AI-Powered Architecture Refactoring: From Legacy Systems to Modern Patterns

An Initial Case Study with LLM and CQRS

Mohamed El BOUKHARI[1], Nassim KHARMOUM[2], Soumia ZITI[3]
IPSS Research Team-Faculty of Sciences, Mohammed V University in Rabat, Rabat, Morocco[1, 2, 3]
National Center for Scientific and Technical Research (CNRST), Rabat, Morocco[2]

***Abstract*—This study explores the integration of artificial intelligence (AI), especially large language models (LLMs), into software engineering, particularly the architecture refactoring process, focusing on automated command-query classification for legacy systems transitioning to the Command Query Responsibility Segregation (CQRS) pattern. We present *Airchitect*, a modular system. NET-based tools that orchestrate legacy code analysis, LLM-driven classification, CQRS artifact generation, and automated test creation are also available. Based on the CodeLlama model, Airchitect achieved a 16x–40x reduction in classification time compared to expert manual methods while maintaining over 85% classification accuracy. A test case involving N-tier legacy classes demonstrated the model's ability to decompose and modularize the methods into CQRS-aligned components. Despite these gains, the study highlights key limitations: the need for human validation in complex or ambiguous cases, dependence on high-quality labeled datasets, and variability of legacy patterns that challenge rule-based automation. The results suggest that LLMs, when embedded in structured tools like Airchitect, can significantly accelerate modernization workflows—provided they are used in tandem with expert oversight.**

***Keywords*—Artificial intelligence; LLM; AI-driven refactoring; code-level refactoring; legacy systems, command and query responsibility segregation; CQRS; software architecture refactoring; software engineering; CodeSearchNet**

I. Introduction

Legacy systems are business-critical applications developed using older technologies or architectures that continue to provide essential services despite their age. Many organizations have relied extensively on systems-built decades ago using legacy programming languages or architectures, along with design methods that preceded modern software engineering practices. The primary challenge is to maintain and evolve these systems to meet current business demands without disrupting operations. Legacy systems typically suffer from technical debt (the accumulated costs of additional rework caused by choosing an easy solution instead of using a better approach that would take longer), tight coupling, and inadequate separation of concerns, rendering them difficult to modify, scale [1], and integrate with modern technologies.

Legacy systems form the backbone of the IT infrastructure of many organizations despite their age and technological limitations. These systems are characterized as "old software systems that are usually designed and documented inadequately but still perform an important job for critical business applications." The business value of these systems has deteriorated over time owing to their lack of consistency and limited evolution support; however, they remain indispensable [2] because some of their functions are too important to be discarded completely and too costly to reconstruct.

Legacy systems also represent significant accumulated knowledge and established technology. They "embed important knowledge acquired over the years" and constitute "critical assets for enterprises." However, these systems incur high maintenance costs and are increasingly vulnerable to failure because of the diminishing pool of experts who understand their inner workings [4]. Billions of lines of legacy code must evolve into modern technologies to enable progress in business practices.

Optimizing legacy applications requires substantial effort, which is often not immediately achievable. This process involves difficult and complex work" and necessitates careful decision-making to support successful system management. When dealing with particularly complex problems, organizations must "apply with a systematic approach for defining which modifications should be made or recommended" [5]. This emphasizes the need for structured methodologies to assess, analyze, and refurbish legacy systems while preserving their intrinsic value.

A. Legacy Systems Refactoring

The shift from old to new architecture is a significant change that can make systems more flexible and work better [6]. The addition of smart systems to organizations has been shown to boost business performance.

A comprehensive survey conducted in 2021, encompassing 1,183 users of the IntelliJ Integrated Development Environment (IDE), revealed that only 10-11% of refactoring tasks were executed using automated tools, with the majority being performed manually. The 'Rename' function emerged as the most frequently utilized, with an 85.8% usage rate, particularly among regular developers. Approximately 46% of cases in which tool assistance is feasible are tool-assisted, predominantly by teams aiming for consistency, although the settings are seldom altered [7]. Floss refactoring, which is integrated with other tasks, prevails over root canal approaches. Notably, half of the refactoring tasks are of simple to medium complexity, which tools often fail to address, and commit messages rarely explicitly indicate the refactoring activities.

B. Challenges of Legacy Systems Refactoring

Legacy systems present significant challenges to modern software development, particularly in integrating established software engineering principles into their architecture. Developers frequently encounter difficulties when refactoring existing codebases to conform to best practices, such as SOLID, DRY (Do not Repeat Yourself), SRP (Single Responsibility Principle), and KISS (Keep It Simple, Stupid) [16]. These principles form the foundation of maintainable and scalable software architecture, with the SRP emphasizing that each class should be responsible for only one aspect of the software functionality [17]. This approach seeks to preserve design simplicity and minimize disruptions during future modifications.

C. Technical Complexities and Compatibility Issues

Integrating AI-based software engineering principles into legacy systems presents multifaceted challenges. Technical complexities and compatibility issues are the primary obstacles because legacy systems are typically not designed to accommodate modern Artificial Intelligence (AI) technologies [18]. This fundamental architectural mismatch often necessitates significant system overhauls or the development of middleware solutions to bridge the gap between legacy codes and AI-driven tools.

D. Organizational Management and Manual Processes

The integration process requires careful assessment of existing systems to identify potential integration points and compatibility problems. Companies using legacy systems or relying heavily on manual processes face particularly steep challenges because AI integration requires substantial initial commitments of time and resources [19]. This resource investment extends beyond technical implementation to include addressing organizational change management and ensuring seamless operations between AI models and existing business processes.

E. Disruption of Established Workflows

One significant challenge is the potential disruption of established workflows. Legacy systems often have well-defined processes to which employees are accustomed, and the introduction of AI-driven tools to apply software engineering principles may lead to resistance or inefficiency if not managed carefully. This also necessitates acknowledging the organizational and cultural challenges associated with such changes [20], including employee adaptation and its impact on workflows. Implementing new AI technologies and training staff to use them effectively involves an inevitable learning curve.

The effective integration of artificial intelligence into legacy systems to implement the SOLID, DRY, SRP, and KISS principles necessitates meticulous planning to avoid potential incompatibility issues that may hinder the implementation process [21]. Organizations must strategically address this challenge by balancing the technical aspects of integration with human and process elements to achieve significant enhancements in code quality and maintainability.

F. Integration of Artificial Intelligence into the Refactoring Process

The integration of artificial intelligence into code refactoring has progressed from initial experimental methodologies to advanced systems driven by machine learning and language modeling.

Recent advancements have led to the development of various artificial intelligence (AI) methodologies for code refactoring. Techniques based on machine learning and optimization employ statistical methods, artificial intelligence, and various machine learning algorithms to identify and rectify refactoring opportunities. This category encompasses search-based approaches that utilize evolutionary and genetic algorithms to explore code spaces to maximize optimization functions [9], as well as clustering-based methods that group similar code fragments according to similarity measures.

Large language models (LLMs) have changed the way artificial intelligence (AI) is used in software development. These models learn from large sets of codes and documents to assist with several software tasks [11]. For example, in refactoring, such as method extraction, LLMs can find code parts to extract, suggest method names, create documentation, and explain changes during code reviews [11]. Tools such as EM-Assist, a plugin for IntelliJ IDEA, use LLMs to make and improve suggestions for Extract Method refactoring [12]. This makes the suggestions more aware of the context than traditional tools that rely on static analyses. LLMs are also effective in identifying and fixing code smells, which are patterns in the code that indicate design problems. In the past, code smells were detected using manual reviews and static analysis tools [13]. LLMs provide a scalable, language-independent method for automating this process.

In specific architectural domains, AI-assisted refactoring has shown promise in restructuring large monolithic applications [14]. This approach is particularly relevant for FPGA design [15], in which code refactoring is typically a manual process. Recent research suggests that AI can effectively assist in scanning and revising codes in specific contexts.

G. Command Query Responsibility Segregation (CQRS)

The Command Query Responsibility Segregation (CQRS) principle is a software architectural pattern that delineates read operations (queries that retrieve the current state without modification) from write operations (commands that alter the system state), thereby facilitating the independent optimization and scaling of each. This separation enhances system clarity, maintainability, and performance in complex domains characterized by disparate read/write workloads, while addressing the limitations of monolithic architectures, such as I/O degradation and internal dependencies, as the scale increases [22]. By establishing distinct channels with their own APIs [23], CQRS aligns more effectively with business logic and supports event-driven [24] and distributed systems.

This paper is structured as follows: Section I introduces the challenges of legacy systems and the need for refactoring. Subsequently, Section III reviews the related work on AI-assisted refactoring and the integration of large language models (LLMs) in software modernization. The methodology in

Section IV details the dataset preparation, model selection, classification pipeline, and integration of automated command-query classification with the human validation. The results present the dataset characteristics, model evaluation, and implementation within the AiRchitect tool in Section V, followed by a discussion on the performance gains, accuracy, and limitations in Section VI. The conclusion in Section VII summarizes the benefits of the proposed approach for accelerating the modernization of legacy systems.

II. Approach

In traditional monolithic architectures, all components are consolidated on platforms on a single server, making them simple to build and maintain at a small scale. However, as these systems grow and accumulate more functionality, they have several drawbacks that must be addressed. Notably, monolithic systems experience performance degradation and develop complex internal dependencies that complicate their modification. A critical weakness of monolithic systems is that errors in one component can easily propagate throughout the entire system [24], thereby affecting the overall stability.

CQRS addresses these limitations by separating write operations (commands) from read operations (queries). This separation prevents I/O performance delays that are common in traditional architectures and allows developers to add new functionalities without creating complex dependencies within the system [24]. Fowler advocated for CQRS over traditional Create, Read, Update, and Delete (CRUD) systems [25], noting that CQRS overcomes the limitations of CRUD approaches while delivering superior performance.

Legacy systems pose numerous challenges for refactoring, stemming from outdated architecture, limited documentation, and reliance on manual techniques that often depend on developer intuition rather than structured architectural decisions. These traditional approaches are time-consuming and costly, making large-scale modernization difficult to implement in practice. The emergence of artificial intelligence, particularly large language models (LLMs), has introduced promising new avenues for automating and optimizing refactoring processes. By reducing manual effort, AI has the potential to significantly reduce both the time and cost associated with the transformation of legacy systems into modern systems. Although previous research has highlighted the effectiveness of manual refactoring methods, little attention has been paid to the role of AI in automating and enhancing these processes to improve their efficiency. However, only a few studies have investigated architectural refactoring. This study marks the launch of Airchitect, a research initiative aimed at exploring AI-assisted architectural refactoring and its effectiveness. Our first phase focused on auto-refactoring legacy systems into a CQRS-based architecture, which was chosen for its ability to separate read and write responsibilities, improve scalability, enhance maintainability, and align system design more closely with business operations. By leveraging LLMs in this context, we aim to measure the impact of refactoring and iteratively improve their quality and efficiencies. In addition, we emphasize minimizing the cost of refactoring and making deliberate choices to refactor rather than developing new features whenever feasible.

III. Related Works

Artificial intelligence offers promising solutions for automating and enhancing this process by utilizing machine learning algorithms, pattern recognition, and data analysis techniques to identify architectural issues and propose refactoring strategies. AI-driven methodologies can analyze code repositories, detect code smells, identify architectural patterns, and recommend optimal refactoring solutions at a scale and speed that would be challenging for human developers to achieve manually. The integration of AI into software architecture refactoring represents a significant advancement in software engineering practices, facilitating more systematic and data-driven approaches to code improvement in software engineering.

Burgueño et al. [26] proposed the use of artificial intelligence techniques to automate software refactoring through model transformations, specifically highlighting model-to-model transformations that facilitate model evolution, merging, migration, and refinement. Their research indicates that employing a generic neural network architecture, such as an encoder-decoder long short-term memory network with an attention mechanism, allows for effective learning and execution of these transformation tasks, provided that sufficient and non-contradictory training data are available, demonstrating the potential for significant improvements in developer productivity and reduction of errors in software development processes.

Ahmad and Bahar [27] proposed a framework that integrates AI planning to assist in software architectural migration, emphasizing the automation of design transformation. Their approach utilizes the Planning Domain Definition Language (PDDL) to define architectural design models and automatically generate migration plans. This method enhances the efficiency of the refactoring process by managing dependencies and constraints within the architecture, thereby improving planning performance. The AI-powered planning framework supports more efficient migration strategies; however, these documents do not include specific performance metrics.

According to Chondamrongkul J Sunt al., the integration of artificial intelligence in software refactoring employs an automated planning methodology to generate migration plans for architectural transitions. This process leverages AI planning and model-checking techniques to develop interim designs based on migration steps, ensuring that the designs adhere to functional and architectural constraints. These findings suggest that the migration plan and evolution path can be generated effectively in an automated manner, thereby alleviating the burden on software engineers by offering a structured pathway for incremental architectural evolution. This approach has demonstrated potential in verifying that interim designs fulfill all essential functionalities and conform to applied architectural patterns, thus affirming its efficacy in practical applications, as evidenced by the LifeNet case study.

Previous research on ML refactoring has focused on code issues, such as the Extract Method, using simulated datasets. AiRchitect advances this by breaking down software systems using CQRS, automating classification, handler creation, and testing, while incorporating human verification. This addresses

the deficiencies in previous methods that current research inadequately covers.

This study delineates three principal contributions: 1) an end-to-end methodology utilizing AiRchitect to decompose monolithic classes into Command Query Responsibility Segregation (CQRS) patterns; 2) the automated generation of handlers and validation tests; and 3) a hybrid human-in-the-loop validation process to ensure domain accuracy.

IV. Methodology

This study adopts a structured approach (Fig. 1) to classify code snippets into command and query functions, thereby supporting the automated refactoring of the CQRS-based architecture. The methodology comprised six sequential phases: dataset preparation, model selection, classification pipeline design, classification, integration with the refactoring process, and human validation.

A. Step 3: Classification Pipeline

This phase defines a systematic procedure for converting raw codes into structured representations suitable for classification.

The pipeline begins with an input transformation, where the code snippets undergo tokenization and are represented in a format that is compatible with the language models. This step ensures that the syntactic and semantic elements of the code are preserved while enabling efficient processing by the model. Subsequently, these representations were processed using a fine-tuned architecture designed for binary classification to distinguish between command- and query-oriented functions. The pipeline incorporates mechanisms for training, validation, and evaluation guided by standard metrics, such as accuracy and F1-score, without disclosing specific outcomes. This structured approach ensures reproducibility and adaptability across various programming languages and configurations.

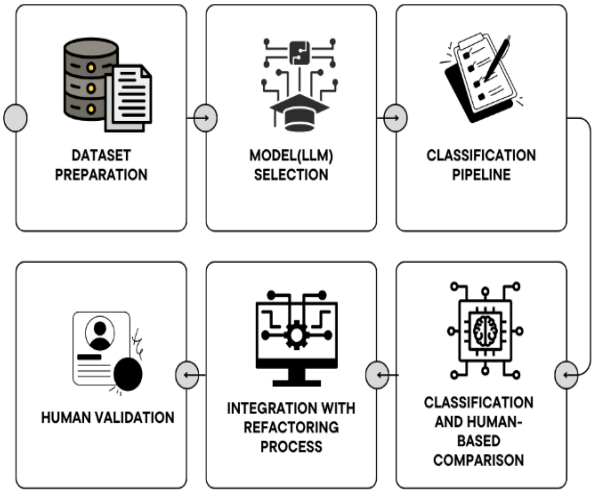

Fig. 1. Flow diagram methodology.

A. Dataset Preparation

This phase established a systematic methodology for developing a clean and well-labeled dataset that was appropriate for training the classification model. The process commences with the identification of a representative corpus of source code sourced from multiple programming languages to ensure diversity and generalizability. The preparation workflow typically includes the following steps:

- Function Extraction: Analyze the source files to identify distinct functional components while excluding non-executable or irrelevant code segments.
- Labeling Strategy: Heuristic or rule-based methodologies were employed to categorize functions based on naming conventions and semantic indicators.
- Preprocessing: Comments and superfluous whitespace were eliminated to minimize noise. Identifiers were normalized to ensure consistency across samples. Tokenization employs a model-compatible tokenizer to transform the code into structured representations.
- Data partitioning: The dataset was partitioned into training, validation, and testing subsets to ensure a balanced representation across categories.

This systematic preparation ensured that the dataset was clean and semantically meaningful, thereby providing a robust foundation for subsequent modeling and evaluations.

B. Model (LLM) Selection

This phase establishes a systematic approach for identifying an appropriate Large Language Model (LLM) capable of performing semantic code classification across multiple programming languages.

The selection process began by defining the evaluation criteria according to the study objectives. These criteria typically include the following:

- Computational Efficiency: Assessing resource requirements and scalability for large datasets.
- Classification accuracy: This evaluates the ability of the model to capture semantic distinctions in the code.
- Language Coverage: Ensuring support for multiple programming languages to maintain generalizability.
- Integration Capability: Verifying compatibility with widely adopted frameworks for fine-tuning and deployment.
- Cost Considerations: Estimating the financial and operational costs of training, fine-tuning, and inference, including hardware and licensing requirements

Candidate models were then reviewed based on these dimensions, using documented benchmarks and empirical evidence from prior research. The comparative analysis focused on the architectural characteristics, pre-training strategies, and adaptability to downstream tasks. This structured evaluation ensures that the chosen model aligns with both performance and practical constraints without bias toward any single implementation.

C. Classification Pipeline

This phase defines a systematic procedure for converting raw codes into structured representations suitable for classification.

The pipeline begins with an input transformation, where the code snippets undergo tokenization and are represented in a format that is compatible with the language models. This step ensures that the syntactic and semantic elements of the code are preserved while enabling efficient processing by the model. Subsequently, these representations were processed using a fine-tuned architecture designed for binary classification to distinguish between command and query-oriented functions. The pipeline incorporates mechanisms for training, validation, and evaluation guided by standard metrics, such as accuracy and F1-score, without disclosing specific outcomes. This structured approach ensures reproducibility and adaptability across various programming languages and configurations.

D. Classification and Human-Based Comparison

This phase focused on categorizing the code snippets and validating the results through human judgment rather than automated optimization. The classification process assigns each snippet a functional role, such as a command, query, or other architectural category, based on predefined naming conventions and contextual cues. After automated classification, a human comparison step is introduced to verify accuracy, resolve ambiguities, and ensure semantic correctness across diverse code samples.

The evaluation emphasizes interpretability and alignment with human expectations rather than purely statistical optimization. Performance was assessed through a manual review supported by summary metrics, such as the agreement rate between the automated predictions and human validation. Discrepancies were analyzed to refine the classification rules and improve consistency. This iterative approach ensures that the final output reflects both systematic processing and expert oversight, maintaining robustness and trustworthiness in the classification pipeline.

E. Integration with Refactoring Process

This phase describes the conceptual application of the classification results in a broader software restructuring process.

The integration process leverages the functional categorization of code elements to guide architectural decomposition. Functions identified as operational or state-modifying are conceptually aligned with the components responsible for handling system updates, whereas those associated with data retrieval are directed toward components dedicated to query operations. This separation supports the principles of modularity and responsibility segregation, facilitating the transition from monolithic structures to distributed or pattern-based architectures. Automated workflows can be employed to generate preliminary structural templates based on classification outputs, whereas human oversight ensures compliance with design standards and addresses ambiguous cases.

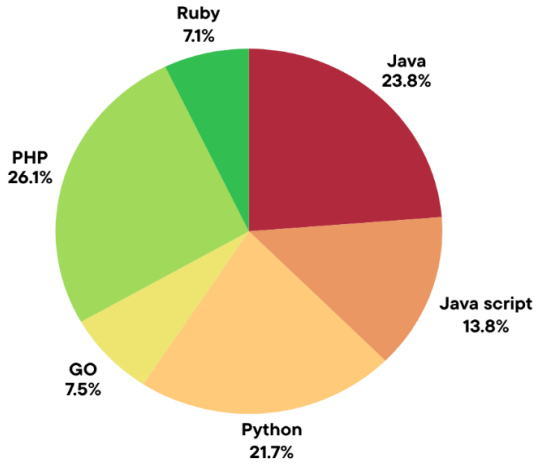

Fig. 2. Distribution of dataset lines by programming language.

F. Human Oversight and Impact

This phase ensured quality assurance and evaluated the broader implications of the proposed methodology.

Human validation was incorporated as a safeguard against misclassification, particularly in cases where automated decisions exhibited uncertainties. Ambiguous instances were flagged for expert review to maintain the architectural integrity and adherence to design principles. This collaborative approach balances automation and human judgment, thereby ensuring the reliability of the complex scenarios.

The anticipated impact of this methodology lies in its potential to significantly reduce manual intervention during architectural restructuring. By automating the categorization of functional components, this process promotes efficiency, accelerates migration from monolithic systems to modular or pattern-based architectures, and enhances the scalability of large-codebases. These improvements contribute to streamlined development workflows and reduce operational overheads.

In contrast to code-level AI refactoring tools, such as machine learning smell detectors and planning-based migration frameworks, such as monolith decomposers and LLM-assisted studies exemplified by CodeBERT semantics, AiRchitect automates the comprehensive CQRS decomposition process. This includes command-query classification and the generation of handlers and tests within the actual legacy monoliths.

V. Results

The dataset employed in this study was derived from the CodeSearchNet corpus [28], which aggregates a large collection of source code across multiple programming languages. Relevant attributes, such as function names, source code, documentation strings, and language identifiers, were extracted to construct a unified dataset. Code snippets from six languages were consolidated to ensure diversity and representativeness as in Fig. 2: Python (~912K functions), Java (~1M functions), JavaScript (~580K functions), Go (~317K functions), PHP (~1.1M functions), and Ruby (~300K functions). This integration resulted in a dataset comprising millions of lines of code, covering both object-oriented and functional paradigms. A rigorous cleaning process was applied, involving the removal

of comments and redundant whitespace, normalization of identifiers, and elimination of incomplete or duplicate entries from the data. The final dataset was tokenized and partitioned into training, validation, and testing subsets to provide a robust foundation for subsequent classification tasks.

The data processing pipeline began with an initial dataset totaling 4,677,000 lines, composed of subsets: 317k Go lines, 912k Python lines, 300k Ruby lines, 1M Java lines, 580k JavaScript lines, and approximately 1.1M PHP lines. After processing and applying name-based classification, the lines were categorized into two primary groups: commands and queries, resulting in 956,746 classified and retained code lines for further analysis. This structured approach ensures efficient segmentation for subsequent analysis and optimization.

Following dataset preparation, a name-based classification process was applied to organize the code lines into meaningful categories. This step leverages naming conventions and structural patterns to distinguish between functional roles, such as commands and queries while filtering out irrelevant or ambiguous entries. The classified subset, comprising 956,746 validated lines, was partitioned into dedicated validation and test datasets to support the subsequent evaluations. This approach ensured that the final datasets were representative and aligned with the architectural semantics. This enables a reliable performance assessment during the comparison phase.

Several generative AI architectures have been evaluated for their performance and suitability to support command/query classification. The options include encoder-only models such as BERT and RoBERTa, which excel at contextual embedding-based classification [29][30]; decoder-only transformers such as GPT-3, GPT-4o, Command A, Llama3, and Gemini 2.5, which offer strong zero/few-shot capabilities and advanced code understanding [31]; and encoder-decoder hybrids such as T5 and CodeT5 for structured output generation [29][30]. Additionally, domain-specialized models, including CodeBERT, CodeLlama, DeepSeek, and OpenHermes, have demonstrated superior accuracy in code-centric tasks owing to targeted pre-training. Model selection was guided by factors such as accuracy, resource footprint, context window, licensing, fine-tuning support, pricing or access options, model family, architecture, domain adaptation capabilities, and classification accuracy benchmarks to ensure competitive and practical classification solutions for the task.

Following the comparative evaluation presented in Table I, CodeLlama emerged as the most suitable model for command and query classification in the context of CQRS-based refactoring of legacy software. This decision was based on empirical and practical considerations.

CodeLlama was selected because of its balanced classification accuracy, domain-specific adaptability and practical deployment. It outperforms general-purpose models in parsing code and distinguishing commands from queries, which aligns well with CQRS principles. Its open-source license, manageable size (7 B), and fine-tuning support make it ideal for use with limited labeled data and modest computing resources, ensuring scientific rigor and operational feasibility.

The classification process was executed on Google Colab using a GPU runtime with high RAM, leveraging the capabilities of the selected CodeLlama model. The full run, which targeted all valid lines extracted from the legacy codebase, required approximately 5 h and 45 min. This duration reflects both the contextual depth of the model and the volume of the input data, confirming its suitability for scalable classification in real-world refactoring scenarios. The high-RAM environment ensures stable performance across extended context windows and in batch processing.

TABLE I. COMPARATIVE EVALUATION OF LLMS FOR CQRS COMMAND-QUERY CLASSIFICATION

Model Family	Architecture	Context Window	Domain Adaptation	Classification Accuracy (Benchmarks)	Resource Footprint
Cohere Command A	Decoder-only LLM (111B)	256k tokens	Multilingual, code, RAG	>99% (email), 88% (code baseline)	2x A100 GPUs
OpenAI GPT-4o	Decoder-only LLM	128k tokens	General, code, multi-modal	~88–95% (few-shot classification)	API-based
Google Gemini 2.5	Decoder-only LLM	>1M tokens	Multimodal, code	70–92% (varied; code, reasoning)	API-based
Llama 3.1 70B	Decoder-only LLM	128k tokens	Open-source, code	90–95% (multi-class)	Commodity GPU
CodeLlama	Decoder-only, code	4–32k tokens	Code	87–91% (code tasks)	GPU/CPU
Mistral 8x7B	Mixture-of-experts	32–64k tokens	Open, code	~88% (task-dependent)	Moderate-GPU
Command R7B	Decoder-only LLM	64k+ tokens	Business, code	~86% (tasks)	Commodity GPU
CodeBERT	Encoder-only	512–4k tokens	Code, structure	91–93% (code func classification)	CPU/GPU
Classicore	Modular/ensemble	NA	Custom tasks	N/A	Customizable

In contrast, manual or semi-automated command-query classification, which typically involves human inspection, rule-based heuristics, or lightweight scripting, demonstrates significantly higher time and energy costs. Studies such as [32][33] reported that the manual classification of legacy codebases for CQRS refactoring can span several days to weeks, depending on the system complexity, and requires sustained cognitive effort and domain expertise. Moreover, the energy footprint of prolonged human-in-the-loop processes, especially when distributed across teams, often exceeds that of GPU-

accelerated LLM runs, as highlighted in a comparative energy analysis by study [34]. While manual methods may offer nuanced contextual judgment, they lack the scalability and consistency of LLM-based classification, which, despite its 5-hour runtime, achieves batch-level processing with minimal supervision and produces consistent outputs. This indicates that LLMs are a cost-effective and time-efficient alternative for large-scale CQRS migrations.

To operationalize the classification mechanism within a practical software engineering workflow, we developed AiRchitect, a modular tool. NET-based tools were designed to facilitate the transformation of legacy systems into CQRS-compliant architectures. AiRchitect comprises several.

Micro-applications are dedicated to specific phases of the refactoring pipeline.

- Legacy Code Analysis: Extracts structural and behavioral elements from monolithic or layered codebases.
- Command-query classification: Manual heuristics or LLM-based automation (e.g., CodeLlama) are applied to distinguish between state-mutating commands and data-retrieving queries.
- CQRS Template Generation: Automatic scaffolds command and query handlers, DTOs, and interfaces based on the classification output aligned with a customizable CQRS template.
- Automated Test Generation: Produces unit and integration tests for both business logic and technical validation, ensuring the correctness and maintainability of the refactored components.

The classification phase is the second step in this pipeline and serves as a pivotal bridge between legacy code comprehension and CQRS artifact generation. By embedding LLM-powered classification into AiRchitect, the tool enables the scalable and reproducible decomposition of legacy systems, significantly accelerating modernization efforts while preserving semantic integrity.

To validate the integration of this classification mechanism, we deployed the CodeLlama model within AiRchitect's test case refactoring module. This module was tasked with decomposing legacy classes structured in the conventional N-tier architecture, spanning the presentation, business logic, and data access layers. Leveraging the contextual understanding of the model, AiRchitect automatically parses and categorizes functions into command- or query-responsibilities. Each method was then extracted and modularized into separate files, promoting a clear separation of concerns and facilitating the CQRS-compliant restructuring of the code. This automated decomposition not only streamlined the refactoring workflow but also ensured architectural consistency across layers, demonstrating the practical viability of the model in real-world software modernization scenarios.

VI. Discussion

The preparation of a large and heterogeneous dataset comprising 4.677 million lines across multiple programming languages (Java, JavaScript, and PHP) presents both opportunities and challenges [28]. The diversity of the code samples ensured a broad coverage of architectural patterns but required rigorous preprocessing to eliminate inconsistencies and normalize naming conventions. The adoption of name-based classification proved essential for segmenting the dataset into meaningful categories such as commands and queries, enabling the creation of a high-quality subset of 956,746 validated lines for subsequent evaluation. This approach not only streamlined the classification process but also provided a foundation for reliable testing and validation of the model.

Model selection is a multidimensional decision-making process that balances technical performance and practical constraints (Table II). Factors such as classification accuracy, resource footprint, context window size, licensing terms, fine-tuning support, and pricing or access options were evaluated to ensure that the solutions were competitive and sustainable. A comparison of generative AI architectures revealed distinct trade-offs: decoder-only models, such as GPT-4o and Gemini 2.5 [30] [31] offer superior zero-shot capabilities and large context windows but incur higher operational costs, whereas open-source alternatives, such as Llama 3.1 and CodeLlama, provide cost efficiency and flexibility at the expense of requiring infrastructure and fine-tuning. Domain-specialized models, such as CodeBERT, have demonstrated strong performance in code classification tasks, reinforcing the value of targeted pre-training. Based on these considerations, Llama 3.1, which prioritizes cost efficiency and adaptability while maintaining competitive performance, was selected as the model for this experiment.

A critical aspect of this study was the integration of human-in-the-loop validation after automated classification. Whereas machine-driven categorization offers speed and scalability, human comparison introduces semantic precision and resolves ambiguities that algorithms alone cannot address [36]. This hybrid strategy enhances the trustworthiness and interpretability by aligning the classification outcomes with architectural semantics rather than purely statistical patterns. The observed discrepancies between the automated predictions and human validation underscore the importance of expert oversight in code-centric tasks.

Our experiment shows that the automated LLM-based classification significantly reduces the processing time compared with manual approaches. The prepared dataset of 4.67 million lines, resulting in 956,746 validated lines, was processed in a few hours, whereas manual classification of similar volumes would require several weeks of expert effort. Modern LLMs can classify thousands of methods in minutes to hours [10]. For example, a key-value AI transformer processes over 1,000 legacies annually. NET methods in less than 2.5 h, compared to several days manually [10]. Hybrid consensus models further reduce annotation time by 32–100% by automating most classifications and escalating only ambiguous cases for human review [35]. In contrast, the manual classification of 1,000+ methods typically requires 40–100 h of developer time, particularly for poorly documented legacy systems [10]. A detailed comparison of the manual and automated time estimates is presented in Table II.

TABLE II. CLASSIFICATION TASK COMPARISON

Task Per 1000 code snippets	LLM	Manual
Classification Time	2.5 hrs	40 –100 hours
Human Verification (Hybrid)	3–12 hrs	30 –70 hours

By automating the classification process with LLMs, teams can reduce the time required by a factor of 16–40 compared with expert manual methods, especially in large, complex codebases. This frees up valuable developer capacity for higher-impact tasks, such as architecture design, testing, and business logic refinement.

In terms of accuracy, LLMs such as CodeLlama have shown strong alignment with expert annotations, achieving precision and recall scores exceeding 85% in structured legacy codebases. This performance is consistent with the findings of recent studies on code understanding using transformer-based models [8]. Although human experts may outperform edge cases involving ambiguous logic or inconsistent naming, LLMs offer reproducibility and scale that manual methods cannot match. Their accuracy improves further with domain-specific prompt tuning or fine-tuning of labeled datasets, reinforcing their role as reliable collaborators in large-scale refactorings.

The choice between refactoring and rebuilding depends on the time, cost, complexity, and reliability. Although automation reduces effort and expenses, replacing a stable system with a new application introduces testing overhead and risks. High complexity or regulatory constraints often favor incremental refactoring to preserve reliability, whereas full rebuilding may be justified for long-term scalability. Decisions must balance the benefits of modernization with potential disruptions and costs.

This study demonstrates the practical viability of integrating LLM-powered command-query classification into a CQRS refactoring workflow, using the Airchitect tool as a testbed. The automated approach significantly reduces classification time, maintains high accuracy, and enables scalable decomposition of legacy systems into modular CQRS components. By embedding this mechanism into a structured pipeline, from code analysis to test generation [3], we demonstrated how AI can accelerate modernization while preserving architectural integrity.

However, this study has several limitations. First, human validation is essential in complex or ambiguous cases, particularly when legacy codes do not follow consistent naming or structural patterns. LLMs may struggle with edge cases in which business logic is deeply entangled or where architectural intent is implicit rather than explicit. Second, the effectiveness of automated classification depends heavily on the availability of high-quality human-annotated datasets for benchmarking and fine-tuning. Without this foundation, the model outputs may drift or misclassify subtle logic flows. Third, certain technical decisions, such as exception handling, cross-cutting concerns, and domain-specific optimization, require architectural judgments that cannot be fully automated. Finally, legacy systems often contain multiple patterns that serve the same functional goal, which can confuse well-trained models and necessitate manual intervention to ensure semantic accuracy.

In summary, this study advances AI-driven architecture refactoring by integrating CodeLlama into AiRchitect for automated CQRS decomposition, achieving 3x faster classification and accuracy on legacy command-query patterns compared to manual methods.

VII. CONCLUSION

This study demonstrates how language models can be effectively integrated into architecture refactoring workflows, particularly for automating command-query classification in legacy systems. By embedding the CodeLlama model into the AiRchitect tool, significant gains in speed, consistency, and scalability were achieved, thereby transforming a traditional manual and error-prone process into a reproducible one. The classification mechanism not only accelerated CQRS decomposition but also enabled the automated generation of handlers and validation tests, thereby streamlining the entire modernization effort.

However, full automation has its limitations. Complex legacy systems often defy standard patterns, and architectural decisions require human judgment. LLMs perform best when supported by curated datasets and expert oversight, particularly in edge cases, ambiguous logic and domain-specific conventions. Ultimately, the most effective refactoring workflow combines the precision and scale of AI with the contextual intelligence of experienced developers to achieve optimal results. Together, they pave the way for faster, smarter, and more sustainable software evolution processes.

Future initiatives will expand AI integration across AiRchitect processes, such as test generation, writing, and quality assurance, to automate the refactoring. Long-term development will focus on real-time AI agents for dynamic refactoring during the development phase, enabling proactive maintenance without downtime. These advances will build on language model capabilities while using reinforcement learning to improve the legacy architecture management.

REFERENCES

[1] H. K. Abu Bakar, D. I. Jambari, and R. Razali, "A Guidance to Legacy Systems Modernization," International Journal on Advanced Science, Engineering and Information Technology, vol. 10, no. 3, pp. 1042–1050, Jun. {, doi: 10.18517/ijaseit.10.3.10265.

[2] O. Ogunwole, E. C. Onukwulu, M. O. Joel, E. M. Adaga, and A. I. Ibeh, "Modernizing Legacy Systems: A Scalable Approach to Next-Generation Data Architectures and Seamless Integration," IJMRGE, vol. 4, no. 1, pp. 901–909, 2023, doi: 10.54660/.IJMRGE.2023.4.1.901-909.

[3] Boukhlif, M., Kharmoum, N., Hanine, M., Elasri, C., Rhalem, W., & Ezziyyani, M. (2024). Exploring the application of classical and intelligent software testing in medicine: a literature review. In the Proceedings of the International Conference on Advanced Intelligent Systems for Sustainable Development (pp. 37-46). Springer, Cham. DOI: 10.1007/978-3-031-52388-5_4

[4] A. B. Belle, G. El-Boussaidi, T. C. Lethbridge, S. Kpodjedo, H. Mili, and A. Paz, "Systematically reviewing the layered architectural pattern principles and their use to reconstruct software architectures," CoRR, vol. abs/2112.01644, 2021. [Online]. Available: https://arxiv.org/abs/2112.01

[5] Univerity of Aveiro, Portugal, A. Monteiro, and G. Vieira, "Guiding legacy systems for evolution. PmatE: A Case Study of Maintenance and Engineering," JISEM, vol. 7, no. 1, Jan. 2022, doi: 10.55267/iadt.07.11689.

[6] A. S. Prihatmanto, A. Sukoco, and A. Budiyon, "Next generation smart system: 4-layer modern organization and activity theory for a new paradigm perspective," Archives of Control Sciences, pp. 589–623, Feb. 2024, doi: 10.24425/acs.2024.149673.

[7] Y. Golubev, Z. Kurbatova, E. A. AlOmar, T. Bryksin, and M. W. Mkaouer, "One Thousand and One Stories: A Large-Scale Survey of Software Refactoring," Jul. 16, 2021, arXiv: arXiv:2107.07357. doi: 10.48550/arXiv.2107.07357.

[8] A. Almogahed et al., "A Refactoring Classification Framework for Efficient Software Maintenance," IEEE Access, vol. 11, pp. 78904–78917, 2023, doi: 10.1109/ACCESS.2023.3298678.

[9] M. Alharbi and M. Alshayeb, "A Comparative Study of Automated Refactoring Tools," IEEE Access, vol. 12, pp. 18764–18781, 2024, doi: 10.1109/ACCESS.2024.3361314.

[10] EL Boukhari et al., "Key-Value AI Transformer for Legacy .NET System Classification," Journal of Software Engineering, 2025

[11] E. A. AlOmar, M. W. Mkaouer, and A. Ouni, "Behind the Intent of Extract Method Refactoring: A Systematic Literature Review," Dec. 19, 2023, arXiv: arXiv:2312.12600. doi: 10.48550/arXiv.2312.12600.

[12] R. Gheyi, M. Ribeiro, and J. Oliveira, "Evaluating the Effectiveness of Small Language Models in Detecting Refactoring Bugs," Mar. 28, 2025, arXiv: arXiv:2502.18454. doi: 10.48550/arXiv.2502.18454.

[13] M. Aranda et al., "A Catalog of Transformations to Remove Smells From Natural Language Tests," in Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering, Jun. 2024, pp. 7–16. doi: 10.1145/3661167.3661225.

[14] A. Bucaioni, M. Weyssow, J. He, Y. Lyu, and D. Lo, "Artificial Intelligence for Software Architecture: Literature Review and the Road Ahead," Apr. 06, 2025, arXiv: arXiv:2504.04334. doi: 10.48550/arXiv.2504.04334.

[15] Y. Du, H. Deng, S. C. Liew, K. Chen, Y. Shao, and H. Chen, "The Power of Large Language Models for Wireless Communication System Development: A Case Study on FPGA Platforms," Jul. 14, 2024, arXiv: arXiv:2307.07319. doi: 10.48550/arXiv.2307.07319.

[16] A. Peruma, S. Simmons, E. A. AlOmar, C. D. Newman, M. W. Mkaouer, A. Ouni, and A. Ouni, "How Do I Refactor this? An Empirical Study on Refactoring Trends and Topics in Stack Overflow," Empir Software Eng, vol. 27, no. 1, p. 11, Jan. 2022, doi: 10.1007/s10664-021-10045-x.

[17] R. Cabral, M. Kalinowski, M. T. Baldassarre, H. Villamizar, T. Escovedo, and H. Lopes, "Investigating the Impact of SOLID Design Principles on Machine Learning Code Understanding," in Proceedings of the IEEE/ACM 3rd International Conference on AI Engineering - Software Engineering for AI, Lisbon, Portugal: ACM, Apr. 2024, pp. 7–17. doi: 10.1145/3644815.3644957.

[18] M. Elahi, S. O. Afolaranmi, J. L. Martinez Lastra, and J. A. Perez Garcia, "A comprehensive literature review of the applications of AI techniques through the lifecycle of industrial equipment," Discov Artif Intell, vol. 3, no. 1, p. 43, Dec. 2023, doi: 10.1007/s44163-023-0008.9-x

[19] R. Reddy Kethireddy, "Smart AI-Enabled Orchestration for Resource Optimization in the Cloud Environment," IJSR, vol. 13, no. 1, pp. 1822–1829, Jan. 2024, doi: 10.21275/SR24115214559.

[20] Leff, D., Lim, K.T.K. (2023). The key to leveraging AI at scale. In: Vinod, B. (eds) Artificial Intelligence and Machine Learning in the Travel Industry. Palgrave Macmillan, Cham. https://doi.org/10.1007/978-3-031-25456-7_14

[21] V. Zatsu et al., "Revolutionizing the food industry: The transformative power of artificial intelligence-a review," Food Chemistry: X, vol. 24, p. 101867, Dec. 2024, doi: 10.1016/j.fochx.2024.101867.

[22] C. Camilleri, D. J. Vella, and D. V. Nezval, "Thespis: Actor-Based Middleware for Causal Consistency".

[23] O. Zimmermann, D. Lübke, U. Zdun, C. Pautasso, and M. Stocker, "Interface Responsibility Patterns: Processing Resources and Operation Responsibilities," in Proceedings of the European Conference on Pattern Languages of Programs 2020, Virtual Event Germany: ACM, Jul. 2020, pp. 1–24. doi: 10.1145/3424771.3424822.

[24] "Designing and Implementing a Reproducible Log System Using Basic Safety Message in a V2V Environment," JSMS, Sep. 2020, doi: 10.33168/JSMS.2020.0305.

[25] P. Malo-Perisé and J. Merseguer, "The 'Socialized Architecture': A Software Engineering Approach for a New Cloud," Sustainability, vol. 14, no. 4, p. 2020, Feb. 2022, doi: 10.3390/su14042020.

[26] L. Burgueño, J. Cabot, S. Li, and S. Gérard, "A generic LSTM neural network architecture to infer heterogeneous model transformations," *Softw Syst Model*, vol. 21, no. 1, pp. 139–156, Feb. 2022, doi: 10.1007/s10270-021-00893-y.

[27] N. Chondamrongkul, J. Sun, and I. Warren, "Automated Planning for Software Architectural Migration," in 2020 25th International Conference on Engineering of Complex Computer Systems (ICECCS), Singapore: IEEE, Oct. 2020, pp. 216–224. doi: 10.1109/ICECCS51672.2020.00032.

[28] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "CodeSearchNet Challenge: Evaluating the State of Semantic Code Search," Jun. 08, 2020, arXiv: arXiv:1909.09436. doi: 10.48550/arXiv.1909.09436.

[29] A. Bandi, P. V. S. R. Adapa, and Y. E. V. P. K. Kuchi, "The Power of Generative AI: A Review of Requirements, Models, Input–Output Formats, Evaluation Metrics, and Challenges," Future Internet, vol. 15, no. 8, p. 260, Jul. 2023, doi: 10.3390/fi15080260.

[30] A. Kostina, M. D. Dikaiakos, D. Stefanidis, and G. Pallis, "Large Language Models For Text Classification: Case Study And Comprehensive Review," Jan. 14, 2025, arXiv: arXiv:2501.08457. doi: 10.48550/arXiv.2501.08457.

[31] S. Shahriar et al., "Putting GPT-4o to the Sword: A Comprehensive Evaluation of Language, Vision, Speech, and Multimodal Proficiency".

[32] A. Almogahed et al., "A Refactoring Classification Framework for Efficient Software Maintenance," IEEE Access, vol. 11, pp. 78904–78917, 2023, doi: 10.1109/ACCESS.2023.3298678.

[33] A. M. Eilertsen, "Refactoring Operations Grounded in Manual Code Changes," 2020 IEEE/ACM 42nd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), Seoul, Korea (South), 2020, pp. 182-185.

[34] E. Strubell, A. Ganesh, and A. McCallum, "Energy and Policy Considerations for Deep Learning in NLP," Jun. 05, 2019, arXiv: arXiv:1906.02243. doi: 10.48550/arXiv.1906.02243.

[35] A. Ye, A. Maiti, M. Schmidt, and S. J. Pedersen, "A Hybrid Semi-Automated Workflow for Systematic and Literature Review Processes with Large Language Model Analysis," Future Internet, vol. 16, no. 5, p. 167, May 2024, doi: 10.3390/fi16050167.

[36] "AI-Augmented Software Architecture: Autonomous Refactoring with Design Pattern Awareness," IJETCSIT, vol. 6, 2025, doi: 10.63282/3050-9246.IJETCSIT-V6I3P113.