Identification and Optimization of Redundant Code Using Large Language Models

Shamse Tasnim Cynthia
Department of Computer Science
University of Saskatchewan
Saskatoon, Canada
shamse.cynthia@usask.ca

Abstract—**Redundant code is a persistent challenge in software development that makes systems harder to maintain, scale, and update. It adds unnecessary complexity, hinders bug fixes, and increases technical debt. Despite their impact, removing redundant code manually is risky and error-prone, often introducing new bugs or missing dependencies. While studies highlight the prevalence and negative impact of redundant code, little focus has been given to Artificial Intelligence (AI) system codebases and the common patterns that cause redundancy. Additionally, the reasons behind developers unintentionally introducing redundant code remain largely unexplored. This research addresses these gaps by leveraging large language models (LLMs) to automatically detect and optimize redundant code in AI projects. Our research aims to identify recurring patterns of redundancy and analyze their underlying causes, such as outdated practices or insufficient awareness of best coding principles. Additionally, we plan to propose an LLM agent that will facilitate the detection and refactoring of redundancies on a large scale while preserving original functionality. This work advances the application of AI in identifying and optimizing redundant code, ultimately helping developers maintain cleaner, more readable, and scalable codebases.**

Index Terms—**code redundancy, LLM, code optimization**

I. Problem Statement

In software development, maintaining high code quality is essential for the scalability, maintainability, and reliability of software projects [1]. However, redundant code poses a significant challenge to read and maintain while offering little to no added functionality, impacting the maintainability and evolution of software [2]. Moreover, it frustrates the senior developers and increases the workload for maintaining the code [3], making bug fixing, feature updates, and future improvements become more time-consuming and error-prone [4]. The risk is even more pronounced in test case methods, where long, scenario-based code can violate coding best practices, leading to exaggerated and redundant test logic similar to issues in source code [5]. Furthermore, removing the redundant code is often non-trivial, and manually deleting them can lead to missed dependencies and encourage new bugs [6]. Thus, automatically identifying and optimizing redundant code is necessary to prevent additional challenges for developers while implementing new features or performing bug fixes.

Previous studies have analyzed source code to identify redundant code in terms of dead code or unused code and investigated their harmful impact on overall software quality [6]–[11]. For instance, Simone et al. [7] found that dead code is harmful both in the maintenance and design phases of software development. Shackleton et al. [6] noted that in large Python codebases, software evolution often leads to unused code and data, reducing efficiency and compromising user privacy. Similarly, Dandan et al. [9] found that redundant code complicates debugging and increases the risk of software bugs. Qiong et al. [3] and Malavolta et al. [8] focused on JavaScript, using both static and dynamic analyses to detect and remove redundant code. Suzuki et al. [10] found significant functional redundancy in code repositories, identifying 984 redundant method pairs across 41.17% of the analyzed projects. Similarly, Eduardo et al. [11] showed that redundant code issues persist during refactoring and negatively impact code quality.

While studies show that redundant code is common and harms software quality, they fail to identify the coding patterns that contribute most to redundancy. Additionally, many studies focus solely on specific types of redundant code, such as dead code, leaving other forms unexplored. For instance, dead code refers to programming that is never executed during a program's runtime and remains unused [12]. In contrast, redundant code involves multiple locations containing identical or nearly identical statements, which may have a broader impact on maintainability and code efficiency. This highlights the need for a more comprehensive investigation into all types of redundant code. Moreover, the reasons why developers introduce redundant code have not yet been thoroughly investigated, leaving a significant gap in understanding the root causes of these inefficiencies.

In our work, we aim to improve the overall quality and maintainability of open-source AI projects by identifying and optimizing redundant code. By leveraging LLMs, our research aims to automatically identify common patterns that contribute to redundancy in source code. Since AI systems often manage large-scale data and perform complex computations [13], it is imperative to reduce the redundancy in these systems to enhance maintainability and efficiency. With LLMs being widely adopted for code-related tasks [14]–[16], we plan to harness their capabilities to optimize code by effectively reducing redundancy. Furthermore, we seek to uncover the reasons why developers unintentionally introduce these inefficiencies. Through the creation of a framework capable of analyzing and optimizing these redundancies, this research will contribute to

enhancing code readability, understandability, and maintainability, ultimately supporting the sustainable development of AI systems.

Research Question: We aim to answer the following research questions:

RQ1: To what extent does the source code of AI systems contain redundant code, and what is the impact of the redundant code on the overall code quality?

RQ2: What are the common coding patterns that contribute to the introduction of redundant code in AI systems?

RQ3: What are AI software developers' perspectives on redundant code, including how they address it in their current practices and the challenges they encounter?

RQ4: How effective are LLMs in identifying redundant code in AI systems and providing effective optimization techniques?

II. EXPECTED OUTCOMES

This doctoral research aims to enhance the understanding of redundant code and propose strategies to optimize them effectively. The expected outcomes include -

1) Understanding the prevalence of redundant code in source code, its impact on overall code quality, and the reasons behind its introduction is essential for improving software development practices. Identifying the factors that lead developers to introduce redundancy—such as time constraints, lack of awareness, or insufficient adherence to software engineering principles—can provide critical insights into addressing the root causes.
2) Developing a tool powered by LLMs to eliminate redundant code while preserving its original functionality.

III. EXPECTED CONTRIBUTION

To achieve our research goal, we define the following contributions:

Analysis of redundant code prevalence: To analyze the presence of redundant code in source code, we will collect open-source software (OSS) projects focused on AI, following the methodology of Li et al. [17], and leverage three widely used LLMs: GPT-4 [18], Gemini [19], and Llama [20]. For each project, we will submit an entire file to the LLM to identify and optimize redundant code. The optimized code is then integrated back into the original codebase, followed by running test cases to ensure all functionalities remain intact. Additionally, we will use static analysis tools to evaluate code quality metrics such as Lines of Code (LOC) [21], Cyclomatic Complexity (logical complexity) [22], and Code Churn (rate of code changes) [23]. We will document the optimized code that successfully passes the test cases along with its metric values. If the optimized code fails, we will also record the failure details and reasons for the failed test cases.

Building a Catalog of Redundant Code Reasons and Identifying Common Patterns: We will create a comprehensive catalog of reasons behind the introduction of redundant code by analyzing existing studies and gathering insights from developers based on their experience, commit messages, code reviews, and other established resources. Simultaneously, we will leverage LLMs to identify recurring patterns of redundancy in codebases, such as copy-paste effects, repetitive logic, and overused conditional statements. These patterns will be analyzed for their impact on key metrics like the Code Maintainability Index (MI) [24] and Bug Density [25], providing actionable insights into the most detrimental patterns.

Prototype of an automated tool: We aim to utilize insights from our prior objectives to develop an LLM-agent capable of analyzing a codebase, optimizing files one at a time, reintegrating them into the original codebase, and running test cases to ensure functionalities remain intact. The agent will also analyze failed test cases to identify scenarios where optimizations were ineffective, enabling it to refine its approach for better results. This implementation has the potential to revolutionize how developers identify and optimize redundant code.

IV. PLANNED EVALUATION

The evaluation of this project involves multiple components. First, we will ensure that the code generated by the LLMs maintains the functional and operational characteristics of the original codebase by executing test cases to verify there are no regressions. A systematic literature review will also be conducted to position our work within the current state-of-the-art and identify relevant research gaps. To understand the reasons behind redundant code, we will collaborate with developers via semi-structured interviews and surveys to validate identified patterns through their feedback and insights. To evaluate the automated tool, we will conduct user studies to gather feedback on its ease of use and effectiveness. Additionally, we plan to use the NASA Task Load Index (NASA-TLX) [26] to assess the cognitive workload associated with using the tool. As the project is in its early stages, the evaluation framework may be refined based on future design decisions.

V. LIMITATIONS

The proposed research has limitations that may affect its scope and applicability. It focuses on open-source AI projects, limiting generalizability to proprietary systems with different coding practices. The reliance on LLMs like GPT-4, Gemini, and Llama introduces potential biases from training data, potentially overlooking edge cases or uncommon coding patterns. Incomplete test case coverage may fail to detect functionality issues in optimized code. Developer feedback, used to validate redundancy patterns, is subjective and context-dependent. Additionally, reliance on metrics e.g., Code Maintainability Index and Cyclomatic Complexity may not fully capture all aspects of software quality. To mitigate these limitations, in future, the research will incorporate diverse datasets from various domains, prioritize comprehensive test coverage, and refine metrics to include subjective quality factors. Feedback will be gathered from developers with diverse backgrounds to ensure broader validity, and the framework will be iteratively refined based on real-world testing and performance evaluations.

REFERENCES

[1] E. Van Emden and L. Moonen, "Java quality assurance by detecting code smells," in *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.* IEEE, 2002, pp. 97–106.

[2] S. Charalampidou, E.-M. Arvanitou, A. Ampatzoglou, P. Avgeriou, A. Chatzigeorgiou, and I. Stamelos, "Structural quality metrics as indicators of the long method bad smell: An empirical study," in *2018 44th Euromicro Conference on software engineering and advanced applications (SEAA).* IEEE, 2018, pp. 234–238.

[3] G. Qiong and W. Li, "An optimization method of javascript redundant code elimination based on hybrid analysis technique," in *2020 17th International Computer Conference on Wavelet Active Media Technology and Information Processing (ICCWAMTIP).* IEEE, 2020, pp. 300–305.

[4] A. Sundelin, J. Gonzalez-Huerta, and K. Wnuk, "The hidden cost of backward compatibility: when deprecation turns into technical debt-an experience report," in *Proceedings of the 3rd International Conference on Technical Debt*, 2020, pp. 67–76.

[5] E. Alégroth and J. Gonzalez-Huerta, "Towards a mapping of software technical debt onto testware," in *2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA).* IEEE, 2017, pp. 404–411.

[6] W. Shackleton, K. Cohn-Gordon, P. C. Rigby, R. Abreu, J. Gill, N. Nagappan, K. Nakad, I. Papagiannis, L. Petre, G. Megreli *et al.*, "Dead code removal at meta: Automatically deleting millions of lines of code and petabytes of deprecated data," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 1705–1715.

[7] S. Romano, C. Vendome, G. Scanniello, and D. Poshyvanyk, "A multi-study investigation into dead code," *IEEE Transactions on Software Engineering*, vol. 46, no. 1, pp. 71–99, 2018.

[8] I. Malavolta, K. Nirghin, G. L. Scoccia, S. Romano, S. Lombardi, G. Scanniello, and P. Lago, "Javascript dead code identification, elimination, and empirical assessment," *IEEE Transactions on Software Engineering*, vol. 49, no. 7, pp. 3692–3714, 2023.

[9] G. Dandan, W. Tiantian, S. Xiaohong, and M. Peijun, "Rc-finder: Redundancy detection for large scale source code," in *2012 Second International Conference on Instrumentation, Measurement, Computer, Communication and Control.* IEEE, 2012, pp. 243–248.

[10] M. Suzuki, A. C. de Paula, E. Guerra, C. V. Lopes, and O. A. L. Lemos, "An exploratory study of functional redundancy in code repositories," in *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM).* IEEE, 2017, pp. 31–40.

[11] E. Carneiro Oliveira, H. Keuning, and J. Jeuring, "Investigating student reasoning in method-level code refactoring: A think-aloud study," in *Proceedings of the 24th Koli Calling International Conference on Computing Education Research*, 2024, pp. 1–11.

[12] S. Romano, "Dead code," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME).* IEEE, 2018, pp. 737–742.

[13] J. Do, V. C. Ferreira, H. Bobarshad, M. Torabzadehkashi, S. Rezaei, A. Heydarigorji, D. Souza, B. F. Goldstein, L. Santiago, M. S. Kim *et al.*, "Cost-effective, energy-efficient, and scalable storage computing for large-scale ai applications," *ACM Transactions on Storage (TOS)*, vol. 16, no. 4, pp. 1–37, 2020.

[14] C. Zhang, Z. Wang, R. Zhao, R. Mangal, M. Fredrikson, L. Jia, and C. Pasareanu, "Attacks and defenses for large language models on coding tasks," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 2268–2272.

[15] B. Berabi, A. Gronskiy, V. Raychev, G. Sivanrupan, V. Chibotaru, and M. Vechev, "Deepcode ai fix: Fixing security vulnerabilities with large language models," *arXiv preprint arXiv:2402.13291*, 2024.

[16] H. Koziolek and A. Koziolek, "Llm-based control code generation using image recognition," in *Proceedings of the 1st International Workshop on Large Language Models for Code*, 2024, pp. 38–45.

[17] X. Li, S. Moreschini, Z. Zhang, and D. Taibi, "Exploring factors and metrics to select open source software components for integration: An empirical study," *Journal of Systems and Software*, vol. 188, p. 111255, 2022.

[18] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[19] G. Team, R. Anil, S. Borgeaud, J.-B. Alayrac, J. Yu, R. Soricut, J. Schalkwyk, A. M. Dai, A. Hauth, K. Millican *et al.*, "Gemini: a family of highly capable multimodal models," *arXiv preprint arXiv:2312.11805*, 2023.

[20] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar *et al.*, "Llama: Open and efficient foundation language models," *arXiv preprint arXiv:2302.13971*, 2023.

[21] E. Morozoff, "Using a line of code metric to understand software rework," *IEEE software*, vol. 27, no. 1, pp. 72–77, 2009.

[22] C. Ebert, J. Cain, G. Antoniol, S. Counsell, and P. Laplante, "Cyclomatic complexity," *IEEE software*, vol. 33, no. 6, pp. 27–29, 2016.

[23] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities," *IEEE transactions on software engineering*, vol. 37, no. 6, pp. 772–787, 2010.

[24] K. D. Welker, P. W. Oman, and G. G. Atkinson, "Development and application of an automated source code maintainability index," *Journal of Software Maintenance: Research and Practice*, vol. 9, no. 3, pp. 127–159, 1997.

[25] T. Bach, A. Andrzejak, R. Pannemans, and D. Lo, "The impact of coverage on bug density in a large industrial software project," in *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM).* IEEE, 2017, pp. 307–313.

[26] S. G. Hart, "Nasa-task load index (nasa-tlx); 20 years later," in *Proceedings of the human factors and ergonomics society annual meeting*, vol. 50, no. 9. Sage publications Sage CA: Los Angeles, CA, 2006, pp. 904–908.