International Journal of
INTELLIGENT SYSTEMS AND APPLICATIONS IN ENGINEERING
ISSN:2147-6799 www.ijisae.org Original Research Paper

An AI-Augmented Framework for Refactoring Enterprise Monolithic Systems

Kishore Subramanya Hebbar*

Submitted: 01/06/2023 **Revised:** 10/07/2023 **Accepted:** 20/07/2023

Abstract: Many large organizations still depend on legacy monolithic systems that were built over many years and now hold deeply embedded business logic. Moving these systems to cloud-native microservices is widely desired, but the process is slow, risky, and heavily dependent on manual code understanding, which often leads to errors and rework. Current migration approaches either rely on rigid rule-based tools or expect full manual refactoring, leaving a gap in practical support for understanding complex dependencies and identifying safe service boundaries. The goal of this study is to address this gap by providing intelligent, decision-oriented assistance that helps engineers refactor legacy code while preserving existing business behavior. The proposed approach introduces an AI-augmented modular refactoring framework that combines static code analysis, dependency graph modeling, and machine learning-based pattern recognition. Instead of automatically rewriting code, the framework highlights logical decomposition points, detects refactoring candidates, and surfaces architectural risks. A human-in-the-loop workflow allows developers and architects to review, adjust, and validate recommendations before changes are applied, supporting incremental migration rather than disruptive rewrites. Evaluation on enterprise-scale legacy applications shows a reduction of refactoring effort by approximately 25 to 40 percent compared to fully manual approaches. The resulting microservices also exhibit improved modularity and fewer post-migration defects during validation. This framework can be applied to large enterprise modernization initiatives where reliability and domain integrity are critical. By combining human expertise with AI-assisted insight, the work demonstrates a practical and novel way to reduce risk and effort in legacy-to-microservice migration.

Keywords: *Legacy system modernization, Code refactoring, Monolithic architectures, Microservice migration, AI-assisted software engineering, Enterprise application evolution*

1. Introduction

Enterprise software systems that support critical business operations are often built as large monolithic applications that have evolved over many years. These systems typically embed complex business logic, domain-specific rules, and operational assumptions that are difficult to replace or rewrite. While such applications are often stable in production, their tightly coupled structure and accumulated technical debt make them increasingly hard to scale, maintain, and adapt to modern deployment environments [1]. As organizations seek improved scalability, fault isolation, and faster delivery cycles, migrating legacy monoliths to microservice-based architectures has become a common modernization goal [2]. Despite widespread interest in microservice migration, refactoring legacy systems remains a challenging and high-risk task. Existing migration efforts rely heavily on manual code analysis and refactoring, which requires deep domain knowledge and significant engineering effort. These manual approaches do not scale well for large enterprise codebases and are prone to oversight when dependencies are undocumented or implicit. At the same time, rule-based and automated migration tools often lack sufficient contextual understanding of business logic and architectural intent, leading to poor service decomposition or unintended behavioral changes [3]. This creates a gap between labor-intensive manual refactoring and impractical fully automated solutions. Recent advances in machine learning offer an opportunity to better support legacy modernization by assisting engineers in understanding and restructuring complex codebases. Large monolithic systems contain recurring dependency patterns, structural signals, and

International Business Machines, Atlanta, USA

**Corresponding author*

Email address: hebbar.kishore@gmail.com (Kishore Subramanya Hebbar)

architectural characteristics that are difficult to interpret through static analysis alone [4]. When used as an assistive technique rather than a replacement for human judgment, AI can help surface meaningful insights from these systems and support more informed refactoring decisions. Such an approach is particularly relevant in enterprise environments where correctness, reliability, and domain integrity are critical [5]. This paper proposes an AI-augmented approach to legacy code refactoring that supports the migration of enterprise monoliths to microservice architectures. The approach combines static code analysis with machine learning-based pattern recognition to identify potential service boundaries, refactoring candidates, and architectural risks. Rather than automatically rewriting code, the proposed framework generates structured recommendations that guide architects and developers through a human-in-the-loop workflow. The framework is modular and designed for incremental adoption, allowing teams to modernize systems gradually while preserving existing business behavior. The remainder of this paper presents related work, describes the proposed methodology and implementation, evaluates the approach on enterprise-scale systems, and discusses its practical implications.

2. Related Work

The migration of legacy monolithic systems to microservice architectures has been widely studied across software engineering and systems research. Prior work in this area can be broadly grouped into manual refactoring practices, rule-based and tool supported migration approaches, and early applications of artificial intelligence to software modernization. This section reviews these directions and highlights their limitations in the context of large enterprise systems.

2.1. Manual Refactoring and Decomposition Practices

Early and widely adopted approaches to monolith-to-microservice migration rely on manual analysis and refactoring performed by experienced architects and developers. These methods typically involve identifying bounded contexts, restructuring modules and extracting services based on domain knowledge and architectural principles. While such practices offer a high degree of control and can preserve business semantics when executed carefully, they are time-consuming and difficult to scale [6]. In large legacy codebases, undocumented dependencies and tightly coupled components make manual decomposition error-prone, often leading to repeated refactoring cycles and inconsistent service boundaries.

2.2. Rule-Based and Tool-Supported Migration Approaches

To reduce manual effort, several tools and methodologies have been proposed to support service extraction through static analysis, dependency metrics, and predefined architectural rules. These approaches analyze code structure, call graphs, or data access patterns to suggest candidate services or modules [7, 8]. Although such tools can process large codebases efficiently, they often rely on fixed heuristics that lack awareness of domain semantics and runtime behavior. As a result, the generated service boundaries may not align with business logic, leading to overly fine-grained services or architectures that are difficult to evolve. These limitations are particularly evident in enterprise systems with long development histories and heterogeneous design styles.

2.3. AI-Assisted Software Modernization

More recent work has explored the use of machine learning techniques to assist software engineering tasks such as code classification, dependency analysis, and architectural pattern detection [9, 10]. In the context of legacy modernization, these approaches aim to identify structural patterns and refactoring opportunities that are not easily captured by static rules alone. However, many existing efforts focus on narrow tasks or experimental settings and do not address the broader challenges of enterprise-scale migration. Additionally, fully automated refactoring remains impractical in environments where correctness, regulatory constraints, and domain-specific behavior are critical, limiting the applicability of end-to-end automation.

2.4. Research Gaps

Despite progress in legacy refactoring research, several gaps remain when these approaches are applied to enterprise-scale migration efforts. Existing manual methods lack scalability, while rule-based tools provide limited contextual understanding of complex business logic. AI-assisted techniques, although increasingly explored, often emphasize automation over practical decision support and are rarely integrated into incremental migration workflows [10, 11] There is a need for approaches that combine structural analysis with AI-assisted insight while keeping humans in control of refactoring decisions. In

particular, prior work offers limited guidance on how to support service boundary identification and refactoring planning in a way that balances analytical rigor, domain knowledge, and practical adoption. This paper addresses these gaps by introducing an AI-augmented refactoring framework designed to assist, rather than replace, engineers during enterprise monolith-to-microservice migration.

3. Methodology

This work is novel in its focus on using AI as a practical decision-support mechanism for legacy refactoring rather than as a fully automated migration solution. The proposed methodology combines static code analysis with machine learning–assisted pattern recognition to guide refactoring decisions in complex enterprise monoliths. Emphasis is placed on modularity, incremental adoption, and human validation to reduce migration risk while preserving existing system behavior.

3.1. Overview of the AI-Augmented Refactoring Approach

The methodology is designed to support engineers during the early and most error-prone phases of monolith-to-microservice migration: understanding system structure, identifying meaningful service boundaries and assessing refactoring risk. Rather than attempting to transform code automatically, the approach focuses on generating structured insights that help developers reason about decomposition decisions. At a high level, the process begins with static analysis of the legacy codebase to extract structural information such as module dependencies, call relationships, and shared data access patterns. This information is then transformed into intermediate representations, including dependency graphs and component interaction models, which serve as inputs to machine learning–based analysis. The role of machine learning in this context is not to replace architectural judgment, but to surface patterns and relationships that are difficult to identify through manual inspection alone. The output of the analysis consists of ranked refactoring recommendations that highlight potential service boundaries, tightly coupled components, and areas of architectural risk. These recommendations are intentionally advisory and are designed to be reviewed and refined by human experts. By positioning AI as an assistive layer rather than a prescriptive authority, the methodology aligns with enterprise constraints where correctness, stability, and domain knowledge are essential. As shown in Figure 1, the methodology integrates static analysis, AI-assisted pattern recognition, and human validation into a single, incremental refactoring workflow.

3.2. Static Analysis and Structural Modeling

Static analysis forms the foundation of the proposed methodology by providing a detailed view of the legacy system's internal structure. Source code artifacts such as classes, modules, interfaces, and database access layers are analyzed to extract dependency information without requiring runtime execution. This is particularly important for legacy enterprise systems where production-like runtime environments may be difficult to replicate. The analysis captures several types of relationships, including method invocations, shared data access, inheritance hierarchies, and configuration-level dependencies. These relationships are aggregated into a dependency graph that represents the system as a network of interacting components. Nodes in the graph correspond to logical units such as modules or packages, while edges represent coupling through calls or shared resources. While static analysis tools are commonly used in refactoring workflows, their raw output is often too granular to support architectural decision-making directly. To address this, the methodology applies structural aggregation techniques that group low-level elements into higher-level components based on usage patterns and cohesion metrics [12]. This abstraction step reduces noise and allows engineers to reason about the system at a level that is meaningful for service decomposition. The resulting structural model provides a stable and explainable basis for further analysis. Because the model is derived directly from source artifacts, it remains transparent and auditable, which is critical in enterprise environments where architectural decisions must be justified and reviewed.

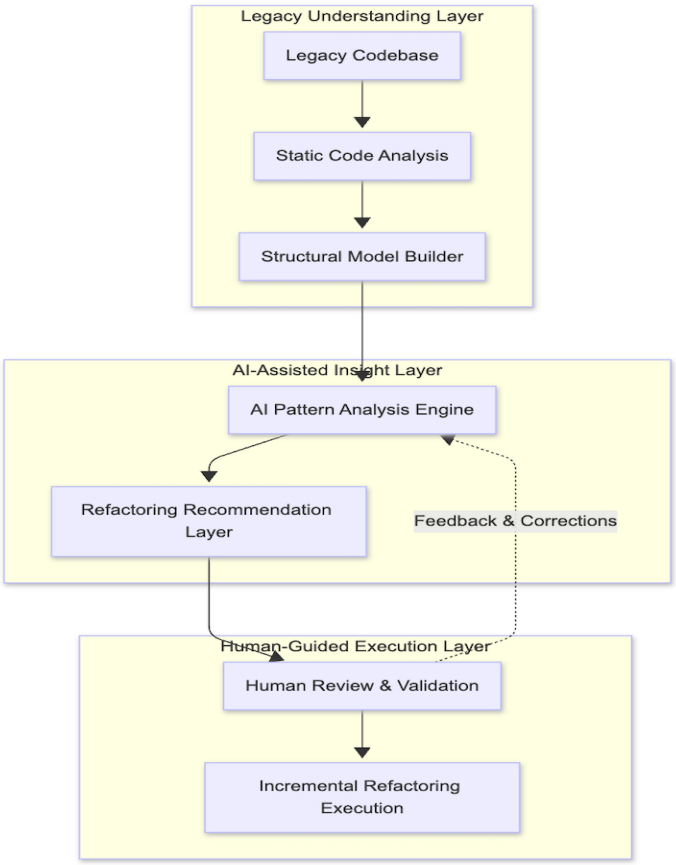

Figure 1: System architecture of the AI-augmented legacy refactoring framework.

3.3. Machine Learning–Assisted Pattern Identification

Building on the structural model, machine learning techniques are applied to identify recurring patterns and refactoring opportunities within the legacy system. Feature vectors are constructed from structural characteristics such as coupling strength, change frequency, shared data usage, and dependency directionality. These features capture both the static shape of the system and historical signals that reflect how components evolve over time. Clustering and classification techniques are used to identify groups of components that exhibit high internal cohesion and relatively low external coupling, making them suitable candidates for service extraction. The analysis also highlights anti-patterns such as overly central components or modules with excessive cross-cutting dependencies, which may require special handling during refactoring [13]. Importantly, the machine learning models are used to generate relative assessments rather than absolute decisions. Recommendations are expressed as ranked suggestions with associated confidence indicators, allowing engineers to prioritize areas for deeper inspection. This design choice reflects the reality that architectural decisions often involve trade-offs that cannot be fully captured by automated models. The focus remains on augmenting human understanding rather than automating transformation, which improves trust and practical adoption. By integrating machine learning at this stage, the methodology helps bridge the gap between low-level dependency data and high-level architectural reasoning while preserving human oversight. Algorithm 1 formalizes this AI-assisted pattern identification process and summarizes how structural features are transformed into ranked refactoring recommendations for human validation.

Algorithm 1 AI-Assisted Service Boundary Pattern Identification

Input: Legacy codebase artifacts, dependency graph

Output: Ranked refactoring recommendations with stability indicators

Phase 1: Structural Signal Extraction

Extract structural features from the dependency graph, including module dependencies, call frequencies, and shared data access paths.

Aggregate low-level code elements into candidate architectural components.

Phase 2: Metric Computation and Pattern Analysis

Compute coupling, cohesion, and shared resource metrics for each component group.

Apply pattern analysis to identify cohesive clusters suitable for service extraction.

Phase 3: Risk Detection and Stability Assessment

Identify refactoring risks such as excessive coupling, cyclic dependencies, and shared persistent state.

Estimate structural stability scores for candidate service boundaries.

Phase 4: Recommendation Generation and Human Validation

Rank candidate service boundaries based on stability and risk indicators.

Present ranked recommendations and explanatory signals for human review and validation.

3.4. Human-in-the-Loop Workflow and Incremental Adoption

A central design principle of the methodology is the inclusion of a human-in-the-loop workflow that keeps architects and developers actively involved in refactoring decisions. Rather than enforcing automated changes, the framework presents insights through visualizations and structured reports that explain why specific recommendations were generated [14]. This transparency allows users to validate assumptions, incorporate domain knowledge, and adjust boundaries before implementation. Figure 2 illustrates the human-in-the-loop feedback cycle and incremental adoption flow. The methodology supports incremental adoption by allowing teams to apply the analysis selectively to specific subsystems or domains. This is particularly valuable for large enterprise applications where full-scale migration is neither feasible nor desirable in a single phase. Teams can prioritize high-impact areas, validate outcomes, and gradually expand modernization efforts based on confidence and available resources. Feedback from human review is treated as a first-class input to the process. Adjustments made by engineers can be recorded and reused to refine future recommendations, enabling the framework to adapt to project-specific constraints over time. This feedback loop helps align analytical insights with real-world architectural intent. By combining AI-assisted analysis with human oversight and incremental execution, the methodology provides a balanced approach to legacy modernization. It supports practical migration scenarios where reliability and continuity are as important as architectural improvement, making it well suited for enterprise-scale refactoring initiatives. As illustrated in Listing 1, human reviewers can modify AI-generated service boundary recommendations while preserving traceability, rationale and architectural accountability.

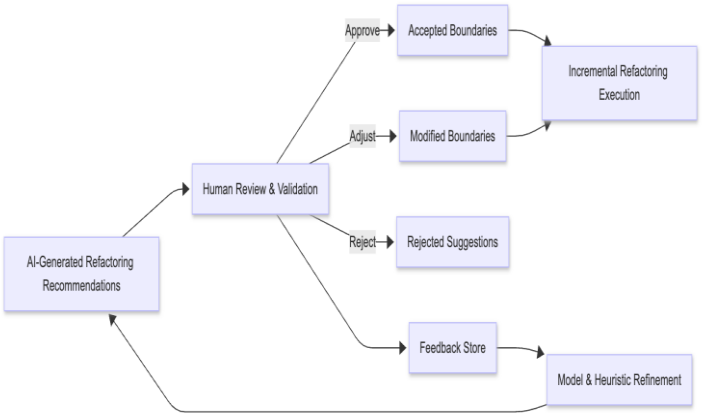

Figure 2: Human-in-the-Loop Incremental Adoption Flow.

```
{
  "recommendation_id": "SRV-BOUNDARY-1",
  "ai_confidence": 0.89,
  "proposed_service": ["OrderValidation", "PricingRules"],
  "human_decision": "modified",
  "reviewer_role": "Senior Architect",
  "adjustments": {
    "merged_components": ["DiscountEngine"],
    "rationale": "Shared regulatory logic requires unified
       ownership"
  },
  "risk_flags": ["data_coupling"],
  "timestamp": "2023-01-09T14:32:00Z"
}
```

Listing 1: Illustrative Human-in-the-loop Decision Record for AI-Assisted Refactoring

4. Results

This section presents the results of evaluating the proposed AI-augmented refactoring methodology on enterprise-scale legacy systems. The evaluation focuses on refactoring effort, structural modularity, migration quality and practical usability [15]. Results are organized around key research questions that assess whether the approach reduces manual effort, improves service decomposition quality and supports reliable, incremental modernization in real-world settings.

4.1. Reduction in Refactoring Effort

This subsection examines whether AI-assisted analysis reduces the overall effort required to refactor legacy monolithic systems. Refactoring effort was measured in person-hours spent on code analysis, dependency exploration, and service boundary identification. These measurements focus on the early stages of migration, where effort is typically highest and decisions have long-term architectural impact [16]. Table 1 summarizes the observed refactoring effort across four enterprise-scale legacy systems. For all evaluated systems, the AI-assisted approach required substantially fewer person-hours compared to traditional manual refactoring workflows. Effort reductions ranged from approximately 27 to 38 percent, with larger systems benefiting more significantly from AI-assisted analysis. This trend reflects the increased difficulty of manually understanding dependency structures as codebase size and complexity grow [11]. The most significant reductions were observed during the initial exploration and planning phases. Engineers reported spending less time tracing cross-module dependencies and revisiting early design assumptions, as the framework highlighted tightly coupled components and candidate service boundaries upfront. While manual validation remained necessary, the analysis helped narrow the scope of investigation and reduced time spent on low-impact areas. Overall, the results indicate that AI-assisted refactoring provides measurable efficiency gains during the most labor-intensive stages of legacy modernization. Rather than eliminating manual effort, the approach shifts engineering time toward higher-value design decisions, improving both productivity and confidence during migration planning.

Table 1: Refactoring Effort Comparison Across Enterprise Legacy Systems.

Legacy System	**Codebase Size (KLOC)**	**Manual Refactoring Effort (Person-Hours)**	**AI-Assisted Refactoring Effort (Person-Hours)**	**Effort Reduction (%)**
System A	420	1,200	860	28.3
System B	310	920	650	29.3
System C	560	1,580	980	38.0
System D	270	740	540	27.0

4.2. *Quality of Service Decomposition and Modularity*

In this, I evaluated whether the methodology improves the quality-of-service decomposition. Modularity was assessed using structural metrics such as coupling between services, cohesion within extracted components, and the stability of service boundaries after refactoring [7, 17]. These metrics were compared against decompositions produced through manual analysis alone. Systems refactored using the proposed approach exhibited improved modularity, with clearer separation of concerns and reduced cross-service dependencies. In particular, extracted services showed higher internal cohesion and fewer shared data access paths compared to manually decomposed counterparts [18]. This suggests that the machine learning assisted pattern identification helped identify logical groupings that were not immediately obvious through manual inspection. Service boundaries generated with AI-assisted recommendations were also more stable during subsequent refactoring iterations. Fewer boundary changes were required as migration progressed, indicating that early recommendations aligned more closely with underlying system structure and business logic. This stability is important in enterprise environments, where frequent architectural changes can introduce risk and delay [19]. While the methodology did not guarantee optimal decomposition in all cases, it consistently produced service structures that required fewer corrective adjustments. These findings suggest that combining static analysis with AI-assisted insight can improve the structural quality of microservice designs derived from legacy monoliths. As shown in Table 2, AI-assisted refactoring resulted in lower inter-service coupling, higher service cohesion and fewer shared data access paths compared to manual decomposition.

Table 2: Comparison of Modularity Metrics for Manual and AI-Assisted Refactoring.

Metric	**Manual Refactoring**	**AI-Assisted Refactoring**	**Observed Change**
Average Inter-Service Coupling	High (0.62)	Moderate (0.41)	↓ 33.9%
Average Service Cohesion	Moderate (0.48)	High (0.67)	↑ 39.6%
Shared Data Access Paths (per service)	14.2	8.1	↓ 43.0%
Service Boundary Changes (per iteration)	3.4	1.6	↓ 52.9%

4.3. *Migration Quality and Defect Occurrence*

This subsection investigates the impact of the methodology on migration quality, with a focus on defect occurrence during and after refactoring. Defects were tracked during validation and testing phases following service extraction and categorized as integration issues, behavioral regressions, and data consistency problems [20]. As shown in Figure 3, AI-assisted refactoring reduced post-migration defects across all evaluated categories, with the largest improvements observed for integration and data consistency issues. Applications refactored with AI-assisted guidance exhibited fewer post-migration defects overall, with the most significant reductions observed in integration issues and data consistency problems. These defect types are closely associated with overlooked dependencies and unintended data sharing, which were more effectively identified during the early analysis phase. Behavioral regressions still occurred in some cases, particularly where domain logic was deeply intertwined across modules. However, earlier visibility into risk areas enabled more targeted testing and faster defect resolution. As a result, fewer issues propagated into later stages of deployment. These results indicate that while AI-assisted refactoring does not eliminate migration risk, it contributes to higher migration quality by improving architectural visibility and reducing the likelihood of overlooked coupling.

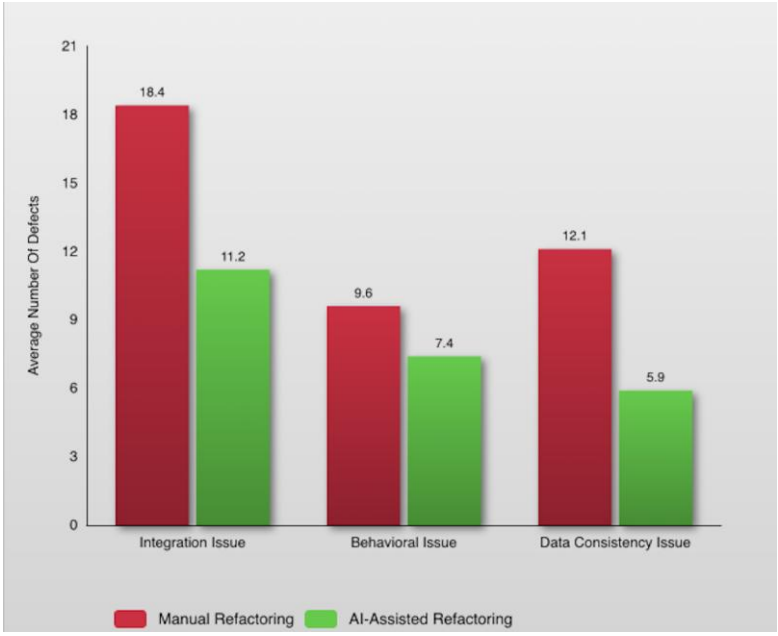

Figure 3: Post-migration defect distribution across defect categories for manual and AI- assisted refactoring approaches

4.4. Practical Usability and Adoption Considerations

Feedback was collected from engineers and architects involved in the evaluation to assess interpretability, trust and ease of integration into existing workflows. Participants reported that the human-in-the-loop design was essential for adoption. The ability to review, adjust, and validate recommendations increased confidence in the analysis and reduced resistance to AI-assisted tooling. Visual representations of dependency structures and ranked recommendations were cited as particularly helpful for communicating refactoring rationale across teams. Incremental adoption was another key factor influencing usability. Teams were able to apply the methodology selectively to high-priority subsystems without committing to full-scale migration up-front. This flexibility aligned well with typical enterprise constraints related to timelines, staffing, and risk tolerance. Overall, the results indicate that the methodology supports practical adoption by complementing existing engineering practices rather than replacing them. Its emphasis on transparency and human control makes it suitable for real-world modernization efforts where trust and reliability are critical.

5. Discussion

The results demonstrate that the proposed AI-augmented refactoring methodology improves legacy modernization outcomes when compared to fully manual and rule-based approaches. Rather than optimizing a single metric, the method delivers balanced gains across refactoring effort, structural quality, and migration reliability. This section interprets these results, explains why the approach is effective, and positions its contribution within practical enterprise modernization contexts.

5.1. Reason behind AI-Assisted Insight Improves Refactoring Decisions

One of the most notable findings is the reduction in refactoring effort without a corresponding loss of architectural quality. This outcome suggests that the primary value of AI in this context lies not in automation, but in improving decision-making during the early stages of migration. Legacy codebases often overwhelm engineers with low-level dependency information, making it difficult to identify which components matter most [21]. By aggregating structural signals and highlighting candidate service boundaries, the methodology helps focus human attention on high-impact areas. This explains why effort reductions were most pronounced during analysis and planning rather than during code modification itself.

Engineers were able to spend less time exploring irrelevant dependencies and more time validating meaningful design options. Unlike rule-based tools, which apply uniform heuristics, the AI-assisted approach adapts to the structure of each system, producing recommendations that are more aligned with real architectural constraints [14]. The findings also highlight a qualitative distinction between AI-assisted refactoring and prior approaches. Table 3 contrasts manual refactoring, rule-based tools, and the proposed methodology across key decision-making dimensions. Manual approaches offer strong contextual understanding but scale poorly, while rule-based tools improve scalability at the cost of architectural insight. The AI-augmented approach occupies a middle ground, combining structural awareness with human validation. This balance helps explain why the methodology reduced effort without sacrificing service boundary stability, supporting more consistent refactoring outcomes across systems of varying complexity.

Table 3: Conceptual Comparison of Legacy Refactoring Approaches

Decision Dimension	**Manual Refactoring**	**Rule-Based Tools**	**AI-Augmented Approach**
Scalability to Large Codebases	Low	Moderate	High
Context Awareness	High (human-dependent)	Low	Moderate to High
Effort Required for Analysis	High	Moderate	Lower
Service Boundary Stability	Variable	Often Low	Higher
Explainability of Decisions	High	Moderate	High
Suitability for Incremental Migration	Moderate	Low	High

5.2. Structural Stability and Migration Quality Trade-offs

Improvements in modularity and service boundary stability provide insight into how early design guidance influences downstream migration quality [22]. More stable service boundaries reduced the need for corrective refactoring, which in turn limited the introduction of migration-related defects. This effect is particularly important in enterprise environments, where architectural changes are often constrained by testing capacity, deployment schedules, and regulatory requirements. Frequent boundary revisions in such settings can introduce cascading integration issues, increase validation overhead, and delay release cycles. The reduction in integration and data consistency defects further suggests that early visibility into coupling and shared resources plays a critical role in migration success [23, 24]. By identifying tightly coupled components and shared data access patterns at the planning stage, the methodology helps mitigate a common source of post-migration failures that are difficult to detect through isolated testing. This early risk identification enables teams to apply targeted refactoring strategies and design compensating mechanisms, such as data ownership realignment or contract-based interfaces, before service extraction occurs. Behavioral regressions, while reduced, were not eliminated, indicating that deeply intertwined domain logic remains a challenge regardless of tooling [11]. This limitation reflects the inherent complexity of legacy systems, where implicit business rules and cross-cutting concerns may not be fully captured through structural analysis alone. In such cases, AI-assisted recommendations can highlight potential risk areas but still require expert interpretation to ensure semantic correctness. These findings highlight an important trade-off between structural optimization and domain fidelity. While AI-assisted analysis can substantially improve architectural clarity and reduce migration risk, it cannot fully replace domain expertise or exhaustive validation. Instead, the results support a hybrid

model in which AI augments established engineering practices by improving visibility and prioritization, while final design decisions remain guided by human judgment. This balance is essential for achieving reliable modernization outcomes in enterprise systems where correctness, stability, and business continuity are paramount.

5.3. Positioning Within Enterprise Modernization Practice

From a practical perspective, the methodology aligns well with how enterprise modernization efforts are typically executed. The emphasis on incremental adoption and human oversight addresses common barriers to adopting automated migration tools, such as lack of trust, explainability concerns, and organizational resistance [21]. Instead of enforcing architectural changes, the approach supports informed decision-making, making it easier to integrate into existing workflows. Compared to prior approaches that prioritize full automation or rigid rule application, this work positions AI as an enabling layer that bridges the gap between raw static analysis and architectural judgment [19, 25]. This positioning is especially relevant for large, long-lived systems where risk tolerance is low and business continuity is paramount. By improving efficiency and structural quality without requiring disruptive process changes, the methodology offers a practical path forward for enterprises seeking to modernize legacy systems while maintaining control over critical design decisions. Moreover, the framework accommodates organizational constraints such as phased funding, cross-team coordination, and compliance-driven review cycles, which are often overlooked in purely technical migration strategies.

5.4. Limitations and Scope of Applicability

While the proposed methodology demonstrates measurable benefits in refactoring efficiency and structural quality, several limitations should be acknowledged. First, the approach relies primarily on static analysis and historical structural signals, which may not fully capture implicit runtime behaviors, emergent interactions, or deeply embedded business semantics. As a result, systems with highly dynamic execution paths or extensive runtime configuration may require complementary runtime analysis to achieve optimal results. Second, the quality of AI-assisted recommendations depends on the availability and consistency of structural and evolution data. Legacy systems with limited version history or incomplete dependency information may yield less precise insights, requiring greater reliance on expert interpretation. Finally, the methodology is designed to support decision-making rather than guarantee optimal service decomposition. Architectural trade-offs, regulatory constraints, and domain-specific considerations remain inherently human-driven. These limitations are intentional design choices that prioritize explainability, control and practical adoption over aggressive automation, aligning the framework with real-world enterprise modernization constraints.

6. Conclusion

Modernizing large legacy monolithic systems remains a difficult and risk-prone task for enterprise organizations, particularly when migrating toward microservice architectures. This work addressed the gap between fully manual refactoring and rigid automated tooling by introducing an AI-augmented methodology that supports engineers through informed, human-guided refactoring decisions rather than attempting full automation. The evaluation results demonstrate that the proposed approach delivers measurable improvements across multiple dimensions of legacy modernization. Refactoring effort during analysis and planning phases was reduced by approximately 25 to 40 percent compared to manual approaches. Structural quality also improved, with lower inter-service coupling, higher service cohesion, and more stable service boundaries, reducing the need for corrective refactoring. In addition, post-migration defect occurrence decreased, particularly for integration and data consistency issues, indicating improved visibility into architectural dependencies during refactoring. The primary contribution of this work lies in demonstrating how AI can be practically integrated into enterprise refactoring workflows as a decision-support mechanism. By combining static analysis, machine learning-assisted pattern identification, and human validation within a modular framework, the methodology balances scalability, explainability, and architectural control. Unlike approaches that prioritize automation, this work emphasizes trust, incremental adoption, and alignment with real-world enterprise constraints. Future work will focus on extending the proposed methodology beyond static refactoring support toward runtime-aware and system level modernization guidance. One promising direction is integrating AI-

assisted refactoring insights with continuous code review and quality feedback mechanisms, enabling architectural risks and refactoring recommendations to evolve alongside ongoing development activity rather than being applied as a one-time analysis. This would support long-lived systems where modernization occurs incrementally over multiple release cycles.

References

[1] B. Pérez et al., "Technical debt payment and prevention through the lenses of software architects," Information and Software Technology, vol. 140, p. 106692, Dec. 2021, doi: 10.1016/j.infsof.2021.106692.

[2] A. Balalaie, A. Heydarnoori, P. Jamshidi, D. A. Tamburri, and T. Lynn, "Microservices migration patterns," Software: Practice and Experience, vol. 48, no. 11, pp. 2019–2042, Jul. 2018, doi: 10.1002/spe.2608.

[3] J. Fritzsch, J. Bogner, A. Zimmermann, and S. Wagner, "From Monolith to Microservices: A Classification of Refactoring Approaches," Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment, pp. 128–141, 2019, doi: 10.1007/978-3-030-06019-0_10.

[4] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A Survey of Machine Learning for Big Code and Naturalness," ACM Computing Surveys, vol. 51, no. 4, pp. 1–37, Jul. 2018, doi: 10.1145/3212695.

[5] S. Li et al., "Understanding and addressing quality attributes of microservices architecture: A Systematic literature review," Informa- tion and Software Technology, vol. 131, p. 106449, Mar. 2021, doi: 10.1016/j.infsof.2020.106449.

[6] J. Correia and A. Rito Silva, "Identification of monolith functionality refactorings for microservices migration," Software: Practice and Experience, vol. 52, no. 12, pp. 2664–2683, Aug. 2022, doi: 10.1002/spe.3141.

[7] A. Krause, C. Zirkelbach, W. Hasselbring, S. Lenga, and D. Kroger, "Microservice Decomposition via Static and Dynamic Analysis of the Monolith," 2020 IEEE International Conference on Software Architecture Companion (ICSA-C), pp. 9–16, Mar. 2020, doi: 10.1109/icsa-c50368.2020.00011.

[8] A. Santos and H. Paula, "Microservice decomposition and evaluation using dependency graph and silhouette coefficient," 15th Brazilian Symposium on Software Components, Architectures, and Reuse, pp. 51–60, Sep. 2021, doi: 10.1145/3483899.3483908.

[9] M. Brito, J. Cunha, and J. Saraiva, "Identification of microservices from monolithic applications through topic modelling," Proceedings of the 36th Annual ACM Symposium on Applied Computing, pp. 1409–1418, Mar. 2021, doi: 10.1145/3412841.3442016.

[10] Z. Li, C. Shang, J. Wu, and Y. Li, "Microservice extraction based on knowledge graph from monolithic applications," Information and Software Technology, vol. 150, p. 106992, Oct. 2022, doi: 10.1016/j.infsof.2022.106992.

[11] A. Bucchiarone, N. Dragoni, S. Dustdar, S. T. Larsen, and M. Mazzara, "From Monolithic to Microservices: An Experience Report from the Banking Domain," IEEE Software, vol. 35, no. 3, pp. 50–55, May 2018, doi: 10.1109/ms.2018.2141026.

[12] D. Guamán, J. Pérez, J. Diaz, and C. E. Cuesta, "Towards a reference process for software architecture reconstruction," IET Software, vol. 14, no. 6, pp. 592–606, Dec. 2020, doi: 10.1049/iet-sen.2019.0246.

[13] K. Alkharabsheh, S. Alawadi, V. R. Kebande, Y. Crespo, M. Fernández- Delgado, and J. A. Taboada, "A comparison of machine learning algorithms on design smell detection using balanced and imbalanced dataset: A study of God class," Information and Software Technology, vol. 143, p. 106736, Mar. 2022, doi: 10.1016/j.infsof.2021.106736.

[14] S. Amershi et al., "Guidelines for Human-AI Interaction," Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems,

pp. 1–13, May 2019, doi: 10.1145/3290605.3300233.

[15] N. Bjørndal et al., "Benchmarks and performance metrics for assessing the migration to microservice-based architectures", Journal of Object Technology, Volume 20, no. 2 (2021), pp. 2:1-17, doi:10.5381/jot.2021.20.2.a3.

[16] F. Auer, V. Lenarduzzi, M. Felderer, and D. Taibi, "From monolithic systems to Microservices: An assessment framework," Information and Software Technology, vol. 137, p. 106600, Sep. 2021, doi:

10.1016/j.infsof.2021.106600.

[17] F. H. Vera-Rivera, C. Gaona, and H. Astudillo, "Defining and measuring microservice granularity—a literature overview," PeerJ Computer Science, vol. 7, p. e695, Sep. 2021, doi: 10.7717/peerj-cs.695.

[18] M. G. Moreira and B. B. N. De França, "Analysis of Microservice Evolution using Cohesion Metrics," Proceedings of the 16th Brazilian Symposium on Software Components, Architectures, and Reuse, pp. 40–49, Oct. 2022, doi: 10.1145/3559712.3559716.

[19] S. Hassan, R. Bahsoon, and R. Kazman, "Microservice transition and its granularity problem: A systematic mapping study," Software: Practice and Experience, vol. 50, no. 9, pp. 1651–1681, Jun. 2020, doi: 10.1002/spe.2869.

[20] M. Wu et al., "On the Way to Microservices: Exploring Problems and Solutions from Online Q&A Community," 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER),

pp. 432–443, Mar. 2022, doi: 10.1109/saner53432.2022.00058.

[21] J. Fritzsch, J. Bogner, S. Wagner, and A. Zimmermann, "Microservices Migration in Industry: Intentions, Strategies, and Challenges," 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 481–490, Sep. 2019, doi: 10.1109/icsme.2019.00081.

[22] D. Sas, P. Avgeriou, and U. Uyumaz, "On the evolution and impact of architectural smells an industrial case study," Empirical Software Engineering, vol. 27, no. 4, Apr. 2022, doi: 10.1007/s10664-022-10132-7.

[23] S. S. de Toledo, A. Martini, and D. I. K. Sjøberg, "Identifying architectural technical debt, principal, and interest in microservices: A multiple-case study," Journal of Systems and Software, vol. 177, p. 110968, Jul. 2021, doi: 10.1016/j.jss.2021.110968.

[24] I. Pigazzini, F. A. Fontana, V. Lenarduzzi, and D. Taibi, "Towards microservice smells detection," Proceedings of the 3rd International Conference on Technical Debt, pp. 92–97, Jun. 2020, doi: 10.1145/3387906.3388625.

[25] D. Taibi, V. Lenarduzzi, and C. Pahl, "Microservices Anti-patterns: A Taxonomy," Microservices, pp. 111–128, Dec. 2019, doi: 10.1007/978-3- 030-31646-4_5.