Universal Library of Engineering Technology Research Article

ISSN: 3064-996X | Volume 3, Issue 2

Open Access | PP: 20-25

DOI: https://doi.org/10.70315/uloap.ulete.2026.0302004

Limitations and Technological Risks of Using Language Models for Refactoring Legacy Software Systems

Sergei Kuznetsov

Lead Software Engineer, Malaga, Spain.

Abstract

This article provides a comprehensive analysis of the technological limitations and risks associated with the application of large language models in the refactoring of legacy software systems. The study is conducted as a structured review and analytical synthesis of peer-reviewed publications devoted to AI-assisted software engineering, automated refactoring, and the modernization of legacy codebases. The analysis focuses on empirical and conceptual studies that examine the capabilities of language models in code transformation tasks and their interaction with complex software architectures. Particular attention is paid to the influence of contextual constraints, architectural dependencies, and accumulated business logic typical for legacy systems, which significantly complicate automated code transformation. The findings show that while large language models can effectively improve certain structural characteristics of code and assist in eliminating repetitive design defects, their performance becomes unstable when transformations affect functional semantics and inter-module dependencies. It is established that improvements in code quality metrics do not necessarily correspond to the preservation of program behavior, which creates additional technological risks during modernization. The paper proposes a structured classification of risks associated with LLM-based refactoring and demonstrates that the most reliable approach currently lies in hybrid workflows combining automated code suggestions with human validation, testing, and static analysis. The results contribute to a better understanding of the safe integration of intelligent assistants into software modernization processes and may inform the design of risk-aware AI-supported refactoring practices.

Keywords: *Large Language Models; Software Engineering; Code Refactoring; Legacy Software Systems; Software Modernization; Artificial Intelligence.*

INTRODUCTION

Modern software systems are becoming increasingly complex. At the same time, a significant portion of critical digital infrastructure continues to operate on legacy software. Such systems are widely used across banking, manufacturing, and the public sector. Frequently, they are structured as large monolithic applications written in Java, C/C++, or COBOL. Upgrading them remains a challenging endeavor [7]. Any modification to this type of code can disrupt mission-critical logic. Against this backdrop, large language models have begun to actively penetrate software development practices. They are already being deployed for code generation, bug detection, and program analysis. As a result, they are increasingly being viewed as a tool for automating refactoring.

Refactoring, however, is a more complex task than simply generating a code snippet. Simply obtaining plausible program text is insufficient in this context. It is necessary to preserve the system's behavior without breaking the dependencies between its components. It is precisely here that the limitations of language models become apparent. They can suggest changes that appear convincing on the surface but are fundamentally incorrect [3]. Their output is highly dependent on the training data and the specific context provided to the model. Meanwhile, the majority of existing research focuses primarily on the accuracy of code generation and automated bug fixing. The risks associated with applying these models to the modernization of legacy systems have been studied considerably less. This creates a research gap that demands distinct and systematic examination. Unlike existing studies, this research proposes a structured model of technological risks in LLM-assisted refactoring, accounting not only for code generation errors but also for the architectural, contextual, and methodological constraints of using language models in the modernization of complex software systems [13].

Citation: Sergei Kuznetsov, "Limitations and Technological Risks of Using Language Models for Refactoring Legacy Software Systems", Universal Library of Engineering Technology, 2026; 3(2): 20-25. DOI: https://doi.org/10.70315/uloap.ulete.2026.0302004.

The aim of this study is to identify and systematize the technological risks of applying large language models to the refactoring of legacy software systems. To achieve this objective, the paper addresses the following tasks:

– review modern intelligent software development tools based on language models;

– compare the outcomes of source code refactoring performed by human developers versus language models;

– identify the primary technological limitations of using language models during code transformation;

– systematize the main types of risks associated with automated refactoring;

– develop practical recommendations for the safe application of language models in the process of software system modernization.

The research hypothesis is as follows. Large language models can improve specific code quality metrics. Yet, their application in refactoring legacy software systems is fraught with significant technological risks. These risks emerge because the models fail to adequately account for complex program architectures. Furthermore, they do not consistently preserve the functional logic of the system. Consequently, the proposed transformations may appear correct while ultimately leading to runtime errors.

The scientific novelty of this research encompasses several aspects. The paper introduces an original classification of technological risks arising from the use of language models in software refactoring. A systematic review of the limitations of such models in the context of legacy software modernization is conducted. Based on the findings, practical recommendations are formulated for a safer utilization of intelligent refactoring tools during software system upgrades.

MATERIALS AND METHODS

The research methods employed include a theoretical synthesis of academic publications on the application of large language models in software engineering, a structural systematization of limitations in automated code refactoring, and a comparative analysis of program transformations executed by language models and human developers.

The study is structured as a systematic review of open-access publications from 2023 to 2026, sourced from international peer-reviewed journals and academic repositories. The literature search was conducted across databases such as Google Scholar, IEEE Xplore, ACM Digital Library, ScienceDirect, and SpringerLink, as well as the arXiv and MDPI repositories. The search strategy relied on combinations of keywords: “large language models software engineering”, “LLM code refactoring”, “AI code modernization”, “legacy system modernization”, “automated code refactoring”, “LLM code review”, and “AI program repair”, utilizing AND/OR Boolean operators.

During the identification phase, 47 publications were found. Following the removal of duplicates and an initial screening of titles and abstracts, papers unrelated to the tasks of refactoring and modernizing software systems were excluded. After a full-text assessment, 15 studies were included in the final sample.

The publication analysis procedure encompassed several stages: an initial source search, duplicate removal, selection based on thematic relevance, full-text content evaluation, and subsequent thematic classification of the research results. In the final stage, an analytical interpretation of the identified limitations regarding the use of language models in code refactoring tasks was conducted.

The modest size of the final sample is attributed to the narrow specialization of the topic under review, as the vast majority of publications are dedicated to code generation tasks, whereas studies directly analyzing the application of LLMs in refactoring and modernizing legacy software systems are significantly less common.

The selected publications cover the core domains of language model applications in software engineering: code generation, automated refactoring, defect and vulnerability detection, and the modernization of legacy software systems. The extracted findings were utilized to analyze the technological risks of deploying language models for the refactoring of legacy software systems.

RESULTS

Modern artificial intelligence tools have already become an integral part of software development practice. They are employed for code generation, bug fixing, program analysis, and the automation of specific refactoring stages [9]. The most prominent group includes GitHub Copilot, GPT-based assistants, and specialized language models designed for code [6]. These systems are trained on massive repositories of source code and can execute tasks that previously required human developer involvement at every step.

Nevertheless, their capabilities vary across different types of tasks. Such tools perform most reliably where the transformation relies on repetitive patterns and local modifications [8]. When transitioning to more complex tasks involving system architecture and extensive cross-module dependencies, their reliability decreases noticeably [12]. This leads to an initial conclusion: while modern artificial intelligence tools are already valuable in the development environment, they currently function more as assistants in refactoring tasks rather than fully autonomous code transformation mechanisms. Table 1 presents a comparison of refactoring effectiveness between a language model and human developers.

Table 1. Comparison of the effectiveness of refactoring by LLMs and developers (Compiled by the author based on source: [3])

Indicator	LLM (StarCoder2)	Developers
Reduction of code smells	44.36%	24.27%
Average improvement in quality metrics	19.32%	17.46%
Unit test pass rate	28.36–57.15%	100%
Improvement in class cohesion	lower	higher

The data in Table 1 indicate that a language model can be effective in eliminating repetitive design flaws. It yields a superior outcome in reducing code smells and provides a slightly better enhancement of specific code quality metrics. This picture shifts, however, when functional correctness is taken into account. Human developers preserve program behavior with significantly greater reliability, whereas a large portion of the automatically generated transformations fails testing.

This finding is significant for two reasons. Primarily, it demonstrates that improving quality metrics does not equate to maintaining the correct operation of the system. Furthermore, it confirms that architecturally sensitive modifications are still better handled by humans [4]. Consequently, automated refactoring cannot be evaluated solely on the basis of superficial improvements to code structure. Table 2 outlines the outcomes of automated loop refactoring across different repositories [11].

Table 2. Results of automatic loop refactoring in different repositories[1]

Repository	Loops Detected	Compilable Transformations	Non-compilable Transformations
iridium	36	16	20
rapid dweller-benerator-ce	729	357	368
openrefine	1200	196	1004

Table 2 reveals that the proportion of viable transformations varies drastically from project to project. In some instances, nearly half of the suggestions compile successfully, while in others, only a small fraction does. This highlights a strong dependence of the output on the source code context. If the model lacks access to the complete dependency graph, it is more prone to producing incomplete or erroneous transformations.

A correlation has been established between the complexity of program constructs and the likelihood of errors during automated code transformation. As the number of nested conditions, side effects, and interconnected elements increases, the risk of incorrect modifications also rises [5]. This issue is of particular relevance to legacy software systems, as they are often characterized by lengthy, tightly coupled code segments shaped by extended evolutionary cycles [10]. Under these circumstances, automated code transformation necessitates mandatory correctness verification, encompassing compilation, testing, and human oversight. Figure 1 depicts the structure of language model suggestions during the generation of the Extract Method refactoring.

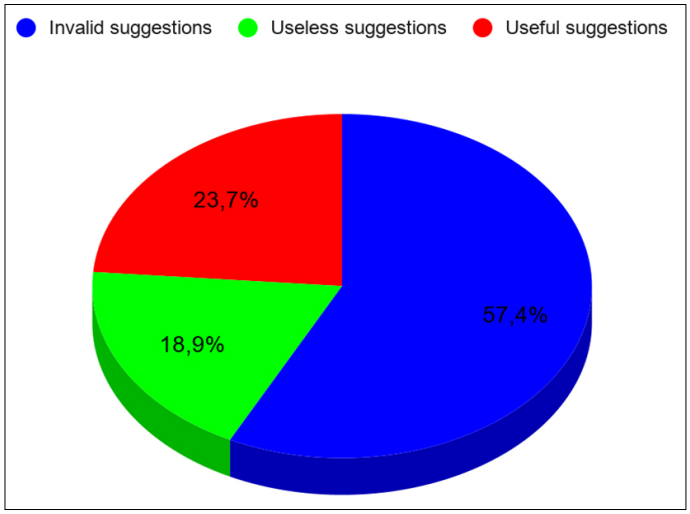

Figure 1. Structure of LLM Suggestions When Generating Extract Method[2]

1 A. Midolo and E. Tramontana, "Refactoring loops in the era of LLMs: A comprehensive study," Future Internet, vol. 17, no. 9, p. 418, 2025. https://doi.org/10.3390/fi17090418

2 A. Midolo and E. Tramontana, "Refactoring loops in the era of LLMs: A comprehensive study," Future Internet, vol. 17, no. 9, p. 418, 2025. https://doi.org/10.3390/fi17090418

Figure 1 indicates that the majority of the suggestions consist of invalid and useless variants. Invalid transformations form the largest category, accounting for 57.4% of all generated proposals. These variants cannot be applied in practice due to syntax errors, incorrect variable references, or missing dependencies. Additionally, 18.9% of the suggestions fall into the useless category, representing formally correct transformations that fail to improve the program structure or lack practical value for refactoring. Combined, the share of invalid and useless suggestions reaches 76.3%, implying that more than three-quarters of the generated outputs cannot be utilized by a developer without substantial rework or outright rejection. Useful transformations make up merely 23.7% of all proposals. Therefore, only about one in four suggestions from the language model has the potential to be implemented during the refactoring process. The root cause of this situation is the limited context window of the model and the lack of direct access to the complete dependency graph of the software system. Consequently, the model generates transformations based on a localized code analysis, completely overlooking the broader architecture of the software project.

This finding carries direct practical implications. Even if the model is capable of suggesting a successful transformation, the developer is forced to sift through a vast volume of unsuitable recommendations. Such a scenario diminishes the engineering value of automation and elevates the cost of making an error. Thus, at its current stage of maturity, the language model is more appropriate as a source of preliminary drafts rather than as a standalone refactoring tool.

The obtained results demonstrate a consistent pattern. Language models are already beneficial in software development and localized code transformation tasks. They are capable of improving specific quality metrics and proposing viable modification options [2]. Nevertheless, when addressing the refactoring of legacy systems, the importance of context, architectural cohesion, and functional correctness increases sharply.

It follows that the primary challenge of modern automated refactoring lies not in a lack of generative capabilities within the models, but rather in the absence of reliable mechanisms for filtering and validating the outputs. For this reason, the most realistic and secure approach remains a hybrid paradigm, where the model generates alternatives while compilation, testing, static analysis, and the human developer serve as the filter to validate the changes.

DISCUSSION

The research findings illustrate that large language models have already emerged as a prominent instrument in software engineering. They possess the ability to suggest code transformation options, eliminate a subset of common defects, and enhance isolated software quality metrics [15]. This does not imply, however, that these models can be considered a reliable mechanism for refactoring complex software systems.

The core problem begins at the semantic level of the code. A language model can render a snippet cleaner and more concise. Yet, this does not guarantee the preservation of its behavior. Surface-level structural improvements do not inherently translate into intact program logic. This poses an acute danger for legacy systems. Within such frameworks, even a minor alteration can disrupt critical business logic.

Another substantial limitation is intrinsically tied to context. A language model typically processes an isolated fragment rather than the entire system. Consequently, it frequently fails to recognize connections across files, methods, libraries, and internal project dependencies [14]. As the system scales, this risk amplifies. In complex projects, the behavior of a single code segment is often determined not locally, but through multiple interconnected components.

Difficulties also manifest at the architectural level. Language models handle local modifications more adeptly, capable of simplifying a construct or proposing a method extraction. Conversely, they are noticeably weaker in tasks that demand consideration of the overall system structure. In these scenarios, modularity, cohesion, and the general organization of the project are paramount. This is precisely why the efficacy of LLM-based refactoring cannot be appraised solely through code quality metrics. The primary criterion remains the preservation of program behavior and the overall stability of the system.

The conducted analysis demonstrates that the risks associated with LLM refactoring are not confined to isolated generation errors. Instead, they form a robust system of constraints. To describe these constraints more precisely, the paper introduces an original classification of the technological risks of LLM refactoring. Table 3 presents the proposed risk structure.

Table 3. Author's Classification of Technological Risks of LLM-Based Refactoring (Author's development)

Risk Type	Nature of Manifestation
Semantic	violation of the program's functional behavior
Architectural	deterioration of the system structure
Contextual	loss of dependencies between components
Computational	high processing cost
Methodological	overestimation of model effectiveness

Semantic risks stem from the possibility that the transformed code may operate differently than the original [1]. Architectural risks emerge when a localized enhancement leads to the degradation of the overall program structure. This can manifest as excessive code fragmentation, a proliferation of helper methods, and increased difficulty in navigating the system. Contextual risks arise when the model lacks

visibility into the complete software context, causing it to lose critical cross-component links. Computational risks are associated with the rising costs of processing, particularly during repeated iterations of generating and refining the output. Methodological risks materialize when the model's effectiveness is evaluated against convenient, yet incomplete, benchmarks. In such instances, an increase in code quality metrics can fabricate a false impression of reliability, even if functional correctness remains unverified.

The deployment of AI assistants is altering the development process and the very role of the developer. As the quality of generation improves, the professional writes code from scratch less frequently, increasingly shifting toward reviewing, correcting, and curating the model's suggestions. This fundamental shift modifies the logic of professional labor. On the one hand, this transition can indeed accelerate the execution of routine tasks. On the other hand, it cultivates a novel dependency on automated prompts. Should a developer begin to lean too heavily on the model, the depth of their independent system analysis inevitably declines. This represents a profound danger when handling legacy code, where the objective is not simply to modify the structure, but to comprehend the historical rationale behind its specific design. A distinct issue relates to professional development. Under the constant influence of automated recommendations, certain engineering skills may begin to atrophy. This primarily concerns the skills required for manual refactoring, architectural analysis, and deciphering complex codebases. Consequently, the AI assistant moves beyond merely aiding the developer to partially replacing the very activities that historically forged their expertise.

For this reason, the integration of LLMs into development must be approached as both a technical and a professional issue. The greater the degree of automation, the more critical it becomes to maintain the human in the role of an active interpreter rather than a passive corroborating operator. The proposed classification is valuable not merely as a descriptive framework for these problems. It establishes a more precise analytical lens. Under this paradigm, the LLM is treated as a source of beneficial, yet potentially risky, transformations that mandate external engineering validation.

In practice, a blended operational model is most frequently adopted. The language model drafts a potential variant for code transformation. Subsequently, the developer scrutinizes the proposed modification and evaluates its impact on system behavior. A decision regarding the implementation of the transformation is then finalized. At the current juncture, this format is widely regarded as the most robust approach for real-world development scenarios.

A multi-tiered verification must be a mandatory component of this process. Any automatically proposed alteration must undergo static analysis, compilation, and unit testing. Absent these steps, even an ostensibly successful refactoring cannot be deemed reliable. Another vital condition relates to the scope of application. It is most justifiable to deploy LLMs for localized and constrained transformations. These include simplifying minor constructs, eliminating code duplication, refining naming conventions, and preliminarily extracting methods. Conversely, tasks impacting system architecture, inter-module dependencies, and the logic of outdated platforms require far stricter oversight and should not be fully delegated to the model. In practical terms, this dictates the following approach. An LLM is highly useful as a mechanism for accelerating preliminary engineering groundwork. However, it must not act as an autonomous engine for system modernization. As the cost of failure escalates, the permissible zone of automation must correspondingly narrow.

CONCLUSION

In an era marked by the proliferation of large language models, refactoring legacy software systems should be viewed not as a simple task of automated code generation, but as a governed engineering process where automated suggestions serve as merely one component of decision-making. The ability of developers to verify the impact of proposed changes on system behavior and to maintain the integrity of architectural dependencies assumes critical importance.

The primary challenge in applying language models is no longer the generation of transformation variants itself, but rather their interpretation and engineering validation. As the architecture of the software system grows more complex and the density of accumulated dependencies increases, result-filtering procedures—including static analysis, testing, and architectural review—become exponentially more vital. Without such mechanisms, the automation of transformations can construct an illusion of improved code structure while simultaneously escalating the risks of disrupting functional logic.

The greatest practical benefit is realized when language models are employed as accelerators for preliminary engineering tasks, assisting in the formulation of localized transformation options without dictating the final software architecture. Under these conditions, intelligent assistants augment the developer's capabilities without supplanting their architectural judgment or responsibility for system correctness.

The subsequent advancement of intelligent refactoring tools is tied less to the continued expansion of model generative capacities and more to the establishment of integrated engineering loops for change verification. Future research prospects lie in the creation of formal procedures to evaluate the reliability of automated transformations, alongside the development of methodologies to integrate language models with dependency analysis tools and architectural verification frameworks for software systems.

REFERENCES

1. B. Althani, "A multi-objective statistical framework for evaluating LLM-based code modernization: Transformation pattern analysis and effect size validation," Computers, vol. 15, no. 3, p. 148, 2026. https://doi.org/10.3390/computers15030148
2. N. Baumgartner, P. Iyenghar, T. Schoemaker, and E. Pulvermüller, "The scalable detection and resolution of data clumps using a modular pipeline with ChatGPT," Software, vol. 4, no. 1, p. 3, 2025. https://doi.org/10.3390/software4010003
3. J. Cordeiro, S. Noei, and Y. Zou, "An empirical study on the code refactoring capability of large language models," arXiv, 2024. https://doi.org/10.48550/arXiv.2411.02320
4. P. Corona-Fraga et al., "Question–answer methodology for vulnerable source code review via prototype-based model-agnostic meta-learning," Future Internet, vol. 17, no. 1, p. 33, 2025. https://doi.org/10.3390/fi17010033
5. D. de-Fitero-Dominguez, E. Garcia-Lopez, A. Garcia-Cabot, and J.-J. Martinez-Herraiz, "Enhanced automated code vulnerability repair using large language models," Engineering Applications of Artificial Intelligence, vol. 138, part A, p. 109291, 2024. https://doi.org/10.1016/j.engappai.2024.109291
6. A. Fan et al., "Large language models for software engineering: Survey and open problems," arXiv, 2023. https://doi.org/10.48550/arXiv.2310.03533
7. B. Liu et al., "An empirical study on the potential of LLMs in automated software refactoring," arXiv, 2024. https://doi.org/10.48550/arXiv.2411.04444
8. H. Mahfoodh et al., "Evaluating LLMs for source code generation and summarization using machine learning classification and ranking," Computers, vol. 15, no. 2, p. 119, 2026. https://doi.org/10.3390/computers15020119
9. G. G. Maru et al., "Neural methods for programming: A comprehensive survey and future directions," Applied Sciences, vol. 15, no. 22, p. 12150, 2025. https://doi.org/10.3390/app152212150
10. I.Melo,D.Polónia,andL.Teixeira,"Human–AIcollaboration in the modernization of COBOL-based legacy systems: The case of the Department of Government Efficiency (DOGE)," Computers, vol. 14, no. 7, p. 244, 2025. https://doi.org/10.3390/computers14070244
11. A. Midolo and E. Tramontana, "Refactoring loops in the era of LLMs: A comprehensive study," Future Internet, vol. 17, no. 9, p. 418, 2025. https://doi.org/10.3390/fi17090418
12. D. Pomian et al., "Together we go further: LLMs and IDE static analysis for extract method refactoring," arXiv, 2024. https://doi.org/10.48550/arXiv.2401.15298
13. C. Pornprasit and C. Tantithamthavorn, "Fine-tuning and prompt engineering for large language models-based code review automation," Information and Software Technology, vol. 175, p. 107523, 2024. https://doi.org/10.1016/j.infsof.2024.107523
14. A. Rgaguena, I. Chlioui, and M. Radgui, "Generative AI for code translation: A systematic mapping study," Engineering Proceedings, vol. 112, no. 1, p. 33, 2025. https://doi.org/10.3390/engproc2025112033
15. M. S. H. Shaon and M. S. Akter, "Modern approaches to software vulnerability detection: A survey of machine learning, deep learning, and large language models," Electronics, vol. 14, no. 22, p. 4449, 2025. https://doi.org/10.3390/electronics14224449