ISSN: 2583-9888
DOI: doi.org/10.51219/JAIMLD/Sriram-Ghanta/649

URF PUBLISHERS
connect with research world

Journal of Artificial Intelligence, Machine Learning and Data Science

https://urfpublishers.com/journal/artificial-intelligence

Vol: 1 & Iss: 1

Research Article

Quality-Driven Microservice Refactoring of Legacy Java Systems: Patterns, Automation, and Migration Challenges

Sriram Ghanta*

Citation: Ghanta S. Quality-Driven Microservice Refactoring of Legacy Java Systems: Patterns, Automation, and Migration Challenges. *J Artif Intell Mach Learn & Data Sci 2018* 1(1), 3197-3202. DOI: doi.org/10.51219/JAIMLD/Sriram-Ghanta/649

Received: 02 January, 2018; **Accepted:** 18 January, 2018; **Published:** 20 January, 2018

***Corresponding author:** Sriram Ghanta, MTS III Consultant, USA

ABSTRACT

Legacy Java applications, often engineered as large, tightly coupled monolithic systems, commonly suffer from scalability bottlenecks, brittle deployment processes, extended-release cycles, and weak fault isolation, all of which hinder their ability to meet modern business demands for agility and resilience. With the emergence of microservices as a dominant architectural paradigm emphasizing independent deployment, decentralized data management, and fine-grained scalability, systematic and controlled refactoring approaches have become essential for transforming legacy Java monoliths into loosely coupled, independently deployable services. This paper presents a structured analysis of microservice-oriented refactoring patterns for legacy Java applications, grounded in foundational software refactoring theory, service-oriented architecture (SOA) principles, and quality-driven architectural transformation methodologies. By leveraging quality-attribute-based architectural assessment, service decomposition workflows, and automation-assisted analysis techniques, this study synthesizes both practical and research-backed patterns that support incremental, low-risk migration strategies. The paper further consolidates core architectural strategies, highlights critical technical and organizational challenges encountered during legacy refactoring, and proposes a reference methodology tailored to enterprise-scale Java modernization initiatives, enabling organizations to achieve improved scalability, reliability, maintainability, and delivery velocity without disruptive system rewrites.

Keywords: Microservices, Legacy Java Refactoring, Monolithic to Microservices Migration, Service Decomposition, Software Architecture Modernization, Distributed Systems, Quality Attributes, SOA, Design Patterns

1. Introduction

Enterprise Java applications were predominantly designed using monolithic, layered, or service-oriented architectures (SOA). These architectural styles were well-suited to the technological and organizational constraints of their time, enabling rapid enterprise adoption, centralized governance, and structured modularization. For many organizations, such systems successfully supported mission-critical business operations across finance, healthcare, telecommunications, and retail for extended periods. Their tightly integrated nature simplified early development efforts and facilitated strong transactional consistency within centralized databases. However, as these systems evolved in scale and complexity, structural and operational limitations became increasingly evident. The continuous growth of codebases led to limited horizontal scalability, as monolithic deployment models constrained independent scaling of computationally intensive components. Tight coupling across modules introduced rigid interdependencies, making changes in one subsystem propagate unintended consequences throughout the application. Furthermore, long build and deployment cycles emerged as a major bottleneck, often requiring full system rebuilds and

coordinated release windows. These factors severely impacted development velocity and responsiveness to business demands.

In addition, such tightly integrated architectures exhibited low fault isolation, where failures in a single component frequently cascaded across the entire system, resulting in widespread service outages. From a maintainability perspective, testing and debugging became increasingly difficult, as tightly coupled code paths and shared state complicated regression testing, environment parity, and defect localization. These challenges were further exacerbated by modern enterprise expectations of continuous integration, continuous deployment, elastic scalability, and near-zero downtime operations. Against this backdrop, the microservices architectural style emerged as a response to the growing need for scalability, agility, and operational resilience. Microservices emphasize small, autonomous, and independently deployable services, each aligned to a well-defined business capability and owning its own data and lifecycle. By promoting decentralized governance, lightweight communication mechanisms, and independent scalability, microservices address many of the structural weaknesses inherent in monolithic enterprise Java systems.

Despite these advantages, direct redevelopment or large-scale rewrites of legacy systems into microservices are rarely feasible in enterprise environments. Such approaches introduce substantial risks related to cost overruns, functional regression, operational disruption, and the loss of institutional domain knowledge embedded within legacy codebases. Consequently, refactoring-based migration has emerged as the preferred modernization strategy, allowing organizations to incrementally extract services while preserving system stability, business continuity, and backward compatibility. This paper analyzes microservice-oriented refactoring patterns specifically tailored for legacy Java ecosystems, drawing from foundational software refactoring principles, service-oriented transformation methodologies, and early microservice migration research. The objective is to provide a structured and research-backed foundation for architects and engineers undertaking incremental modernization of long-lived enterprise Java platforms.

2. Background and Motivation

2.1. Legacy java monolith characteristics

Legacy Java applications, particularly those developed between the early 2000s and mid-2010s, were predominantly engineered as large, tightly coupled monolithic systems. A defining characteristic of such systems is tight database coupling, where multiple functional modules directly depend on a shared relational database schema. This shared persistence layer enforces structural rigidity, making schema evolution risky and complex. Additionally, these systems often rely on shared in-process object graphs, where domain objects are deeply intertwined across layers and packages. This design promotes hidden dependencies, increases ripple effects during code changes, and complicates modular isolation an essential requirement for independent service deployment.

Another critical limitation of legacy monoliths is centralized transaction management, typically implemented using global ACID transactions across multiple business modules via container-managed transactions or JTA. While effective for consistency in tightly integrated systems, such centralized control severely restricts horizontal scalability and introduces performance bottlenecks. Furthermore, heavy framework entanglement, such as deeply embedded Enterprise Java Beans (EJBs) or large monolithic Spring application contexts, tightly binds business logic to infrastructure concerns. These architectural traits directly violate core microservice principles, including independent deploy-ability, decentralized data governance, and fault isolation, thereby necessitating structural refactoring rather than superficial system decomposition.

2.2. From SOA to microservices

Service-Oriented Architecture (SOA) emerged as an early response to monolithic system complexity by promoting service abstraction, reusability, and loose coupling. However, in practice, SOA implementations frequently introduced centralized Enterprise Service Buses (ESBs) to orchestrate service interactions. While ESBs simplified service integration, they inadvertently became architectural bottlenecks and single points of failure. Additionally, SOA governance models often enforced heavy centralized control, with strict service registries, standardized contracts, and enterprise-wide data schemas. This high level of control, although beneficial for standardization, significantly reduced development agility and slowed down service evolution.

Microservices evolved as a natural architectural progression from SOA by explicitly rejecting centralized orchestration and heavy governance models. Instead, microservices emphasize lightweight communication mechanisms, primarily through RESTful HTTP APIs and asynchronous messaging. Equally important is the principle of decentralized data ownership, where each service manages its own persistence layer to ensure autonomy and scalability. Microservices also align tightly with DevOps-driven continuous integration and continuous delivery (CI/CD) practices, enabling rapid, independent deployments and faster innovation cycles. Consequently, legacy SOA-based and monolithic Java systems cannot be effectively modernized through interface exposure alone; they instead require deep structural refactoring of code, data, deployment pipelines, and operational models.

3. Related Work

Early foundational contributions to microservice-oriented refactoring are rooted in classical software refactoring and service-oriented design research. Opdyke laid the theoretical groundwork for object-oriented refactoring by formalizing behavior-preserving transformation principles, enabling structural changes without altering system functionality. Fowler later systematized these ideas into a comprehensive catalogue of refactoring patterns applicable to large-scale software systems, establishing practical guidance for incremental codebase evolution. Building upon modular design principles, Erl introduced foundational service-oriented design concepts and service normalization patterns, which emphasized loose coupling, contract-based interfaces, and service reusability concepts that later became central to microservice-based architectures.

As microservices emerged as a distinct architectural paradigm, Dragoni and colleagues provided one of the earliest comprehensive surveys of microservices, positioning

them as an evolution of distributed, service-based systems. Industrial migration perspectives were further enriched by Taibi, Lenarduzzi, and Pahl through their analysis of real-world transitions from monolithic systems to microservice architectures, highlighting both technical and organizational challenges. Bogner and collaborators introduced quality-driven decomposition approaches that align microservice extraction with explicit architectural quality goals such as scalability, performance, and maintainability. Complementing these methodological advances, the IBM Mono2Micro initiative proposed automated static-analysis-based techniques for identifying service boundaries within large monolithic systems. Collectively, these contributions establish the combined theoretical and applied foundation for microservice-oriented refactoring.

4. Quality-Driven Refactoring Methodology

A quality-driven migration strategy evaluates architectural refactoring decisions against explicitly defined non-functional requirements, which serve as the true drivers of microservice adoption rather than purely structural decomposition. Among the most critical quality attributes considered in legacy Java modernization are scalability, performance, reliability, maintainability, and security. These attributes capture the primary limitations of monolithic systems and provide measurable targets for architectural transformation. By prioritizing quality scenarios early in the migration process, organizations can avoid arbitrary service decomposition and instead ensure that each extracted microservice contributes measurably to system-wide operational goals **(Figure 1)**.

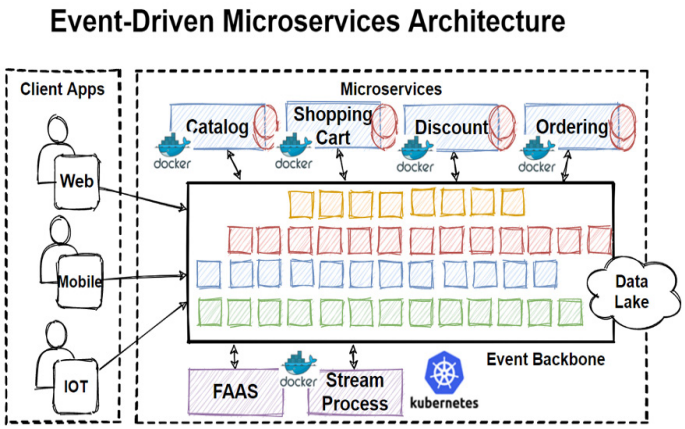

Figure 1: Quality-Driven Refactoring Framework.

The quality-driven refactoring framework **(Figure 1)** conceptualizes migration as a structured, iterative process comprising four fundamental stages: assessment of the monolithic architecture, identification of quality attribute scenarios, candidate service identification, and iterative refactoring with continuous validation. This systematic progression ensures that service boundaries emerge from both technical dependencies and quality requirements, rather than from code structure alone. By embedding continuous validation into the refactoring lifecycle, the framework enables early detection of architectural regressions while progressively improving system scalability, resilience, and maintainability. As a result, microservice extraction becomes a goal-oriented architectural evolution process, rather than a purely mechanical partitioning exercise.

5. Taxonomy of Microservice-Oriented Refactoring Patterns

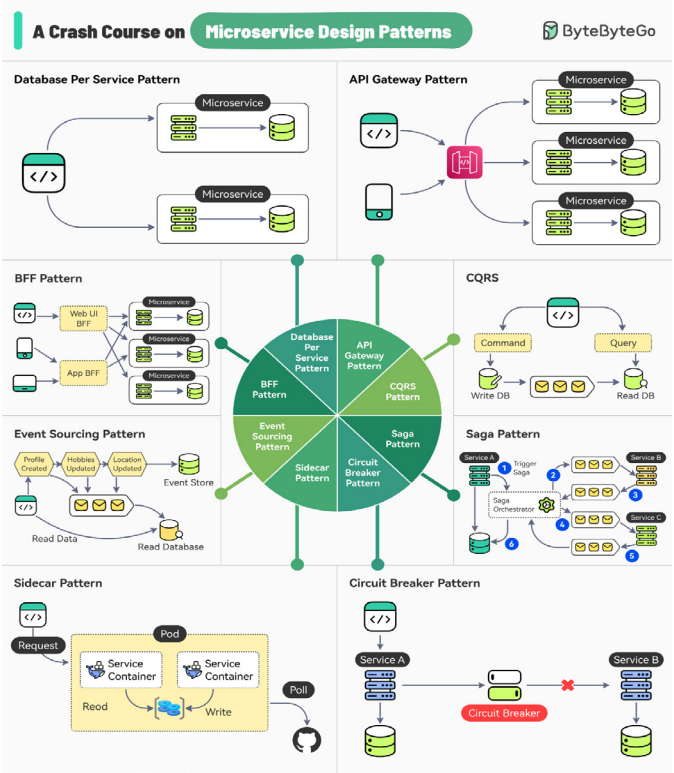

Figure 2: Refactoring Patterns Mapped to Quality Attributes.

5.1. Decomposition patterns

Decomposition patterns form the foundation of microservice-oriented refactoring, as they directly address the structural disassembly of monolithic systems into independently deployable services. The Strangler Pattern enables gradual replacement of legacy functionality by incrementally routing specific features through newly developed microservices while the monolith continues to operate. This approach minimizes migration risk, supports continuous delivery, and allows for controlled validation of new services under real production workloads. By avoiding large-scale rewrites, the strangler approach enables progressive modernization while preserving business continuity.

Domain-Driven Design (DDD)-based bounded context decomposition and business capability partitioning further strengthen decomposition strategies by aligning service boundaries with domain semantics and organizational responsibilities. Bounded context decomposition isolates services around clearly defined domain models, preventing schema leakage and semantic coupling across services. Business capability partitioning, on the other hand, organizes services around end-to-end business functions such as billing, authentication, or order processing. Together, these patterns ensure that service boundaries are both technically sound and business-aligned, enabling long-term architectural stability and organizational scalability.

5.2. Data refactoring patterns

Data refactoring represents one of the most challenging dimensions of microservice migration, as legacy monolithic systems typically rely on a single, tightly coupled relational database. The database-per-service pattern enforces strict service

autonomy by requiring that each microservice manages its own persistence layer. This pattern eliminates cross-service schema dependencies and enables independent schema evolution, but it also introduces challenges related to data duplication, distributed queries, and eventual consistency.

To mitigate migration risk during transitional phases, many modernization efforts adopt a read-only shared legacy database strategy, where newly extracted services initially consume data from the monolith's database without modification privileges. This approach enables progressive decoupling while maintaining transactional safety. Complementing this, event-based data synchronization allows services to propagate state changes asynchronously through messaging mechanisms. This pattern supports eventual consistency while improving scalability and fault isolation, making it particularly suitable for highly distributed microservice ecosystems.

5.3. Communication refactoring patterns

Communication refactoring patterns focus on restructuring how services interact once decomposition has begun. Synchronous REST extraction is commonly adopted as an initial step, where service boundaries are exposed through HTTP-based APIs while maintaining compatibility with existing call flows. This approach allows legacy components to interact with newly externalized services with minimal protocol changes, enabling low-friction integration during early migration stages.

As systems evolve toward higher scalability and resilience, asynchronous event-driven migration becomes increasingly significant. In this model, services communicate via message brokers or event streams, enabling loose temporal coupling and improved fault tolerance. To prevent legacy domain models from contaminating newly refactored services, anti-corruption layers are introduced at integration boundaries. These layers translate data formats and domain semantics between old and new systems, preserving architectural integrity and protecting microservices from legacy design constraints.

5.4. Reliability and fault-tolerance refactoring

Reliability centred refactoring patterns address one of the most critical weaknesses of monolithic architectures system-wide failure propagation. The Circuit Breaker pattern is widely applied to prevent cascading failures by immediately terminating calls to unstable services after failure thresholds are exceeded. This ensures that service faults remain localized and that dependent components can recover gracefully without exhausting system resources.

Additional resilience is achieved through the introduction of bulkheads and timeouts, which enforce isolation across service execution contexts and prevent resource starvation. Bulkheads limit the impact of localized failures by segregating thread pools, memory, or connection resources, while timeouts prevent indefinite blocking during remote calls. At a higher level, service-level fallback mechanisms provide degraded but functional responses during periods of partial failure. Collectively, these fault-tolerant refactoring patterns significantly enhance system availability, operational stability, and user experience qualities that monolithic systems inherently struggle to maintain under failure conditions.

6. Automated Service Decomposition Workflow

Manual decomposition of large-scale Java monoliths is widely recognized as both resource-intensive and highly error-prone, primarily due to the scale of codebases, implicit dependencies, and tightly coupled architectural layers. To address these limitations, the Mono2Micro workflow **(Figure 3)** introduced a semi-automated decomposition approach that systematically combines static and dynamic analysis techniques for identifying candidate microservice boundaries. The workflow begins with static code dependency analysis, which examines package-level, class-level, and call-graph dependencies to uncover tightly coupled components within the monolith. This step provides a structural foundation for understanding coupling and cohesion at scale.

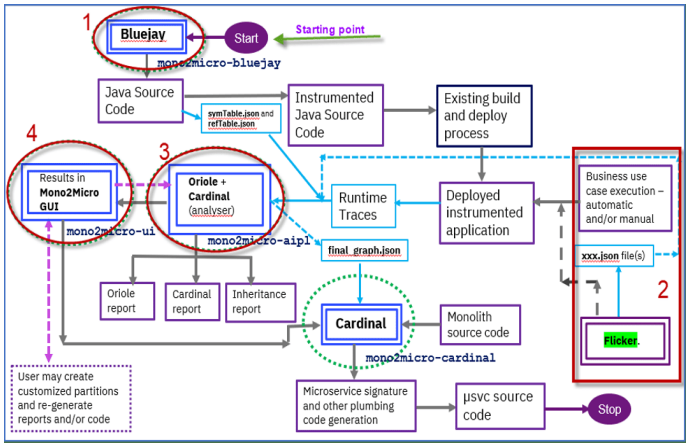

Figure 3: Mono2Micro Automated Decomposition Workflow.

The process is further refined through execution trace mining, which captures runtime behavior and user-driven execution paths to reveal real operational dependencies that static analysis alone cannot detect. These insights are then processed using service clustering algorithms that group related functionalities into potential service candidates. Following automated clustering, manual validation and refinement allow domain experts and architects to correct algorithmic biases and align service boundaries with business semantics. Finally, microservice interface extraction formalizes the external contracts for each identified service. This semi-automated methodology significantly reduces architectural subjectivity, improves decomposition repeatability, and accelerates large-scale legacy modernization efforts while maintaining architectural and domain correctness.

7. Key Challenges in Legacy Java Refactoring

Beyond structural and transactional concerns, organizational and architectural coupling also present significant barriers during microservice refactoring. Legacy Java systems are often developed by large, functionally siloed teams over extended time periods, resulting in implicit ownership boundaries and undocumented dependencies. When services are extracted without realigning team responsibilities and DevOps workflows, organizations frequently encounter coordination bottlenecks, delayed deployments, and fractured accountability. Moreover, existing CI/CD pipelines designed for monolithic builds are typically ill-suited for microservice ecosystems, where dozens or hundreds of independently deployable services require automated testing, versioning, and release orchestration.

Another frequently underestimated challenge lies in performance regression and network overhead. In monolithic systems, method invocations occur in-process with negligible latency. After refactoring into microservices, these same interactions become remote network calls subject to serialization cost, network latency, partial failures, and timeout management. Without careful application of communication refactoring patterns and caching strategies, organizations may observe severe throughput degradation and unpredictable latency behaviors. This transformation also necessitates the introduction of distributed tracing, centralized logging, and fine-grained performance monitoring, without which diagnosing production failures becomes significantly more difficult than in monolithic environments.

Finally, data governance, security enforcement, and regulatory compliance grow substantially more complex in distributed architectures. Legacy Java platforms often enforce security policies centrally through container-managed security, shared authentication modules, and monolithic authorization rules. When decomposed into microservices, these responsibilities must be consistently enforced across multiple independently deployed services without introducing security gaps. Additionally, regulatory requirements such as data residency, audit logging, and access control must be revalidated across every newly extracted service. These factors reinforce the necessity of controlled, incremental, and pattern-guided refactoring, where each migration step is validated not only for functionality, but also for security, compliance, and operational stability.

8. Case Study

8.1. Study 1: Industrial migration case studies

Industrial migration case studies conducted by Taibi and collaborators provide some of the earliest empirical evidence on the practical impact of refactoring monolithic systems into microservice architectures. The study demonstrated that organizations adopting microservices experienced a significant reduction in deployment time, primarily due to the ability to release services independently rather than through synchronized monolithic deployments. In addition to deployment agility, the study observed measurable improvements in system resilience, as service failures became localized rather than propagating across the entire system.

A critical insight from this study was the identification of DevOps maturity as a decisive success factor in microservice migration. Organizations that had already adopted automated testing, continuous integration, continuous delivery, and infrastructure-as-code practices were able to transition more successfully and with fewer operational disruptions. Conversely, teams lacking DevOps capabilities struggled with service orchestration, monitoring, and release coordination. This finding reinforces the notion that microservice refactoring is not purely an architectural transformation, but also an organizational and operational evolution.

8.2. Study 2: Quality-driven decomposition

The quality-driven decomposition study conducted by Bogner and colleagues established that microservice extraction must be explicitly guided by architectural quality attributes, rather than being based solely on static structural decomposition. The study demonstrated that performance, scalability, and adaptability scenarios should serve as primary drivers for defining service boundaries. By mapping system quality goals to architectural decisions, the approach ensured that microservice refactoring directly addressed the operational deficiencies of legacy monolithic systems.

An important outcome of this work was the identification of the over-fragmentation risk, often referred to as the creation of "nano-services." The study showed that uncontrolled decomposition can lead to an excessive number of extremely small services, resulting in increased communication overhead, complex dependency management, and degraded system performance. By enforcing quality-driven constraints during service extraction, the methodology successfully balanced service granularity, achieving architectural scalability without sacrificing maintainability or runtime efficiency.

8.3. Study 3: Automated decomposition

The IBM Mono2Micro initiative validated the feasibility of using automated tooling to identify candidate microservice boundaries within large-scale enterprise monolithic Java applications. By applying static code analysis and execution trace mining, the system generated data-driven service clusters that reflected both structural and runtime dependencies. This approach significantly reduced reliance on manual architectural intuition, which is often subjective and vulnerable to oversight in large codebases.

One of the most significant findings of this study was the reduction in migration planning time for large enterprise systems. By providing architects with automated recommendations for service boundaries, the tool accelerated decision-making and enabled rapid exploration of multiple decomposition alternatives. While still requiring expert validation, the Mono2Micro approach demonstrated that automation can substantially improve repeatability, objectivity, and scalability of microservice refactoring initiatives in complex legacy Java environments.

9. Discussion

Microservice-oriented refactoring extends far beyond a purely technical decomposition of software artifacts; it constitutes a fundamental socio-technical transformation that simultaneously reshapes system architecture, DevOps practices, organizational team structures, and governance models. From an architectural perspective, systems evolve from tightly coupled, centrally managed codebases to autonomous, independently deployable services. This technical decentralization must be mirrored at the organizational level, where development teams transition from functionally siloed structures to cross-functional, service-aligned teams with end-to-end ownership of design, development, deployment, and operations.

The results from early empirical and methodological studies strongly indicate that successful microservice migration depends on disciplined architectural governance, quality-driven decision-making, and robust automation support. Architectural discipline ensures that service boundaries remain coherent and aligned with both business capabilities and system quality goals. Quality-driven reasoning prevents arbitrary decomposition and mitigates risks such as over-fragmentation and operational instability. Strong automation spanning testing, deployment,

monitoring, and recovery serves as the enabling infrastructure that makes large-scale service autonomy operationally viable. Collectively, these factors confirm that microservice refactoring must be executed as a coordinated evolution of technology, process, and organization, rather than as an isolated software restructuring effort.

10. Conclusion

This paper presented a structured and systematic analysis of microservice-oriented refactoring patterns for legacy Java applications, synthesizing foundational refactoring theory, the evolution of service-oriented architecture, quality-driven architectural design, and early advances in automation-assisted service decomposition. Through the consolidation of decomposition, data, communication, and reliability refactoring patterns, the study demonstrated that successful legacy modernization cannot be achieved through ad hoc structural decomposition alone, but instead requires an incremental, pattern-based refactoring strategy guided by explicitly defined quality attributes such as scalability, performance, reliability, maintainability, and security. By grounding architectural transformation in measurable quality goals, organizations can ensure that microservice adoption directly addresses the fundamental limitations of monolithic systems rather than introducing new forms of architectural complexity.

The analysis further emphasized that automation-assisted decomposition techniques, such as static dependency analysis, execution trace mining, and service clustering, significantly improve the objectivity, repeatability, and scalability of microservice extraction. While manual validation by domain experts remains essential, semi-automated workflows substantially reduce migration planning effort and mitigate the risk of architecturally inconsistent service boundaries. At the same time, the study highlighted that microservice-oriented refactoring is inherently a socio-technical transformation, requiring coordinated evolution across software architecture, DevOps practices, organizational structures, and governance models. Without such alignment, technical refactoring efforts alone are unlikely to yield sustainable modernization outcomes.

Despite the promising results demonstrated across early studies and industrial case reports, several open limitations remain. Legacy systems frequently exhibit deeply entangled business logic, shared data ownership, and undocumented runtime dependencies, which continue to challenge both automated and manual decomposition approaches. Furthermore, the operational overhead introduced by microservice ecosystems including distributed monitoring, fault management, and security enforcement can offset the agility benefits of decomposition if not carefully managed. These factors reaffirm the necessity of controlled, iterative migration strategies rather than monolithic system rewrites or overly aggressive service fragmentation.

Future research will focus on the deeper integration of runtime monitoring, dynamic dependency extraction, and AI-assisted service boundary refinement to improve the adaptability and precision of refactoring decisions. Continuous telemetry-driven analysis can enable the dynamic evolution of service boundaries based on real workload behavior, failure propagation patterns, and performance bottlenecks. In parallel, machine learning–based optimization techniques hold promise for predicting service cohesion, anticipating architectural drift, and reducing long-term maintenance overhead. Additionally, tighter integration between microservice refactoring frameworks and continuous delivery pipelines represents a critical area for future advancement, enabling modernization efforts to proceed as part of routine software evolution rather than as disruptive transformation projects.

In conclusion, microservice-oriented refactoring represents a long-term evolutionary strategy rather than a one-time architectural event. When guided by quality-driven reasoning, supported by automation, and aligned with organizational transformation, it enables legacy Java systems to evolve toward highly scalable, resilient, and continuously deployable platforms. The continued convergence of architectural design, automation tooling, and intelligent analytics will ultimately define the next generation of legacy modernization methodologies.

11. References

1. https://silab.fon.bg.ac.rs/wp-content/uploads/2016/10/Refactoring-Improving-the-Design-of-Existing-Code-Addison-Wesley-Professional-1999.pdf
2. Parnas DL. On the criteria to be used in decomposing systems into modules. Communications of the ACM, 1972;15: 1053-1058.
3. https://fabiofumarola.github.io/nosql/readingMaterial/Evans03.pdf
4. https://www.researchgate.net/publication/224001127_Software_Architecture_In_Practice
5. https://book.northwind.ir/bookfiles/building-microservices/Building.Microservices.pdf
6. Padur SKR. Network Modernization in Large Enterprises: Firewall Transformation, Subnet Re-Architecture, and Cross-Platform Virtualization. In International Journal of Scientific Research & Engineering Trends, 2016;2.
7. Routhu KK. Seamless HR Finance Interoperability: A Unified Framework through Oracle Integration Cloud. In International Journal of Science, Engineering and Technology, 2018;6.
8. Dragoni N, Giallorenzo S, Lafuente AL, et al. Microservices: Yesterday, today, and tomorrow. In Present and ulterior software engineering. Springer, 2017.
9. Vishnubhatla S. Scalable Data Pipelines for Banking Operations: Cloud-Native Architectures and Regulatory-Aware Workflows. In International Journal of Science, Engineering and Technology, 2016;4.
10. https://martinfowler.com/articles/microservices.html
11. Routhu KK. The Evolution of HR from On-Premise to Oracle Cloud HCM: Challenges and Opportunities. In International Journal of Scientific Research & Engineering Trends, 2017;3.
12. https://dl.acm.org/doi/abs/10.1007/s00450-016-0337-0
13. Taibi D, Lenarduzzi V, Pahl C. Processes, motivations, and issues for migrating to microservices architectures. IEEE Cloud Computing, 2017;4: 22-32.
14. Padur SKR. Network Modernization in Large Enterprises: Firewall Transformation, Subnet Re-Architecture, and Cross-Platform Virtualization. In International Journal of Scientific Research & Engineering Trends, 2016;2.
15. Bogner J, Wagner S, Zimmermann A. Towards a practical maintainability quality model for service- and microservice-based systems. ECSA Proceedings, 2017.