Benchmarking Large Language Models for Autonomous Run-time Error Repair: Toward Self-Healing Software Systems

Alessio Bucaioni
Mälardalen University
Sweden
alessio.bucaioni@mdu.se

Gabriele Gualandi
Mälardalen University
Sweden
gabriele.gualandi@mdu.se

Johan Toma
Mälardalen University
Sweden

ABSTRACT

As software systems grow in complexity and become integral to daily operations, traditional approaches to software testing, maintenance, and evolution are increasingly inadequate. Recent advances in artificial intelligence, particularly in large language models, offer promising avenues for achieving self-healing software—software capable of autonomously detecting, diagnosing, and repairing faults without human intervention. However, while much of the existing literature focuses on on code repair of vulnerabilities or repository-level bugs, the application of large language models for autonomously repairing run-time errors—which require dynamic analysis and execution context awareness—remains largely uncharted.

In this study, we empirically benchmark ten distinct large language models—ChatGPT-4o, ChatGPT-4o-mini, Claude 3.5 Sonnet, Claude 3.5 Haiku, Gemini 1.5 Flash, Llama 3.2, Mistral Nemo, Grok Beta, Command R+, and Jamba 1.5 Large—to assess their ability to repair run-time errors in code. We conducted our evaluation on a dataset of 76 programming problems manually sourced from Leetcode, implemented in C++ (48 problems) and Java (28 problems). Each model was provided with a single opportunity to generate a corrected solution, which was then evaluated based on its ability to pass all associated test cases.

Our experimental results provide early empirical evidence of the potential of large language models to drive a paradigm shift in artificial intelligence-driven software engineering. The findings reveal that while certain large language models demonstrate strong code-fixing capabilities, others struggle, highlighting significant performance disparities across models. This work not only fills a critical gap in empirical software engineering, but also opens avenues for refining artificial intelligence-driven software engineering, particularly for self-healing software.

CCS CONCEPTS

• **Software and its engineering → Software verification and validation**.

KEYWORDS

Self-healing code, code repair, large language models, bechmarking

ACM Reference Format:
Alessio Bucaioni, Gabriele Gualandi, and Johan Toma. 2018. Benchmarking Large Language Models for Autonomous Run-time Error Repair: Toward Self-Healing Software Systems. In *Proceedings of The 29th International Conference on Evaluation and Assessment in Software Engineering (EASE 2025).* ACM, New York, NY, USA, 6 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

As software systems grow increasingly complex and critical to daily operations, traditional approaches to software testing, maintenance and evolution are rapidly becoming insufficient [7]. Downtime, unexpected run-time errors, and escalating maintenance costs demand a paradigm shift toward self-healing software [6, 9], hence autonomic software capable of detecting, diagnosing, and repairing faults and anomalies without human intervention [6, 9]. Recent advancements in Artificial Intelligence (AI), particularly in the domain of Large Language Models (LLMs), have opened new avenues for addressing long-standing challenges in software engineering such as self-healing software [3]. While models such as ChatGPT have demonstrated exceptional capabilities in natural language understanding and programming [1], leveraging these models to enable self-healing in software remains largely under-explored- making this an emerging and futuristic research direction in empirical software engineering.

Our study investigates how LLMs can fuel this paradigm shift by autonomously repairing run-time errors, thereby bridging AI-driven software engineering and next-generation autonomous systems. In our research, we empirically benchmark ten distinct LLMs-ChatGPT-4o, ChatGPT-4o-mini, Claude 3.5 Sonnet, Claude 3.5 Haiku, Gemini 1.5 Flash, Llama 3.2, Mistral Nemo, Grok Beta, Command R+, and Jamba 1.5 Large-to assess their ability to autonomously repair run-time errors. The evaluation is conducted on a dataset of 76 programming problems manually sourced from Leetcode—a widely recognized online platform that offers a vast repository of coding problems and solutions—implemented in two widely used programming languages: C++ (48 problems) and Java (28 problems). Our approach is grounded in a rigorous empirical methodology. Each LLM was provided a single attempt to correct run-time errors, and success was quantified by measuring the percentage of problems for which the errors were effectively resolved. This study not only examines the effectiveness of each individual LLM, but also provides comparative insights into their relative performance across a diverse set of challenges. By doing so, we address a critical research gap: while the integration of AI into software engineering has been widely discussed, empirical studies that systematically benchmark the self-healing potential of different LLMs remain scarce. Building on the results of our experiment, we draw research implications and outline potential future research avenues for the community. In summary, this paper makes the following contributions:

- It provides early empirical evidence on the capability of LLMs to autonomously repair run-time errors in code.
- It benchmarks ten distinct LLMs using a curated set of 76 programming problems, offering a comparative analysis of their performance.
- It lays the groundwork for future research on integrating AI-driven self-healing mechanisms into real-world software systems, ultimately advancing the field of autonomous systems.

The remainder of this paper is organized as follows. In Section 2, we review existing studies related to our work. In Section 3, we describe the methodology used to benchmark LLMs for run-time error repair and detail its implementation. In Section 4, we present the results of our experiments. In Section 5, we discuss these results along with the potential threats to validity. Finally, in Section 6, we conclude the paper and outline possible future research avenues. Section 7 provides a public replication package.

2 RELATED WORK

While a significant body of literature exists on code repair and its automation and improvement using AI techniques, the vast majority of these studies focus on identifying and correcting code vulnerabilities—known flaws, weak points, glitches, or other security issues in software. In contrast, our study explicitly benchmarks LLMs for run-time error repair—an aspect that remains under-explored in AI-driven software engineering. Run-time errors often arise due to complex execution dependencies, making their resolution more nuanced than static vulnerability fixes. To the best of our knowledge, *our work is the first documented, peer-reviewed effort* to benchmark LLMs for their ability to autonomously repair run-time errors. An opportunistic review of the related literature, using the search string "llm* AND ("code repair" OR "self-healing")" on abstracts and titles, yielded only 11 unique results—8 from Scopus, 1 from the ACM Digital Library, and 2 from IEEE Xplore. Notably, *we deliberately excluded papers that had not undergone peer review*, such as those uploaded solely to arXiv or similar platforms, as they do not provide sufficient assurance of methodological rigour.

One of the most significant differences between our research and prior studies is that those studies primarily focus on fixing code vulnerabilities rather than run-time errors. For instance, De et al. investigate the effectiveness of LLMs in addressing Java vulnerabilities [4] by benchmarking only two LLMs, namely Code Llama and Mistral. Similarly, Zhang et al. evaluate LLMs for repairing memory corruption vulnerabilities—specifically in C/C++—and benchmark only ChatGPT-4 and Claude [10]. Additionally, Jain et al. present a related study that, instead of leveraging LLMs, employs deep learning techniques to fix security vulnerabilities such as code size issues, the presence of security-sensitive data structures, and conditional statements [8]. Unlike De et al., who evaluated only two LLMs on Java vulnerabilities, our study benchmarks ten distinct LLMs across a diverse dataset of 76 run-time error problems from Leetcode. Additionally, while Chen et al. assessed repository-level bug fixes, our focus is on single-file, functionally constrained run-time errors, which demand different debugging capabilities.

Chen et al. investigated the performance of ChatGPT-3.5 in handling repository-level repair tasks [2]. They defined repository-level repair tasks as those that repair bugs arising from complex interactions or dependencies among multiple code files, such as interface inconsistencies, incorrect error handling, global variable misuse, and race conditions. To evaluate this, they introduced RepoBugs—a new benchmark comprising 124 typical repository-level bugs collected from open-source repositories. Their preliminary experiments demonstrated that ChatGPT-3.5 achieved a success rate of 22.58% in fixing these issues. It is important to note that, in contrast to our study, Chen et al. concentrated on repository-level repair tasks rather than on run-time errors.

In their paper, Diaz et al. shed light on the automated patching of infrastructure-as-code projects with the help of LLMs [5]. They evaluated six LLMs across three different scenarios. In Scenario 0, they introduced an error in the module name, which caused execution to fail. In Scenario 1, they injected an error in the module state by providing an invalid value, as the state was restricted to a specific set of acceptable values. Finally, in Scenario 2, they misspelled the name of a common package.

Given the above, our study represents a crucial step in AI-driven software engineering, demonstrating how LLMs can autonomously repair run-time errors—an emerging direction in self-healing software. By providing the first systematic benchmark of LLMs for this task, our work lays the foundation for future research.

3 THE PIPELINE FOR BENCHMARKING LLMS ON AUTONOMOUS CODE REPAIR

In this section, we describe the methodology used to benchmark LLMs for run-time error repairing. We divided the process into four primary steps: Data Collection, Prompt Generation, Communication with LLMs, and the Evaluation Pipeline (Figure 1).

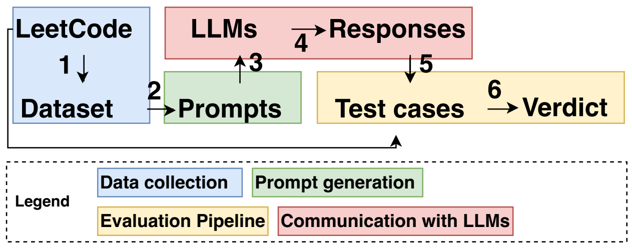


Figure 1: Overview of the experimental pipeline

Data Collection. We executed a pivotal and labour-intensive stage in our benchmarking workflow by gathering programming solutions that exhibit run-time errors. To minimize threats to validity, we selected Leetcode as our primary source, which ensured that our dataset remained objective. Since we focused specifically on run-time errors—as opposed to compile-time or other error types—we manually selected each candidate solution. We individually reviewed each solution to verify that it produced a run-time error, ensuring our dataset precisely aligned with our research objectives. We further refined our dataset by including only solutions written in Java or C++, thereby maintaining consistency across experiments. We recorded essential metadata for every eligible solution—such as the problem name, description, programming language, difficulty level, and a unique numerical identifier—in JSON files. We maintained separate JSON files for Java and C++ to streamline subsequent processing. We preprocessed each solution to remove extraneous comments and personal identifiers, ensuring we preserved only the relevant code.

Prompt Generation. In this phase, we transformed each programming problem and its corresponding erroneous solution into a clear, structured prompt. We designed these prompts to include all pertinent details—such as the problem description, code snippet, and explicit instructions for rectifying run-time errors—ensuring that each LLM received a comprehensive and unambiguous input. We predefined and standardized the prompt structure to include:

- Explicit request to correct the code,
- The programming language in question,
- The name and overview of the problem,
- A clear outline of the task and expected behaviour, and
- The code generating the run-time error.

Although we automated prompt generation using custom Python scripts, the process is technology-agnostic manner so that anyone can apply the underlying process regardless of the automation tools they use.

Communication with LLMs. During the communication phase, we interfaced with the LLMs and captured their corrected code outputs. We systematically transmitted the structured prompts to each LLM and stored their responses. We developed custom applications to automate prompt delivery, minimize manual intervention, and standardize communication across all models. We ensured that every prompt transmitted via the appropriate APIs reached its target consistently. We queried each LLM individually, sent structured prompts over the API, and collected real-time responses that we expected to contain the corrected code. To support subsequent evaluations, we stored each response in dedicated, structured files and applied a consistent naming convention that linked each output to its respective LLM. This systematic organization allowed us to compare model performance methodically. Moreover, we designed the entire process to be reproducible so that we delivered each prompt to all ten LLMs under identical conditions.

Evaluation Pipeline. In the final phase, we assessed the performance of each LLM based on its ability to correct run-time errors. We validated the outputs generated by the LLMs against a comprehensive set of test cases to determine if the corrections enabled successful execution. Notably, LeetCode provided these test cases for each selected problem, which helped us reduce threats to validity by grounding our evaluation in a standardized framework. We categorized the results from the testing phase into two distinct outcomes: success or failure, and we deemed a solution successful only if it passed all test cases without error. Since LeetCode does not publicly expose the full set of test cases via an API, we manually monitored each submission. We individually tracked each submission and documented its outcome to ensure the accuracy and integrity of our benchmarking results. We systematically logged these outcomes in a structured format, typically within a spreadsheet, and we included pertinent metadata for each problem—such as the problem name, programming language, difficulty level, and the corresponding result for each LLM.

3.1 Implementing the Pipeline

In this section, we detail the concrete tools and technologies we used to implement our benchmarking process across all four key phases. In our experiment, we selected ChatGPT-4o, ChatGPT-4o-mini, Claude 3.5 Sonnet, Claude 3.5 Haiku, Gemini 1.5 Flash, Llama 3.2, Mistral Nemo, Grok Beta, Command R+, and Jamba 1.5 Large. We provide a public replication package containing all our experimental artifacts—scripts, prompts, and results—at in Section 7.

We began with data collection by focusing on LeetCode. We manually scrutinized each candidate solution to confirm the presence of a run-time error. We stored the selected problems and their erroneous solutions in JSON files grouped by programming language. We recorded key metadata like the problem name, description, programming language, difficulty level, and a unique numerical identifier in each entry. Listing 1 shows an excerpt from the JSON file for the Java programs, focusing on problem number 15, "Permutation Sequence"[1]. We cleansed each solution by applying preprocessing steps that removed extraneous comments and personal identifiers before storage. We developed a Python-based application to automate the creation of prompts from the stored JSON files (Listing 2[1]). This application systematically reads each entry and generates a corresponding prompt designed to steer the LLMs toward correcting run-time errors. An example of prompt

```
{
    "id" : 15,
    "programming_problem_name": "Permutation Sequence",
    "difficulty" : "Hard",
    "desc" : "The set [1, 2, 3, ..., n] contains total ...",
    "additional_information" : "",
    "solution": "class Solution { public String ...",
    "programming_language" : "java",
    "new_error" : ""
}
```

Listing 1: Excerpt of the JSON file containing the JAVA problems.

```
for java_problem in java_data:
    prompt = f'''
    Please help me solve this code with {java_problem[ ...
        {java_problem['programming_problem_name']}
        This is the programming problems desc: { ...
        This code contains a run-time error.
        {java_problem['additional_information']}
        {java_problem['solution']}'''
    java_prompt_data.append({"id": i, "programming_ ...
    i += 1
```

Listing 2: Excerpt of the Python-based application.

[1]Please note that some rows have been truncated for readability and to conserve space.

generated by the Python-based application in Listing 2 is displayed in Listing 3

```
Please help me solve this code with java Permutation
Sequence

This is the programming problems desc: The set [1, 2, 3,
..., n] contains total of n! unique permutations. By listing
 and labeling all of the permutations in order, we get the
following sequence for n = 3: 1. 123, 2. 132, 3. 213, 4.
231, 5. 312, 6. 321, given n and k return k-th permutation
sequence

This code contains a run-time error.
class Solution { public String getPermutation(int n, int k)
{ List<Integer> numbers = new ArrayList<>(); int[]
factorials = new int[n + 1]; for (int i = 1; i <= n; i++) {
numbers.add(i); } factorials[0] = 1; for (int i = 1; i <= n;
 i++) { factorials[i] = factorials[i - 1] * i; if (
factorials[i] < 0) { factorials[i] = Integer.MAX_VALUE; } }
k--; if (k > factorials[n - 1]) { k = k * 10; }
StringBuilder sb = new StringBuilder(); for (int i = n; i >
0; i--) { int index = k / factorials[i - 1]; k %= factorials
[i - 1]; sb.append(numbers.get(index)); numbers.remove(index
); } return sb.toString(); } }
```

Listing 3: Example of a prompt.

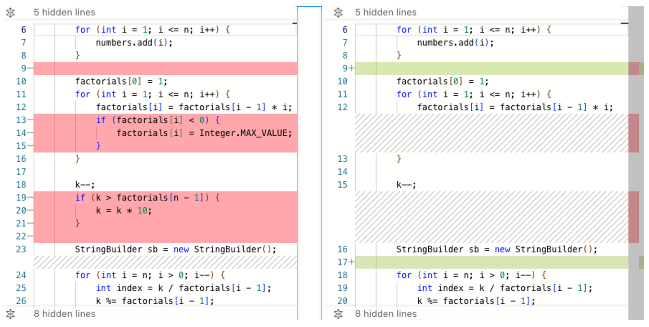
We saved the generated prompts in separate JSON files based on programming language (Listing 4[1]), and we organized each file to facilitate further processing and analysis. Next, we established robust communication with the LLMs by developing custom Python applications that interfaced with the models using APIs. To control costs and expand access to multiple models, we used the OpenRouter API—an OpenAI-compatible completion API that connects us to 318 models and providers. These APIs provided the necessary endpoints to transmit prompts and capture responses. We automated the entire process with Python scripts that sequentially sent each prompt to the designated LLM and captured the resulting output (i.e., the corrected code). We stored each model's outputs in dedicated JSON files and applied a naming convention that clearly associated each file with its respective model. Listing 5[1] displays an excerpt from the JSON file generated for the Claude 3.5 Haiku model and for the Java programs, and shows the solution for problem number 15, "Permutation Sequence." Building on the example of problem number 15, "Permutation Sequence," Figure 2 presents a side-by-side comparison of the original Java program and the version corrected by Claude 3.5 Haiku, clearly illustrating the modifications performed by the LLMs. In the final phase, we implemented the evaluation pipeline, where we rigorously tested each corrected solution generated by the LLMs against LeetCode's hidden test cases. We submitted each corrected solution individually to the testing environment and carefully monitored and documented its outcome. We deemed a solution successful only if it passed all test cases. We systematically logged the results in a Google Sheet, including metadata such as the problem name, programming language, and difficulty level alongside the outcomes for each LLM's submission.

```
{
    "id": 15,
    "programming_problem_name": "Permutation Sequence",
    "prompt": "\n    Please help me solve ...
    "extra error": ""
},
```

Listing 4: Excerpt of the JSON file containing the JAVA problems.

```
{
    "id": 15,
    "programming_problem_name": "Permutation Sequence",
    "fixed_solution": "```java\nclass Solution ..."
    }
```

Listing 5: Excerpt of the JSON file generated for the Claude 3.5 Haiku model and Java programs.



```
5 hidden lines
        for (int i = 1; i <= n; i++) {
            numbers.add(i);
        }

        factorials[0] = 1;
        for (int i = 1; i <= n; i++) {
            factorials[i] = factorials[i - 1] * i;
            if (factorials[i] < 0) {
                factorials[i] = Integer.MAX_VALUE;
            }
        }

        k--;
        if (k > factorials[n - 1]) {
            k = k * 10;
        }

        StringBuilder sb = new StringBuilder();
        for (int i = n; i > 0; i--) {
            int index = k / factorials[i - 1];
            k %= factorials[i - 1];
8 hidden lines
```

```
5 hidden lines
        for (int i = 1; i <= n; i++) {
            numbers.add(i);
        }

        factorials[0] = 1;
        for (int i = 1; i <= n; i++) {
            factorials[i] = factorials[i - 1] * i;
        }

        k--;
        StringBuilder sb = new StringBuilder();

        for (int i = n; i > 0; i--) {
            int index = k / factorials[i - 1];
            k %= factorials[i - 1];
8 hidden lines
```

Figure 2: Comparison between the original JAVA program and the JAVA program fixed by Claude 3.5 Haiku.

4 EXPERIMENTAL RESULTS AND ANALYSIS

In this section, we present and analyse the performance of the LLMs in achieving autonomous code repair. We computed the success rate by dividing the number of problems that an LLM successfully resolved by the total number of problems it attempted. Expressed

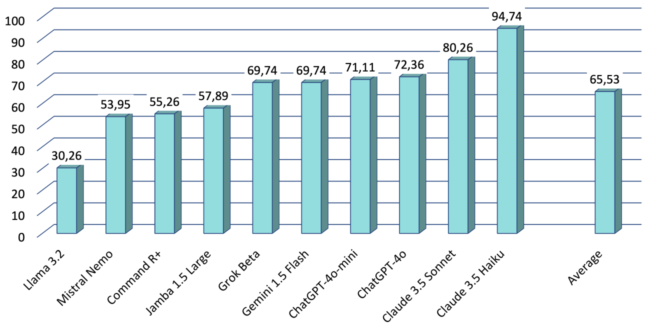


Figure 3: Success rate across different LLMs and average success rate

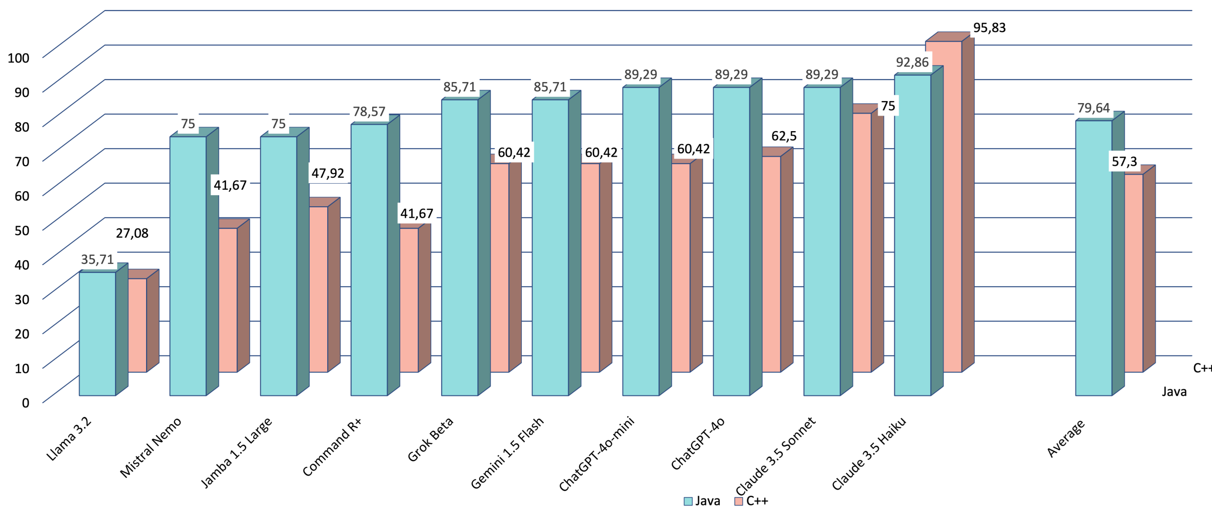


Figure 4: Success rate per programming language across different LLMs and average success rate.

as a percentage, this metric enabled us to quantitatively compare each model's ability to correct run-time errors across different programming languages. Figure 3 summarizes the overall success rates for each LLM.

Among the evaluated models, Claude 3.5 Haiku emerged as the top performer, achieving a success rate of 94.74%, while Claude 3.5 Sonnet also performed strongly with a success rate of 80.26%. The ChatGPT series showed comparable results, with ChatGPT-4o achieving 72.36% and ChatGPT-4o-mini reaching 71.11%. In contrast, Mistral Nemo, Command R+, and Jamba 1.5 Large exhibited relatively lower performance, recording success rates of 53.95%, 55.26%, and 57.89%, respectively. Grok Beta and Gemini 1.5 Flash achieved moderate results, both with success rates of 69.74%, while Llama 3.2 was the lowest-performing model, with a success rate of only 30.26%. When we analyse the distribution of success rates across all models, we calculate an average success rate of 65.53%, with performance ranging from a minimum of 30.26% (Llama 3.2) to a maximum of 94.74% (Claude 3.5 Haiku).

Figure 4 further illustrates the distribution of success rates across the two programming languages—Java and C++—for each LLM. We observed that, overall, Java exhibits higher success rates compared to C++ across most models, with the exception of Claude 3.5 Haiku. In Java, Claude 3.5 Haiku achieved the highest success rate of 92.86%, while models such as Claude 3.5 Sonnet, ChatGPT-4o, and ChatGPT-4o-mini each reached 89.29%. For C++, the overall performance was lower, with Claude 3.5 Haiku achieving 95.83% and Claude 3.5 Sonnet reaching 75%, whereas other models, including ChatGPT-4o, Gemini 1.5 Flash, and ChatGPT-4o-mini, recorded success rates around 60%. When we analyse the distribution across all models, the average success rate for Java is 79.64% and for C++ is 57.3%, with ranges from a minimum of 35.71% for Java and 27.08% for C++ (both observed in Llama 3.2) to a maximum of 92.86% for Java and 95.83% for C++ (both observed in Claude 3.5 Haiku).

Overall, these results not only demonstrate that the pipeline we implemented effectively supports a comprehensive comparison of LLMs for autonomous code repair, but also highlight the viability of using LLMs for autonomous error correction.

5 DISCUSSION

Despite significant variation among the ten models, our experiments show that LLMs exhibit promising potential for autonomously repairing run-time errors, aligning with the self-healing principles of autonomic computing. Claude 3.5 Haiku emerged as the top performer, achieving a success rate of 94.74%, while Llama 3.2 recorded the lowest at 30.26%. This disparity underscores the effects of differences in model architecture, training data, and optimization strategies. Thus, a hybrid approach that leverages the strengths of multiple LLMs may deliver more robust and reliable performance—especially in critical systems where even a minor failure in error correction could have severe consequences.

Our analysis also revealed variations in performance across programming languages. Overall, Java problems tended to achieve higher success rates compared to C++ problems. This may be attributed to biases in training data or inherent differences in the complexity of Java versus C++ problems. Furthermore, we observed that several models, especially with C++ problems, occasionally produced extraneous text in their code responses. This issue, likely stemming from misinterpretations of additional key-value fields in the prompts, sometimes led to failed submissions. Improving prompt engineering to eliminate such extraneous outputs represents an important area for enhancing model performance.

We opted not to fully automate our testing pipeline because we wanted to focus exclusively on problems that generated exceptions. An alternative approach would have been to automate the entire process, execute all problems, automatically identify those that produce exceptions, and then consider only those for evaluation. An automated testing pipeline, though currently unfeasible due to Leetcode's undisclosed test cases, would be a valuable enhancement for future studies. Developing such an automated framework could streamline evaluations and improve accuracy.

5.1 Threats to validity

Regarding internal validity, our manual approach to test and data collection introduces the potential for human error. Moreover, the limited number of problems in our dataset can create a bias in the reported success rates. External validity threats may also arise from our dataset's limitations. Since our dataset is exclusively sourced from Leetcode and focuses only on two programming languages—C++ and Java—the findings may not generalize to other languages such as Python or JavaScript, nor to more complex applications. In addition, the LLMs evaluated in our study represent only a subset of available models, and their performance might not reflect other LLMs or future iterations. Regarding conclusion validity, although the 76 programming problems provide a meaningful sample, a larger dataset would enhance the generalizability of our results and support more robust conclusions.

6 CONCLUSION AND LOOKING AHEAD

To the best of our knowledge, our study is the first peer-reviewed research that explicitly benchmarks ten LLMs for run-time error repair. The results confirm that LLMs hold significant promise for autonomously repairing run-time errors, thereby contributing to the evolution of self-healing software. While challenges remain—particularly in achieving consistent performance across diverse models and programming languages—our work lays a solid foundation for future research and practical implementations in AI-driven, self-healing software systems.

By benchmarking a diverse set of LLMs, our study underscores the importance of further research into hybrid approaches that combine multiple models to overcome individual weaknesses. Such integrated systems could be especially beneficial in safety-critical domains, such as aerospace or medical software, where reliability is of utmost importance. To this end, several technical approaches warrant investigation. One possible direction is to develop ensemble techniques and voting mechanisms, where outputs from different LLMs are combined through weighted voting or consensus methods. In this context, researchers could focus on dynamic weighting schemes or decision layers that adaptively prioritize outputs based on e.g., context, confidence scores, or historical performance. Another direction is the integration of multiple LLMs into structured pipelines. Such pipelines would allow for a multi-stage process where different models have different roles (e.g., one model diagnoses the error, another suggests a fix), possibly integrating feedback loops to achieve iterative refinement. Moreover, cooperative reasoning among models presents an exciting avenue for exploration. By employing chain-of-thought strategies, models could collaborate—using sequential processing where one model's output informs the next or through protocols for inter-model communication—to synergize their reasoning processes and generate more robust solutions. These directions not only aim to harness the complementary strengths of multiple LLMs, but also address the challenges of achieving consistent, scalable, and reliable autonomous code repair in real-world scenarios.

Building on the above results, future research could focus on developing a comprehensive LLM-based agent that autonomously monitors software execution, detects anomalies or failures, and triggers a self-healing process. This process would involve capturing contextual information about the failure, querying a LLM (or a combination of them) for potential fixes, and then dynamically integrating the generated code into the running system. Such an agent would not only react to errors as they occur, but also continuously learn from its environment, improving its repair strategies over time. Researchers could further explore adaptive mechanisms for error detection, where the agent uses historical data and real-time metrics to predict potential failure points before they lead to runtime errors. In parallel, developing robust methods for validating the generated fixes—possibly through a combination of automated testing and runtime monitoring—would be crucial to ensure that the self-healing process does not introduce new issues.

In addition, future work could explore expanding the dataset to include more diverse and complex software systems as well as additional programming languages. Automating the testing process would help reduce human error and improve evaluation efficiency. Further analysis of model performance by error type could reveal important insights, as different error categories may significantly impact repair success. Finally, investigating the inference time of each model across various problem types could inform trade-offs between accuracy and responsiveness.

7 DATA AVAILABILITY

We provide a public replication package at the following repository https://zenodo.org/records/14871706

REFERENCES

[1] Alessio Bucaioni, Hampus Ekedahl, Vilma Helander, and Phuong T Nguyen. 2024. Programming with ChatGPT: How far can we go? *Machine Learning with Applications* 15 (2024), 100526.

[2] Yuxiao Chen, Jingzheng Wu, Xiang Ling, Changjiang Li, Zhiqing Rui, Tianyue Luo, and Yanjun Wu. 2024. When Large Language Models Confront Repository-Level Automatic Program Repair: How Well They Done?. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings.* 459–471.

[3] McKinsey & Company. 2025. How an AI-enabled software product development life cycle will fuel innovation. https://www.mckinsey.com/industries/technology-media-and-telecommunications/our-insights/how-an-ai-enabled-software-product-development-life-cycle-will-fuel-innovation Accessed: 2025-03-04.

[4] David de Fitero-Dominguez, Eva Garcia-Lopez, Antonio Garcia-Cabot, and Jose-Javier Martinez-Herraiz. 2024. Enhanced automated code vulnerability repair using large language models. *arXiv preprint arXiv:2401.03741* (2024).

[5] Josu Diaz-de Arcaya, Juan López-de Armentia, Gorka Zárate, and Ana I Torre-Bastida. 2024. Towards the self-healing of Infrastructure as Code projects using constrained LLM technologies. In *Proceedings of the 5th ACM/IEEE International Workshop on Automated Program Repair.* 22–25.

[6] David Garlan and Bradley Schmerl. 2002. Model-based adaptation for self-healing systems. In *Proceedings of the first workshop on Self-healing systems.* 27–32.

[7] Vahid Garousi, Michael Felderer, Marco Kuhrmann, Kadir Herkiloğlu, and Sigrid Eldh. 2020. Exploring the industry's challenges in software testing: An empirical study. *Journal of Software: Evolution and Process* 32, 8 (2020), e2251.

[8] Ridhi Jain, Nicole Gervasoni, Mthandazo Ndhlovu, and Sanjay Rawat. 2023. A code centric evaluation of c/c++ vulnerability datasets for deep learning based vulnerability detection techniques. In *Proceedings of the 16th Innovations in Software Engineering Conference.* 1–10.

[9] Franco Zambonelli, Nicola Bicocchi, Giacomo Cabri, Letizia Leonardi, and Mariachiara Puviani. 2011. On self-adaptation, self-expression, and self-awareness in autonomic service component ensembles. In *2011 Fifth IEEE Conference on Self-Adaptive and Self-Organizing Systems Workshops.* IEEE, 108–113.

[10] Lan Zhang, Qingtian Zou, Anoop Singhal, Xiaoyan Sun, and Peng Liu. 2024. Evaluating Large Language Models for Real-World Vulnerability Repair in C/C++ Code. In *Proceedings of the 10th ACM International Workshop on Security and Privacy Analytics.* 49–58.