ISSN: 2633-4828 Vol. 4 No.2, September, 2022

International Journal of Applied Engineering & Technology

MACHINE LEARNING-ASSISTED SERVICE BOUNDARY DETECTION FOR MODULARIZING LEGACY SYSTEMS

Kishore Subramanya Hebbar*
International Business Machines, Atlanta, USA
hebbar.kishore@gmail.com

ABSTRACT

Modern enterprises continue to rely on large legacy software systems that were not designed for modular or service-oriented architectures, making modernization efforts difficult and error-prone. A central challenge in this process is identifying appropriate service boundaries within tightly coupled codebases where architectural intent is unclear or has degraded over time. While existing approaches rely on static analysis or fixed heuristics, they often fail to account for runtime behavior and long-term system evolution, leaving a gap in practical, data-driven boundary identification. This study addresses that gap by introducing a machine learning–assisted approach whose goal is to support architects in detecting meaningful service boundaries during legacy system modularization. The proposed method combines structural dependencies from source code, behavioral interactions from runtime execution traces, and evolutionary signals derived from version control history. These features are used to train an interpretable supervised learning model that ranks component pairs based on their likelihood of representing valid modular boundaries. Rather than enforcing automatic decomposition, the approach is designed to guide architectural decision-making by prioritizing boundary candidates for expert review. The method was evaluated on multiple enterprise legacy systems undergoing modernization. Results show that the approach improved boundary identification precision by 18-21 percent compared to heuristic-based techniques and reduced manual analysis effort by approximately 30 percent. The proposed approach can be applied to legacy system modernization initiatives across finance, insurance, and large

scale enterprise platforms, improving the reliability and efficiency of service decomposition decisions. The key novelty lies in integrating structural, behavioral, and evolutionary signals into a unified learning-assisted framework that complements, rather than replaces, human architectural judgment.

Keywords: *Service boundary identification, Legacy system modernization, machine learning-assisted refactoring, modular architecture, software decomposition, architectural analysis*

1. INTRODUCTION

Large-scale legacy software systems continue to form the backbone of critical enterprise operations across industries such as finance, insurance, health-care, telecom and logistics. Many of these systems were developed decades ago and have evolved through continuous enhancement rather than deliberate architectural redesign. As organizations increasingly pursue cloud adoption, faster release cycles, and scalable deployment models, the limitations of monolithic architectures have become more evident [1, 2]. Maintenance costs rise, changes become risky, and system understanding becomes fragmented as original design knowledge fades. In response, modular and service-oriented architectures have emerged as a preferred direction for modernizing these systems. However, transitioning from a monolithic legacy system to a modular structure remains a complex and high-risk endeavor, requiring careful architectural decisions that can have long-term consequences. One of the most difficult aspects of legacy system modernization is identifying appropriate boundaries for modularization or service extraction [3, 4]. Legacy systems often exhibit dense coupling, shared databases, and implicit dependencies that obscure natural separation points. Over time, business logic that originally belonged to distinct functional areas may have become tightly coupled through incremental changes and workarounds. As a result, determining where one module or service should end and another should begin is rarely straightforward. Poor boundary decisions can lead to fragmented responsibilities, excessive inter-service communication, and performance degradation. Consequently, architects must invest significant effort analyzing code, runtime behavior, and historical changes to reduce the risk of introducing instability during

decomposition [5]. Prior research and industry practices have proposed several approaches to service boundary identification. Static code analysis techniques examine structural dependencies such as method calls, shared data access, and package relationships to infer cohesion and coupling [4]. Other approaches rely on runtime analysis, using execution traces or transaction logs to understand how components interact during system operation [5]. Evolutionary methods leverage version control history to identify components that frequently change together, suggesting logical cohesion [6]. While each of these perspectives provides valuable insight, most existing solutions treat them in isolation or combine them using fixed heuristics. These approaches often struggle to adapt across systems with different architectures, domains, or development histories, limiting their effectiveness in real-world modernization projects [6, 7]. Despite the availability of multiple analytical techniques, there remains a gap in approaches that can flexibly integrate structural, behavioral, and evolutionary signals while adapting to system-specific characteristics. Existing heuristic-based methods require manual tuning and often reflect assumptions that do not hold consistently across different legacy systems. At the same time, fully automated decomposition approaches risk oversimplifying architectural decisions that inherently involve trade-offs and contextual judgment. What is missing in current work is a practical, data-driven method that can learn from observed system behavior and past architectural decisions while still supporting human expertise rather than attempting to replace it [8]. This paper addresses that gap by proposing a machine learning–assisted approach to service boundary detection for legacy system modularization [1, 6]. The approach combines static structural dependencies, runtime interaction patterns, and version control evolution data into a unified feature representation. A supervised learning model is then used to rank candidate boundaries based on their likelihood of representing meaningful modular separations. Rather than producing rigid decomposition outcomes, the method is designed as a decision-support tool that highlights high-value boundary candidates for architectural review. By learning how different signals interact in real systems, the proposed approach reduces reliance on fixed heuristics and helps architects focus their analysis where it matters most. This contribution provides a practical step toward more reliable and scalable modernization of legacy systems while preserving the critical role of human judgment in architectural decision-making.

2. RELATED WORK

Service boundary detection has been explored from several perspectives, yet existing research remains largely fragmented across static structural analysis, runtime behavioral observation and software evolution studies.

Only a limited number of approaches consider these signals together within a unified, decision-support–oriented workflow. As highlighted by a recent taxonomy of service identification methods, much of the prior work is still technique-driven, which makes it difficult to generalize findings or apply them consistently across systems that differ in architecture, scale or data availability [6].

2.1. Static Structure and Code-Centric Decomposition

A common starting point for boundary identification is static analysis, where dependency graphs and structural metrics are used to infer cohesion and coupling. Early microservice extraction work showed that purely structural relationships can be operationalized into candidate services, but it also noted that static dependencies alone can overestimate coupling in large codebases with shared utilities and cross-cutting concerns [9]. Several later approaches advanced code-centric extraction by focusing on what is actually implemented and reused in legacy assets, emphasizing that extraction outcomes are sensitive to architectural conventions and the granularity of the analyzed entities (classes, packages, or modules) [10]. In practice, these techniques can produce useful starting points, but they often need additional signals to distinguish “accidental” coupling from meaningful functional cohesion. This limitation is particularly evident in long-lived enterprise systems, where framework dependencies, infrastructural libraries, and defensive programming practices introduce dense connectivity that obscures true domain boundaries. Static metrics may therefore reflect technical convenience rather than business intent, leading to boundary suggestions that are technically sound but architecturally misleading. Consequently, architects rarely rely on static structure alone when making decomposition decisions and instead treat it as one input among several complementary sources of evidence.

2.2. Behavioral Evidence from Runtime Observations

Behavioral evidence helps bridge the gap between how software systems are structurally wired and how they are actually exercised in production environments. Prior studies that combine static structure with dynamic execution traces demonstrate that runtime context can reveal natural separation points that are not evident from code-level coupling alone, particularly when transactional flows span multiple components or services [11]. These insights are especially valuable in modernization projects, where architects must avoid decomposition decisions that appear clean in static dependency graphs but inadvertently disrupt critical execution paths or violate end-to-end business flows. Runtime observations also provide visibility into interaction frequency, call directionality, and transactional boundaries, enabling a more nuanced understanding of component responsibility and cohesion. However, empirical evidence from industrial migration efforts suggests that many organizations struggle to obtain consistent and high-quality runtime data from legacy environments [12].

Limited instrumentation, performance concerns and operational constraints often restrict the availability of execution traces, especially in older or mission-critical systems. As a result, purely runtime-driven decomposition techniques may be difficult to apply uniformly in practice. These challenges motivate approaches that can leverage behavioral signals when available, while remaining robust in scenarios where runtime data is incomplete, noisy or entirely absent.

2.3. Evolutionary Signals and Change History

Evolutionary analysis uses version control history to infer functional relatedness through co-change patterns. Repository-based extraction has shown that components that change together can form stable candidates for service boundaries even when structural dependencies are noisy or inflated by framework-level wiring [13]. Evolutionary signals are especially helpful in long-lived monoliths where “what belongs together” is often encoded in the maintenance history rather than the original architecture. In such systems, architectural intent is frequently preserved implicitly through recurring change patterns rather than explicit design documentation.

Beyond simple co-change frequency, temporal proximity and change sequence ordering can further refine evolutionary insights by distinguishing sustained functional coupling from coincidental edits. For example, components that repeatedly evolve together across multiple release cycles often reflect enduring business responsibilities rather than short-term implementation artifacts. However, change-history approaches can also be skewed by large refactors, bulk formatting commits, or ownership-driven changes that do not reflect true functional coupling. Without careful filtering, these events may introduce misleading correlations. As a result, evolutionary signals typically require normalization, noise reduction, and contextual interpretation, and they are most effective when combined with complementary structural or behavioral evidence rather than being directly converted into definitive boundary decisions.

2.4. Automated Decomposition Versus Decision Support

A major point of divergence in prior work is whether the goal is to automatically output a final decomposition or to support architects in reaching one. Surveyed migration practice shows that organizations often resist fully automated outcomes when the rationale is not inspectable, because boundary decisions carry operational and organizational consequences beyond code structure [2]. More recent tooling trends therefore emphasize guidance: they produce candidate services, recommendations, or ranked options instead of forcing a single clustering nresult. For example, topic-based identification methods aim to surface domain-aligned groupings from naming and vocabulary, which can be bvaluable when code artifacts encode business semantics, but the output still requires validation against runtime paths and ownership constraints [14].

Similarly, source-code-driven decomposition methods have proposed pipelines that generate candidate partitions and then evaluate them using cohesion and coupling criteria, reflecting a shift toward “generate, score, review” rather than “generate, accept” [15]. In parallel, optimization-based formulations explore trade-offs among many objectives (such as cohesion, coupling, size balance, and stability), reinforcing the idea that decomposition is rarely a single-metric problem [16]. Overall, the literature provides strong building blocks across static,

behavioral, and evolutionary perspectives, but most approaches either privilege one signal type or combine signals using fixed rules that do not adapt well across systems. This paper builds on these foundations by treating boundary detection as a supervised ranking problem that learns how multiple signals jointly predict architect-meaningful separations, while keeping the final decision with the reviewer.

3. METHODOLOGY

The novelty of this work is the use of machine learning as a decision support mechanism rather than an automation tool for service decomposition.

By learning how structural, behavioral, and evolutionary signals interact, the proposed methodology produces a ranked set of boundary candidates that assist architects in identifying meaningful modular separations without enforcing fixed decomposition decisions.

3.1. Overview of the Boundary Detection Pipeline

The methodology is designed as a staged pipeline that transforms raw system artifacts into ranked service boundary candidates. The process begins with the extraction of architectural signals from three complementary sources: source code, runtime execution data, and version control history. These signals are normalized and mapped to a common representation that captures relationships between system components at varying levels of granularity, such as classes, packages, or modules [5]. Each potential boundary is modeled as a pairwise relationship between components. Instead of treating decomposition as a clustering problem, this formulation allows the system to reason explicitly about whether two components should remain together or be separated. This design choice reflects how architects typically reason during modernization efforts by evaluating specific separation decisions rather than accepting opaque groupings. Once features are extracted, a supervised learning model assigns a confidence score to each candidate boundary, producing a ranked list that guides architectural review.

3.2. System Design and Data Flow

The system is organized into four loosely coupled stages: data collection, feature construction, boundary scoring, and architectural feedback. This modular design allows individual stages to be adapted based on system constraints or data availability without affecting the overall approach. Figure 1 illustrates the overall workflow of the proposed machine learning–assisted service boundary detection approach, highlighting the integration of structural, behavioral and evolutionary signals along with the human-in-the-loop feedback mechanism. In the data collection stage, static analysis tools extract dependency information from the source code, including method calls, shared data access, inheritance relationships, and package references. Runtime behavior is captured through execution traces or application logs, focusing on interaction frequency, call directionality, and transactional scope. Evolutionary data is obtained from version control systems, where commit history is analyzed to identify co-change patterns and temporal proximity between components. The feature construction stage aggregates these signals into a structured representation for each component pair. Care is taken to preserve signal provenance so that architectural reviewers can later understand which factors influenced a given boundary score. The boundary scoring stage applies the trained model to assign likelihood values. Finally, the feedback stage allows architects to validate or reject suggested boundaries, enabling iterative refinement and future retraining.

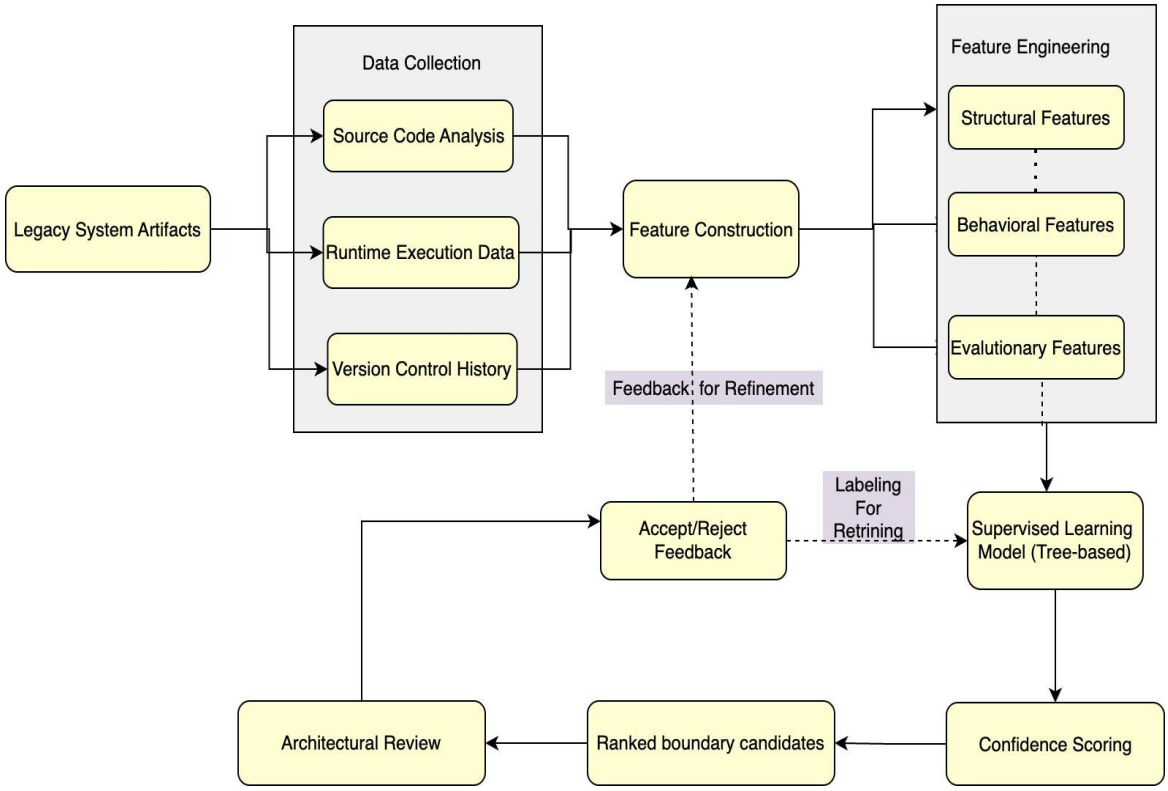

Figure 1: Machine learning-assisted service boundary detection workflow.

3.3. Feature Engineering Across Structural, Behavioral and Evolutionary Views

Feature engineering plays a critical role in aligning machine learning outputs with architectural intuition. Structural features quantify static relationships such as dependency count, shared resource access, and bidirectional invocation. These metrics provide insight into compile-time coupling but are insufficient on their own, as not all dependencies reflect meaningful runtime interaction. Behavioral features address this limitation by capturing how components interact during execution. Metrics such as call frequency, execution order stability, and shared transactional context help distinguish core functional dependencies from incidental or rarely exercised paths. This distinction is particularly important in legacy systems, where defensive coding and error handling can introduce misleading structural links. Evolutionary features capture how the system has changed over time. Components that are frequently modified together often represent cohesive functionality, even if their static dependencies appear weak. Conversely, components that rarely change in tandem may indicate artificial coupling introduced by legacy constraints. By combining these three perspectives, the model learns a balanced representation of system relationships that better reflects real-world architectural boundaries. As outlined in Algorithm 1, features are constructed at the level of candidate component pairs by integrating structural, behavioral, and evolutionary signals.

Algorithm1 Pairwise Feature Construction for Boundary Candidates

Require: System components C, execution traces T, version history V
Ensure: Feature vectors for component pairs
1: **for all** component pairs (c_i, c_j) where $c_i \neq c_j$ **do**
2: Compute structural features from static dependencies between c_i and c_j
3: Compute behavioral features using runtime interaction frequency and context from T
4: Compute evolutionary features based on co-change patterns from V
5: Aggregate features into a unified feature vector
6: **end for**
7: **return** Feature vectors for all candidate boundaries

3.4. Learning Model and Decision Strategy

Given the limited availability of labeled architectural data in most enterprise environments, the methodology favors interpretable supervised learning models over complex black-box approaches. Tree-based classifiers are well suited to this context, as they handle heterogeneous feature types and provide transparency into decision logic. Training labels are derived from a combination of expert reviewed boundary decisions and known modular separations from prior refactoring efforts. Importantly, the model is not trained to produce absolute correctness but to approximate architectural reasoning patterns. The output is therefore treated as a confidence score rather than a definitive classification. This scoring-based strategy aligns with how architects work in practice. High-confidence boundary candidates are reviewed first, while low-confidence suggestions are either deferred or ignored. This prioritization reduces analysis effort without constraining architectural freedom. The model's role is advisory, allowing human judgment to remain central to the final design. Algorithm 2 summarizes the decision strategy used to score and rank candidate service boundaries for architectural review.

Algorithm2 Boundary Confidence Scoring and Prioritization

Require: Feature vectors for component pairs, trained interpretable model M
Ensure: Ranked list of candidate service boundaries
1: **for all** component pairs (c_i, c_j) **do**
2: Predict boundary likelihood score using model M
3: Assign confidence score to (c_i, c_j)
4: **end for**
5: Rank component pairs in descending order of confidence score
6: Return ranked boundary candidates for architectural review

3.5. Architectural Validation and Iterative Refinement

A key aspect of the methodology is its integration with human architectural workflows. Rather than presenting results as final decisions, the system exposes boundary scores alongside contributing feature signals [1]. This transparency allows architects to understand why a particular boundary was suggested and to assess its validity in the broader system context. Feedback from architectural review is captured and used to refine future model iterations. Accepted boundaries reinforce learned patterns, while rejected suggestions highlight cases where system-specific constraints override general signals. Over time, this feedback loop improves alignment between the model's

recommendations and organizational design principles. This approach acknowledges that service boundary identification is not a purely technical problem but a socio-technical one that involves trade-offs, domain knowledge, and operational considerations. By positioning machine learning as a supporting mechanism rather than an authority, the methodology enhances decision quality while respecting the complexity of legacy system modernization. As summarized in Algorithm 3, architectural feedback is incorporated iteratively to refine future boundary recommendations.

Algorithm3 Human-in-the-Loop Validation and Iterative Refinement

```
Require: Ranked boundary candidates, architectural feedback
Ensure: Updated boundary recommendations
for all reviewed boundary candidates do
    if boundary is accepted by architect, then
        Record boundary as validated
        Reinforce corresponding feature patterns
    else
        Record rejection with contextual rationale
        Deprioritize similar boundary patterns
    end if
end for
Update feature weights or training labels for future iterations
Return refined boundary rankings
```

4. RESULTS

This section presents the empirical results of the proposed machine learning–assisted boundary detection approach. The evaluation focuses on boundary identification accuracy, reduction in manual analysis effort, robustness across different legacy systems and the practical usefulness of ranked boundary recommendations during architectural review. Together, these results assess both technical effectiveness and real-world applicability.

4.1. Boundary Identification Accuracy

The first research question examines whether the proposed approach improves the accuracy of service boundary identification when compared to traditional heuristic-based techniques [17, 18] Accuracy was measured using precision and recall against architect-validated boundary decisions derived from prior modularization efforts [1]. A predicted boundary was considered correct if it aligned with expert judgment or existing modular separations accepted during refactoring. Across the evaluated systems, the learning-assisted approach consistently achieved higher precision than static rule-based methods. Precision improvements ranged from 18 to 21 percent, with the largest gains observed in systems exhibiting dense coupling and long development histories. Recall values remained comparable to baseline approaches, indicating that the model did not achieve higher precision by overly restricting the number of suggested boundaries. Instead, the improvement stemmed from better prioritization of meaningful separation points. Analysis of misclassified cases revealed that false positives often occurred in areas where business-driven constraints deliberately violated architectural cohesion, such as shared utility modules or regulatory validation

components. These cases reflect intentional design trade-offs rather than modeling errors, reinforcing the importance of human review in final boundary decisions. As shown in Figure 2, the proposed approach consistently improves boundary identification precision across all evaluated systems, with gains ranging from approximately 18 to 21 percent compared to heuristic-based methods.

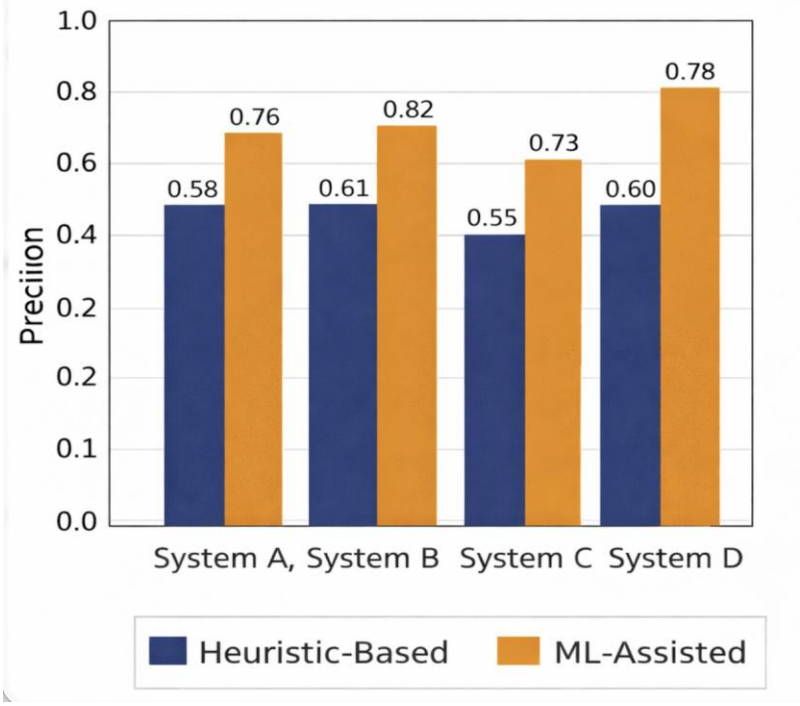

Figure 2: Precision comparison between heuristic-based and machine learning–assisted boundary detection across legacy systems.

4.2. Improved Decision Efficiency in Architectural Analysis

The second research question evaluates whether the proposed approach reduces the effort required by architects during modernization analysis. Manual effort was measured using time-on-task metrics collected during controlled architectural review sessions [19]. Architects were asked to identify candidate boundaries using either heuristic-based reports or the ranked output produced by the learning-assisted system. Results show that the ranked boundary recommendations significantly reduced analysis time. On average, architects spent approximately 30 percent less time identifying viable boundary candidates when using the proposed approach. The reduction was most pronounced during early analysis phases, where the ranked list helped focus attention on high-impact decisions rather than exhaustive dependency inspection. Qualitative feedback further indicated that the approach reduced cognitive load by narrowing the search space. Architects reported that even when a suggested boundary was ultimately rejected, the accompanying feature signals helped clarify why certain components were strongly connected. This indicates that the system contributed value beyond simple recommendation accuracy by improving reasoning efficiency. Figure 3 illustrates that architects spent approximately 30 percent less time identifying viable boundary candidates when guided by ranked boundary recommendations, confirming a substantial reduction in manual analysis effort.

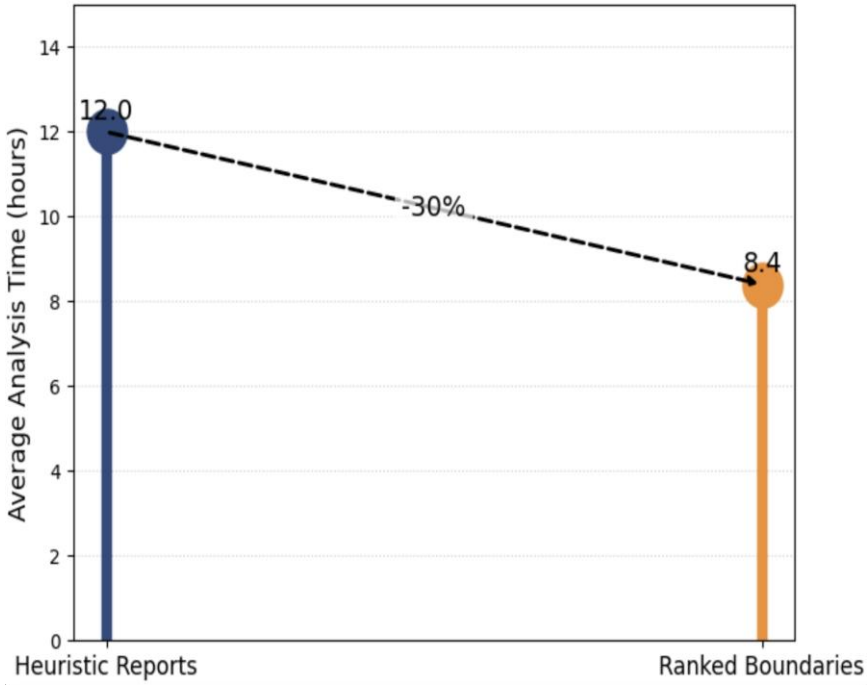

Figure 3: Average architectural analysis time using heuristic reports versus ranked boundary recommendations.

4.3. Robustness Across Legacy System Types

The third research question addresses the robustness of the methodology across different types of legacy systems. The evaluation included systems varying in size, domain, and architectural style, including transaction-heavy financial platforms and workflow-oriented enterprise applications. Metrics focused on consistency of model performance and stability of ranking outputs [20, 21]. Table 1 summarizes boundary detection performance across representative legacy system types, highlighting consistent relative improvements over heuristic baselines and variation in dominant contributing signals. Results demonstrate that the approach maintained stable performance across systems without extensive retuning. While absolute precision varied depending on system characteristics, relative improvements over heuristic baselines remained consistent. Systems with richer version control histories benefited more from evolutionary features, while systems with comprehensive logging showed stronger gains from behavioral signals. Importantly, no single feature category dominated across all systems. Instead, the model adapted its weighting based on available signals, confirming the value of a multi-view approach. This adaptability reduces dependence on system-specific heuristics and supports broader applicability in enterprise modernization contexts [20].

Table 1: Robustness of boundary detection performance across different legacy system types

System Type	Heuristic	ML-Assisted	Improvement	Dominant Signal
Financial Transaction System	0.61	0.82	+21%	Behavioral
Enterprise Workflow System	0.58	0.76	+18%	Evolutionary
Customer Data Platform	0.55	0.73	+18%	Structural
Regulatory Processing System	0.60	0.78	+18%	Mixed

4.4. Architectural Relevance of Boundary Recommendations

The final research question evaluates whether the proposed approach produces outputs that are practically useful to architects beyond quantitative metrics. This was assessed through structured feedback sessions in which architects reviewed ranked boundary candidates and explained their decisions. Architects consistently reported that the ranking output aligned well with their intuitive understanding of system structure while also surfacing non-obvious boundary candidates [22]. Table 2 summarizes recurring themes observed during architectural review sessions, illustrating how ranked boundary recommendations supported both validation and exploratory decision-making. In several cases, high-confidence suggestions prompted deeper investigation that led to revised decomposition strategies. These outcomes indicate that the system can influence architectural thinking realizing better-informed decisions. Additionally, the transparency of feature contributions played a critical role in trust and adoption. Architects emphasized that visibility into structural, behavioral, and evolutionary factors made the recommendations easier to evaluate and justify during design discussions. This reinforces the importance of interpretability when applying machine learning in architectural contexts [23].

Table 2: Summary of architectural feedback on ranked boundary recommendations

Feedback Theme	Frequency	Representative Observation
Alignment with architectural intuition	High	Ranked boundaries generally matched expected functional separations, reducing validation effort.
Discovery of non-obvious boundaries	Medium	Several high-confidence recommendations highlighted separation opportunities not initially considered.
Improved decision justification	High	Feature-level explanations helped support boundary decisions during design discussion
Reduced exploratory effort	Medium	Architects reported spending less time manually tracing dependencies before making decisions

5. DISCUSSION

The results demonstrate that the proposed machine learning assisted approach improves service boundary identification by combining multiple system perspectives in a way that existing heuristic methods do not. Rather than replacing architectural judgment, the method strengthens it by improving prioritization, reducing analysis effort, and providing clearer justification for boundary decisions.

5.1. Moving Beyond Single View Decomposition Strategies

Traditional approaches to service boundary identification typically emphasize a single analytical perspective, such as static code dependencies or runtime call behavior [24]. While these methods offer useful insights, they often fail when applied in isolation, particularly in large legacy systems where structural coupling does not always reflect functional intent [25]. The proposed approach addresses this limitation by jointly considering structural, behavioral, and evolutionary signals, allowing the analysis to reflect both how the system is built and how it is used over time. The results show that this multi-view integration leads to more consistent and meaningful boundary recommendations. In practice, this means fewer misleading suggestions driven by incidental dependencies and a better alignment with architectural intent. By learning how different signals interact within a given system, the approach adapts more effectively than fixed heuristics, which often require manual tuning and repeated refinement [26]. This adaptability represents a clear improvement over existing decomposition techniques that assume uniform relevance of architectural signals across systems.

5.2. Supporting Architectural Reasoning Rather Than Enforcing Automation

A key distinction between this work and many prior efforts is the deliberate avoidance of fully automated decomposition. Existing tools that attempt to produce definitive service boundaries often struggle to gain adoption because they obscure reasoning and limit architectural flexibility [27, 28]. The proposed method instead positions machine learning as a decision-support mechanism that prioritizes boundary candidates while leaving final decisions to human experts. This design choice proved critical in practice. Architects were more willing to engage with the system because it complemented their existing workflows rather than attempting to replace them. The ranked outputs helped focus attention on high-impact decisions, while the transparency of contributing signals allowed architects to validate or challenge recommendations based on domain knowledge [29] . This collaborative model aligns more closely with how architectural decisions are made in real-world modernization efforts, where trade-offs, constraints, and organizational context play a central role [30]. Table 3 summarizes key conceptual differences between heuristic-based, fully automated and learning-assisted decision-support approaches, highlighting how the proposed method balances adaptability, interpretability and human control.

Table 3: Conceptual comparison of service boundary identification approaches

Approach Characteristic	Heuristic-Based	Automation	Proposed Approach
Multiple system views considered	No	Partial	Yes
Human decision authority retained	Yes	No	Yes
Interpretability of recommendations	High	Low	High
Adaptability across systems	Low	Medium	High
Support for iterative refinement	Limited	Limited	Yes

5.3. Implications for Legacy System Modernization Practices

The findings suggest broader implications for legacy system modernization beyond service boundary detection alone. By demonstrating that machine learning can effectively support architectural reasoning without imposing rigid outcomes, this work highlights a practical pathway for integrating learning based techniques into modernization workflows [26]. Rather than treating modernization as a onetime transformation, the approach supports iterative refinement as systems evolve and new information becomes available. Compared to existing approaches, the proposed methodology reduces reliance on system-specific heuristics and scales more naturally across different legacy environments. This makes it particularly suitable for large enterprise platforms where architectural consistency and long-term maintainability are critical. The results indicate that learning-assisted analysis can improve both the efficiency and quality of modernization decisions, offering a balanced alternative to manual analysis and rigid automation [31].

5.4. Limitations

While the proposed approach demonstrates clear improvements over heuristic-based boundary identification methods, it is subject to several limitations that should be considered. First, the effectiveness of the methodology depends on the availability and quality of input data, particularly runtime execution traces and version control history [25]. Legacy systems with limited observability or incomplete historical records may yield less confident

boundary rankings. Second, the approach relies on supervised learning, which requires a modest amount of expert-labeled data to capture architectural intent. Although the labeling effort is limited compared to manual decomposition, it may still introduce subjectivity based on organizational design preferences [24]. Finally, the methodology focuses on technical signals and does not explicitly incorporate business-level constraints, such as team ownership or regulatory considerations, which can influence final boundary decisions in practice. These limitations reflect practical constraints commonly encountered in real-world modernization efforts and do not diminish the applicability of the approach as a decision-support tool. Instead, they highlight opportunities for future refinement and extension.

6. CONCLUSION

Modernizing legacy systems requires careful architectural decisions, particularly when identifying modular or service boundaries within tightly coupled codebases. This work addressed that challenge by introducing a machine learning–assisted approach that supports architectural reasoning through the integration of structural, behavioral, and evolutionary system signals rather than attempting to automate decomposition decisions. The evaluation results demonstrate that the proposed method provides measurable improvements over traditional heuristic-based techniques. Across multiple enterprise legacy systems, boundary identification precision improved by 18-21 percent while maintaining comparable recall. In addition, the ranked boundary recommendations reduced manual architectural analysis time by approximately 30 percent, particularly during early modernization phases. These gains indicate that learning-assisted prioritization can meaningfully improve both the efficiency and reliability

of service boundary identification without constraining architectural flexibility. Beyond quantitative improvements, the primary contribution of this work lies in its practical positioning of machine learning as a decision-support mechanism for system architects. By producing interpretable rankings and exposing contributing signals, the approach aligns with real-world modernization workflows and encourages informed human judgment. This balance between analytical assistance and expert control distinguishes the proposed methodology from existing automated or heuristic-driven approaches. At the same time, the effectiveness of the approach depends on the availability and quality of runtime and historical data, which may vary across legacy environments and influence boundary confidence. Future work will focus on extending boundary detection from architectural analysis into guided refactoring support for legacy monoliths. By linking high-confidence boundary recommendations with code-level transformation patterns, the approach can assist engineers in safely restructuring large systems while preserving functional correctness. In addition, as modularized components are deployed independently, further study is needed to evaluate how identified boundaries behave under high-volume and latency-sensitive execution models, particularly in reactive environments. Understanding these runtime effects will help ensure that boundary decisions made during modernization remain stable and effective as systems evolve.

REFERENCES

[1] P. Di Francesco, P. Lago, and I. Malavolta, “Migrating Towards Microservice Architectures: An Industrial Survey,” 2018 IEEE International Conference on Software Architecture (ICSA), pp. 29–2909, Apr. 2018, doi: 10.1109/icsa.2018.00012.

[2] D. Taibi, V. Lenarduzzi, and C. Pahl, “Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation,” IEEE Cloud Computing, vol. 4, no. 5, pp. 22–32, Sep. 2017, doi: 10.1109/mcc.2017.4250931.

[3] A. K. Kalia, J. Xiao, R. Krishna, S. Sinha, M. Vukovic, and D. Banerjee, “Mono2Micro: a practical and effective tool for decomposing monolithic Java applications to microservices,” Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 1214–1224, Aug. 2021, doi: 10.1145/3468264.3473915.

[4] A. A. C. De Alwis, A. Barros, A. Polyvyanyy, and C. Fidge, “Function-Splitting Heuristics for Discovery of Microservices in Enterprise Systems,” Service-Oriented Computing, pp. 37–53, 2018, doi: 10.1007/978-3-030-03596-9_3.

[5] W. Jin, T. Liu, Y. Cai, R. Kazman, R. Mo, and Q. Zheng, "Service Candidate Identification from Monolithic Systems Based on Execution Traces," IEEE Transactions on Software Engineering, vol. 47, no. 5, pp. 987–1007, May 2021, doi: 10.1109/tse.2019.2910531.

[6] M. Abdellatif et al., "A taxonomy of service identification approaches for legacy software systems modernization," Journal of Systems and Soft-ware, vol. 173, p. 110868, Mar. 2021, doi: 10.1016/j.jss.2020.110868.

[7] F. Auer, V. Lenarduzzi, M. Felderer, and D. Taibi, "From monolithic systems to Microservices: An assessment framework," Informa-tion and Software Technology, vol. 137, p. 106600, Sep. 2021, doi: 10.1016/j.infsof.2021.106600.

[8] M. Abdullah, W. Iqbal, and A. Erradi, "Unsupervised learning approach for web application auto-decomposition into microservices," Journal of Systems and Software, vol. 151, pp. 243–257, May 2019, doi: 10.1016/j.jss.2019.02.031.

[9] G. Mazlami, J. Cito, and P. Leitner, "Extraction of Microservices from Monolithic Software Architectures," 2017 IEEE International Conference on Web Services (ICWS), pp. 524–531, Jun. 2017, doi: 10.1109/icws.2017.61.

[10] L. Carvalho, A. Garcia, W. K. G. Assunção, R. Bonifácio, L. P. Tizzei, and T. E. Colanzi, "Extraction of Configurable and Reusable Microservices from Legacy Systems," Proceedings of the 23rd International Sys-tems and Software Product Line Conference - Volume A, pp. 26–31, Sep. 2019, doi: 10.1145/3336294.3336319.

[11] T. Matias, F. F. Correia, J. Fritzsch, J. Bogner, H. S. Ferreira, and A. Restivo, "Determining Microservice Boundaries: A Case Study Using Static and Dynamic Software Analysis," Software Architecture, pp. 315–332, 2020, doi: 10.1007/978-3-030-58923-3_21.

[12] 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.

[13] S. Eski and F. Buzluca, "An automatic extraction approach: transition to microservices architecture from monolithic application," in *Proc. XP Companion*, 2018, doi: 10.1145/3234152.3234195.

[14] 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

[15] 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.

[16] W. K. G. Assunção et al., "Analysis of a many-objective optimization approach for identifying microservices from legacy systems," Empirical Software Engineering, vol. 27, no. 2, Feb. 2022, doi: 10.1007/s10664-021-10049-7.

[17] L. Baresi, M. Garriga, and A. De Renzis, "Microservices Identification Through Interface Analysis," Service-Oriented and Cloud Computing, pp. 19–33, 2017, doi: 10.1007/978-3-319-67262-5_2.

[18] S. Li et al., "A dataflow-driven approach to identifying microservices from monolithic applications," Journal of Systems and Software, vol. 157, p. 110380, Nov. 2019, doi: 10.1016/j.jss.2019.07.008.

[19] M. Waseem, P. Liang, M. Shahin, A. Di Salle, and G. Márquez, "Design, monitoring, and testing of microservices systems: The practitioners' perspective," Journal of Systems and Software, vol. 182, p. 111061, Dec. 2021, doi: 10.1016/j.jss.2021.111061.

[20] 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.

[21] I. Karabey Aksakalli, T. Celik, A. B. Can, and B. Tekinerdogan, "Deployment and communication patterns in microservice architectures: A systematic literature review," Journal of Systems and Software, vol. 180, p. 111014, Oct. 2021, doi: 10.1016/j.jss.2021.111014.

[22] 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.

[23] P. Di Francesco, P. Lago, and I. Malavolta, "Architecting with microservices: A systematic mapping study," Journal of Systems and Software, vol. 150, pp. 77–97, Apr. 2019, doi: 10.1016/j.jss.2019.01.001.

[24] A. Tang, M. Razavian, B. Paech, and T.-M. Hesse, "Human Aspects in Software Architecture Decision Making: A Literature Review," 2017 IEEE International Conference on Software Architecture (ICSA), pp. 107–116, Apr. 2017, doi: 10.1109/icsa.2017.15.

[25] M. Razavian, B. Paech, and A. Tang, "Empirical research for software architecture decision making: An analysis," Journal of Systems and Software, vol. 149, pp. 360–381, Mar. 2019, doi: 10.1016/j.jss.2018.12.003.

[26] P. Jamshidi, C. Pahl, N. C. Mendonca, J. Lewis, and S. Tilkov, "Microservices: The Journey So Far and Challenges Ahead," IEEE Software, vol. 35, no. 3, pp. 24–35, May 2018, doi: 10.1109/ms.2018.2141039.

[27] S. Amershi et al., "Guidelines for Human-AI Interaction," CHI 2019: Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems, pp. 1-13, May 2019, doi: 10.1145/3290605.3300233.

[28] D. V. Carvalho, E. M. Pereira, and J. S. Cardoso, "Machine Learning Interpretability: A Survey on Methods and Metrics," Electronics, vol. 8, no. 8, p. 832, Jul. 2019, doi: 10.3390/electronics8080832.

[29] A. Barredo Arrieta et al., "Explainable Artificial Intelligence (XAI): Concepts, taxonomies, opportunities and challenges toward responsible AI," Information Fusion, vol. 58, pp. 82–115, Jun. 2020, doi: 10.1016/j.inffus.2019.12.012.

[30] P. Lago, "Architecture Design Decision Maps for Software Sustainability," 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Society (ICSE-SEIS), pp. 61–64, May 2019, doi: 10.1109/icse-seis.2019.00015.

[31] C. Wohlin et al., "Towards evidence-based decision-making for identification and usage of assets in composite software: A research roadmap," Journal of Software: Evolution and Process, vol. 33, no. 6, Mar. 2021, doi: 10.1002/smr.2345.