LLM-Driven Code Refactoring: Opportunities and Limitations

1st Jonathan Cordeiro
Queen's University
Kingston, Canada
19jac16@queensu.ca

2nd Shayan Noei
Queen's University
Kingston, Canada
s.noei@queensu.ca

3rd Ying Zou
Queen's University
Kingston, Canada
ying.zou@queensu.ca

***Abstract*—Refactoring is a systematic process of improving code quality while preserving the functional behavior of the software. In recent years, integrated development environments (IDEs) have added or improved automatic refactoring in their features, to enhance developers' productivity and reduce the likelihood of human errors. With the advancement and increasing popularity of large language models (LLMs), coding automation using them has gained enormous attention and has shown to be effective in performing refactorings on the source code automatically. However, this automation can carry the risk of introducing errors or hallucinations that may break or alter the software functionality. The error-proneness and the possibility of hallucinations in LLMs limit their ability to be fully integrated into an automated refactoring pipeline (e.g., IDEs) and often require humans in the loop to verify the performed modifications. In this position paper, we examine the limitations of existing LLM-based refactoring techniques. We propose research directions to address these limitations and improve the quality of LLM-based code refactoring for reliable software maintenance.**

***Index Terms*—code refactoring, code quality improvement, large language models, integrated development environments**

I. Introduction

Refactoring is a systematic process aimed at improving code quality without changing its external functionality [1]–[3]. To improve developers' productivity, the refactoring capabilities are integrated into the integrated development environments (IDEs) to ease refactoring operations. Therefore, developers can automatically perform refactoring without altering the code manually. In recent years, LLMs have been applied for automating various programming tasks (e.g., code repairing and generation) traditionally performed by developers to save time and mitigate human errors. LLMs have also demonstrated remarkable potential in generating and refactoring code for improved quality [4]. Despite this potential, the refactorings generated by LLMs suffer from the following limitations:

(1) Difficulty in integrating LLMs into IDEs and their limited contextual awareness. IDEs are commonly used for software development and offer features such as syntax highlighting, code suggestions, and refactoring aids. However, the integration of LLMs into IDEs for refactoring purposes presents unique challenges. Although LLMs can automate refactoring tasks, they lack the contextual understanding and user-specific adaptability that tools integrated within IDEs are designed to offer. This misalignment can lead to frustration for developers, as LLM-generated changes may conflict with project-specific conventions or fail to address the nuanced goals of the refactoring process. Furthermore, LLMs require significant computational resources [5], which can impact the responsiveness of the IDE and disrupt developers' workflows. Therefore, the seamless interaction between LLMs and IDE workflows, such as incremental refactoring or live debugging, remains an area with significant potential for improvement.

(2) Challenging to handle hallucinations in automated refactorings by LLMs. Refactorings often introduce errors or "hallucinations" — modifications that appear syntactically correct but lead to incorrect, suboptimal, or unexpected behavior when integrated into the codebase [6]. The hallucinations arise due to the LLM's tendency to make contextually inappropriate changes, particularly when the full scope of the code is unavailable or the refactoring requires a semantic understanding of the underlying program logic. Moreover, hallucinations in LLM-based refactorings can be traced back to deficiencies in the training data. LLMs are trained on vast datasets, but high-quality refactoring examples are underrepresented. This gap in the training data leads to LLMs making contextually inappropriate modifications that are not best practices or fail to meet the intended objectives of refactoring.

(3) Lack of an understanding of the purposes behind the refactoring process. Human developers refactor code to achieve specific purposes, such as removing code smells, improving readability, or adhering to design principles. LLMs operate without an explicit rationale for their modifications, often leading to changes that are misaligned with the goal of a project. For instance, an LLM may focus on simplifying code structure without addressing underlying technical debts, or it may improve readability while introducing new performance issues. In our previous work [7], we observe that StarCoder2 [8] demonstrates superior performance in reducing cohesion and complexity metrics, achieving greater modularity and structure. However, developers outperform StarCoder2 in reducing class coupling, highlighting developers' advantages in understanding the complexity of the code. Despite this, StarCoder2's ability to simplify code logic and reduce cyclomatic complexity makes it valuable for improving code refactoring, though its lack of explicit purpose alignment limits its broader applicability in context-dependent refactoring tasks.

(4) Inadequate or unavailable test cases to examine the logic introduced by LLM-generated refactorings. Test

cases, such as unit tests, are essential to validate if refactored code maintains functionality. Moreover, test cases are critical to ensure that LLM-generated refactorings do not introduce hallucinations and align with the intended functionality of a project, as LLMs may produce suggestions that are syntactically correct but semantically inappropriate. Test cases can be automatically generated using frameworks such as EvoSuite [9]. Existing approaches for automatic test case generation are often inadequate for comprehensive evaluation. Many methods focus on generating minimal or basic test cases, which may not sufficiently cover edge cases. Therefore, the refactored code can pass simple test cases but may fail under real-world conditions. With the lack of comprehensive test cases, verifying the correctness and performance of LLM-generated refactorings is challenging, and often requires human evaluation. In our previous work [7], we observe that the refactorings generated by StarCoder2 [8] pass 28.36% of the unit tests at pass@1 setting, improving to 57.15% at pass@5. The pass@ evaluates the effectiveness of the refactoring generated by considering that at least one of the top-k generated solutions passes all test cases. This improvement highlights the importance of regenerating refactorings multiple times to increase test pass rates but also underscores the challenges of achieving the same level of reliability as human developers.

Paper Organization. The remainder of this paper is structured as follows. Section II summarizes existing work on automatic refactoring, testing, and automated refactoring repair, highlighting their limitations and relevance to LLM integration in IDEs. Section III outlines the potential research directions for addressing the limitations of LLM-based refactoring. Finally, Section IV concludes the paper by summarizing our contributions and outlining future research directions.

II. Existing Work

In this section, we review existing work on automatic refactoring, testing, and automatic refactoring repair, and explore the challenges and opportunities of LLM-based refactoring and the integration of LLM-based refactoring into IDEs.

A. LLM Integration in IDEs

Integrated development environments (IDEs), such as IntelliJ IDEA [10] and Visual Studio [11] have incorporated AI-driven tools such as GitHub Copilot [12] to assist developers in code suggestions. However, AI-driven tools primarily focus on code completion and generation, offering limited support for complex refactoring tasks that require a deep understanding of code semantics, code architecture, and external functionalities. Recent research has explored the potential of LLMs in enhancing refactoring capabilities within IDEs [13]. The EM-Assist plugin for IntelliJ IDEA combines LLM-based suggestions with static analysis to recommend extract method refactorings that align more closely with developers' practices. In evaluations, EM-Assist successfully replicates developer-performed refactorings in 53.4% of cases, surpassing the 39.4% recall rate of the previous best-in-class static analysis tool. Despite the high recall, a human in the loop is still required to validate the refactorings, as there remains a chance of incorrect refactorings.

B. Automated Code Refactoring

Automated code refactoring using machine learning and LLMs has generated significant attention due to their potential to assist developers in improving code quality and reducing technical debts. Early approaches, such as *JRefactory* [14] and *Eclipse's Refactoring Engine* [15], rely on static and rule-based tools to apply common refactorings, such as renaming methods or extracting code fragments. While effective for straightforward tasks, these tools require substantial manual intervention to handle high-level or context-specific refactorings, limiting their scalability for real-world projects. Recent advancements in LLMs, such as Codex [16], GPT-4 [17], and StarCoder [8], [18], have introduced more flexible and context-aware solutions for automated refactoring. Previous studies [19] demonstrate the capabilities of the LLMs in code generation and refactoring tasks, showcasing their ability to generate human-like code and structural changes. However, these studies also highlight critical shortcomings, particularly in preserving semantic correctness and ensuring the refactorings pass rigorous quality checks, using unit tests.

Recent studies provide additional insights into the challenges and opportunities in LLM-based refactoring. AlOmar et al. [20] analyze 17,913 ChatGPT prompts and responses related to refactoring, revealing that developers often make generic refactoring requests, while ChatGPT typically specifies the intended improvements. The study highlights that developers prefer using explicit textual descriptions of refactoring needs alongside code fragments, with 41.9% of interactions following this pattern. The work emphasizes the importance of refining prompts to guide LLMs toward more context-aware and effective refactorings. Liu et al. [21] empirically evaluate the refactoring capabilities of ChatGPT and Gemini on a dataset of 180 real-world refactorings from 20 projects. ChatGPT identifies 15.6% of the refactoring opportunities initially, but explaining specific subcategories in the prompts increases its success rate to 86.7%. ChatGPT recommends 176 solutions, with 63.6% being as good as or better than those by human experts. However, 13 solutions introduce syntax errors or alter functionality. To mitigate such risks, the study proposes the RefactoringMirror tactic, which re-applies LLM-generated refactorings using tested engines. The accuracy of reapplication achieves 94.3%, successfully avoiding all buggy refactorings.

C. Testing LLM-generated Code

Frameworks, such as *LLM4TDD* [22] integrate LLMs into Test-Driven Development (TDD) and enable LLMs to generate code based on test cases. The TDD approaches can improve code quality. However, the TDD approach depends on highly structured prompts and high-quality tests. Therefore, the TDD approach lacks robustness when handling unstructured or context-dependent code refactoring tasks. The TDD approach could be used to generate refactorings by inputting the test

case into the prompt and asking LLM to generate a refactoring that can pass the test case. Moreover, mutation testing has also been extensively used to evaluate test suite quality and identify subtle flaws in automatically generated refactorings. For example, Papadakis *et al.* [23] explores the integration of mutation testing with refactoring tools to reveal behavioral changes introduced by automated refactorings. These studies [22], [23] underscore the importance of evaluating the downstream effects of refactorings but do not propose solutions for correcting code (e.g., refactorings) that fail tests.

D. Automated Refactoring Repair

Previous studies [4], [24], [25] show that LLMs can propose meaningful structural changes, but they often introduce bugs or fail to account for the broader context of the codebase. This could be problematic for large projects where changes in one part of the code can have cascading effects elsewhere. Automated Program Repairing (APR) systems [26], [27], have provided techniques for identifying and fixing bugs by generating minimal changes that pass test cases. While APR focuses on repairing faults, its methodologies can be applied to automated refactoring correction, which involves continuous code enhancement and validating them against test cases, enabling the refinement of refactoring suggestions based on the results of each iteration. In this context, approaches such as those by Shirafuji [4] aim to enhance refactoring suggestions by leveraging few-shot prompting to provide the model with additional information. However, the generalizability of these methods is limited by their reliance on the context of the provided prompts and do not dynamically adapt based on real test scenarios.

III. Future Research Directions

In this section, we discuss the possible future research directions to address the limitations of LLM-generated refactorings, discussed in Section I. To improve the automation and integration of LLMs into IDEs, future research directions can focus on providing developers with reliable and contextually appropriate refactoring recommendations for code quality improvement. A comprehensive framework is needed to focus on integrating automated test generation, refining training datasets, enhancing contextual understanding, and mitigating hallucinations. Below, we discuss the key research directions and propose some initial solutions.

Tighter integration of automated test generation can improve LLM-based refactoring reliability. To address the limitation of IDEs' inability to validate the functional correctness of the generated code, it is essential to build a tighter integration of automated test generation with refactoring processes. By integrating LLM-based refactoring tools with frameworks like EvoSuite [9] or Randoop [28], developers can automatically generate unit tests for modified code when there is no existing test case available. These tests can ensure that the generated refactorings not only preserve functionality but also enhance code quality. An important research direction involves enhancing the integration of automated test generation with LLM-based refactoring. These mechanisms can assess the correctness and effectiveness of refactorings on code quality in real-time, providing feedback to developers. Examples of such mechanisms include:

- Static analysis tools that detect potential issues in the refactored code before execution.
- Test case prioritization approaches to identify existing test cases for verifying the refactored code.
- Mutation testing to evaluate the robustness of generated unit tests.
- Symbolic execution to ensure that all paths within the refactored code adhere to the expected functional behaviors.

Such approaches aim to ensure that the generated refactorings align more closely with project goals and reduce the risk of introducing errors, thereby improving the overall reliability of LLM-generated refactorings.

Curating high-quality datasets enhances LLM refactoring performance. The quality of LLM-based refactorings is heavily influenced by the training data. Current code generation models are often trained on general-purpose code repositories, which may lack the depth and specificity needed for effective refactoring tasks. To enhance the performance of generating high-quality code, it is critical to create and curate datasets that explicitly focus on high-quality refactoring tasks. These datasets should include:

- Real-world examples of code refactorings with detailed justifications explaining the rationale behind each change (e.g., improving readability, reducing code smells, or optimizing performance).
- Clear patterns that demonstrate best practices and common mistakes in refactoring.
- Examples tailored to specific domains, such as embedded systems, machine learning pipelines, or web development frameworks, to improve domain-specific refactoring capabilities.

These datasets can be used to fine-tune LLMs or serve as benchmarks for evaluating refactoring tools, ensuring that LLMs generate more contextually relevant and semantically accurate suggestions.

Enriched prompts improve contextual understanding in LLM-based refactorings. Another challenge in LLM-based refactoring is the lack of sufficient contextual information provided to the LLM. Current prompts often focus on "how" to refactor without addressing the "why." By enriching prompts with explicit information about the purpose of refactoring, such as the removal of specific code smells (e.g., long methods or duplicate code) or the improvement of maintainability and performance, LLMs can generate more targeted and effective suggestions. Future research can explore the use of prompting techniques (e.g., chain-of-thought prompting and few-shot prompting) to break down refactoring tasks into sequential steps that align with developers' intentions. For example, a prompt might guide the LLMs by identifying a code smell, suggesting a solution, and providing a justification for the

suggested change. Such approaches can enhance the practical utility of LLM-based tools in real-world scenarios.

Mitigating hallucinations ensures semantic correctness in refactorings. LLMs are prone to generating "hallucinations," for various reasons. For example, the LLMs suggest changes that are syntactically valid but semantically incorrect or the generated refactorings are irrelevant to the given context. Identifying the root causes of hallucinations requires analysis of failure cases, and conducting targeted interventions. To mitigate hallucinations in LLM-generated refactorings, model calibration techniques, such as uncertainty quantification, can be used to help identify instances where the LLM lacks confidence in its suggestions. Additional validation steps can be incorporated when necessary. Moreover, implementing post-processing pipelines, such as rule-based or heuristic filtering methods, can serve to eliminate contextually irrelevant refactoring suggestions before they are presented to developers. Furthermore, requiring the LLM to generate justifications alongside its refactorings can enhance its transparency, and allow developers to assess the validity and rationale of the proposed changes. Developing the aforementioned mechanisms will improve the semantic correctness of LLM-generated refactorings, making the generated refactorings align more closely with developers' objectives and project requirements.

IV. Summary

In this paper, we discuss the current limitations of LLMs in refactoring generation and highlight the challenges of integrating LLM-based refactoring into IDEs. We also explore the potential research directions to address the challenges. We envision that LLM-based refactoring tools could be fully automated, without human intervention, and effectively integrated into real-world software development workflows, such as those found in IDEs, if the current limitations are overcome. To address the limitations of LLM-based refactoring hallucinations, future research can focus on generating comprehensive test cases. Additionally, future research can focus on training or fine-tuning refactoring expert LLMs with rich refactoring datasets. Future work can also use more context-aware and hallucination-preventative prompts to improve the quality of LLM-based refactorings. Advancing the identified research directions on LLM-based refactoring tools can improve developers' productivity, code quality, and maintainability in software systems.

References

[1] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[2] S. Noei, H. Li, S. Georgiou, and Y. Zou, "An empirical study of refactoring rhythms and tactics in the software development process," *IEEE Transactions on Software Engineering*, vol. 49, no. 12, pp. 5103–5119, 2023.

[3] S. Noei, H. Li, and Y. Zou, "Detecting refactoring commits in machine learning python projects: A machine learning-based approach," *ACM Transactions on Software Engineering and Methodology*, 2024.

[4] A. Shirafuji, Y. Oda, J. Suzuki, M. Morishita, and Y. Watanobe, "Refactoring programs using large language models with few-shot examples," in *2023 30th Asia-Pacific Software Engineering Conference (APSEC)*, pp. 151–160, 2023.

[5] M. Xu, W. Yin, D. Cai, R. Yi, D. Xu, Q. Wang, B. Wu, Y. Zhao, C. Yang, S. Wang, Q. Zhang, Z. Lu, L. Zhang, S. Wang, Y. Li, Y. Liu, X. Jin, and X. Liu, "A survey of resource-efficient llm and multimodal foundation models," 2024.

[6] L. Huang, W. Yu, W. Ma, W. Zhong, Z. Feng, H. Wang, Q. Chen, W. Peng, X. Feng, B. Qin, and T. Liu, "A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions," *ACM Transactions on Information Systems*, Nov. 2024.

[7] J. Cordeiro, S. Noei, and Y. Zou, "An empirical study on the code refactoring capability of large language models," *arXiv preprint arXiv:2411.02320*, 2024.

[8] A. Lozhkov *et al.*, "Starcoder 2 and the stack v2: The next generation," 2024.

[9] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, (New York, NY, USA), p. 416–419, Association for Computing Machinery, 2011.

[10] I. IDEA. https://www.jetbrains.com/idea, November 2024.

[11] V. Studio. https://visualstudio.microsoft.com, November 2024.

[12] G. Copilot. https://github.com/features/copilot, November 2024.

[13] D. Pomian, A. Bellur, M. Dilhara, Z. Kurbatova, E. Bogomolov, A. Sokolov, T. Bryksin, and D. Dig, "Em-assist: Safe automated extractmethod refactoring with llms," in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, FSE '24, p. 582–586, ACM, July 2024.

[14] M. Bittman, R. Roos, and G. Kapfhammer, "Creating a free, dependable software engineering environment for building java applications," in *1st Workshop on Open Source Software Engineering at ICSE 2001*, 2001.

[15] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd ed., 2009.

[16] M. Chen *et al.*, "Evaluating large language models trained on code," 2021.

[17] OpenAI, J. Achiam, *et al.*, "Gpt-4 technical report," 2024.

[18] R. Li *et al.*, "Starcoder: may the source be with you!," 2023.

[19] D. G. et al., "Graphcodebert: Pre-training code representations with data flow," 2021.

[20] E. A. AlOmar, A. Venkatakrishnan, M. W. Mkaouer, C. Newman, and A. Ouni, "How to refactor this code? an exploratory study on developer-chatgpt refactoring conversations," in *Proceedings of the 21st International Conference on Mining Software Repositories*, MSR '24, (New York, NY, USA), p. 202–206, Association for Computing Machinery, 2024.

[21] B. Liu, Y. Jiang, Y. Zhang, N. Niu, G. Li, and H. Liu, "An empirical study on the potential of llms in automated software refactoring," 2024.

[22] S. Piya and A. Sullivan, "Llm4tdd: Best practices for test driven development using large language models," in *Proceedings of the 1st International Workshop on Large Language Models for Code*, LLM4Code '24, (New York, NY, USA), p. 14–21, Association for Computing Machinery, 2024.

[23] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. Le Traon, and M. Harman, "Mutation testing advances: an analysis and survey," in *Advances in computers*, vol. 112, pp. 275–378, Elsevier, 2019.

[24] A. Peruma, S. Simmons, E. A. AlOmar, C. D. Newman, M. W. Mkaouer, and A. Ouni, "How do i refactor this? an empirical study on refactoring trends and topics in stack overflow," *Empirical Software Engineering*, vol. 27, Oct. 2021.

[25] D. Pomian, A. Bellur, M. Dilhara, Z. Kurbatova, E. Bogomolov, A. Sokolov, T. Bryksin, and D. Dig, "Em-assist: Safe automated extractmethod refactoring with llms," in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, FSE 2024, (New York, NY, USA), p. 582–586, Association for Computing Machinery, 2024.

[26] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pp. 691–701, 2016.

[27] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 54–72, 2012.

[28] C. Pacheco and M. D. Ernst, "Randoop: feedback-directed random testing for java," in *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion*,

OOPSLA '07, (New York, NY, USA), p. 815–816, Association for Computing Machinery, 2007.