaev et al., 2016) approached bug fixing through genetic programming, constraint solving, or symbolic analysis. These techniques required substantial engineering effort and were typically bound to specific languages or benchmarks. Tools like Astor (Monperrus, 2018; Martinez & Monperrus, 2019) and Nopol (Xuan et al., 2017) refined patch generation mechanisms but had problems with scalability and generalization. CURE (Jiang et al., 2021) and CodeBERT (Feng et al., 2020) showed that transformer-based models can generate patches with minimal human input. There are many benchmarks for APR such as ManyBugs (Le Goues et al., 2012) and Defects4J (Just et al.,2014). However, they suffer from the limitations of less diversity in evaluation datasets (Ye, et al., 2020). To address evaluation diversity, Lin et al. (2017) developed QuixBugs, a benchmark with 40 algorithmic bugs in Java and Python, specifically designed for evaluating APR tools across languages. Ye et al. (2020) used QuixBugs to assess 10 APR tools, highlighting concerns like patch overfitting and the need for better patch validation methods. The rise of LLMs like Codex and ChatGPT introduced data-driven approaches to program synthesis and repair. Codex, trained on GitHub data, demonstrated capabilities in solving introductory programming tasks (Finnie-Ansley et al., 2023). ChatGPT has not been explicitly trained for code; however, it has demonstrated strong debugging capabilities (Wuisang et al., 2023; Sobania et al., 2023). Fan et al. (2023) demonstrated that LLMs can outperform some APR baselines on QuixBugs. Curtis (2023) The literature shows that big language models are promising tools for automated program repair, but there are still challenges, such as ensuring their fixes are correct and avoiding new bugs (Wuisang et al., 2023).

Most of the literature in the area of APR is focused on the performance measurement of one APR tool against the benchmark. A comparative analysis of LLMs in APR is essential to evaluate their bug-fixing accuracy, language adaptability, and prompt sensitivity across standardized benchmarks.

3. Methodology

The bug-fixing performance of ChatGPT and Gemini is analyzed using the QuixBugs benchmark and prompt-based instructions, with bug-fixing accuracy manually verified. Five independent requests were made to ChatGPT and Gemini for each of the 40 programming problems to assess its ability to identify the bug and correct the code. The responses were recorded and compared. Figure 1 shows an example request to ChatGPT for the bitcount problem and Figure 2 shows the ChatGPT response to the above prompt.

```
def bitcount(n):
  count = 0
  while n:
    n ^= n - 1
    count += 1
  return count
```

Does the following code have a bug? Yes/No If yes, clearly state the bug. give the corrected code

Figure 1. ChatGPT prompt for bitcount problem.

Yes, the code does have a bug.

Bug Explanation:

The statement `n ^= n - 1` is incorrect for counting the number of set bits (1s) in the binary representation of `n`. This operation performs a bitwise XOR between `n` and `n - 1`, which does not remove the rightmost set bit.

The intended logic is to remove the rightmost set bit in each iteration. This is correctly done using `n &= n - 1`.

Figure 2. ChatGPT response for bitcount problem.

The corrected code as given by ChatGPT is shown in Figure 3.

```python
def bitcount(n):
    count = 0
    while n:
        n &= n - 1
        count += 1
    return count
```

Figure 3. ChatGPT corrected code for bitcount problem.

The Gemini response and corrected code for the same prompt are shown in Figure 4 and Figure 5 respectively.

Yes, the provided Python code has a bug.

The Bug:

The bug lies in the line `n ^= n - 1`. This line is intended to use the bitwise XOR assignment operator (`^=`), but due to the order of operations and the way `n - 1` is evaluated, it does not correctly count set bits in all cases.

Let's break down `n ^= n - 1`:

- `n - 1` is calculated first.
- Then, `n` is XORed with the result of `n - 1`.
- Finally, the result of the XOR operation is assigned back to `n`.

The intended logic for counting set bits (also known as Hamming weight or population count) using this kind of bit manipulation usually involves `n & (n - 1)` to clear the least significant set bit. The XOR operation (`^`) does something different, which is not suitable for this purpose.

Figure 4. Gemini response for bitcount problem.

```
def bitcount(n):
    count = 0
    # The loop continues as long as n is not zero
    while n > 0:
        # This operation clears the least significant set bit of n
        n = n & (n - 1)
        count += 1
    return count
```

Figure 5. Gemini corrected code for bitcount problem.

For some problems, the bug is not identified correctly and an incorrect fix is proposed by ChatGPT. Figure 6(a) shows original rpn_eval (Reverse Polish Notation) problem, 6(b) shows the incorrect fix proposed by ChatGPT and 6(c) shows the correct solution given by Quixbugs.

```
def rpn_eval(tokens):
    stack = []
    for token in tokens:
        if token in '+-*/':
            b = stack.pop()
            a = stack.pop()
            if token == '+':
                stack.append(a + b)
            elif token == '-':
                stack.append(a - b)
            elif token == '*':
                stack.append(a * b)
            else:
                stack.append(a / b)
        else:
            stack.append(token)
    return stack[0]
```

```
def rpn_eval(tokens):
    stack = []
    for token in tokens:
        if token in '+-*/':
            b = stack.pop()
            a = stack.pop()
            if token == '+':
                stack.append(str(a + b))
            elif token == '-':
                stack.append(str(a - b))
            elif token == '*':
                stack.append(str(a * b))
            else:
                stack.append(str(a / b))
        else:
            stack.append(token)
    return int(stack[0])
```

```
def rpn_eval(tokens):
    stack = []
    for token in tokens:
        if token in '+-*/':
            b = stack.pop()
            a = stack.pop()
            if token == '+':
                stack.append(a + b)
            elif token == '-':
                stack.append(a - b)
            elif token == '*':
                stack.append(a * b)
            else:
                stack.append(a // b)
        else:
            stack.append(int(token))
    return stack[0]
```

Figure 6(a) Original problem (b) Incorrect fix by ChatGPT (c) Correct fix.

5. Results and Discussion

The responses of ChatGPT and Gemini for each of the 40 buggy programs from QuixBugs benchmark were recorded and analyzed. To decide whether response is a correct answer, we judge by looking at the output given and comparing it with the correct python program provided by QuixBugs. Each response was marked as Yes (The bug was correctly identified and fixed), No (The bug was not identified or fixed). In some cases, it is observed that a bug is identified and along with this some other issue is also identified and corrected. Also, in a few runs, the bug is not identified but other issue is identified and corrected. Table 1 gives the analysis done on the responses provided by ChatGPT.

Table 1. Analysis of ChatGPT results.

Benchmark	**ChatGPT**		
	Bug successfully identified Y/N	**Bug identified and some other issue identified and corrected**	**Bug not identified but other issue identified and corrected**
bitcount	Y (5/5)	0	0
breadth_first_search	Y (5/5)	5	0
bucketsort	Y (5/5)	0	0
depth_first_search	Y (5/5)	0	0
detect_cycle	Y (5/5)	0	0
find_first_in_sorted	Y (5/5)	0	0
find_in_sorted	Y (5/5)	5	0
flatten	Y (5/5)	0	0
gcd	Y (3/5)	0	2
get_factors	Y (4/5)	0	1
hanoi	Y (5/5)	0	0
is_valid_parenthesization	Y (5/5)	0	0
kheapsort	Y (3/5)	1	1
knapsack	Y (5/5)	0	0
kth	Y (5/5)	5	0
lcs_length	N (0/5)	0	0
leveshtein	Y (5/5)	1	0
lis	N (0/5)	0	0
longest_common_subsequence	Y (4/5)	3	0
max_sublist_sum	Y (5/5)	0	0
mergesort	Y (1/5)	0	0
minimum_spanning_tree	Y (3/5)	1	0
next_palindrome	Y (5/5)	4	0
next_permutation	Y(4/5)	0	1
node	Y (5/5)	0	0
pascal	Y (5/5)	0	0
possible_change	Y (5/5)	0	0
powerset	Y(1/5)	0	4
quicksort	Y (2/5)	0	3
reverse_linked_list	Y (5/5)	0	0
rpn_eval	N (0/5)	0	0
shortest_path_length	Y (2/5)	0	3
shortest_path_lengths	N (0/5)	0	0
shortest_paths	Y(4/5)	0	1
shunting_yard	Y (5/5)	0	0
sieve	Y (5/5)	0	0

sqrt	Y (2/5)	0	3
subsequences	Y (4/5)	0	1
to_base	N (0/5)	0	0
topological_ordering	Y (4/5)	1	0
wrap	Y (5/5)	4	0
Total	**35**	**30**	**20**

Table 2. Analysis of Gemini results.

Benchmark	**Gemini**		
	Bug successfully identified Y/N	**Bug identified and some other issue identified and corrected**	**Bug not identified but other issue identified and corrected**
bitcount	Y (5/5)	0	0
breadth_first_search	Y (5/5)	5	0
bucketsort	Y (5/5)	0	0
depth_first_search	Y (5/5)	0	0
detect_cycle	Y (2/5)	2	0
find_first_in_sorted	Y (3/5)	3	0
find_in_sorted	Y (2/5)	0	0
flatten	Y (3/5)	0	0
gcd	Y (2/5)	0	2
get_factors	Y (5/5)	0	0
hanoi	Y (5/5)	0	0
is_valid_parenthesization	Y (5/5)	0	0
kheapsort	N (0/5)	0	5
knapsack	Y (5/5)	0	0
kth	Y (1/5)	1	0
lcs_length	N (0/5)	0	2
leveshtein	Y (2/5)	2	0
lis	N (0/5)	0	1
longest_common_subsequence	Y (4/5)	0	0
max_sublist_sum	Y (3/5)	0	0
mergesort	Y (3/5)	2	0
minimum_spanning_tree	Y (5/5)	5	0
next_palindrome	Y (1/5)	1	0
next_permutation	Y (4/5)	4	0
node	Y (2/5)	0	0
pascal	Y (5/5)	5	0
possible_change	Y (4/5)	4	0
powerset	Y (1/5)	1	0
quicksort	Y (4/5)	4	1
reverse_linked_list	Y (5/5)	5	0
rpn_eval	N (0/5)	0	2
shortest_path_length	Y (1/5)	0	0
shortest_path_lengths	N (0/5)	0	1
shortest_paths	Y (5/5)	0	0
shunting_yard	Y (1/5)	1	0
sieve	Y (5/5)	0	0
sqrt	Y (5/5)	0	0
subsequences	Y (5/5)	0	0
to_base	N (0/5)	0	0
topological_ordering	Y (5/5)	0	0
wrap	N (0/5)	0	5
Total	**33**	**45**	**19**

An example of inconsistent responses for the quicksort problem is shown in Figure 7.

```
def quicksort(arr):
    if not arr:
        return []

    pivot = arr[0]
    lesser = quicksort([x for x in arr[1:] if x < pivot])
    greater = quicksort([x for x in arr[1:] if x > pivot])
    return lesser + [pivot] + greater
```

Figure 7(a) Original problem.

Yes, the code has a bug.

Bug:
The code does not handle duplicate elements properly. It only includes one copy of the pivot and excludes other elements equal to the pivot from both the `lesser` and `greater` lists. This causes duplicates to be lost during sorting.

Figure 7(b) Bug identified by ChatGPT.

No, the code does not have a bug. It correctly implements the quicksort algorithm to sort a list `arr`.

The quicksort algorithm recursively divides the input list `arr` into two sublists: one containing elements less than the pivot (`lesser`) and the other containing elements greater than the pivot (`greater`). It then recursively sorts these sublists and combines them with the pivot to produce the sorted list.

The base case of the recursion is when the input list `arr` is empty, in which case the function returns an empty list.

Overall, the code correctly implements the quicksort algorithm to sort a list without any bugs.

Figure 7(c) Bug not identified by ChatGPT.

In a few programs, it was seen that the bug was successfully identified and corrected and an additional correction was also carried out. For example, in the kheapsort problem, the bug was fixed with an additional check provided for ‘k’ to prevent invalid heap initialization as shown in Figure 8.

Yes, the code has a bug. Specifically, it initializes the heap with the first `k` elements but then starts iterating over the entire array again, which includes these first `k` elements. This causes the initial elements to be processed twice.

The correct approach is to:

1. Initialize the heap with the first `k` elements.
2. Iterate over the remaining elements, starting from index `k`.

Here is the corrected code:

```
def kheapsort(arr, k):
    import heapq

    if k <= 0:
        raise ValueError("k must be greater than 0")

    # Initialize the heap with the first k elements
    heap = arr[:k]
    heapq.heapify(heap)

    # Iterate over the remaining elements
    for x in arr[k:]:
        yield heapq.heappushpop(heap, x)

    # Extract the remaining elements from the heap
    while heap:
        yield heapq.heappop(heap)
```

Explanation of Corrections:

- Initialize the heap with the first `k` elements using `arr[:k]`.
- Iterate over the remaining elements starting from index `k` with `for x in arr[k:]`.
- Added a check to ensure `k` is greater than 0 to prevent invalid heap size initialization.

Figure 8. Additional correction proposed by ChatGPT.

In some programs, it was seen that a bug was identified but the correct fix was not suggested. In the shortestpaths problem, an invalid update in the condition was made as shown in Figure 9.

Yes, the code has a bug. The bug lies in the inner loop where the weight is updated. The calculation of the updated weight for `weight_by_edge[u, v]` should be `weight_by_node[u] + weight`, but the current code mistakenly uses `weight_by_node[v]` instead of `weight_by_node[u] + weight`.

Here's the corrected code:

```
def shortest_paths(source, weight_by_edge):
    weight_by_node = {
        v: float('inf') for u, v in weight_by_edge
    }
    weight_by_node[source] = 0

    for i in range(len(weight_by_node) - 1):
        for (u, v), weight in weight_by_edge.items():
            weight_by_edge[u, v] = min(
                weight_by_edge[u, v],          # Corrected line
                weight_by_node[u] + weight
            )

    return weight_by_node
```

Explanation:

In the inner loop, I've replaced `weight_by_node[v]` with `weight_by_edge[u, v]` to correctly update the weight. This ensures that the shortest paths are computed accurately.

Figure 9. Incorrect fix for shortestpaths problem.

From the analysis of the responses, it can be observed that ChatGPT has better and more consistent bug detection and fixing capability as compared to Google Gemini. Bugs were successfully identified in more problems by ChatGPT (35) than Gemini (33). However, the response inconsistencies have been observed in both. There are many instances of successful bug identification and fix with additional fixes as well as non-identification of bugs with incorrect corrections or fixes reported by both.

6. Conclusion

APR has seen significant evolution, from rule-based and test-driven tools to sophisticated deep learning models and LLMs. This study evaluates and compares the automatic bug fixing capability of LLMs such as ChatGPT and Google Gemini on the QuixBugs benchmark dataset. ChatGPT was found to have a remarkable ability for the detection and correction of several common programming mistakes while giving readable and maintainable fixes in most of the cases. Gemini also demonstrated good debugging capabilities and came very close to identifying bugs in most problems. However, in terms of consistency, both were inconsistent in the bug identification and fixes suggested. In some instances, they missed the detection, or put forward improper fixes for some complex mistakes. As the LLM's become more powerful in terms of scale, training, code generation and learn more from ever-increasing corpora, these results will improve with newer versions. Hence, periodic evaluations are recommended. The flexibility and accuracy of these tools and response to prompts can be leveraged for APR in software development.

Conflicts of interest

The authors declare no conflicts of interest.

References

[1] Fan, Y., Wang, S., Liu, Y., & Zhang, L. (2023). Towards generalizable program repair with large language models: An empirical study. *Proceedings of the 45th International Conference on Software Engineering*.

[2] Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., ... & Zhou, M. (2020). CodeBERT: A pre-trained model for programming and natural languages. *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing* (EMNLP), 1536–1547.

[3] Finnie-Ansley, J., Sivaraman, A., Vasilescu, B., & DeLine, R. (2023). Robots need social skills: Exploring social behavior in code generation tools. *IEEE Transactions on Software Engineering*.

[4] Jiang, J., Zhang, D., Wang, S., Yin, G., & Zhou, J. (2021). CURE: Code-aware neural machine translation for automatic program repair. *IEEE Transactions on Software Engineering*.

[5] Just, R., Jalali, D., & Ernst, M. D. (2014). Defects4J: A database of existing faults to enable controlled testing studies for Java programs. *In Proceedings of the 2014*

International Symposium on Software Testing and Analysis (ISSTA) (pp. 437–440). ACM. https://doi.org/10.1145/2610384.2628055

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

[7] Lin, D., Koppel, J., Chen, A., & Solar-Lezama, A. (2017). QuixBugs: A Multi-Lingual Program Repair Benchmark Set Based on the Quixey Challenge. SPLASH Companion 2017. https://doi.org/10.1145/3135932.3135941. GitHub. https://github.com/jkoppel/QuixBugs

[8] Martinez, M., & Monperrus, M. (2019). Astor: A program repair library for Java. *Proceedings of ISSTA*, ACM.

[9] Mechtaev, S., Yi, J., & Roychoudhury, A. (2016). Angelix: Scalable multiline program patch synthesis via symbolic analysis. *Proceedings of the 38th International Conference on Software Engineerin*g, 691–701.

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

[11] Nguyen, H. D. T., Qi, D., Roychoudhury, A., & Chandra, S. (2013). SemFix: Program repair via semantic analysis. *Proceedings of the 2013 International Conference on Software Engineering*, 772–781.

[12] Prenner, J. A., Babii, H., & Robbes, R. (2022). Can OpenAI's Codex Fix Bugs? An Evaluation on QuixBugs. International Workshop on Automated Program Repair (APR'22). https://doi.org/10.1145/3524459.3527351

[13] Sobania, D., Briesch, M., Hanna, C., & Petke, J. (2023). An analysis of the automatic bug fixing performance of chatgpt. In 2023 IEEE/ACM *International Workshop on Automated Program Repair* (APR) (pp. 23-30). IEEE.

[14] Wuisang, M. C., Kurniawan, M., Santosa, K. A. W., Gunawan, A. A. S., & Saputra, K. E. (2023). An Evaluation of the Effectiveness of OpenAI's ChatGPT for Automated Python Program Bug Fixing Using QuixBugs. 2023 *International Seminar on Application for Technology of Information and Communication* (iSemantic), IEEE. https://doi.org/10.1109/iSemantic59612.2023.10295323

[15] Xia, C. S., Wei, Y., & Zhang, L. (2023). Automated program repair in the era of large pre-trained language models. *Proceedings of the 2023 IEEE/ACM 45th International Conference on Software Engineering* (ICSE), 1482–1494. https://doi.org/10.1109/ICSE48619.2023.00129

[16] Xuan, J., Martinez, M., Demarco, F., Clement, M., Danglot, B., Le Berre, D., & Monperrus, M. (2017). Nopol: Automatic repair of conditional statements in Java programs. *IEEE Transactions on Software Engineering*, 43(1), 34–55.

[17] Ye, H., Martinez, M., Durieux, T., & Monperrus, M. (2020). A comprehensive study of automatic program repair on the QuixBugs benchmark. *Journal of Systems and Software*, 171, 110825. https://doi.org/10.1016/j.jss.2020.110825

[18] Zhang, D., Liu, Y., Wang, S., & Zhou, J. (2023). A survey of learning-based automated program repair. *ACM Computing Surveys* (CSUR), 55(9), 1–39.