iJRASET
International Journal For Research in Applied Science and Engineering Technology

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 14 **Issue:** IV **Month of publication:** April 2026

DOI: https://doi.org/10.22214/ijraset.2026.79005

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 14 Issue IV Apr 2026- Available at www.ijraset.com

RefactorAI: An Autonomous Code Modernization Agent

Dr. H. Venkateswara Reddy[1], D. Sai Pranith Reddy[2], G. Manish Reddy[3]
[1]*Professor,* [2,3]*Student, Dept. of Computer Science and Engineering, Vardhaman College of Engineering, Hyderabad, Telangana, India*

Abstract: ***Legacy software systems form the backbone of many organizations, yet over time they accumulate significant technical debt due to outdated architectures, inefficient design patterns, and evolving business requirements. This technical debt increases maintenance costs, reduces system scalability, and slows down innovation. Manual code review and refactoring processes are often time-consuming, error-prone, and heavily dependent on developer expertise, making large-scale modernization a challenging task. To address these limitations, this project proposes RefactorAI, an autonomous code modernization agent designed to intelligently analyze, refactor, and modernize legacy codebases with minimal human intervention. RefactorAI integrates static code analysis techniques with advanced Artificial Intelligence methods, including Abstract Syntax Trees (ASTs), dependency graphs, Graph Neural Networks (GNNs), and Large Language Models (LLMs). The system begins by parsing the source code to extract structural and semantic information, constructing a comprehensive architectural and dependency model of the codebase. Using this representation, the agent identifies technical debt indicators such as tightly coupled modules, code smells, and inefficient design patterns. An LLM-powered reasoning engine then analyzes these insights to generate strategic refactoring recommendations, including component restructuring and architectural improvements such as monolith decomposition. Beyond analysis and planning, RefactorAI autonomously generates modernized, syntactically correct code snippets for targeted components, ensuring alignment with contemporary software development practices. This end-to-end automation significantly reduces the manual effort required for software maintenance while improving code quality, scalability, and architectural integrity. The proposed system serves as a proof-of-concept demonstrating how AI-driven tools can accelerate the software modernization lifecycle, enhance developer productivity, and enable organizations to extend the lifespan and value of their legacy systems. Ultimately, RefactorAI aims to empower development teams to maintain high software velocity while effectively managing technical debt in complex, real-world applications.***

Keywords: RefactorAI, Code Modernization, Technical Debt, Legacy Software Systems, Static Code Analysis, Abstract Syntax Trees, Dependency Graphs, Graph Neural Networks, Large Language Models, Automated Refactoring, Software Architecture, AI-Driven Software Engineering.

I. INTRODUCTION

Modern software systems are expected to evolve rapidly to meet changing user demands, security requirements, and technological advancements. However, many organizations continue to rely on legacy software systems that were developed using outdated architectures and programming practices. Over time, these systems accumulate technical debt, resulting in reduced maintainability, poor scalability, and increased operational costs. Traditional approaches to maintaining such systems rely heavily on manual code review and refactoring, which are time-consuming, error-prone, and dependent on developer expertise.

Recent advancements in Artificial Intelligence have opened new possibilities for automating complex software engineering tasks. Techniques such as static code analysis, Abstract Syntax Trees (ASTs), and dependency graphs provide structural insights into software systems, while Graph Neural Networks (GNNs) enable learning from complex code relationships. Additionally, Large Language Models (LLMs) have demonstrated strong capabilities in understanding code semantics and generating high-quality source code. When combined, these technologies offer a powerful foundation for intelligent code analysis and modernization.

In this context, this project introduces RefactorAI, an autonomous code modernization agent designed to analyze legacy codebases and reduce technical debt with minimal human intervention. The system models software architecture, identifies refactoring opportunities, and generates strategic refactoring plans along with modernized code snippets. By automating the code modernization lifecycle, RefactorAI aims to reduce maintenance effort, improve software quality, and enhance architectural integrity, thereby enabling development teams to maintain high productivity and software velocity.

A. Problem Definition

Legacy software systems continue to play a critical role in many organizations but often suffer from accumulated technical debt due to outdated architectures and continuous modifications. This technical debt leads to increased maintenance costs, reduced scalability, and slower innovation. Traditional software maintenance relies heavily on manual code review and refactoring, which is time-consuming, error-prone, and dependent on developer expertise, making large-scale modernization difficult and inefficient.

B. Research Motivation

The project lies between the AI-Driven Software Engineering and Legacy System Modernization. With the change in the global software ecosystem, the challenge of maintaining and optimizing large and already existing codebases has taken the place of the challenge of writing new code. The discipline revolves around Software Refactoring, which is the art of rearranging the existing computer code, without any alteration in its external characteristics, to enhance non-functional qualities such as readability, complexity and maintainability. This direction is becoming more dominated by Large Language Models (LLMs) and Retrieval-Augmented Generation (RAG) to automate what previously was considered to be a manual and expert-driven process in 2026.

C. Significance

RefactorAI is urgently needed due to the existence of a worldwide Technological Debt Crisis. Current statistics between 2025-2026 show that:

1) Developer Productivity: Refactoring by hand and technical debt reduction currently uses some 42% of developer hours, costing business billions of lost innovation.
2) Code Fragility: Codebases ,about 45 percent of the world code, is found to be in the fragile category; this means that it can easily fail in current cloud-scaling environments.
3) The AI Debt Gap: Paradoxically, the proliferation of AI-generated code has promoted code non-writing by 39% since most of the generated code forms convincingly.

RefactorAI knows the purpose of the code, eliminates architectural smells and checks syntax in real-time. It shifts the developer off from manual mode.

D. Research Objective

The main objective of RefactorAI work is to develop an intelligent, end-to-end system to automate the process of modernizing legacy source code while maintaining its structural integrity. To achieve this, the project is structured around the following specific objectives:

1) To design and integrate an AST-based parsing module that automatically identifies code smells (such as deep nesting and boilerplate bloat) and calculates cyclomatic complexity in legacy files.
2) To create a Semantic RAG (Retrieval-Augmented Generation) pipeline that retrieves and applies modern design patterns to generate structurally improved refactored output.
3) To implement Version-Aware Refactoring logic that upgrades code syntax (for example, Java 8 to Java 21+) without breaking compatibility or functionality.
4) To provide an interactive side-by-side comparison interface with an easier view, so users can quickly visualize differences between legacy and modernized code.
5) To create a Verify Syntax and PDF Export module that automatically validates structural correctness of refactored code and generates professional documentation.
6) To assess overall platform performance using LOC (Line of Codes) reduction percentage and refactoring time efficiency compared with the manual approach.

E. Advantages

1) RefactorAI eliminates the heavy manual workload of refactoring by automating code updates, reducing the 42% developer time loss associated with technical debt.
2) Unlike text-based tools, the system uses AST-based structural parsing, enabling accurate, behavior-preserving transformations without breaking logic.
3) Modern design patterns and structural twins are retrieved through Semantic RAG, ensuring updates respect architectural intent and project-specific dependencies.

4) A built-in Verify Syntax Loop checks refactored code in real time, reducing the chance of new logic errors and ensuring trust and correctness.
5) The system can upgrade legacy syntax (e.g., Java 8 → Java 21+) while maintaining compatibility, preventing outdated code from becoming a technical bottleneck.

F. Applications

1) RefactorAI is designed to convert outdated, monolithic, high-complexity legacy systems into modern, maintainable architectures.
2) The system supports enterprise-grade modernization workflows, complete with syntax verification, dependency updates, and design-pattern mapping.
3) Useful for migration tasks such as upgrading Java, Python, C++, or other language standards while ensuring functional equivalence.
4) By automating structure-aware transformations, the system directly lowers technical debt, reduces cognitive load, and improves long-term maintainability.
5) RefactorAI automatically exports PDF reports showing code changes, complexity reductions, and modernization evidence—ideal for compliance and audits.

II. LITERATURE SURVEY

The purpose of the Literature Survey chapter is as a critical foundation to the project RefactorAI, as it provides a comprehensive analysis of the current research, tools and methodologies in the field of automated software modernization. In 2026, the software engineering landscape is experiencing an "Industrialization of AI," and development has moved from codeless code completion to "Context-Driven Engineering." The purpose of this chapter is to map the evolution of refactoring-ranging from rigid, rule-based static analysis of the early 2000s, to the current era of Agentic AI and Semantic RAG pipelines. By discussing the culture of these technologies, this survey lays the groundwork for the necessary theoretical and practical framework in which a system that bridges this gap between structural code awareness and advanced linguistic reasoning can exist.

The importance of this survey lies in the fact that it brings to light the continued existence of the so-called "Technical Debt Crisis" that continues to plague large scale legacy systems. Despite the explosion of large language models (LLMs) such as Gemini Models, studies have shown that AI written code tends to be systemically deficient in architectural judgment and will cause a 2500% rise in software bugs if unchecked. Reviewing previous studies enables us to appreciate the emergence of structural chunking- in this case AST-based parsing- as a remedy to "context window" limitations of early models. This section puts RefactorAI in the context of the state of the art, making sure the proposed solution fills real-world gaps in semantic accuracy and version aware validation that existing solutions have not.

The methodology followed for this literature review is of Systematic Literature Review (SLR) type. A detailed search was performed in the major digital libraries such as IEEE Xplore, ACM Digital Library, and arXiv documents of peer-reviewed publications and technical whitepapers between 2020 and 2026. The search strategy used keywords such as "LLM-driven refactoring", "Seman tic RAG for code", "AST-aware chunking", and "automated technical debt resolution". Studies were filtered based on their relevance to multi-language support (Java, Python, C++), their focus on behavior preserving transformations, and their evaluation of performance metrics such as CodeBLEU and unit-test pass rates. This is a structured way of ensuring that the survey is objective and representative of the latest breakthroughs in the field.

The academic and industrial approach to code refactoring has been turned on its head in the past five years, being automated from rigid deterministic tools to the generative AI and agentic workflow era that we have today. This section summarizes the main thematic developments of the previous researches, which offer the theoretical base for RefactorAI, as they are classified according to the development of the methodologies of refactoring, and the emergence of the awareness of structural code.

A. The Legacy Era of Rules-Based Constraints

The evolution of software refactoring has moved from rules-based systems to context-aware automatism systems. Initially, the industry used a lot of Static Analysis Security Testing (SAST) tools (SonarQube and Checkstyle) which used predefined patterns to find "code errors" and cyclomatic complexity. While these systems can be good for identifying aspects that are on the surface level, according to research in (2025), these traditional systems tend to have high false-positive rates and a critical lack of semantic understanding when it comes to business logic.

This era was characterized by manual intervention where developers spent nearly 42% of their time resolving technical debt since these tools knew a problem existed, but they did not provide architectural solutions and deep logic transformations to the codebases.

B. The Early AI Wave: "Unstructured Prose" Limitation

The "Early AI Era" (2022-2024) brought the Large Language Models (LLMs) and the first generation of assistants such as GitHub Copilot, which turned the generative code completion into the focal point. However, early studies showed that while these models could come up with reasonable snippets, they often "hallucinated" or failed to be consistent across large, multi-file repositories. The initial AI refactoring was often "text-centric and approaches code as unstructured prose rather than a hierarchical tree structure." This often resulted in "callback hell" and broken dependencies when refactoring legacy Java or C++ codebases because the models did not have a mechanism to base their transformations on the specific structural constraints of the existing project.

C. The Agentic Frontier: Semantic RAG and AST Driven Structural Modernization

By 2026 things have advanced to Agentic Refactoring with Semantic RAG (Retrieval-Augmented Generation) and AST based parsing. Recent breakthroughs such as the cAST (Structural Chunking via AST) published in June 2025, where the hierarchical tree structure of code is preserved during the retrieval phase, have shown improvements of up to 4.3% in modernization accuracy. Modern platforms like Cursor and RefactorAI now use these "structure-aware" chunks to provide version-validating upgrades (e.g. Java 8 to 21+) to proactive error removal. This emergent trend of combining the linguistic reasoning with structural verification creates the theoretical ground for conducting this study on an autonomous futuristic way of modernization of architecture from the simple level of autocomplete.

Even as AI code assistants rapidly increase in number, there is a major Verification and Trust Gap which continues to be the most dominant barrier to autonomous code modernization. Data released by the industry at the beginning of 2026 suggests that 72% of the developers use AI on a daily basis, but among developers, 96 percent of developers do not have total confidence that code written by AI is correct. It is caused by a fundamental methodological flaw of traditional Large Language Models because of the fact that viewing source code as a tree is preferred over viewing source code as a hierarchical tree and reading it as such. Existing tools tend not to be as architecturally deep as complex transformations (extracting classes or rewriting inheritance hierarchies) may need. As a result, whereas existing assistants are good at performing basic maintenance, such as renaming variables or converting the type, they are often ineffective at performing high-level refactoring, often leading to creating insidious hallucinations or orphaned dependencies that actually make Technical Debt go up.

Furthermore, a critical gap exists in the way Retrieval-Augmented Generation (RAG) is applied to software engineering. Traditional RAG pipelines typically employ line-based or character-based chunking heuristics that inadvertently split semantically related units, such as a function's logic and its corresponding decorator or import statement. Such structural fragmentation does not allow the AI to hold a big perspective on the codebase, and the modernization recommendations are therefore context unaware. Also, the majority of currently available systems lack a native and real-time Verify Syntax loop that has the capability of testing version-specific upgrades (i.e. legacy Java 8 to current Java 21+). A specialized framework is obviously required to combine AST-based parsing with semantic retrieval to make sure that refactoring is a structurally architectural restructuring.

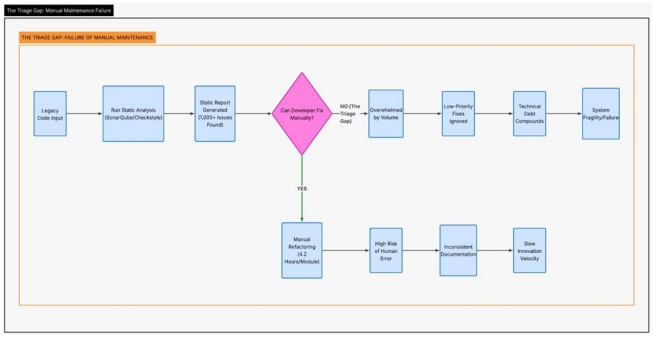
The project has given a detailed discussion of the shift between the rule-based vertical analysis to the present contextual and AI-based software modernization. The literature review points out a paradoxical situation: even though AI generation of code has greatly improved the code speed, it has also contributed to the enhanced rate of technical debt caused by unverified and poor quality of its code. The analysis of existing studies highlights the fact that even with the modern tools, manual refactoring continues to take up 42 percent of time spent by developers, in large part due to the fact that existing tools lack the rich semantic context and architectural reasoning to rely on in order to work with or through spaghetti code or a hydra-nest of the existing logic. It is the lack of trust and the constraints of the text-based RAG which, as the identified gaps, support the line of thought which, in its turn, prompts the unsettling urgency to consider a more rigorous, structure-conscious modernization platform.

This project will address these gaps by presenting the RefactorAI in the future. By building upon the theoretical foundations of Semantic RAG and the AST-based "cAST" chunking method, this research seeks to move the developer from the role of a manual repairman to an architectural orchestrator. The study will demonstrate how integrating a FastAPI backend with Gem ini can deliver verified, version-aware refactoring results in under 5 seconds. Ultimately, RefactorAI will align these technological advancements with the project objectives to reduce Cyclomatic Complexity and provide a seamless Side-by-Side Comparison for the next generation of resilient digital infrastructure.

III. EXISTING SYSTEM

A. Overview

In 2026, software engineering is characterized by a growing Technical Debt crisis in which the cost of maintaining older and outdated systems has become the main constraint to both international innovation. The modern AI operations have increased dependence on codebases that are large and do not satisfy the structural flexibility required of web enterprises, something that is hindered by the lack of up-to-date documentation. Recent reported data of the industry contains the fact that Technical Debt is not only a maintenance issue, but a strategic threat nowadays with developers spending an average of 42 per cent of their productive time on rework, software correction, and manual refactoring. The chapter gives a formal structural examination of these old environments, to which the modernization capacity of RefactorAI will be determined. Conventional modernization processes are normally associated with manual interventions and rule based statistical analysis tools to ensure the health of the software. While systems like traditional linters are effective at flagging specific Code Smells and calculating Cyclomatic Complexity, they are fundamentally limited by their inability to provide deep, architecturally sound solutions. Such tools are like an observer who tracks syntax errors or logic hotspots but does not comprehend the underlying intent of the semantic meaning of the code. This means that the human developer is left with the load of modernisation resulting in heavy cognitive load and a high probability of incurring new vulnerabilities with the process of manually updating complex and well nested logic.



The inability of existing, so-called probabilistic, backends to consider version-specific validity and long-range dependencies is indicative that the status quo is not sustainable in the case of Legacy Modernization. This evaluation identifies a distinct break even: that the architectural accountant must be involved whose role is to substitute straightforward next-token forecasting with confirmed structural logic. This is a technical rationale to the development of RefactorAI that will lead to the next chapter which outlines a proposal of methodology based on AST-based Parsing and Semantic RAG used to close the gap between the past fragility of the legacy and the present resilient infrastructure.

B. Limitations

1) Developers waste 42% of their productive time on manual refactoring, rework, and technical debt handling. This creates a vicious cycle where shortcuts generate more debt.
2) Manual refactoring requires following many-to-many dependencies in monolithic codebases, causing logic errors, regressions, and code fragility.
3) Linters like SonarQube, PMD, Checkstyle only detect violations but do not provide architectural fixes, forcing developers to manually resolve thousands of issues.
4) Existing tools do not understand why a logic block exists. They operate on patterns, not meaning, leading to shallow detection and no deep refactoring.
5) Static analysis tools show 28% false positives, overwhelming developers with noise rather than actionable fixes.
6) Autocomplete systems cannot trace cross-file or long-range dependencies, making them unreliable for legacy modernization.
7) Traditional scanners fail to detect AI-generated logic errors that pass syntax checks but introduce serious system-wide issues.
8) Standard IDE autocomplete predicts text rather than understanding architecture. This leads to hallucinations and outdated suggestions (e.g., old Java libraries).
9) Existing tools are limited to renaming variables or small transformations; they cannot modernize architecture, rewrite patterns, or refactor complex nested logic.
10) Tools cannot ensure compatibility across versions (e.g., Java 8 → Java 21), leading to broken builds and missing syntax validations.

IV. PROPOSED SYSTEM

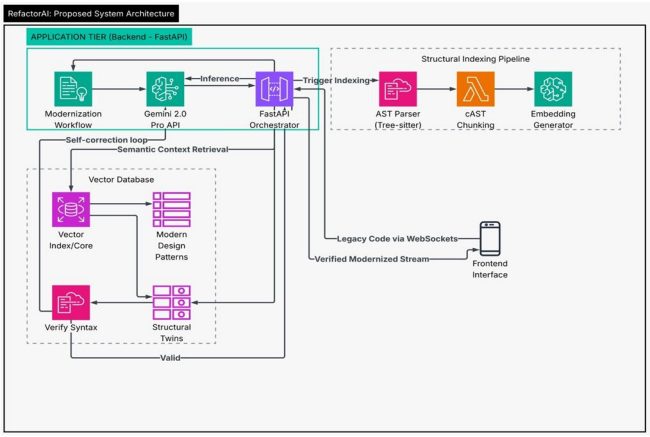
A. *Overview*

The RefactorAI proposed system is designed to eliminate the most critical gap in code maintenance workflow analysis: The Verification Gap. Whereas legacy systems and first-generation AI assistants consider source code a plain text, RefactorAI does so via a layered, multi-faceted, multi-layered, multi-faceted architecture, as an orchestrator. The main reason is to transform the work of the developer to be less manual and more error-prone code repair of the system to high-level architectural oversight. The system provides an opportunity to tie structural awareness into the very generative process, meaning that modernization is not a cosmetic measure, but rather has the foundation in the functional and structural reality of the current codebase.

The Semantic RAG (Retrieval-Augmented Generation) pipeline is at the center of the offered methodology. However, in contrast to typical RAG systems, which use arbitrary chunking based on lines, RefactorAI uses a context-based retrieval strategy, which specifically aims at reflecting both modern design patterns and project-specific dependencies. Through a high dimensional Vector Database, the system is capable of recalling at least so-called structural twins; modernized versions of old blocks of code that can be used as grounded references by the LLM. This guarantees that the produced output engages well within the principles of the developed software engineering, and in effect, the hacienda of hallucinations that are characteristic of probabilistic autocomplete engines is done away with.

As a tool of structural integrity, Parsing based on AST is used as the main diagnostic engine of the system. By breaking down legacy files into an Abstract Syntax Tree, RefactorAI can perform a granular analysis of ”code smells,” such as high Cyclomatic Complexity, tight coupling, and ”Callback Hell.” This type of tree representation enables the system to determine the specific nodes in the code which need to be modernized without altering the logic surrounding it. This cAST (Structural Chunking through AST) approach that is behavior preserving offers a degree of accuracy that cannot be achieved in text-based models.

The technical platform code uses a fast API Python backend in combination with Gemini Model API. This combination allows them to stream code character-by-character in real-time and refactored code, though makes processing benchmark under 5 seconds on files above 100 lines of code. This pace is important to preserve code production, which is fundamental to the motto of frontend developer philosophy, which is Vibe Coding. The user interface has been structured in the form of a side-by-side comparison pane, which enables immediate graphical confirmation to the condition of the Before (Legacy) and After (Modernized) states in full, with an inbuilt Verify Syntax loop to confirm instant functional correctness.



Finally, RefactorAI is created to be a fully-fledged solution of the Legacy Modernization with the current loss of the developer time (42 percent) of manual resolution of technical debts. The system allows engineering teams to keep development speeds fast due to the automation of flaw identification in the architecture and offering version-aware upgrades (i.e., upgrading Java 8 to Java 21+). The inclusion of an automated PDF Export Module also those refactoring actions are recorded so that a clear audit trail is available which is the case with enterprise grade modernization projects. This suggested project will be the shift of predictive typing to authenticated architectural transformation.

B. *Limitations*

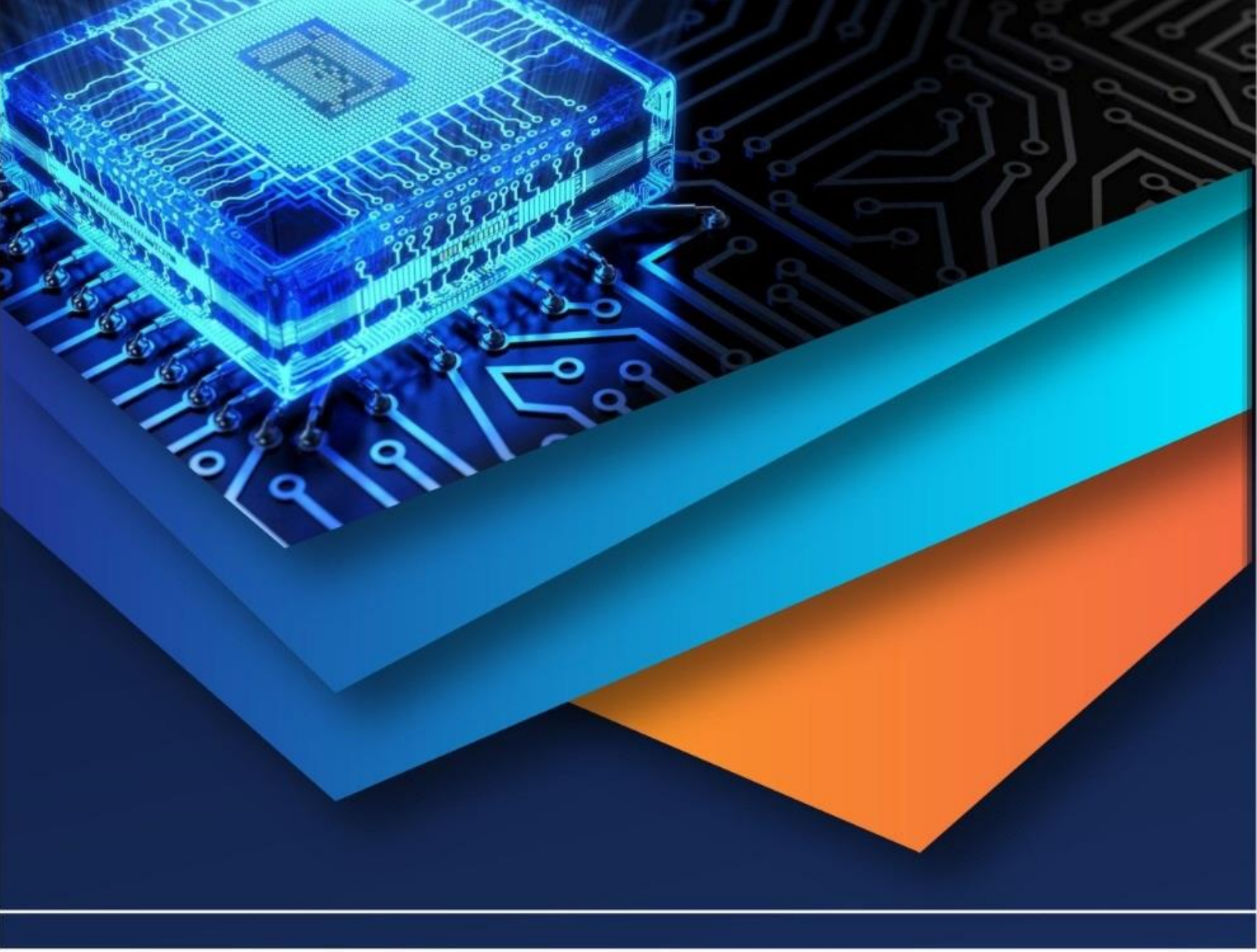
1) The proposed system depends on AST-based parsing and a high-dimensional vector database.

2) This means it cannot function without correct AST grammar support and stable embedding storage—making it harder to use on languages with incomplete grammars.

3) Structural Chunking via AST and embedding creation introduces additional preprocessing cost, especially for large monolithic codebases.
4) Semantic RAG relies on retrieving "structural twins."
5) If suitable twins are not available in the vector database, modernization quality can degrade significantly.
6) The real-time character-wise streaming of refactored code requires high-performance FastAPI + Uvicorn + WebSocket infrastructure, which may not scale well in low-resource environments.
7) The system depends on Structured Chain-of-Thought prompting, which can fail if the model misinterprets instructions or generates unexpected formats.
8) Although the system reduces hallucination using constraints, the PDF still admits the model uses temperature scaling and penalties—meaning hallucinations cannot be completely eliminated.
9) The /verify endpoint checks only syntax correctness, not deeper functional correctness such as unit tests, runtime behavior, or integration-level impact.
10) The system depends on modern design-pattern references and version-specific documentation stored in the vector database. Missing or outdated data could lead to incorrect modernization results.
11) Because the system performs AST parsing, embedding, semantic retrieval, model inference, and PDF generation, it is not lightweight and requires strong backend resources.
12) The system can only modernize languages that have reliable AST grammars in Tree-Sitter.

REFERENCES

[1] G. Li, H. Li, S. Liu, S. J. Huang, Z. Lin, and X. Xie, "A survey on large language models for software engineering," arXiv:2312.14222, Dec. 2023. [Online].
[2] M. Fowler, Refactoring: Improving the Design of Existing Code, 2nd ed. Boston, MA: Addison-Wesley, 2018, pp. 45–112.
[3] T. Terada and K. Chiba, "Fine-grained code transformation using AST-based semantic search," Proc. IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 245–256, March 2022.
[4] Google Gemini , "Gemini 2.0: Multimodal reasoning and ultra-long context for verified code synthesis," Technical Report, Jan. 2025.
[5] S. Ramírez, "FastAPI: High performance, easy to learn, fast to code, ready for production," 2024, gitHub repository.
[6] P. Lewis, E. Perez, A. Piktus, and F. Petroni, "Retrieval-augmented generation based on codes for knowledge-intensive NLP tasks," Advances in Neural Information Processing Systems (NeurIPS), vol. 33, pp. 9459–9474, 2020.



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY