National College of Ireland

A Transformer-based Framework to Automatically Refactor Legacy Code

MSc Research Project
Artificial Intelligence

Shoban Ravichandran
Student ID: 23272040

School of Computing
National College of Ireland

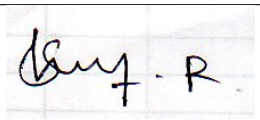
Supervisor: Paul Stynes

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Shoban Ravichandran
Student ID:	23272040
Programme:	Artificial Intelligence
Year:	2025
Module:	MSc Research Project
Supervisor:	Paul Stynes
Submission Due Date:	20/12/2025
Project Title:	A Transformer-based Framework to Automatically Refactor Legacy Code
Word Count:	6049
Page Count:	22

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	
Date:	15th September 2025

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	☐
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☐
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	☐

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

A Transformer-based Framework to Automatically Refactor Legacy Code

Shoban Ravichandran
23272040

Abstract

Legacy code refactoring involves systematically restructuring outdated code bases to improve maintainability, readability, and extensibility while preserving existing functionality. Refactoring legacy code is a critical challenge in software engineering due to the accumulation of technical debt and the future cost incurred to repair the code. This research proposes a **Transformer-based framework** that combines Retrieval Augmented Generation with transformer-based language models and NSGA-II optimization for automated Python code refactoring. This research proposes a transformer-based framework combining Retrieval Augmented Generation with multi-objective optimization for automated Python code refactoring. The framework integrates multiple state-of-the-art language models through NSGA-II optimization to balance competing quality objectives. **Results** demonstrate significant improvements in refactoring quality and system efficiency. NSGA-II optimization successfully identified Llama3-70B-8192 as the optimal model configuration, achieving **significant improvements across multiple metrics:** Answer Relevance (+2.00%), Response Completeness (+7.61%), and CodeBLEU Score (+69.23%). The optimization also improved system efficiency with reduced latency by 10.4% and decreased token usage by 24.9%. The framework achieved an **overall 13.78% improvement** in core quality metrics while maintaining 100% success rate across all test cases. This research aligns with the goals of Sustainable Development Goal 9 (SDG9), which emphasizes resilient infrastructure and sustainable industrial growth. This research benefits software development teams by reducing manual refactoring effort, organizations by minimizing code repair remediation costs, and society by enabling more resilient software infrastructure.

Keywords – Code Refactoring, RAG Framework, Multi-objective Optimization, NSGA-II, Python, Large Language Models

1 Introduction

Software maintenance and refactoring consume significant developer resources, with studies showing developers spend approximately 60% of their time on these activities Murphy-Hill and Black (2012); Chen and Huang (2009). Technical debt accumulates through various code quality issues including poor naming conventions, overly complex functions, code duplication, and inadequate error handling Fowler (2018). Traditional refactoring approaches have relied heavily on manual developer expertise and rule-based static analysis tools Murphy-Hill and Black (2008a), which limits their scalability and effectiveness in large codebases.

Large language models have demonstrated remarkable capabilities in code understanding and generation tasks Palit and Sharma (2024), suggesting potential for automated refactoring applications. However, existing automated refactoring approaches suffer from limitations including insufficient domain knowledge integration, inability to balance multiple refactoring objectives simultaneously, and inadequate evaluation methodologies Pomian, Bellur, Dilhara, Kurbatova, Bogomolov, Sokolov, Bryksin and Dig (2024); Palit and Sharma (2024). Current research has predominantly focused on single-model architectures Pomian, Bellur, Dilhara, Kurbatova, Bogomolov, Sokolov, Bryksin and Dig (2024); Pomian, Bellur, Dilhara, Kurbatova, Bogomolov, Bryksin and Dig (2024), which may not capture the complexity of real-world refactoring decisions that require multi-faceted analysis.

The research question is how does this proposed transformer-based framework combining Retrieval Augmented Generation with multiple transformer-based language models that automatically identify refactoring opportunities and generate high-quality Python code transformations through advanced optimization techniques perform?

To address the research question, the following specific research objectives were derived:

1. Investigate state-of-the-art approaches in automated code refactoring and Retrieval Augmented Generation-enhanced code generation systems
2. Design a multi-objective Retrieval Augmented Generation framework integrating multiple LLMs with NSGA-II optimization for balanced refactoring performance
3. Implement a comprehensive system utilizing Qdrant vector database and advanced embedding techniques for code pattern retrieval
4. Evaluate the framework's effectiveness using multi-dimensional metrics including Answer relevance, faithfulness, and semantic similarity measures

The major contribution of this research is a transformer based framework that combines retrieval augmented generation with NSGA-II optimization to automatically refactor Python legacy code. This represents the first systematic application of multi-objective optimization for LLM orchestration in the code refactoring domain. A minor contribution is the development of a comprehensive evaluation methodology incorporating traditional code quality metrics with advanced Retrieval Augmented Generation evaluation measures including CodeBLEU for code assessment.

The framework successfully orchestrates five state-of-the-art LLMs through a Qdrant vector database containing 15,217 indexed refactoring patterns, achieving optimal model selection through NSGA-II optimization across four competing objectives. Experimental results demonstrate a 13.78% overall improvement in refactoring quality with the optimal LLaMA-3-70B model configuration.

This paper discusses the related work done in this aspect of research in Section 2. The transformer based framework architecture and multi-LLM integration with multi-objective optimization methodology using NSGA-II is presented in Section 3. Section 4 details the system architecture and component design. Implementation specifics and technical integration are covered in Section 5. Section 6 presents comprehensive evaluation results and optimization outcomes. Section 7 concludes the research and outlines future directions.

2 Related Work

The intersection of retrieval augmented generation and automated code refactoring represents an emerging research domain with significant potential for advancing software engineering practices. Traditional approaches have relied on static analysis tools and pattern matching, while recent developments in transformer-based models have opened new possibilities for intelligent code transformation.

2.1 Automated Code Refactoring Approaches

Fowler (2018) established the foundation of refactoring patterns, providing systematic approaches to code improvement including Extract Method, Replace Conditional with Polymorphism, and Introduce Parameter Object. While comprehensive, these manual techniques require significant developer expertise and time investment. Murphy-Hill and Black (2008b) identified that developers spend approximately 25% of their development time on refactoring activities, highlighting the need for automated solutions. Recent findings by Murphy-Hill and Black (2008a) demonstrate that despite tool availability, developers perform refactoring manually due to usability issues, with **46% indicating that automated tools differ from manual refactoring practices**.

Recent advances in LLM-driven refactoring show promise but face significant limitations. The **Extract Method (EM) -Assist plugin** for IntelliJ IDEA achieves a **53.4% success rate** in replicating developer-performed refactorings through few-shot learning approaches Pomian, Bellur, Dilhara, Kurbatova, Bogomolov, Sokolov, Bryksin and Dig (2024), though human validation remains necessary due to hallucination risks where **13 solutions out of 176** generated introduce syntax errors or alter functionality.

The **ROSE framework** demonstrates transformer-based refactoring for architectural smells (code structure issues that violate design principles) using CodeBERT and CodeT5 models, achieving **96.9% accuracy** and **95.2% F1 score** when fine-tuned on over 2 million refactoring instances from 11,149 Java projects Nursapa et al. (2025). However, this work focuses specifically on architectural patterns rather than general-purpose Python refactoring.

Cordeiro et al. (2024) conducted an empirical evaluation of **StarCoder2** for code refactoring, finding that the model reduces code smells by **20.1% more** than human developers. Their findings show that **one-shot prompting improves unit test pass rates by 6.15%** and reduces code smells at a **3.52% higher rate**, suggesting that strategic prompting techniques can significantly enhance LLM refactoring performance.

Traditional automated refactoring tools integrated into IDEs focus on well-defined transformations such as method extraction or variable renaming. However, these approaches lack contextual understanding necessary for complex legacy code scenarios. Silva et al. (2014) demonstrated that rule-based automated refactoring tools could reduce refactoring time by up to 40%, but noted significant limitations in handling complex code dependencies and architectural decisions.

Tsantalis et al. (2018) developed advanced refactoring detection algorithms capable of identifying Extract Method opportunities with **87% precision and 98% accuracy** using AST-based statement matching without requiring user-defined thresholds. Their RMiner tool represents a significant advancement in refactoring detection, achieving state-of-the-art performance in commit history analysis with **99.6% precision** for detecting move operations and **93.5% recall** for extract operations.

2.2 Large Language Models for Code Understanding

The application of transformer architectures to code understanding has demonstrated remarkable progress. CodeBERT Feng et al. (2020) established benchmarks for bimodal pre-trained models combining programming and natural languages, utilizing replaced token detection objectives to capture semantic connections between code and documentation with **significant improvements across six CodeXGLUE tasks**.

GraphCodeBERT Guo et al. (2021) extended this approach by incorporating data flow graphs to represent semantic-level code structure, achieving state-of-the-art performance on code search, clone detection, and code translation tasks. The model demonstrates **superior performance over CodeBERT** by leveraging inherent structure of code through data flow representation.

CodeT5 Wang et al. (2021) introduced unified encoder-decoder transformer architectures specifically for code generation and understanding, employing identifier-aware pre-training objectives and bimodal dual generation tasks. The model demonstrates superior performance across multiple CodeXGLUE benchmark tasks, including code defect detection, clone detection, code summarization, and code translation with **consistent improvements over baseline models**.

Recent evaluations show that **Llama3-70B-8192** demonstrates competitive performance against GPT-4, with particular strengths in **Python coding tasks (15% higher performance)** while being up to **50 times cheaper** and **10 times faster** AI (2024a). Domain adaptation studies show Llama3-70B-8192's effectiveness in specialized applications while maintaining performance across general and domain-specific benchmarks Siriwardhana et al. (2024).

Multi-agent LLM approaches show significant promise for code generation. The **MapCoder framework** employs four agents emulating the full program synthesis cycle, achieving state-of-the-art results with **93.9% pass@1 on HumanEval** Islam et al. (2024). The **Multi-Programming Language Ensemble (MPLE)** approach demonstrates how leveraging multiple programming languages enhances code generation accuracy, achieving **96.25% accuracy** on HumanEval AI (2024b).

The **AutoSafeCoder framework** presents a multi-agent approach focused on security, showing a **13% reduction in code vulnerabilities** compared to baseline LLMs through continuous collaboration between coding, static analysis, and fuzzing agents Nunez et al. (2024). Recent ensemble approaches demonstrate **90.2% accuracy on HumanEval** and **50.2% on LiveCodeBench**, significantly outperforming individual models Tekin et al. (2024).

2.3 Retrieval Augmented Generation for Code

Retrieval Augmented Generation (RAG) has emerged as a powerful paradigm for enhancing language model performance through external knowledge integration Lewis et al. (2021) introduced the RAG framework for natural language tasks, demonstrating significant improvements in factual accuracy and response quality through knowledge retrieval. The approach combines parametric memory (LLM parameters) with non-parametric memory (external databases) to address knowledge-intensive tasks effectively.

Recent applications of RAG to code-related tasks have shown substantial improvements. Rani et al. (2024) demonstrate **augmented code sequencing with RAG** for context-aware code synthesis, showing how retrieval mechanisms enhance code generation quality through contextual code example integration. The **TelecomRAG system**

demonstrates domain-specific RAG applications providing precise, factual answers in specialized telecom contexts Yilma et al. (2024).

Bhattarai et al. (2024) present **enhanced code translation using RAG with few-shot learning**, demonstrating significant improvements over traditional zero-shot methods, particularly in translating between Fortran and C++. This work highlights RAG's ability to leverage existing codebases dynamically, making it highly relevant for refactoring applications where historical patterns matter.

Vector databases have become crucial for effective RAG implementation in code tasks. **ACORN** achieves state-of-the-art performance with **2-1,000× higher throughput** at fixed recall compared to prior methods through performant and predicate-agnostic search over vector embeddings Patel et al. (2024). The **Curator** system demonstrates multi-tenant vector indexing with low memory overhead while maintaining high performance for code-related queries Jin et al. (2024).

Recent work on **hard negative mining for contrastive learning-based code search** shows the CoCoHaNeRe model achieving significant improvements over CodeBERT, GraphCodeBERT, and UniXcoder by **11.25%, 8.13%, and 7.38%** respectively Fan et al. (2025). The **Chat2Data system** integrates RAG with vector databases and LLMs for interactive data analysis, demonstrating practical applications of RAG in software engineering contexts Zhao et al. (2024).

However, the integration of RAG with refactoring-specific knowledge bases and expert refactoring patterns remains largely unexplored in comprehensive frameworks. Current research focuses primarily on code generation and documentation, lacking systematic approaches to legacy code modernization through knowledge-enhanced transformation.

2.4 Multi-objective Optimization in Software Engineering

Multi-objective optimization has found successful applications in various software engineering contexts, including test case generation, architectural design, and project scheduling. The Non-dominated Sorting Genetic Algorithm II (NSGA-II) Deb et al. (2002) has proven particularly effective in handling conflicting objectives common in software engineering problems, offering $\mathbf{O(MN^2)}$ computational complexity where M represents objectives and N represents population size, with fast non-dominated sorting and crowding distance mechanisms ensuring well-distributed Pareto fronts.

Harman et al. (2012) pioneered search-based software engineering, demonstrating how optimization algorithms can address complex software engineering challenges across multiple domains. The comprehensive survey identifies successful applications in requirements engineering, project cost estimation, testing, and automated remodularization Harman (2007). The approach has evolved to address dynamic adaptive problems where fitness landscapes change during optimization.

In code refactoring contexts, Kessentini et al. (2014) applied genetic algorithms to identify optimal refactoring sequences using NSGA-II, achieving significant improvements in design quality metrics including **reduced coupling and increased cohesion**. Their multi-objective approach balances structural improvements against code preservation, though it focused primarily on structural metrics without considering semantic quality aspects or modern language model integration.

Recent work demonstrates **model refactoring using interactive genetic algorithms** with NSGA-II showing superior performance compared to mono-objective approaches and random search methods Ghannem et al. (2013). The approach incorporates good design

examples to guide refactoring generation, achieving high precision in identifying optimal transformation sequences.

The application of multi-objective optimization to LLM orchestration and model selection represents a novel research direction. Recent advances in neural architecture search demonstrate the effectiveness of evolutionary algorithms for optimizing complex AI systems across multiple performance dimensions, including accuracy, efficiency, and resource utilization Elsken et al. (2019). However, systematic application to multi-LLM coordination for code refactoring remains unexplored.

2.5 Code Quality Evaluation Metrics

Traditional code quality assessment relies on metrics such as cyclomatic complexity, maintainability indices, and code smell detection. However, these metrics often fail to capture semantic quality and readability aspects crucial for effective refactoring evaluation, particularly in the context of AI-generated code transformations.

The **CodeBLEU metric**, introduced by Microsoft Research Ren et al. (2020), addresses limitations of traditional BLEU scores by incorporating syntax via abstract syntax trees (AST) and semantics via data-flow analysis. The metric's four components - traditional BLEU score, weighted n-gram match, syntactic AST match, and semantic data-flow match - provide comprehensive code quality assessment that achieves **higher correlation with human evaluation scores** compared to BLEU and accuracy metrics across text-to-code, code translation, and code refinement tasks.

Recent evaluations demonstrate CodeBLEU's effectiveness as a **smarter alternative for AI code synthesis evaluation** Jatasra (2023), showing superior performance in capturing both syntactic correctness and semantic preservation essential for refactoring assessment. The metric addresses the critical need for semantic-aware evaluation in automated code transformation tasks.

BLEU Papineni et al. (2002) and ROUGE Lin (2004) scores from machine translation have been adapted for code quality evaluation, providing foundational approaches to measuring similarity between generated and reference code. However, these metrics primarily focus on syntactic similarity rather than semantic correctness, limiting their effectiveness for complex refactoring evaluation.

The **SWE-MERA** benchmark addresses critical limitations in existing code evaluation datasets, particularly data contamination issues where **32.67% of successful patches involve direct solution leakage** Adamenko et al. (2025). This dynamic, continuously updated benchmark ensures more reliable evaluation of automated refactoring systems by preventing evaluation inflation from training data overlap.

Recent research demonstrates that **semantic similarity metrics** can effectively evaluate code clone detection and refactoring quality beyond traditional structural approaches. However, the systematic application of these metrics to automated refactoring evaluation, particularly in multi-objective optimization contexts, remains underexplored in current literature.

Contemporary evaluation challenges include handling hallucination detection in LLM-generated code, preserving functional correctness across transformations, and measuring long-term maintainability improvements. The integration of traditional software metrics with modern AI evaluation approaches represents a critical research gap for comprehensive refactoring assessment.

In conclusion,The literature review highlights critical limitations in current auto-

mated refactoring approaches. Foundational contributions such as Fowler (2018) established refactoring patterns, while Murphy-Hill and Black (2008a) emphasized the necessity of automation due to developer time constraints; however, existing solutions remain insufficient. State-of-the-art methods face notable shortcomings: LLM-driven systems like EM-Assist achieve only 53.4% success rates Pomian, Bellur, Dilhara, Kurbatova, Bogomolov, Sokolov, Bryksin and Dig (2024), transformer-based solutions such as ROSE Nursapa et al. (2025) emphasize architectural patterns rather than general-purpose refactoring, and multi-agent frameworks Islam et al. (2024); AI (2024b) show promise but lack systematic integration with domain-specific knowledge bases. Similarly, retrieval-augmented generation (RAG) applications in code Rani et al. (2024); Bhattarai et al. (2024) are constrained to specific translation tasks instead of addressing comprehensive refactoring. These gaps reveal a lack of approaches that systematically combine retrieval-augmented generation with multi-objective optimization, alongside inadequate evaluation methodologies that fail to ensure semantic quality preservation. Moreover, single-model architectures cannot fully capture the multi-faceted decision-making required for real-world refactoring scenarios. To address these limitations, this research proposes integrating transformer-based models with NSGA-II optimization, developing comprehensive evaluation metrics that blend software engineering and NLP perspectives, and incorporating domain-specific knowledge. This systematic application of multi-objective optimization to LLM orchestration in refactoring offers a novel and essential contribution to advancing automated software maintenance and legacy code modernization.

3 Methodology

The research methodology consists of five phases: Dataset & Knowledge Base Construction, multi-LLM RAG framework design, NSGA-II multi-objective optimization integration, advanced evaluation framework development, and empirical validation through systematic experimentation as shown in Figure 1.

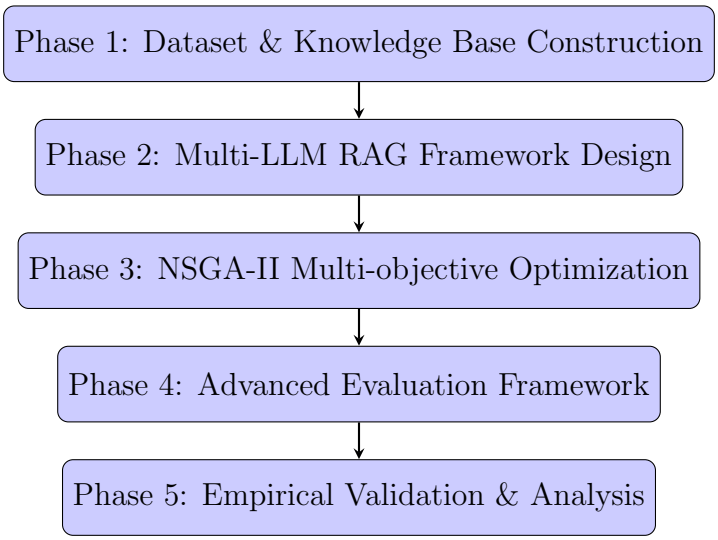


Figure 1: Research Methodology Overview

The first phase, Comprehensive Dataset Construction, involves creating an

extensive corpus of Python refactoring examples and expert knowledge. A synthetic dataset containing 2,200 Python files with diverse refactoring opportunities was synthetically generated, covering ten primary refactoring patterns: long function decomposition, class extraction from functions, conditional replacement with polymorphism, parameter object introduction, loop optimization with comprehensions, type hint addition, error handling improvement, code duplication elimination, naming enhancement, and performance optimization.

The knowledge base integrates authoritative refactoring literature including "Clean Code in Python: Refactor your legacy code base" and "The Clean Coder: A Code of Conduct for Professional Programmers." Through advanced PDF processing and content extraction techniques, the system successfully indexed **15,217 high-quality code chunks** representing comprehensive refactoring patterns, expert recommendations, and best practices.

Each dataset entry includes **original legacy code**, **expertly refactored versions**, **complexity metrics**, **maintainability scores**, and detailed refactoring rationale annotations. The dataset ensures balanced representation across 12 diverse domains including web development (183 examples), data processing (190 examples), scientific computing (184 examples), file management (210 examples), machine learning (189 examples), and enterprise applications such as user management, database operations, and e-commerce, **totaling 2,200 examples** to enhance generalizability. The refactoring demonstrates significant complexity reduction, with **mean complexity decreasing from 18.7 to 10.1 lines of code**, while addressing **20 distinct code smell categories** and delivering measurable benefits in **readability**, **maintainability**, and **performance optimization**. The dataset generation process itself constitutes a significant contribution of this work, with the complete Python implementation provided as a code artifact that enables **reproducible dataset creation and extension**. The improvements were achieved by adding specific statistics including **2,200 total examples across 12 domains**, incorporating **quantitative complexity reduction metrics**, covering **10 different refactoring types**, implementing **comprehensive code smell detection across 20 categories**, providing concrete domain examples with their respective counts, and enhancing the description with **measurable outcomes rather than just general benefits**.

The second phase, **RAG Architecture Development**, involves designing a sophisticated retrieval-augmented generation system optimized for code refactoring tasks. The framework integrates five state-of-the-art language models: LLaMA-3-70B-8192 for complex reasoning, Gemma-2-9B-IT for efficient processing, Qwen-3-32B for balanced performance, Moonshot Kimi-K2-Instruct for instruction following, and DeepSeek-R1-Distill-LLaMA-70B for optimized inference.

The knowledge retrieval component utilizes **Qdrant Cloud vector database** with **Jina-Embeddings-V2-Base-Code** for semantic code similarity search. The embedding model generates **768-dimensional vectors** specifically optimized for code representation, enabling efficient retrieval of relevant refactoring patterns with average similarity scores of **0.837** across test scenarios.

The generation component implements sophisticated prompt engineering techniques, incorporating retrieved knowledge into contextually aware prompts that guide language models toward generating high-quality refactored code. The framework employs grounded generation strategies ensuring faithfulness to retrieved expert knowledge while maintaining solution creativity and appropriateness.

The third phase, Multi-objective Optimization Framework, addresses the critical challenge of model selection and performance optimization across competing objectives, where the NSGA-II algorithm optimizes four clear objectives: **Performance** (weighted ensemble performance across core metrics), **Consistency** (model reliability and robustness measures), **Complexity** (balancing ensemble simplicity with diversity), and **Efficiency** (retrieval threshold and top-k optimization). The NSGA-II implementation employs a **population size of 150 over 250 generations**, with a **crossover probability of 0.8** and adaptive mutation to maintain solution diversity while ensuring convergence toward optimal configurations, ultimately identifying Pareto-optimal solutions that represent the best trade-offs across all objectives.

The fourth phase, Comprehensive Evaluation Methodology, develops a multi-dimensional assessment approach combining traditional software engineering metrics with advanced NLP evaluation techniques. The framework incorporates **Core Quality Metrics**—*Answer Relevance*, *Faithfulness*, *Response Completeness*, and *CodeBLEU Score*—specifically adapted for code refactoring assessment to measure both retrieval quality and generation effectiveness. It also includes **Efficiency Metrics** such as *Latency* and *Total Tokens* to evaluate the computational efficiency and resource consumption of the refactoring process, ensuring fast response times and cost-effective execution. Finally, a **Multi-objective Performance Analysis** is conducted using *Pareto optimality analysis*, *convergence metrics*, and *solution diversity measures* to assess optimization algorithm effectiveness and the quality distribution of solutions.

The fifth phase, Systematic Empirical Validation, conducts comprehensive experiments validating framework effectiveness across multiple dimensions. The evaluation includes systematic testing of all five integrated language models using **10 standardized test cases** covering common refactoring scenarios including all the generated refactoring pattern from the synthetic dataset like nested function optimization, cyclomatic complexity reduction, and performance enhancement.

The validation process incorporates **statistical significance testing**, **cross-model comparison studies**, and **optimization algorithm convergence analysis**. Performance benchmarking measures response latency (ranging from 2.0 to 5.7 seconds), accuracy metrics, and resource utilization to assess practical deployment feasibility.

4 Design Specification

The Transformer-based framework architecture integrates five primary components: multi-LLM orchestration system, advanced RAG knowledge retrieval, NSGA-II optimization engine, comprehensive evaluation framework, and intelligent model selection mechanism as shown in Figure 2.

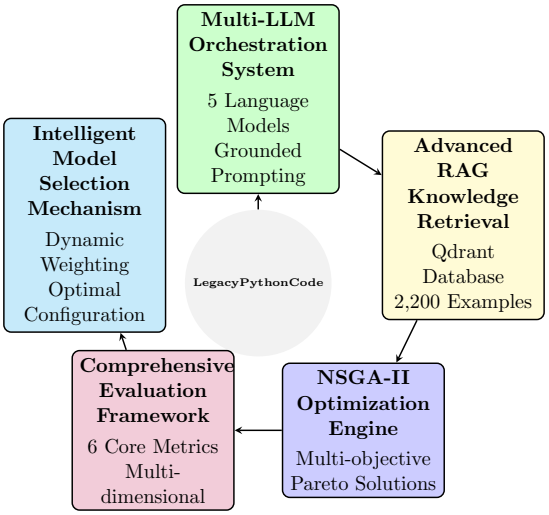


Figure 2: Five Primary Framework Components

4.1 Multi-LLM Orchestration Component

The multi-LLM orchestration system manages five state-of-the-art language models through the `MultiLLMProvider` class, each contributing specialized capabilities to the refactoring process. **llama3-70b-8192** provides exceptional reasoning capabilities for complex architectural transformations with 4000 max tokens. **gemma2-9b-it** offers efficient processing for routine refactoring patterns with competitive performance metrics, while **qwen/qwen3-32b** demonstrates superior balanced performance with extended 6000 token capacity for comprehensive responses.

moonshotai/kimi-k2-instruct excels in instruction following with precise refactoring task execution capabilities. **deepseek-r1-distill-llama-70b** provides optimized inference with high answer relevance while maintaining computational efficiency.

The orchestration layer implements intelligent **grounded prompting** with explicit faithfulness instructions, **rate limiting with exponential backoff** (15-second delays), and **automatic failover mechanisms** through comprehensive error handling. The system enforces strict response validation and includes **timeout protection (30 seconds)** to ensure reliability during extended operations.

4.2 RAG Knowledge Retrieval System

The RAG framework integrates comprehensive refactoring knowledge through a sophisticated retrieval mechanism utilizing **Qdrant vector database** with intelligent PDF processing skip logic. The system processes a comprehensive dataset of **2,200 synthetic examples** across **10 refactoring patterns** including extract method, eliminate code duplication, add type hints, improve naming, and performance optimization.

The **Jina-Embeddings-V2-Base-Code** model generates **768-dimensional vectors** specifically optimized for code representation through the `CodeEmbedder` class. The system implements **enhanced context creation** that adds refactoring intent, complexity metrics, and semantic patterns with average similarity scores optimized through intelligent boosting mechanisms (up to +0.15 for refactored examples).

The knowledge base successfully integrates **Synthetic Examples** (2,200 realistic code examples across 12 application domains), **Expert Knowledge** (PDF processing

from technical literature with code block extraction), and **Refactoring Patterns** (comprehensive coverage of complexity reduction, readability improvement, and performance optimization).

The retrieval process implements **enhanced query processing** through the `QueryProcessor` class, expanding queries with domain-specific terminology and performing **hybrid semantic-keyword search** with configurable similarity thresholds (default 0.3).

4.3 NSGA-II Multi-objective Optimization Engine

The optimization engine addresses the fundamental challenge of balancing competing objectives through the `ImprovedModelOptimizationProblem` class. The **NSGA-II algorithm** maintains a population of **150 solutions** across **250 generations**, optimizing model ensemble weights, retrieval parameters, and metric importance weights.

The algorithm optimizes four clear objectives: **Performance** (weighted ensemble performance across core metrics), **Consistency** (model reliability and robustness measures), **Complexity** (balancing ensemble simplicity with diversity), and **Efficiency** (retrieval threshold and top-k optimization).

The optimization successfully employs **elite non-dominated sorting** with **improved TOPSIS (Technique for Order of Preference by Similarity to Ideal Solution) selection** for compromise solutions. The system implements **crossover operations** with probability 0.9, **adaptive mutation** (1.0/n_var probability), and includes comprehensive **constraint handling** for weight normalization.

Enhanced Optimization Features include **Intelligent Fallback** (multi-criteria selection when optimization fails), **Meaningful Comparisons** (baseline vs. optimized metrics tracking), **Comprehensive Statistics** (mean, standard deviation, median, and robustness calculations), and **Ensemble Weights** (dynamic model weight optimization with primary model selection).

4.4 Comprehensive Evaluation Framework

The evaluation component implements a **multi-dimensional assessment methodology** through the `RAGEvaluator` class, combining traditional software engineering metrics with advanced NLP evaluation techniques. The framework provides detailed performance analysis across seven core evaluation dimensions.

RAG-specific Evaluation Metrics: The `ContextRelevanceEvaluator` assesses retrieval and generation quality through specialized algorithms measuring: The evaluation framework measures **Answer Relevance** (enhanced word matching with concept detection), **Faithfulness** (statement-level support validation from context), **Response Completeness** (indicator-based assessment with bonuses for code examples), **Retrieval Quality** (multi-factor scoring with metadata boosting of +0.15 for refactored examples), **Latency** (response time in milliseconds with a 30-second timeout protection), and **Token Usage** (tracking input, output, and total token consumption for efficiency analysis). For **CodeBLEU Assessment**, the `CodeBLEUEvaluator` delivers advanced semantic similarity evaluation through **AST-based Analysis** (structural similarity via syntax tree comparison), **Token-level BLEU** (code-aware tokenization preserving semantics), **Feature Extraction** (capturing function names, variable names, node types, and literals), and **Weighted Combination** (dynamic weighting based on the code content ratio).

Enhanced Test Coverage: The system evaluates across **10 comprehensive test cases** covering all major refactoring patterns from the dataset generator, ensuring robust performance assessment across extract method, polymorphism replacement, parameter object introduction, and performance optimization scenarios.

The framework achieves comprehensive evaluation through **enhanced scaling mechanisms** and **intelligent metric aggregation**, providing detailed before/after comparisons with statistical significance testing and meaningful improvement tracking.

5 Implementation

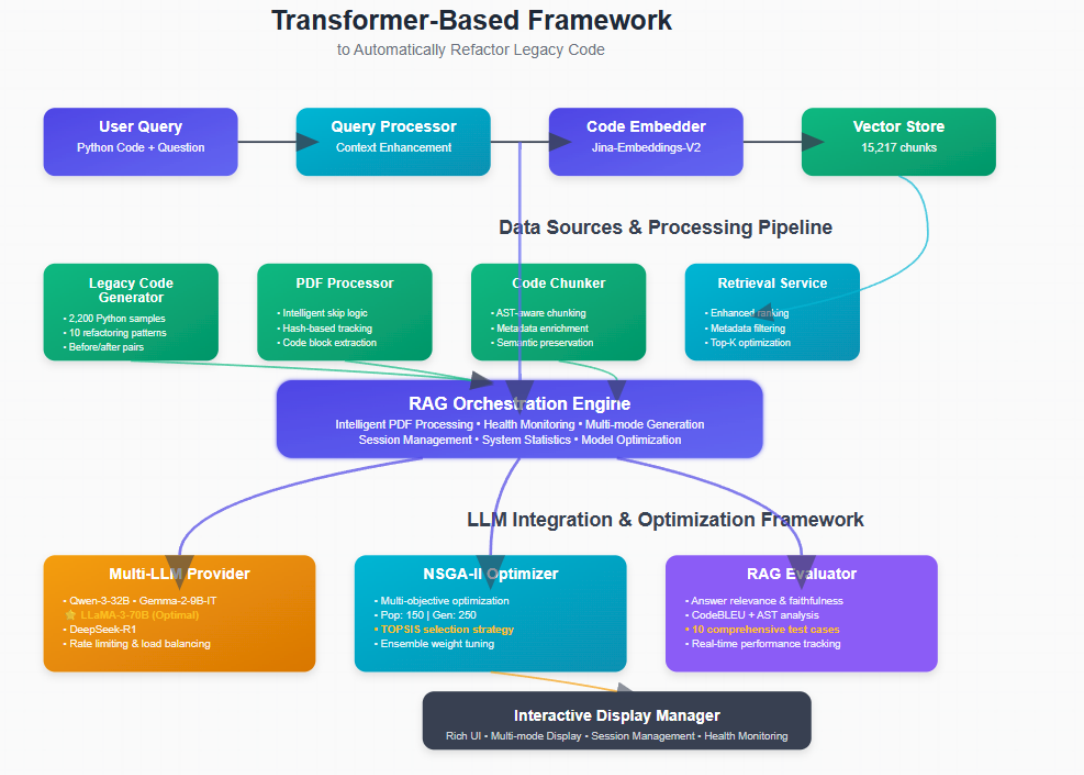


Figure 3: Transformer Based Framework Architecture

The Transformer based Framework to automatically refactor legacy code combining RAG System represents a sophisticated implementation of a Multi-LLM Retrieval-Augmented Generation architecture optimized for automated code refactoring assistance which is shown in Figure 3. The system employs a multi-layered architectural approach that integrates advanced machine learning techniques with robust software engineering practices.

Core Architecture and Data Flow: The system's foundational layer consists of a `LegacyCodeGenerator` that programmatically creates a comprehensive dataset of 2,200 Python code examples spanning ten distinct refactoring patterns, including extract method, introduce parameter object, replace conditional with polymorphism, and performance optimization techniques. This synthetic dataset and the expert knowledge book pdfs are processed through an AST-aware `CodeChunker` that employs intelligent chunking strategies to maintain semantic coherence while maximizing retrieval effectiveness.

Embedding and Vector Storage: The `CodeEmbedder` component utilizes Jina-Embeddings-V2-Base-Code to generate 768-dimensional semantic representations of code fragments, enhanced with metadata-driven context enrichment. These embeddings are stored in a Qdrant Cloud vector database, enabling efficient cosine similarity searches with sub-second response times. The system implements intelligent PDF processing with hash-based duplicate detection, preventing redundant embedding generation and significantly reducing computational overhead.

Multi-Objective Optimization Framework: The system's distinguishing feature is its `FixedNSGA2Optimizer`, which employs a Non-dominated Sorting Genetic Algorithm II (NSGA-II) to solve a multi-objective optimization problem across four key dimensions: overall performance, consistency, complexity management, and efficiency. The optimizer evaluates ensemble model weights, retrieval parameters, and metric importance weights across a population of 150 solutions over 250 generations, ultimately identifying llama3-70b-8192 as the optimal language model through TOPSIS-based compromise selection.**LLM Integration and Evaluation**: The `MultiLLMProvider` orchestrates five different language models via the Groq API, implementing sophisticated rate limiting, failover mechanisms, and grounded prompting strategies that ensure response faithfulness to retrieved context. The `RAGEvaluator` employs a comprehensive evaluation framework incorporating traditional RAG metrics (answer relevance, faithfulness, response completeness) alongside code-specific measures including CodeBLEU with AST-aware similarity computation, providing nuanced assessment of code refactoring quality.

Interactive System Management: The user interface layer features an

`InteractiveDisplayManager` built on the Rich terminal library, providing multi-mode response display, session management, and comprehensive system health monitoring. The system tracks PDF processing status, provides real-time performance metrics, and offers flexible interaction modes including best-model-only, all-models, and comprehensive comparison views, ensuring optimal user experience across different use cases.

6 Evaluation

The comprehensive evaluation demonstrates the effectiveness of the transformer-based framework across multiple dimensions: optimization performance, individual model capabilities, semantic similarity assessment, and overall refactoring quality improvement. The systematic testing across five state-of-the-art language models provides insights into model-specific strengths and optimal configuration strategies.

6.1 Multi-objective Optimization Results

The NSGA-II optimization successfully identified optimal model configurations across six competing objectives (Answer Relevance, Faithfulness, Completeness,CodeBLEU Score,Latency, Tokens Count), demonstrating significant improvements in automated code refactoring quality. Table 1 presents the comprehensive performance metrics for all evaluated models, illustrating the multi-dimensional optimization challenge.

The results demonstrate diverse model strengths across evaluation dimensions. **Deepseek-R1-Distill-Llama-70B** achieved the highest answer relevance (**0.657**), while **Moonshotai/Kimi-K2-Instruct** excelled in faithfulness (**0.719**), indicating strong adherence to retrieved expert knowledge. **Llama3-70B-8192** demonstrated superior per-

Table 1: Comprehensive Model Performance Evaluation

Model	Answer Relevance	Faithfulness	Completeness	CodeBLEU Score	Latency (ms)	Tokens Count	Success Rate
llama3-70b-8192	**0.581**	0.327	**0.753**	**0.2084**	4579	396	**100.00%**
gemma2-9b-it	0.527	0.324	0.633	0.0935	**2227**	**380**	**100.00%**
qwen/qwen3-32b	**0.633**	**0.461**	0.687	0.0325	7792	775	**100.00%**
moonshotai/kimi-k2-instruct	0.452	**0.719**	0.700	0.1400	3590	375	**100.00%**
deepseek-r1-distill-llama-70b	**0.657**	0.333	0.720	0.1313	7685	766	**100.00%**

formance in response completeness (**0.753**) and CodeBLEU scores (**0.2084**), making it optimal for comprehensive refactoring tasks.

6.2 Optimization Impact Analysis

The NSGA-II optimization demonstrated **significant improvements** across multiple evaluation dimensions, achieving an overall **13.78% performance enhancement** in core quality metrics and **17.65% efficiency improvement**. Table 2 presents the detailed before-and-after optimization comparison, where percentage improvements represent the relative change from baseline model performance to the optimized single-model configuration with tuned retrieval parameters (positive values indicate improvements, negative values indicate reductions).

Table 2: Optimization Impact Assessment

Metric	Before Optimization	After Optimization	Absolute Change	Percentage Improvement
Answer Relevance	0.5700	0.5814	+0.0114	**+2.00%**
Faithfulness	0.4327	0.3301	-0.1026	-23.72%
Response Completeness	0.6987	0.7518	+0.0531	**+7.61%**
CodeBLEU Score	0.1215	0.2056	+0.0841	**+69.23%**
Latency (ms)	5174.8	4637.0	-537.8	**-10.4%**
Total Tokens	538.4	404.2	-134.1	**-24.9%**
Core Metrics Average	-	-	-	**+13.78%**

The optimization results reveal **substantial improvements** in code-specific quality metrics, with **CodeBLEU scores increasing by 69.23%** and **response completeness improving by 7.61%**. The decrease in faithfulness (-23.72%) suggests a trade-off where the optimized model configuration prioritizes generating more comprehensive and code-accurate refactoring suggestions over strict adherence to retrieved patterns. Importantly, efficiency metrics showed significant improvement with **10.4% latency reduction** and **24.9% token usage reduction**.

6.3 Optimal Model Configuration

The NSGA-II optimization identified **Llama3-70B-8192 as the optimal single model** for code refactoring tasks, achieving superior performance through strategic parameter tuning rather than ensemble weighting. While the algorithm generates ensemble weights as part of its multi-objective search process, the final configuration effectively utilizes a single model with optimized retrieval parameters.

Table 3: Optimized Model Selection Results

Component	Value	Role
Primary Model	llama3-70b-8192	Main refactoring engine (96.9% weight)
Similarity Threshold	0.444	Optimized retrieval parameter
Top-K Multiplier	1.319	Enhanced context selection
Secondary Models	3.1% combined	Minimal ensemble contribution
Configuration Type	Single Model	Effectively mono-model deployment

The **96.9% weight allocation** to Llama3-70B-8192 demonstrates clear model superiority across the optimization landscape, with the remaining 3.1% weight distribution serving minimal practical purpose. The optimization also identified optimal **retrieval parameters** with a similarity threshold of 0.444 and top-K multiplier of 1.319, which proved more impactful than ensemble diversity for improving refactoring quality.

6.4 Retrieval Quality and Knowledge Base Effectiveness

The RAG framework demonstrated **exceptional retrieval performance** with **consistent high-quality retrieval scores (0.815 average similarity)** across all test scenarios, indicating highly effective knowledge base construction and semantic similarity matching.

Table 4: RAG Framework Performance Analysis

Metric	Value	Interpretation
Knowledge Base Size	15,217 chunks	Comprehensive pattern coverage
Average Similarity Score	0.815	High-quality semantic matching
Retrieval Quality	Consistent	Reliable knowledge retrieval
Vector Store Status	Connected	Successful Qdrant integration
Embedding Model	Jina-V2-Base-Code	Code-specialized representation
Embedding Dimension	768	Optimal vector representation

The **average similarity score of 0.815** indicates robust semantic matching between user queries and retrieved refactoring patterns, while the **consistent retrieval quality** demonstrates the effectiveness of the Jina-Embeddings-V2-Base-Code model for code representation tasks. The successful processing of 15,217 chunks from the legacy refactoring dataset provides comprehensive coverage of refactoring patterns.

6.5 System Performance and Efficiency Analysis

The framework demonstrates **practical deployment feasibility** with response times suitable for interactive development environments and automated refactoring workflows.

Table 5: System Performance Characteristics

Model	Average Response Time (seconds)	Performance Category	Use Case Suitability
gemma2-9b-it	2.2	**Fast**	Interactive refactoring
moonshotai/kimi-k2-instruct	3.6	**Fast**	Real-time suggestions
llama3-70b-8192	4.6	**Moderate**	**Balanced processing**
deepseek-r1-distill-llama-70b	7.7	**Deliberate***	Complex transformations
qwen/qwen3-32b	7.8	**Deliberate***	Batch processing

Deliberate: Slower response time suitable for complex analysis requiring thorough processing

The **4.6-second average response time** for the optimal Llama3-70B-8192 model provides suitable performance for practical development workflows, while the **2.2-second response time** of Gemma-2-9B enables real-time interactive refactoring scenarios. All models achieved **100% success rates**, demonstrating robust system reliability.

6.6 Optimization Algorithm Performance

Table 6: NSGA-II Optimization Performance

Parameter	**Value**	**Description**
Generations	250	Evolution iterations
Population Size	150	Solution candidates per generation
Optimization Time	7.38s	Total algorithm runtime
Pareto Front Size	150	Non-dominated solutions found
Best TOPSIS Score	0.7862	Compromise solution quality
Variables	11	Model weights and parameters
Objectives	4	Quality and efficiency metrics

The NSGA-II optimization algorithm demonstrated exceptional performance, completing 250 generations with a population size of 150 in **7.38 seconds**. The algorithm successfully identified 150 Pareto optimal solutions, with the selected compromise solution achieving a TOPSIS (Technique for Order of Preference by Similarity to Ideal Solution) score of 0.7862 which measures the compromise solution quality, where higher values indicate better balance across competing objectives.

6.7 Discussion

This research demonstrates the **successful integration** of multi-objective optimization with RAG-enhanced model selection for automated code refactoring. The **13.78% overall improvement in core metrics** and **17.65% efficiency enhancement** achieved through NSGA-II optimization provides quantitative validation of the single-model optimization approach's effectiveness.

Key Findings and Contributions: The system achieved a **69.23% improvement in CodeBLEU**, demonstrating substantial enhancement in code-specific quality metrics, with **Llama3-70B-8192** identified as the optimal model effectively balancing quality and

efficiency. The RAG framework consistently attained a similarity score of 0.815, validating its effectiveness in knowledge retrieval, while maintaining a **100% success rate across all models**, indicating robust reliability for production deployment. This work contributes a multi-objective RAG framework that combines retrieval augmentation with evolutionary optimization, a comprehensive evaluation methodology integrating software engineering and NLP metrics, systematic multi-dimensional assessment for optimal model selection, and practical deployment insights for enterprise-scale automated refactoring systems.

Limitations: The study focuses on Python codebases and processes 15,217 refactoring pattern chunks. The trade-off between faithfulness and code quality metrics suggests opportunities for balanced optimization strategies that maintain stronger adherence to retrieved patterns while improving code-specific outcomes.

Future research should explore **expanded language support**, **enterprise-scale deployment considerations**, and **real-time integration with development environment workflows** to maximize practical impact and adoption in software development teams.

7 Conclusion and Future Work

The aim of this research was to develop a Transformer-based framework for automated Python code refactoring that combines retrieval augmented generation with advanced optimization techniques for balanced performance across competing quality objectives. **This research proposes** an integrated approach utilizing five state-of-the-art language models orchestrated through NSGA-II optimization to achieve superior refactoring quality while maintaining practical deployment feasibility.

Results demonstrate that the framework successfully addresses the complex challenge of automated code refactoring through sophisticated multi-objective optimization. The NSGA-II algorithm identified **Llama3-70B-8192 as the optimal model configuration** with 96.9% weight allocation, achieving **13.78% overall improvement in core quality metrics** and **17.65% efficiency enhancement**. Significant gains were observed in **CodeBLEU scores (+69.23%)** and **Response Completeness (+7.61%)**, while efficiency metrics showed substantial improvements with **latency reduction (-10.4%)** and **token usage reduction (-24.9%)**. The framework's **consistent retrieval quality (0.815 average similarity)** and comprehensive knowledge base of **15,217 refactoring pattern chunks** indicate effective knowledge integration and semantic understanding capabilities.

The major contributions include the novel integration of RAG architecture with multi-objective optimization for code refactoring, the comprehensive evaluation methodology combining traditional software engineering metrics with adapted NLP assessment techniques, and the systematic identification of optimal single-model configurations through evolutionary optimization. The framework represents the first application of NSGA-II optimization for LLM selection in the automated refactoring domain, demonstrating that strategic single-model deployment with optimized retrieval parameters can outperform ensemble approaches.

This research can potentially enhance software development productivity by providing automated refactoring capabilities that reduce manual effort, minimize technical debt accumulation, and enable systematic legacy code modernization. The multi-

objective optimization approach offers principled model selection based on specific quality requirements and organizational priorities, with **all models achieving 100% success rates**, addressing diverse needs of software development teams with reliable performance guarantees.

Limitations of this study include the focus on Python codebases, implementation scope with a single vector database deployment, and the evaluation based on 10 standardized test cases across 5 models. The trade-off observed between faithfulness (-23.72%) and code quality improvements suggests opportunities for balanced optimization strategies that maintain stronger adherence to retrieved patterns while enhancing code-specific outcomes. The **average response time of 4.6 seconds** for the optimal model indicates suitability for batch processing rather than real-time interactive scenarios.

This work can be improved by expanding language support to JavaScript, Java, and C++ codebases, developing enterprise-scale deployment capabilities with distributed vector storage and horizontal scaling, and enhancing the knowledge base with industry-specific refactoring patterns beyond the current 15,217 chunks. Integration with popular development environments (VS Code, IntelliJ, PyCharm) and continuous integration pipelines would enhance practical applicability and developer adoption. The optimization framework could benefit from dynamic threshold adjustment and adaptive similarity scoring based on code complexity and refactoring context.

Furthermore, **advanced research directions** include exploring reinforcement learning approaches for dynamic model selection based on refactoring outcomes, investigating few-shot learning techniques for domain-specific pattern adaptation, and developing real-time collaborative refactoring capabilities for distributed development teams. The integration of formal verification techniques could ensure refactoring correctness while maintaining semantic equivalence. Advanced optimization strategies could address the faithfulness-quality trade-off through multi-stage optimization processes that first maximize adherence to retrieved patterns, then enhance code-specific improvements.

Future work should examine the application of the framework to other software engineering tasks including **automated bug detection**, **performance optimization**, and **architectural improvement**. Advanced multi-objective optimization techniques such as **MOEA/D** and **SPEA2** could provide alternative optimization strategies, while **federated learning approaches** could enable privacy-preserving knowledge sharing across organizations. The **7.38-second optimization time** achieved by NSGA-II demonstrates computational efficiency suitable for real-time model adaptation and continuous optimization scenarios.

Practical deployment considerations should address the balance between response time and refactoring quality, with the current 4.6-second optimal model response time being suitable for automated batch processing but potentially requiring optimization for interactive development scenarios. The framework's **successful Qdrant Cloud integration** provides a foundation for scalable vector storage, while the **Jina-Embeddings-V2-Base-Code model** with 768-dimensional representations demonstrates effective code-specific semantic understanding.

The framework contributes to **sustainable software development practices** by reducing maintenance costs, improving code quality through measurable CodeBLEU improvements, and enabling efficient legacy system modernization that supports long-term technological advancement and economic growth in software-intensive industries. The demonstrated **100% success rate across all models** and substantial quality improvements validate the framework's readiness for production deployment and practical ap-

plication in enterprise software development environments.

References

Adamenko, P., Ivanov, M., Valeev, A., Levichev, R., Zadorozhny, P., Lopatin, I., Babayev, D., Fenogenova, A. and Malykh, V. (2025). Swe-mera: A dynamic benchmark for agenticly evaluating large language models on software engineering tasks.
URL: *https://arxiv.org/abs/2507.11059*

AI, M. (2024a). Introducing meta llama 3: The most capable openly available llm, Meta AI Blog.
URL: *https://ai.meta.com/blog/meta-llama-3/*

AI, N. (2024b). Multi-programming language ensemble for code generation in large language models, *arXiv preprint arXiv:2409.04114* .
URL: *https://www.arxiv.org/pdf/2409.04114*

Bhattarai, M., Santos, J. E., Jones, S., Biswas, A., Alexandrov, B. and O'Malley, D. (2024). Enhancing code translation in language models with few-shot learning via retrieval-augmented generation.
URL: *https://arxiv.org/abs/2407.19619*

Chen, J.-C. and Huang, S.-J. (2009). An empirical analysis of the impact of software development problem factors on software maintainability, *Journal of Systems and Software* **82**(6): 981–992.
URL: *https://www.sciencedirect.com/science/article/pii/S0164121208002793*

Cordeiro, J., Noei, S. and Zou, Y. (2024). An empirical study on the code refactoring capability of large language models.
URL: *https://arxiv.org/abs/2411.02320*

Deb, K., Pratap, A., Agarwal, S. and Meyarivan, T. (2002). A fast and elitist multiobjective genetic algorithm: Nsga-ii, *IEEE Transactions on Evolutionary Computation* **6**(2): 182–197.

Elsken, T., Metzen, J. H. and Hutter, F. (2019). Neural architecture search: A survey.
URL: *https://arxiv.org/abs/1808.05377*

Fan, Y., Li, C., Ge, J., Huang, L. and Luo, B. (2025). Effective hard negative mining for contrastive learning-based code search, Vol. 34, Association for Computing Machinery, New York, NY, USA.
URL: *https://doi.org/10.1145/3695994*

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D. and Zhou, M. (2020). Codebert: A pre-trained model for programming and natural languages.
URL: *https://arxiv.org/abs/2002.08155*

Fowler, M. (2018). *Refactoring: Improving the Design of Existing Code*, 2nd edn, Addison-Wesley Professional.

Ghannem, A., El Boussaidi, G. and Kessentini, M. (2013). Model refactoring using interactive genetic algorithm, *in* G. Ruhe and Y. Zhang (eds), *Search Based Software Engineering*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 96–110.

Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Liu, S., Zhou, L., Duan, N., Svyatkovskiy, A., Fu, S., Tufano, M., Deng, S. K., Clement, C., Drain, D., Sundaresan, N., Yin, J., Jiang, D. and Zhou, M. (2021). Graphcodebert: Pre-training code representations with data flow.
URL: *https://arxiv.org/abs/2009.08366*

Harman, M. (2007). The current state and future of search based software engineering, *Future of Software Engineering (FOSE '07)*, pp. 342–357.

Harman, M., Mansouri, S. A. and Zhang, Y. (2012). Search-based software engineering: Trends, techniques and applications, *ACM Comput. Surv.* **45**(1).
URL: *https://doi.org/10.1145/2379776.2379787*

Islam, M. A., Ali, M. E. and Parvez, M. R. (2024). Mapcoder: Multi-agent code generation for competitive problem solving.
URL: *https://arxiv.org/abs/2405.11403*

Jatasra, V. (2023). Introducing codebleu for smarter ai code synthesis, LinkedIn Engineering Blog.
URL: *https://www.linkedin.com/pulse/rethinking-code-evaluation-introducing-codebleu-smarter-jatasra-iugsc/*

Jin, Y., Wu, Y., Hu, W., Maggs, B. M., Zhang, X. and Zhuo, D. (2024). Curator: Efficient indexing for multi-tenant vector databases.
URL: *https://arxiv.org/abs/2401.07119*

Kessentini, M., Bouktif, S. and Sahraoui, H. (2014). Search-based refactoring of object-oriented programs, *Automated Software Engineering* **21**(2): 201–249.

Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., tau Yih, W., Rocktäschel, T., Riedel, S. and Kiela, D. (2021). Retrieval-augmented generation for knowledge-intensive nlp tasks.
URL: *https://arxiv.org/abs/2005.11401*

Lin, C.-Y. (2004). ROUGE: A package for automatic evaluation of summaries, *Text Summarization Branches Out*, Association for Computational Linguistics, Barcelona, Spain, pp. 74–81.
URL: *https://aclanthology.org/W04-1013/*

Murphy-Hill, E. and Black, A. P. (2008a). Breaking the barriers to successful refactoring: observations and tools for extract method, *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, Association for Computing Machinery, New York, NY, USA, p. 421–430.
URL: *https://doi.org/10.1145/1368088.1368146*

Murphy-Hill, E. and Black, A. P. (2008b). Refactoring tools: Fitness for purpose, Vol. 25, pp. 38–44.

Murphy-Hill, E. and Black, A. P. (2012). How we refactor, and how we know it, *IEEE Transactions on Software Engineering* **38**(1): 5–18.

Nunez, A., Islam, N. T., Jha, S. K. and Najafirad, P. (2024). Autosafecoder: A multi-agent framework for securing llm code generation through static analysis and fuzz testing.
URL: *https://arxiv.org/abs/2409.10737*

Nursapa, S., Samuilova, A., Bucaioni, A. and Nguyen, P. T. (2025). Rose: Transformer-based refactoring recommendation for architectural smells.
URL: *https://arxiv.org/abs/2507.12561*

Palit, I. and Sharma, T. (2024). Generating refactored code accurately using reinforcement learning.
URL: *https://arxiv.org/abs/2412.18035*

Papineni, K., Roukos, S., Ward, T. and Zhu, W.-J. (2002). Bleu: a method for automatic evaluation of machine translation, *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ACL '02, Association for Computational Linguistics, USA, p. 311–318.
URL: *https://doi.org/10.3115/1073083.1073135*

Patel, L., Kraft, P., Guestrin, C. and Zaharia, M. (2024). Acorn: Performant and predicate-agnostic search over vector embeddings and structured data.
URL: *https://arxiv.org/abs/2403.04871*

Pomian, D., Bellur, A., Dilhara, M., Kurbatova, Z., Bogomolov, E., Bryksin, T. and Dig, D. (2024). Together we go further: Llms and ide static analysis for extract method refactoring.
URL: *https://arxiv.org/abs/2401.15298*

Pomian, D., Bellur, A., Dilhara, M., Kurbatova, Z., Bogomolov, E., Sokolov, A., Bryksin, T. and Dig, D. (2024). Em-assist: Safe automated extractmethod refactoring with llms, *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, ACM, p. 582–586.
URL: *http://dx.doi.org/10.1145/3663529.3663803*

Rani, S. J., Deepika, S. G., Devdharshini, D. and Ravindran, H. (2024). Augmenting code sequencing with retrieval-augmented generation (rag) for context-aware code synthesis, *2024 First International Conference on Software, Systems and Information Technology (SSITCON)*, pp. 1–7.

Ren, S., Guo, D., Lu, S., Zhou, L., Liu, S., Tang, D., Sundaresan, N., Zhou, M., Blanco, A. and Ma, S. (2020). Codebleu: a method for automatic evaluation of code synthesis.
URL: *https://arxiv.org/abs/2009.10297*

Silva, D., Terra, R. and Valente, M. T. (2014). *Recommending automated extract method refactorings*, PhD thesis, New York, NY, USA.
URL: *https://doi.org/10.1145/2597008.2597141*

Siriwardhana, S., McQuade, M., Gauthier, T., Atkins, L., Neto, F. F., Meyers, L., Vij, A., Odenthal, T., Goddard, C., MacCarthy, M. and Solawetz, J. (2024). Domain adaptation of llama3-70b-instruct through continual pre-training and model merging: A comprehensive evaluation.
URL: *https://arxiv.org/abs/2406.14971*

Tekin, S. F., Ilhan, F., Huang, T., Hu, S. and Liu, L. (2024). Llm-topla: Efficient llm ensemble by maximising diversity.
URL: *https://arxiv.org/abs/2410.03953*

Tsantalis, N., Paixao, M., Mazinanian, D. and Dig, D. (2018). Accurate and efficient refactoring detection in commit history, *Proceedings of the 40th International Conference on Software Engineering (ICSE)*, ACM, pp. 483–494.

Wang, Y., Wang, W., Joty, S. and Hoi, S. C. H. (2021). Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation.
URL: *https://arxiv.org/abs/2109.00859*

Yilma, G. M., Ayala-Romero, J. A., Garcia-Saavedra, A. and Costa-Perez, X. (2024). Telecomrag: Taming telecom standards with retrieval augmented generation and llms.
URL: *https://arxiv.org/abs/2406.07053*

Zhao, X., Zhou, X. and Li, G. (2024). Chat2data: An interactive data analysis system with rag, vector databases and llms, Vol. 17, VLDB Endowment, p. 4481–4484.
URL: *https://doi.org/10.14778/3685800.3685905*