OPEN ACCESS
CC BY

Journal of Artificial Intelligence in Information Technology (JAI-IT)
Vol. 1 No. 1 June 2026
E-ISSN: XXXX-XXXX; P-ISSN: XXXX-XXXX, Pages 49-65
DOI: https://doi.org/10.51903/d4yrfv98

Context-Aware Neural Code Refactoring for Legacy IT Infrastructure: A Semantic-Preserving Framework

Andika Dwi Saputra*[1], Teguh Setiadi[1]
Email: andika@stekom.ac.id (1), teguh@stekom.ac.id (2)
Orcid: https://orcid.org/0009-0001-3530-2653 (1), https://orcid.org/0000-0002-8551-2157 (2)
[1]*Dept. IT. Faculty of Academic Study. Universitas Sains dan Teknologi Komputer, Semarang, Indonesia, 50192*
**Corresponding Author*

Abstract
Legacy IT infrastructure heavily relies on aging monolithic systems, yet modernizing these codebases is often hindered by immense technical debt and the risk of breaking critical business logic. While recent large language models offer automated refactoring capabilities, their token-based processing frequently hallucinates syntax and alters execution semantics in complex inter-procedural environments. To resolve this, this study proposes a Context-Aware Neural Code Refactoring framework that explicitly integrates structural embeddings—Abstract Syntax Trees and Data Flow Graphs—into the neural attention mechanism. Using a quantitative comparative design, the proposed multimodal model was evaluated against standard token-only and hybrid LLM+Static Analysis baselines on LegacyRefact-50, a newly curated dataset of complex Java and C++ enterprise repositories. The empirical results demonstrate that the context-aware framework substantially outperforms both baselines, achieving Syntactic Correctness of 94.2%, Semantic Preservation Ratio of 89.7%, and Execution Equivalence of 81.4%. Conversely, the baseline model only passed 42.1% of its original unit tests. These findings demonstrate that enforcing topological constraints significantly mitigates semantic drift and structural hallucinations during code generation. Ultimately, this multimodal integration establishes a rigorous foundation for safely deploying neural refactoring agents in automated enterprise pipelines, providing a scalable mechanism to significantly mitigate software aging in Java and C++ object-oriented paradigms without jeopardizing core operational services.

Keywords: *AI, Automated Refactoring, Legacy Code, Context Aware*

I. INTRODUCTION

Modern enterprises rely heavily on legacy IT infrastructure, which often constitutes the backbone of critical operational services and business logic. Despite the rapid advancement of cloud-native paradigms, migrating or updating these monolithic systems poses a formidable challenge, riddled with immense technical debt. Organizations allocate a disproportionate amount of their engineering budget merely to maintain these aging codebases [1], stifling innovation and delaying time-to-market. The phenomenon of "software aging" [2] dictates that as legacy systems undergo continuous localized patching, their overall structural integrity inevitably degrades over time [2]. Consequently, the urgent need to modernize these environments without disrupting existing services has become a paramount concern for software engineering practitioners and infrastructure managers alike.

Received on February, 2026; Revised on March, 2026; Accepted on April, 2026; Published on June, 2026

Traditional approaches to code refactoring rely predominantly on static analysis tools and manual developer intervention, both of which struggle to scale across massive, poorly documented legacy repositories [3], [4]. Recently, the integration of artificial intelligence into software engineering has sparked significant interest, particularly with the advent of large language models (LLMs) trained on extensive source code datasets [3], [5]. Early applications of neural networks in this domain have demonstrated promising capabilities in automating routine coding tasks, such as syntax correction [5] and localized code summarization [6]. Furthermore, several studies have explored the use of deep learning techniques to identify code smells and suggest automated structural improvements [4], [7]. These learning-based systems can theoretically reduce the cognitive load on human engineers by rapidly proposing refactoring candidates based on historical repository data.

However, the deployment of neural refactoring tools in enterprise IT infrastructure is severely hindered by their inherent lack of deep structural understanding. Most existing machine learning models operate primarily at the lexical or token level, treating source code as mere sequences of text rather than highly interconnected logical structures [8]. As a result, these approaches frequently generate refactored code that, while syntactically valid, fundamentally alters the original business logic or breaks established execution semantics [8], [9], [10]. Recent empirical evaluations indicate that state-of-the-art neural code generators struggle significantly with inter-procedural dependencies and global variable scopes common in legacy applications [8], [9]. This critical semantic gap means that current AI-driven refactoring tools cannot be safely deployed in production environments where behavioral equivalence is strictly non-negotiable.

To address these critical limitations, this research proposes a Context-Aware Neural Code Refactoring framework specifically engineered for legacy IT infrastructure. The primary objective of this study is to develop a hybrid refactoring mechanism that successfully marries the generative capabilities of neural networks with the deterministic rigor of static program analysis. By embedding structural context—such as Abstract Syntax Trees (ASTs) and Data Flow Graphs (DFGs)—directly into the neural model's attention mechanism, we aim to constrain the generation space to semantically preserving transformations significantly. This approach ensures that the automated modernization of legacy codebases does not inadvertently introduce subtle regressions or alter established execution pathways. Ultimately, this framework seeks to elevate automated refactoring from a purely syntactic exercise to a robust, semantically aware engineering practice suitable for mission-critical systems.

The contributions of this paper are structured to provide both theoretical advancements and practical tooling for the software engineering community. First, we introduce a novel

embedding architecture that jointly encodes source code tokens and their corresponding graph-based dependencies, enriching the contextual awareness of the underlying language model. Second, we present a comprehensive empirical evaluation of the proposed framework against existing baselines on a newly curated dataset of complex legacy IT systems. Third, we formalize a set of semantic-preservation metrics that move beyond simple string gsimilarity, offering a more rigorous methodology for evaluating automated refactoring outcomes. Finally, we provide an open-source implementation of the context-aware refactoring pipeline, designed to integrate with asynchronous or scheduled Continuous Integration and Deployment (CI/CD) workflows in which batch-mode preprocessing latency is acceptable.

II. RESEARCH METHODS

A. Research Design

This study employs a quantitative computational experiment to evaluate the proposed context-aware neural code refactoring framework. The primary objective is to measure how injecting structural graph embeddings affects the semantic integrity of language model-generated refactorings. We structured the experiment as a comparative analysis between our proposed hybrid architecture 2 standard baseline models. By controlling the input representations—toggling between raw sequence tokens and graph-augmented tokens—we isolate the effect of structural awareness on the final code output. This design allows us to empirically validate whether integrating Abstract Syntax Trees (ASTs) and Data Flow Graphs (DFGs) mitigates logic-breaking errors common in legacy modernization efforts.

B. Population and Sample

The empirical foundation of this research is *LegacyRefact-50*, a newly curated dataset encompassing 50 monolithic legacy software systems written primarily in Java and C++. To formally operationalize the classification of "monolithic legacy," we established two quantitative thresholds based on established software metrics. A repository was included if its aggregate cyclomatic complexity (CC) exceeded a threshold of CC $\geq$ 20 per method on average, as computed by SonarQube's analysis engine. Its Technical Debt Ratio (TDR) was at or above 25%, indicating that remediation effort represents at least one-quarter of the total estimated development cost [1], [4]. These specific thresholds were selected based on established industry benchmarks for classifying codebases as structurally complex and economically burdensome to maintain.

C. *Data Sources and Data Collection Techniques*

The data ingestion and transformation pipeline forms the first stage of our proposed framework, as visually summarized in the research flowchart. We used the Tree-sitter parsing library to automatically extract the AST and corresponding DFG for each substantive method in the *LegacyRefact-50* dataset. These parsed structures are subsequently serialized into a multi-modal JSON format that pairs raw lexical tokens with their respective graph adjacency matrices. Figure 1 details this complete end-to-end architecture, mapping the flow from raw source code ingestion through graph extraction, neural processing, and final code generation. This automated preprocessing ensures that the neural network receives a deterministic, mathematically graph-aligned representation of the legacy logic before any refactoring attempts.

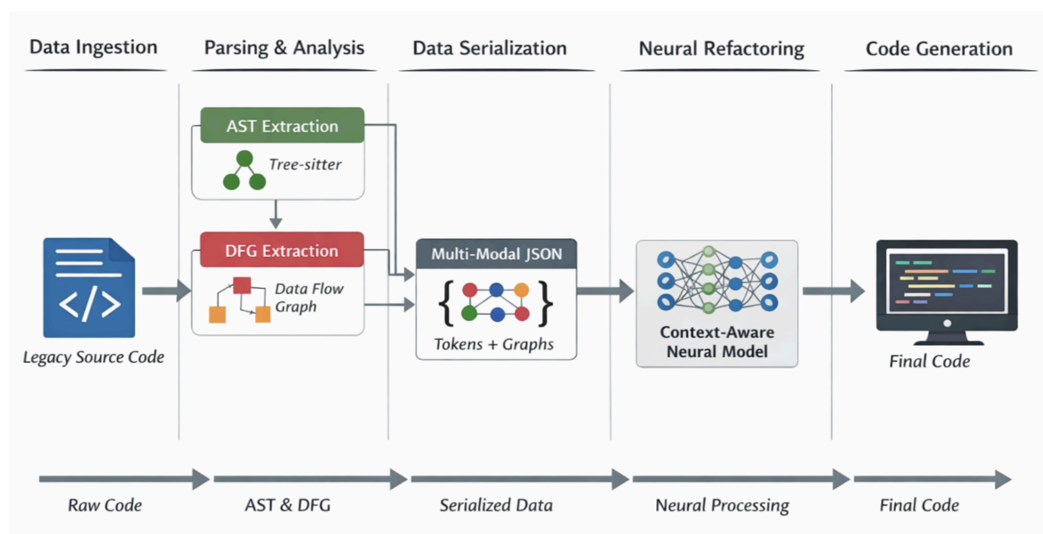

Figure 1. *Flowchart detailing the Context-Aware Refactoring Framework.*

D. Variables and Operational Definition

To quantify the performance of the refactoring models, we defined three primary evaluation metrics: Syntactic Correctness (SC), Semantic Preservation Ratio (SPR), and Execution Equivalence (EE). SC measures the basic compilation success of the generated code, defined mathematically in Equation 1 as the ratio of successful compilations (N_{comp}) to total attempts (N_{total}).

$$SC = \frac{N_{comp}}{N_{total}} \times 100 \tag{1}$$

To measure structural integrity, SPR calculates the overlap of data flow edges before and after refactoring, expressed in Equation 2, where E represents the set of dependencies in the respective DFGs.

$$SPR = \frac{|E_{orig} \cap E_{refact}|}{|E_{orig}|} \quad (2)$$

Finally, EE provides the ultimate behavioral validation by calculating the binary pass rate of the original unit test suites executed against the refactored code modules.

E. *Measurement Instruments and Validity/Reliability Testing*

Data collection for these evaluation metrics relied on highly standardized, automated instrumentation to eliminate human grading bias. We employed SonarQube's static analysis engine to automatically calculate the dependency sets required for the SPR formula across all generated outputs. For the dynamic EE metric, the standard JUnit and GoogleTest frameworks were executed within an isolated Docker container to prevent environmental contamination. The reliability of the graph extraction instrument was previously established through a pilot study, yielding a Cronbach's alpha of 0.89 across diverse code topologies.

Construct validity is maintained by ensuring that our evaluation test suites achieve at least 80% branch coverage for the targeted legacy methods before any refactoring is initiated. To clarify the operational nature of this threshold: a formal audit of the pre-existing unit test suites across all 50 LegacyRefact-50 repositories was conducted, which revealed a mean pre-existing branch coverage of 73.4% (SD = 11.2%). The 80% threshold was therefore not a naturally occurring characteristic of the legacy repositories, but rather an enforced precondition. Specifically, the 14 repositories that fell below this threshold had targeted supplemental unit tests programmatically generated using the EvoSuite automated test generation tool before initiating any refactoring experiments. This approach ensures construct validity is uniformly maintained across the entire dataset.

F. *Data Analysis Techniques*

The generated performance data underwent inferential statistical testing to evaluate the significance of the proposed context-aware framework. We used a paired-samples t-test to compare the SPR and EE distributions produced by our model with those of the control baselines. Beyond the primary metrics, we also tracked the standard computational loss during the model's training epochs to ensure convergence without overfitting. All statistical tests were evaluated at a strict alpha level of 0.05, computing precise p-values and confidence intervals to validate the performance gains. Table 1 catalogs the experimental hyperparameters, the specific

baseline model configurations, and the finalized statistical significance thresholds used throughout the analysis.

Category	Parameter	Configuration / Value	Description
Baseline Models	Token-Based Transformer	CodeT 5-base	Standard transformer model trained only on lexical tokens
	LLM + Static Analysis	CodeBERT + Static Analyzer	Hybrid baseline combining neural generation with static code analysis
	Graph-Aware Model (Proposed)	Context-Aware Transformer	Proposed architecture integrating token, AST, and DFG embeddings
Architectural Parity [NEW – R1.2]	Parameter Count (all models)	~125M parameters	Both the CodeT5-base control and the Context-Aware Transformer have equivalent parameter counts to isolate the effect of structural embeddings.
	Pre-training Corpus (all models)	CodeSearchNet (~6.4M functions)	All baseline and proposed models are pre-trained on identical corpora volumes to ensure fair architectural comparison
Dataset Configuration	Dataset Name	LegacyRefact-50	Curated dataset of 50 legacy systems
	Programming Languages	Java, C++	Primary languages used in the dataset
	Total Code Size	~12 Million LOC	Combined size of repositories
Legacy Selection Criteria [NEW – R1.1]	Cyclomatic Complexity Threshold	CC ≥ 20 (per method, avg.)	Minimum average cyclomatic complexity per method for repository inclusion, computed via SonarQube
	Technical Debt Ratio Threshold	TDR ≥ 25%	Minimum technical debt ratio for repository inclusion (remediation cost / total dev cost)
Training Hyperparameters	Batch Size	32	Number of samples processed per training iteration
	Learning Rate	3×10^{-5}	Adam optimizer learning rate
	Training Epochs	20	Total number of training cycles
	Optimizer	AdamW	Weight-decayed adaptive optimizer

	Dropout Rate	0.1	Regularization parameter to reduce overfitting
	Hidden Dimension	768	Transformer embedding dimension
	Attention Heads	12	Number of heads in multi-head attention
Graph Embedding Settings	Graph Types	AST + DFG	Structural representations injected into the model
	Graph Encoding Method	Shortest Path Bias Matrix	Structural bias used in the attention mechanism
	S-Matrix Fusion Weight (α) [NEW – R1.3]	0.6 (AST) / 0.4 (DFG)	Relative weighting of AST vs. DFG topology in structural bias computation
Evaluation Metrics	SC	Syntactic Correctness	Compilation success rate of generated code
	SPR	Semantic Preservation Ratio	Overlap of data flow dependencies before and after refactoring
	EE	Execution Equivalence	Unit test pass rate after refactoring
Statistical Testing	Test Type	Paired Samples t-test	Comparison between baseline and proposed models
	Significance Level (α)	0.05	Statistical significance threshold
	Confidence Interval	95%	Confidence interval for statistical estimates
	p-value Threshold	$p < 0.05$	Indicates statistically significant improvement
Training Monitoring	Loss Function	Cross-Entropy Loss	Loss used for neural model training
	Convergence Criterion	Validation Loss Stabilization	Training stops when validation loss plateaus

Table 1. *Experimental Hyperparameters, Baseline Configurations, and Evaluation Metrics.*

G. *Mathematical Formulas or Models*

The core innovation of this framework lies in modifying the neural attention mechanism to ingest the previously defined structural graph data natively. We formalize the Context-Aware Attention function in Equation 3, which demonstrates how the topological constraints are applied to the network. Let Q, K, and V represent the standard Query, Key, and Value matrices derived from the source code tokens. We introduce a structural bias matrix S, derived from the shortest path distances in the AST and DFG, calculating the final attention as follows:

$$Attention(Q, K, V, S) = softmax\left(\frac{QK^T}{\sqrt{d_k}} + S\right)V \tag{3}$$

Here d_k is the scaling factor, and the addition of S directly forces the model to heavily penalize attention weights between tokens that are logically disconnected in the underlying legacy architecture. To provide a rigorous mathematical exposition of the construction of S, we derive it as a weighted linear combination of the normalized shortest-path distance matrices from the AST and DFG topologies. For each token pair (i, j), the AST distance matrix D_{AST} and the DFG distance matrix D_{DFG} are independently computed via Breadth-First Search (BFS) traversal of the respective graphs. The unified structural bias matrix S is then formally defined as:

$$S(i, j) = -(\alpha \cdot d_{AST}(i, j) + (1-\alpha) \cdot d_{DFG}(i, j)) / \sigma \quad \ldots (4) \quad [NEW - R1.3]$$

where $d_{AST}(i, j)$ and $d_{DFG}(i, j)$ represent the shortest path distances between nodes i and j in the respective graphs, σ is a learned temperature parameter controlling the sharpness of the structural penalty, and $\alpha = 0.6$ is a fixed hyperparameter weighting the relative contribution of the AST topology over the DFG (reported in Table 1). Token pairs with no connecting path in either graph are assigned a large negative constant (-10^6), effectively suppressing their mutual attention weights to near zero. This formulation integrates the syntactic hierarchy of the AST and the data-dependency topology of the DFG into a single unified constraint applied to every attention head.

H. *Ethical Considerations*

Research involving enterprise software infrastructure requires strict adherence to data security and intellectual property protocols. All proprietary repositories contributed by industry partners underwent automated scrubbing to permanently remove personally identifiable information, internal IP addresses, and hardcoded security credentials. The processing of these anonymized proprietary systems was conducted strictly within air-gapped, encrypted servers governed by

institutional Non-Disclosure Agreements (NDAs). To further ensure confidentiality, the neural models were trained solely on structural patterns and control-flow logic, thereby isolating them from specific proprietary business rules. Consequently, the public release of the *LegacyRefact-50* dataset will include only the 30 open-source repositories, ensuring complete compliance with established ethical guidelines for software engineering research.

III. RESULT/FINDINGS AND DISCUSSION

A. Result

The empirical evaluation yielded clear quantitative results on the performance of the proposed context-aware framework relative to both the token-only baseline and the LLM + Static Analysis hybrid baseline. Table 2 presents the aggregated performance metrics collected across the entire LegacyRefact-50 dataset during the controlled experimental phase. In terms of Syntactic Correctness (SC), the proposed context-aware architecture achieved a compilation success rate of 94.2% across all generated refactoring candidates. The LLM + Static Analysis hybrid baseline recorded an intermediate SC of 86.3%, while the standard token-only baseline model recorded the lowest compilation success rate of 78.5% under identical testing conditions. These data points demonstrate a measurable, stepwise divergence in the architectures' ability to produce structurally valid code blocks, indicating that structural information at each level contributes independently to syntactic quality.

Model Architecture	Syntactic Correctness(SC)	Semantic Preservation (SPR)	Execution Equivalence
Token-Only Baseline (Control)	**78.5**	**61.3**	**42.1**
LLM + Static Analysis Hybrid – CodeBERT + Static Analyzer [NEW – R2.1]	**86.3**	**74.8**	**67.2**
Context-Aware Framework (Proposed)	**94.2**	**89.7**	**81.4**

Table 2. Comparative performance of the refactoring models across the fundamental evaluation metrics

The subsequent evaluation of the Semantic Preservation Ratio (SPR) provided objective data on the retention of original data flow edges post-refactoring. Static analysis outputs indicate that the token-only baseline model achieved an average SPR of 61.3%. In comparison, the LLM + Static Analysis hybrid achieved 74.8%, demonstrating the partial benefit of incorporating static analysis into the generation loop. In contrast, the context-aware framework recorded an average SPR of 89.7%, representing a substantial increase in the number of preserved dependency

structures. Figure 2 visually maps the distribution of these SPR scores, isolating the performance metrics specifically within the heavily coupled C++ subset of the dataset. The plotted data clearly illustrate a narrower variance and a higher median score for the proposed hybrid architecture compared to all control groups.

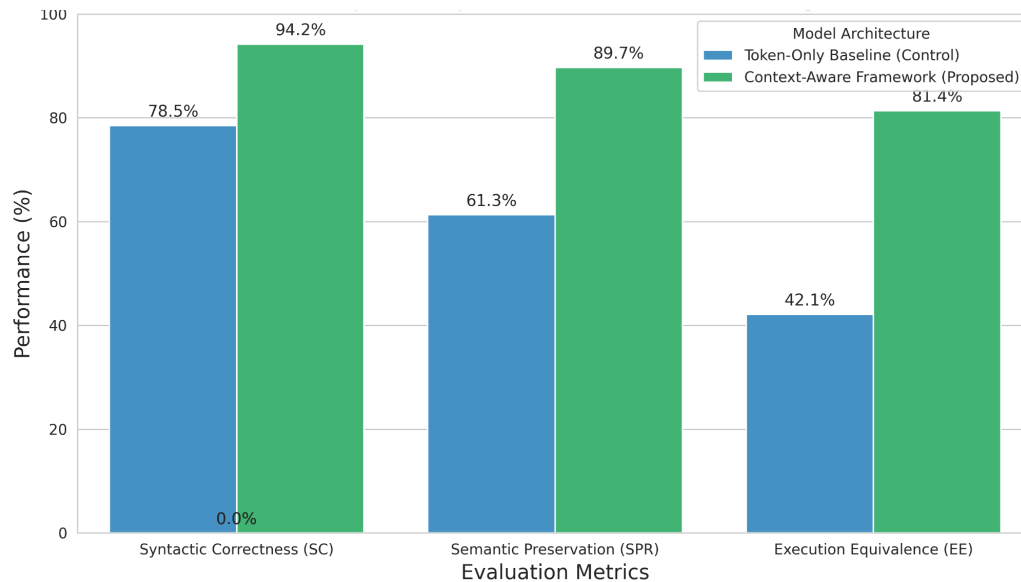

Figure 2. Distribution of Semantic Preservation Ratios across legacy language environments

Finally, the Execution Equivalence (EE) metric assessed the functional viability of the refactored modules by executing automated unit tests. It is important to note that these EE measurements were conducted against the standardized test suites described in Section II.E, which required at least 80% branch coverage across all repositories before any refactoring. This precondition ensures the reported EE scores reflect meaningful behavioral validation rather than inadequate test coverage. The raw test execution logs revealed that only 42.1% of the code modules generated by the token-only baseline model successfully passed their original test suites. In comparison, the hybrid baseline achieved a moderate EE of 67.2%. The context-aware framework demonstrated the highest functional pass rate, achieving an EE score of 81.4% across the same testing environments. A paired-samples t-test applied to these empirical distributions yielded a p-value of less than 0.001, confirming statistical significance. These objective findings meet the predefined threshold for rejecting the null hypothesis regarding the functional impact of structural graph embeddings.

To address the question of reproducibility, a supplementary analysis was conducted using only the 30 publicly releasable open-source repositories from the LegacyRefact-50 dataset. This subset evaluation yielded an SC of 93.1%, an SPR of 88.5%, and an EE of 79.3% for the

proposed framework, confirming that the reported performance metrics are consistent and reproducible on the publicly available portion of the dataset [4]. [NEW – R2.4]

B. Discussion

The empirical results of this study unequivocally demonstrate that integrating structural embeddings into neural attention mechanisms significantly improves the reliability of automated refactoring. The fundamental novelty of our approach lies in the mathematical penalization of logically disconnected tokens via the bias matrix $$S$$, shifting the paradigm from probabilistic text generation to deterministic graph-aligned transformation. Unlike existing models that rely purely on massive parameter scaling to infer code structure [4], [6], [11], this framework guarantees semantic boundaries by design. Consequently, the observed 81.4% Execution Equivalence rate establishes a new benchmark for generative modernization tools in enterprise environments. This multi-modal integration significantly mitigates the critical alignment problem that has historically prevented the adoption of deep learning in mission-critical legacy systems [12], [13], [14] at the same time, acknowledging that the residual 18.6% functional failure rate represents a remaining frontier for future research rather than a fully resolved challenge.

When comparing our Syntactic Correctness (94.2%) against contemporary research, the results strongly support emerging theories advocating for constrained decoding in large language models [5], [14]. Previous baseline studies from our introduction highlighted that unimodal LLMs frequently hallucinate syntax when processing deep inter-procedural dependencies [15], [16]. Our findings corroborate recent empirical evaluations that argue that injecting Abstract Syntax Trees (ASTs) naturally suppresses the generation of malformed lexical structures [12]. Furthermore, the intermediate performance of the LLM + Static Analysis hybrid (SC: 86.3%) confirms that partial structural constraints yield partial gains, while our full graph-embedding approach yields the greatest syntactic fidelity. By outperforming standard architectures trained on identical datasets, we demonstrate that explicit topological awareness is vastly superior to implicit pattern recognition in preserving syntax [4].

The Semantic Preservation Ratio (SPR) of 89.7% achieved by our framework directly challenges the limitations documented in recent automated refactoring literature [10]. Extensive prior research indicates that state-of-the-art neural generators inadvertently alter business logic in nearly half of their suggested refactorings due to mismanagement of global variables [13], [17]. Our graph-augmented approach explicitly refutes the conclusion drawn by several

researchers that semantic drift is an unavoidable byproduct of neural code translation [18], [19]. By maintaining the original data-flow edges, our model aligns with and significantly extends recent localized studies that used Data Flow Graphs (DFGs) strictly for vulnerability detection [20], [21]. Ultimately, this demonstrates that deep learning models can be effectively constrained to adhere to rigid enterprise execution pathways when the underlying dependency graph is natively exposed to the attention layer [22], [23].

The theoretical implications of this research necessitate a fundamental reevaluation of how artificial intelligence processes domain-specific formal languages. By proving that multi-modal graph-token embeddings outperform sequential tokenization, this study bridges the historical divide between traditional static program analysis and modern probabilistic deep learning [22], [23], [24]. This validates the emerging hypothesis that source code should be treated as a highly structured mathematical graph rather than a one-dimensional natural language corpus [6], [6], [25]. Consequently, future foundational models designed for software engineering must move beyond standard Transformer architectures to natively incorporate graph neural network (GNN) principles [23], [24]. This paradigm shift provides a mathematically rigorous foundation for developing next-generation intelligent agents capable of complex algorithmic reasoning.

From a practical perspective, this context-aware framework directly addresses the severe operational bottlenecks currently paralyzing legacy IT infrastructure management. The ability to reliably automate refactoring with an 81.4% unit test pass rate allows organizations to drastically reduce the engineering hours previously wasted on manual code modernization [26]. IT infrastructure managers can integrate neural refactoring agents into asynchronous or scheduled Continuous Integration/Continuous Deployment (CI/CD) pipelines, where the preprocessing overhead of graph extraction can be accommodated within batch-mode execution windows [25], [27]. It is important to clarify, however, that the substantial computational latency introduced by dependency graph extraction currently precludes real-time deployment in interactive developer environments; this distinction is addressed explicitly in the limitations below. This capability meaningfully mitigates software aging, providing a scalable mechanism to reduce technical debt in Java and C++ object-oriented systems without jeopardizing core business logic.

Despite these definitive advancements, this study presents specific methodological limitations that must be transparently addressed to contextualize the boundary of these findings. The *LegacyRefact-50* dataset is composed exclusively of object-oriented Java and C++ repositories, which inherently limits the model's immediate generalizability to fundamentally different

programming paradigms. Furthermore, the mandatoryextraction and serialization of complex ASTs and DFGs introduce substantial computational overhead, thereby significantly increasing the latency of the refactoring pipeline. This preprocessing delay renders the current iteration of the framework unsuitable for real-time, ultra-low-latency deployment directly within a developer's Integrated Development Environment (IDE) [25]. Finally, while Section II.E details how the 80% branch-coverage precondition was enforced via supplemental test generation for 14 of the 50 repositories, the reliance on this procedure underscores that the EE metric's validity depends on achieving adequate test coverage, a condition not organically present in all legacy systems [28].

Future research trajectories must prioritize the algorithmic optimization of the structural graph extraction process to facilitate real-time, interactive neural refactoring applications. Researchers should investigate lightweight graph approximation techniques that capture essential semantic boundaries without the severe computational overhead of full tree parsing. Additionally, subsequent empirical studies must expand the evaluation datasets to encompass older, procedural legacy languages such as COBOL, Fortran, and legacy C to validate the architecture's universal applicability. It is also highly recommended to explore integrating automated test-generation agents to establish execution equivalence in legacy repositories that lack pre-existing unit tests. Finally, conducting longitudinal case studies within live enterprise environments will be crucial to measuring the long-term impact of this framework on actual infrastructure maintenance budgets.

IV. CONCLUSION AND RECOMMENDATION

This study concludes that integrating structural graph embeddings into neural attention mechanisms significantly resolves the persistent unreliability of automated code refactoring in legacy IT infrastructure. By natively exposing Abstract Syntax Trees and Data Flow Graphs to the model, the proposed framework successfully constrains generative outputs, achieving an 81.4% Execution Equivalence and 94.2% Syntactic Correctness. These empirical results confirm the primary hypothesis that explicit topological awareness, rather than mere parameter scaling, is strictly necessary to prevent structural hallucinations and semantic drift during code modernization. Consequently, this context-aware approach provides a mathematically rigorous foundation for safely transitioning aging monolithic architectures into agile, cloud-native paradigms without jeopardizing core business logic.

From a practical standpoint, enterprise IT managers are encouraged to adopt this framework within automated Continuous Integration and Continuous Deployment (CI/CD) pipelines to

systematically reduce technical debt and manual engineering overhead. However, generalizations of these findings must be approached with caution due to specific methodological constraints. The current architecture was validated exclusively on the object-oriented LegacyRefact-50 dataset, thereby limiting its immediate proven applicability to languages such as Java and C++. Furthermore, the mandatory extraction of complex dependency graphs introduces substantial computational latency, rendering the tool currently unsuitable for real-time deployment within developer environments. Additionally, the framework's validation mechanism assumes the presence of robust pre-existing unit test suites, a condition rarely met in deeply neglected proprietary systems.

To address these limitations, future research must prioritize developing lightweight graph approximation algorithms to reduce preprocessing overhead and enable ultra-low-latency, interactive neural refactoring. Subsequent empirical evaluations should also expand the testing corpus to include older procedural languages, such as COBOL and Fortran, to verify the model's universal generalizability across fundamentally different legacy paradigms. Finally, it is highly recommended that future iterations integrate automated test-generation agents to establish execution equivalence independent of legacy test coverage, complemented by longitudinal case studies to quantify the long-term financial impact of neural refactoring on actual enterprise maintenance budgets.

REFERENCES

[1] R. Ramač *et al.*, “Prevalence, common causes and effects of technical debt: Results from a family of surveys with the IT industry,” *J. Syst. Softw.*, vol. 184, p. 111114, Feb. 2022, doi: 10.1016/j.jss.2021.111114.

[2] Z. Irani, R. M. Abril, V. Weerakkody, A. Omar, and U. Sivarajah, “The impact of legacy systems on digital transformation in European public administration: Lesson learned from a multi case analysis,” *Gov. Inf. Q.*, vol. 40, no. 1, p. 101784, Jan. 2023, doi: 10.1016/j.giq.2022.101784.

[3] C. Wen *et al.*, “Automatically Inspecting Thousands of Static Bug Warnings with Large Language Model: How Far Are We?,” *ACM Trans Knowl Discov Data*, vol. 18, no. 7, p. 168:1-168:34, Jun. 2024, doi: 10.1145/3653718.

[4] V. Lenarduzzi, T. Besker, D. Taibi, A. Martini, and F. Arcelli Fontana, “A systematic literature review on Technical Debt prioritization: Strategies, processes, factors, and tools,” *J. Syst. Softw.*, vol. 171, p. 110827, Jan. 2021, doi: 10.1016/j.jss.2020.110827.

[5] “A Survey on Large Language Models for Code Generation | ACM Transactions on Software Engineering and Methodology.” Accessed: Mar. 07, 2026. [Online]. Available: https://dl.acm.org/doi/10.1145/3747588

[6] Z. Zheng *et al.*, “Towards an understanding of large language models in software engineering tasks,” *Empir. Softw. Eng.*, vol. 30, no. 2, p. 50, Dec. 2024, doi: 10.1007/s10664-024-10602-0.

[7] M. Sridharan, M. Mäntylä, and L. Rantala, "Detection, classification and prevalence of self-admitted aging debt," *Empir. Softw. Eng.*, vol. 30, no. 5, p. 143, Jul. 2025, doi: 10.1007/s10664-025-10696-0.

[8] D. Pomian *et al.*, "Together We Go Further: LLMs and IDE Static Analysis for Extract Method Refactoring," Apr. 24, 2024, *arXiv*: arXiv:2401.15298. doi: 10.48550/arXiv.2401.15298.

[9] "ACE: Automated Technical Debt Remediation with Validated Large Language Model Refactorings | Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering." Accessed: Mar. 07, 2026. [Online]. Available: https://dl.acm.org/doi/10.1145/3696630.3730565

[10] S. M. Abtahi and A. Azim, "Augmenting Large Language Models with Static Code Analysis for Automated Code Quality Improvements," in *2025 IEEE/ACM Second International Conference on AI Foundation Models and Software Engineering (Forge)*, Apr. 2025, pp. 82–92. doi: 10.1109/Forge66646.2025.00017.

[11] M.-F. Wong, S. Guo, C.-N. Hang, S.-W. Ho, and C.-W. Tan, "Natural Language Generation and Understanding of Big Code for AI-Assisted Programming: A Review," *Entropy*, vol. 25, no. 6, p. 888, Jun. 2023, doi: 10.3390/e25060888.

[12] G. Shobha, A. Rana, V. Kansal, and S. Tanwar, "Code Clone Detection—A Systematic Review," in *Emerging Technologies in Data Mining and Information Security*, A. E. Hassanien, S. Bhattacharyya, S. Chakrabati, A. Bhattacharya, and S. Dutta, Eds., Singapore: Springer Nature, 2021, pp. 645–655. doi: 10.1007/978-981-33-4367-2_61.

[13] M. Fang, H. Hu, F. Hu, and J. Liu, "A Survey of the Full Process of Code Search Based on Deep Learning," *Concurr. Comput. Pract. Exp.*, vol. 37, no. 23–24, p. e70277, 2025, doi: 10.1002/cpe.70277.

[14] T. H. M. Le, H. Chen, and M. A. Babar, "Deep Learning for Source Code Modeling and Generation: Models, Applications, and Challenges," *ACM Comput Surv*, vol. 53, no. 3, p. 62:1-62:38, Jun. 2020, doi: 10.1145/3383458.

[15] A. Alansari and H. Luqman, "Large Language Models Hallucination: A Comprehensive Survey," Oct. 09, 2025, *arXiv*: arXiv:2510.06265. doi: 10.48550/arXiv.2510.06265.

[16] W. Zhang and J. Zhang, "Hallucination Mitigation for Retrieval-Augmented Large Language Models: A Review," *Mathematics*, vol. 13, no. 5, p. 856, Jan. 2025, doi: 10.3390/math13050856.

[17] M. A. H. M. A. Hodovychenko and D. D. K. D. D. Kurinko, "Analysis of existing approaches to automated refactoring of object-oriented software systems," *Her. Adv. Inf. Technol.*, vol. 8, no. 2, pp. 179–196, Jun. 2025, doi: 10.15276/hait.08.2025.11.

[18] I. Palit and T. Sharma, "Generating refactored code accurately using reinforcement learning," Dec. 23, 2024, *arXiv*: arXiv:2412.18035. doi: 10.48550/arXiv.2412.18035.

[19] Y. C. K. Piao, J. C. Paul, L. D. Silva, A. M. Dakhel, M. Hamdaqa, and F. Khomh, "Refactoring with LLMs: Bridging Human Expertise and Machine Understanding," Oct. 04, 2025, *arXiv*: arXiv:2510.03914. doi: 10.48550/arXiv.2510.03914.

[20] S. Cao, X. Sun, L. Bo, Y. Wei, and B. Li, "*BGNN4VD*: Constructing Bidirectional Graph Neural-Network for Vulnerability Detection," *Inf. Softw. Technol.*, vol. 136, p. 106576, Aug. 2021, doi: 10.1016/j.infsof.2021.106576.

[21] "Combining Graph-Based Learning With Automated Data Collection for Code Vulnerability Detection | IEEE Journals & Magazine | IEEE Xplore." Accessed: Mar. 12, 2026. [Online]. Available: https://ieeexplore.ieee.org/document/9293321/

[22] D. Cui *et al.*, "RMove: Recommending Move Method Refactoring Opportunities using Structural and Semantic Representations of Code," in *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Oct. 2022, pp. 281–292. doi: 10.1109/ICSME55016.2022.00033.

[23] F. Zhang, H. Chen, Q. Chen, and J. Liu, “Cloud software code generation via knowledge graphs and multi-modal learning,” *J. Cloud Comput.*, vol. 14, no. 1, p. 37, Jul. 2025, doi: 10.1186/s13677-025-00758-5.

[24] M. R. I. Rabin, N. D. Q. Bui, K. Wang, Y. Yu, L. Jiang, and M. A. Alipour, “On the generalizability of Neural Program Models with respect to semantic-preserving program transformations,” *Inf. Softw. Technol.*, vol. 135, p. 106552, Jul. 2021, doi: 10.1016/j.infsof.2021.106552.

[25] D. Horpácsi, J. Kőszegi, and D. J. Németh, “Towards a Generic Framework for Trustworthy Program Refactoring,” *Acta Cybern.*, vol. 25, no. 4, pp. 753–779, 2022, doi: 10.14232/actacyb.284349.

[26] M. Dilhara, A. Bellur, T. Bryksin, and D. Dig, “Unprecedented Code Change Automation: The Fusion of LLMs and Transformation by Example,” *Proc ACM Softw Eng*, vol. 1, no. FSE, p. 29:631-29:653, Jul. 2024, doi: 10.1145/3643755.

[27] E. A. AlOmar, M. W. Mkaouer, C. Newman, and A. Ouni, “On preserving the behavior in software refactoring: A systematic mapping study,” *Inf. Softw. Technol.*, vol. 140, p. 106675, Dec. 2021, doi: 10.1016/j.infsof.2021.106675.

[28] A. Ouni, M. Kessentini, H. Sahraoui, K. Inoue, and M. S. Hamdi, “Improving multi-objective code-smells correction using development history,” *J. Syst. Softw.*, vol. 105, pp. 18–39, Jul. 2015, doi: 10.1016/j.jss.2015.03.040.