The Eastasouth Journal of Information System and Computer Science
Vol. 1, No. 02, December, pp. 311-324

Architectural Decomposition Strategies for Migrating Legacy Java EE Monoliths to Cloud-Native Microservices

Tathagata Roy
Independent Researcher, Colorado, USA

Article Info

Article history:

Received December, 2023
Revised December, 2023
Accepted December, 2023

Keywords:

AKS;
Apache Kafka;
Axis;
Cloud Native Architecture;
Distributed Systems;
EJB;
EKS;
Event Driven Architecture;
GKE;
iBATIS;
Java EE Modernization;
JSP;
Kubernetes;
Legacy Application Refactoring;
MDB;
Microservices Decomposition;
Saga Pattern;
Servlets;
Strangler Pattern;
Struts 2.0

ABSTRACT

This paper presents a structured technical roadmap for decomposing legacy Java EE monoliths into independently deployable, cloud native microservices. The modernization effort focuses on enterprise applications built on JSPs, Servlets, Struts 2.0, EJBs, MDBs, Axis based SOAP services, and iBATIS driven persistence layers. These systems typically rely on tightly coupled modules, shared state, and synchronous communication patterns that limit scalability and operational resilience. The proposed approach outlines the architectural refactoring required to migrate these components into a Kubernetes containerized environment. Objectives include the replacement of point to point and JTA-based two-phase commit workflows, the transformation of MDB based asynchronous logic into Kafka consumer services, the evolution of Axis SOAP endpoints into RESTful interfaces, and the transition from shared schema iBATIS implementations to decentralized, database-per-service designs. The resulting framework provides a repeatable methodology for reducing architectural debt, improving throughput, and supporting compliant processing in regulated domains such as financial services.

This is an open access article under the CC BY-SA license.

Corresponding Author:

Name: Tathagata Roy
Institution: Independent Researcher, Colorado, USA
Email: tatha.roy@gmail.com

1. INTRODUCTION

Enterprise information systems in regulated industries such as financial services face persistent challenges as legacy Java Enterprise Edition (Java EE) monoliths struggle to meet modern expectations for scalability, resilience, and rapid delivery [1], [2], [3]. These systems, frequently hosted on Oracle WebLogic Server [4], depend on shared Java Virtual Machine (JVM) resources [18], [28] and global Java Transaction API (JTA) coordination [5], [6], [7], which introduce structural rigidity and weaken fault isolation. Tightly coupled technologies, including JavaServer Pages (JSP), Apache Struts 2.0, Enterprise JavaBeans (EJB), and Message-Driven Beans (MDB), further complicate the independent evolution of

business logic and presentation layers [12], [8], [3]. As a result, accumulated technical debt within these monolithic architectures constrains operational resilience and slows the adoption of elastic, cloud-native infrastructure [1], [9].

This paper examines a structured technical roadmap for migrating this legacy stack to independently deployable Spring Boot microservices operating on Kubernetes [9], [10], [11]. The transformation refactors EJB business logic [12], modernizes Apache Axis-based Simple Object Access Protocol (SOAP) services [12], [21], and evolves iBATIS persistence layers toward decentralized, service-aligned data ownership [13], [12], [14]. Synchronous SOAP interactions are replaced with Representational State Transfer (REST) interfaces [12], [21], [22], while asynchronous processing transitions toward Apache Kafka-based event-driven integration [15], [16], [17]. The primary objective is to establish a repeatable framework for reducing architectural debt, improving throughput, and sustaining compliance in highly regulated domains [2], [3].

2. ANALYSIS OF THE LEGACY JAVA EE ECOSYSTEM

Legacy Java EE systems in regulated industries were designed around centralized runtime control, strong transactional consistency, and predictable container behavior [1], [2], [3]. These assumptions conflict with the requirements of distributed, cloud-native environments that emphasize independent deployment, fine-grained scalability, and fault isolation [1], [3], [18]. The analysis below examines constraints across the application server, presentation tier, business logic, asynchronous processing, and persistence layer to clarify the structural impediments that complicate modernization efforts.

2.1 WebLogic Application Server Characteristics

WebLogic concentrates workloads within a shared JVM, where multiple modules compete for heap and CPU capacity, increasing the risk that a single resource-intensive operation can degrade overall system stability [18], [28]. Heavy reliance on JNDI introduces lookup latency and tight operational coupling to a global registry rather than lightweight service discovery mechanisms [12], [4]. Thread management issues such as stuck threads frequently emerge under load when long-running requests exhaust execution queues, forcing reactive vertical scaling that undermines containerized elasticity [3], [28]. In addition, global JTA coordination enforces distributed two-phase commit across multiple resources, creating lock contention, elevated latency, and a single failure domain that resists effective microservices decomposition [5], [6].

2.2 Presentation Layer Constraints

Presentation logic implemented using Apache Struts 2.0 Action classes tightly couples the user interface to backend orchestration, blending validation, workflow, and business logic within a single layer [12], [8]. This coupling complicates incremental refactoring, reduces testability, and introduces cascading change effects as business rules evolve [3], [8]. JSP relies on server-side rendering and custom tag libraries that intermingle markup with application logic, requiring the application server to generate HTML while simultaneously processing data [12]. This model limits the adoption of modern client-driven interfaces and frequently necessitates full monolith redeployment even for minor user interface changes [3], [8].

2.3 Business Logic Layer Limitations

Business logic implemented with EJB reflects a heavyweight component model closely bound to the WebLogic container lifecycle [12], [8]. Stateful EJB introduces session affinity that complicates horizontal scaling and cluster replication, while stateless EJB continues to incur container overhead

and RMI communication costs [3], [8]. Service exposure through Apache Axis-based SOAP endpoints further increases complexity through large XML payloads, rigid service contracts, and tooling dependencies that impede rapid evolution compared with REST-based API designs [12], [21], [22].

2.4 Asynchronous Processing Challenges

Asynchronous workflows implemented using MDB are tightly coupled to the WebLogic JMS provider, creating middleware lock-in that complicates migration to cloud-managed messaging platforms [4], [16]. Observability into message processing behavior is often fragmented, limiting visibility into latency, throughput, and failure conditions in high-volume workloads [19], [20]. Horizontal scalability remains constrained because concurrency depends on WebLogic thread pool and queue configurations, meaning that increased throughput typically requires scaling the entire application server cluster rather than individual consumers [3], [30].

2.5 Persistence Layer Constraints

Many legacy systems depend on iBATIS SQL Maps that externalize SQL into XML artifacts and encourage direct table-level coupling across functional modules [12], [14]. Over time, this approach increases maintenance complexity and amplifies risk during schema evolution [13], [18]. The widespread shared database anti-pattern allows multiple modules to access the same schema, eliminating clear data ownership boundaries and causing changes in one area to trigger unpredictable cascading effects throughout the monolith [3], [13]. These characteristics significantly slow delivery velocity and increase coordination overhead across development teams [3], [18].

3. PROPOSED CLOUD NATIVE ARCHITECTURE

The proposed architecture replaces the centralized Java EE monolith with a distributed, containerized system composed of independently deployable services [1], [2], [9]. The design prioritizes stateless execution, clear domain boundaries, and platform-managed scalability while preserving enterprise security and compliance requirements [21], [30]. Kubernetes is adopted as the primary execution substrate to standardize deployment, resilience, and operational automation across environments [18], [22].

3.1 Target State Overview

The target state centers on Spring Boot microservices that encapsulate discrete business capabilities aligned to bounded domains [9], [30]. Each service is designed to be stateless so that instances can be created or terminated dynamically without relying on local memory for session data, which is a prerequisite for horizontal scaling and failure isolation [21]. Domain-aligned service boundaries are established using Domain Driven Design (DDD) principles to reduce coupling and limit cross-service coordination [13], [23]. By encapsulating logic within these bounded contexts, each microservice can evolve, deploy, and scale independently while maintaining overall system coherence [9].

3.2 Kubernetes as the Execution Platform

Workloads are deployed on Kubernetes to provide automated scheduling, health management, and resource isolation across clusters [18]. Transitioning to managed environments such as Amazon Elastic Kubernetes Service (EKS), Azure Kubernetes Service (AKS), and Google Kubernetes Engine (GKE) allows organizations to focus on application delivery rather than infrastructure maintenance [18]. Built-in service discovery relies on the internal Domain Name System (DNS) for dynamic routing, while the Horizontal

Pod Autoscaler (HPA) adjusts pod counts based on CPU or memory utilization to sustain throughput under variable demand and mitigate the vertical scaling limitations of the legacy stack [22].

3.3 *Communication Model*

Synchronous integration shifts from SOAP to REST interfaces that rely on lightweight JavaScript Object Notation (JSON) payloads [12], [21]. This transition reduces network overhead and simplifies versioning compared with the complex XML parsing historically associated with Apache Axis [22]. For asynchronous coordination, the architecture adopts Apache Kafka as a durable event backbone that decouples producers from consumers and enables temporal separation between services [15]. Event-driven flows replace tightly coupled point-to-point MDB interactions, allowing services to process events at their own pace while improving resilience and scalability [16], [21].

3.4 *Security and IAM Integration*

The architecture maintains existing enterprise security standards by integrating with the current Identity Provider (IdP) through a centralized Identity and Access Management (IAM) layer [7]. Services leverage Spring Security to implement OpenID Connect (OIDC) and OAuth 2.0 (OAuth2) for authentication and authorization [7]. By utilizing JSON Web Token (JWT) artifacts, security contexts are propagated securely across service boundaries without maintaining stateful sessions [7]. This approach standardizes token validation and role mapping while preserving compatibility with enterprise governance policies and removing dependence on container-managed security in WebLogic [7].

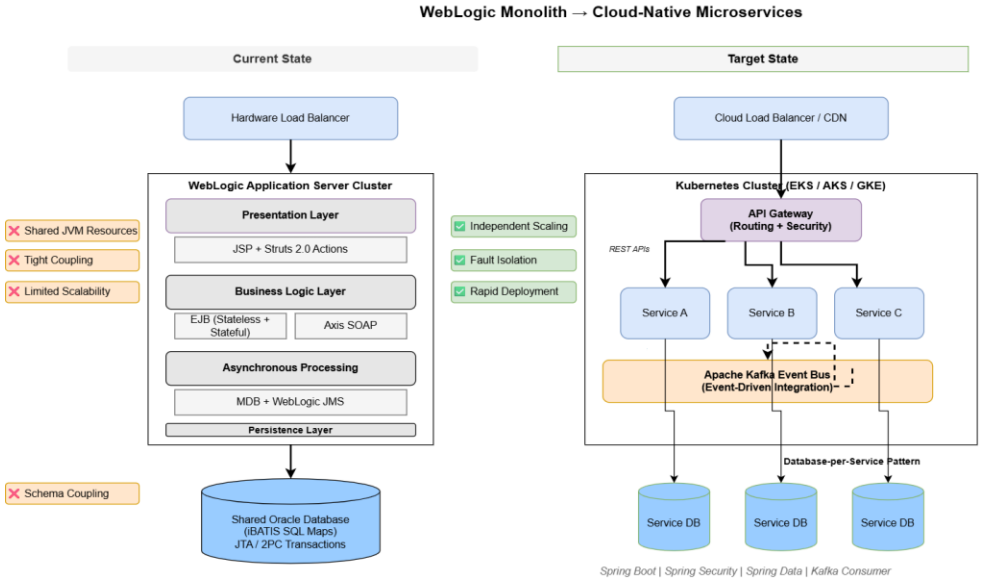


Figure 1. Weblogic monolith to Cloud Native Microservices

4. ARCHITECTURAL REFACTORING AND PATTERN MAPPING

Modernizing a legacy Java EE monolith requires disciplined architectural refactoring rather than a wholesale rewrite [1], [2]. The work examines how entrenched enterprise components can be systematically mapped to cloud native constructs while preserving functional correctness, data integrity, and operational reliability [9]. In practice, decomposition is most stable when domain boundaries and data ownership are clarified before interface and runtime refactoring, because JSP, EJB, and iBATIS logic are deeply coupled to a shared schema

and container-managed transactions [3], [13]. The following subsections describe key refactoring patterns and the practical steps required to transition from the legacy stack to the target architecture.

4.1 Persistence Layer and Domain-Driven Design Refactoring

Persistence refactoring must be guided by domain boundaries to ensure that extraction does not simply reproduce shared schema coupling behind new API layers [13], [23]. The migration begins by applying Domain-Driven Design to identify bounded contexts and assigning explicit ownership of specific table clusters to target microservices [13]. Technical execution involves extracting SQL logic from iBATIS SQL Maps and isolating it within service-specific Data Access Object components [13], [3]. During coexistence, allowing broader read access to legacy tables while restricting writes to a single owning service reduces migration risk and prevents conflicting updates [9]. Identifying and removing cross-module foreign key constraints is critical, as it prepares the system for a gradual transition to a database-per-service model and removes hidden dependencies that previously constrained JSP and EJB refactoring [1], [3]. Over time, newly extracted services can be promoted as primary writers for their owned data while the monolith retains limited read access until full separation is achieved [9].

4.2 Struts and JSP Modernization

Modernization of Struts and JSP begins by mapping user journeys to bounded contexts and identifying which data each context owns [8], [13]. This ensures that new REST endpoints are defined around stable domain resources rather than legacy page actions or shared table access patterns [8]. Migration proceeds by introducing thin REST facades that expose these domain resources while still delegating to existing business logic, effectively functioning as an anti-corruption layer that prevents legacy coupling from leaking into new services [13]. An API gateway can serve as a routing boundary between legacy JSP pages and newly modernized frontends, enabling side-by-side coexistence during transition [24], [2]. UI migration is most effective when executed incrementally, prioritizing high-change or high-value screens first while leaving low-risk pages on the legacy stack [3]. Over time, JSP pages and Struts actions are progressively replaced with a REST-backed UI or SPA that consumes these APIs, allowing presentation logic to decouple from backend orchestration without disrupting existing functionality [8].

4.3 EJB to Spring Service Refactoring

EJB refactoring is stabilized by aligning business services to bounded contexts and explicitly assigning responsibility for domain invariants [13], [3]. Direct translation of EJB methods into Spring services often preserves hidden coupling; instead, logic is reorganized around cohesive domain services that encapsulate business rules [8]. Once a service boundary is defined, EJB methods are extracted into Spring-managed services using @Service and @Transactional annotations, replacing JNDI-based lookups with dependency injection [8], [30]. Each microservice defines a single primary transaction boundary corresponding to its domain aggregate, minimizing cross-service locking and coordination [5], [6]. Any stateful EJBs are refactored into stateless services, with conversational state offloaded to distributed caches so that individual pods remain ephemeral and horizontally scalable [3], [30]. Legacy EJB logic may be wrapped temporarily to enable gradual migration before full decommissioning of WebLogic-specific artifacts [8].

4.4 Axis SOAP to REST Transformation

SOAP to REST migration becomes more maintainable when

service contracts are derived from domain models rather than legacy implementation details [12], [21]. The transition begins by defining REST resources that correspond to domain aggregates, using Spring @RestController to manage requests [8]. During migration, legacy SOAP consumers can continue operating while new consumers adopt REST contracts, creating a temporary dual-protocol environment that minimizes disruption [8], [24]. A pass-through REST facade can initially delegate to existing Axis services, enabling phased traffic migration without forcing simultaneous consumer changes [24]. API versioning protects existing clients while newer services adopt JSON payloads that simplify tooling and schema governance compared with XML [22]. Over time, native REST implementations replace Axis endpoints as confidence and coverage increase [21].

4.5 MDB to Kafka Consumer Migration

Migrating MDB workflows to Kafka is more reliable when event definitions reflect the data ownership implied by bounded contexts [15], [16]. The migration begins by identifying business state transitions currently embedded in MDB logic and expressing them as explicit events with governed schemas [15]. A JMS to Kafka bridge can mirror legacy messages onto Kafka topics so that new consumers can run in parallel with existing MDB processing during validation [16], [24]. Business logic is extracted from the onMessage method and placed into a @KafkaListener component, while partition keys are aligned with domain identifiers such as account or transaction IDs to preserve logical ordering where required [4], [15]. Kafka enables partitioning and consumer group scaling, allowing high-throughput processing that was previously constrained by WebLogic thread pools, while at-least-once delivery semantics combined with idempotent handlers prevent duplicate business effects [16]. Once stability is demonstrated, MDB components can be gradually retired [3].

4.6 Distributed Transaction Replacement

Replacing JTA two-phase commit requires decomposing business processes into domain-level steps that can be committed locally within each bounded context [5], [6]. The migration begins by identifying cross-module transactions that currently depend on global coordination and modeling them as sequences of local state transitions [5]. Some workflows are better suited to centralized Saga orchestration when strict sequencing is required, while others benefit from decentralized choreography when services are loosely coupled [25]. Compensating actions are designed to restore business invariants rather than merely reverting technical state, ensuring semantic correctness from a domain perspective [6], [26]. To ensure reliable event publication, services can adopt an Outbox pattern in which events are written to a local table as part of the same transaction and later emitted to Kafka, replacing brittle JTA coordination with a resilient eventually consistent workflow [5], [6]. This approach reduces lock contention and latency that historically produced stuck threads in WebLogic while preserving business correctness through explicit, auditable state transitions [3].

Table 1. Mapping Patterns

Legacy construct or pattern	Target cloud native equivalent	Primary target library or platform component
WebLogic deployed an EAR with shared runtime	Containerized services with independent lifecycles	Docker; Kubernetes Deployment

Legacy construct or pattern	Target cloud native equivalent	Primary target library or platform component
JNDI-based resource lookup	Configuration and dependency injection	Spring Framework dependency injection; Kubernetes ConfigMap and Secret
Container managed transactions with JTA and two-phase commit	Service local transactions with event driven coordination	Spring transaction management; Saga pattern; Outbox pattern
Struts 2.0 Action classes	API first controller layer aligned to domain services	Spring Web with REST controllers
JSP with server side rendering	REST backed UI or SPA consuming APIs	REST APIs via Spring Web; SPA framework external to backend
EJB stateless session bean	Stateless domain service component	Spring service layer using *@Service*
EJB stateful session bean	Stateless service with externalized state	Distributed cache; token based session model
EJB container declarative transactions	Explicit service boundary transactions	Spring transactions using *@Transactional*
Axis based SOAP endpoints and WSDL contracts	REST endpoints with resource semantics	Spring Web with REST controllers; API versioning strategy
SOAP XML payloads	JSON payloads over HTTP	JSON serialization stack in Spring; HTTP content negotiation
JMS queue based MDB consumers	Kafka consumer services with horizontal scaling	Spring for Apache Kafka using *@KafkaListener*
WebLogic JMS provider coupling	Managed Kafka cluster integration	Kafka client ecosystem; managed Kafka offering
Point to point integration	Event driven integration	Kafka topics and consumer groups
iBATIS SQL Maps	Repository and DAO layer per service	DAO and repository abstraction in Spring; iBATIS retained during transition
Shared database schema	Service aligned schema and ownership boundaries	Database per service target model; coexistence controls during transition
Database used for cross module joins	API composition or event driven replication	REST composition; Kafka driven replication and projections

5. INFRASTRUCTURE AND CONTAINERIZATION STRATEGY

The modernization of application architecture must be accompanied by a corresponding shift in infrastructure practices to support portability, reproducibility, and operational consistency [2], [3]. Containerization and Kubernetes-based deployment create a uniform runtime environment across development, testing, and production, reducing configuration drift and enabling automation [30]. The strategy outlined in this section focuses on building lightweight Java containers, defining predictable resource usage, externalizing configuration, and standardizing deployment through templated packaging [3].

5.1 Dockerization of Java Services

Java services are packaged using Docker containers to ensure consistent execution regardless of underlying infrastructure [30]. Multi-stage builds separate compilation from runtime packaging so that build dependencies

remain in intermediate layers while the final image contains only the application artifact and minimal runtime components [3]. JVM tuning for containers emphasizes memory awareness through flags such as -XX:MaxRAMPercentage and -XX:InitialRAMPercentage, allowing the runtime to respect Kubernetes resource limits and avoid out-of-memory termination caused by cgroup constraints [3], [22]. Runtime images are constructed using either distroless base layers or Alpine Linux to minimize attack surface, reduce image size, and improve startup performance while preserving compatibility with Spring Boot [30]. Security hardening practices, including non-root execution and container health checks, further align the runtime with regulatory expectations in financial environments [30].

5.2 Kubernetes Resource Management

Each microservice declares explicit CPU and memory requests and limits so that Kubernetes can schedule workloads efficiently and prevent resource exhaustion across shared clusters [22]. Liveness and readiness probes provide continuous health signaling, enabling the platform to restart stalled instances and withhold traffic until a service is fully initialized and connected to its dependencies [22]. The HPA dynamically adjusts replica counts based on observed utilization or application metrics, maintaining throughput under variable demand while avoiding unnecessary overprovisioning [22]. This shift from vertical scaling to elastic horizontal scaling addresses capacity constraints inherent in the legacy WebLogic clustering model [3].

5.3 Configuration and Secret Management

Application configuration is externalized using Kubernetes ConfigMaps and Secrets to decouple environment-specific settings from container images and replace reliance on server-bound configuration mechanisms [2], [3]. Non-sensitive properties such as endpoints and feature toggles are supplied through ConfigMaps, allowing the same artifact to be promoted across environments without modification [30]. Sensitive data such as database credentials, API keys, and certificates are stored as Secrets and integrated with enterprise vault systems to provide centralized governance, auditability, and rotation [2], [30]. At runtime, secrets are injected securely into containers rather than embedded in code or images, while remaining compatible with Spring Boot's hierarchical configuration model [30].

5.4 Helm-Based Deployment Model

Deployment standardization is achieved using Helm charts that encapsulate Kubernetes manifests into reusable templates for consistent delivery [30]. A common chart structure defines base resources such as Deployments, Services, and ingress rules, ensuring uniform labels, annotations, and runtime behavior across services [30]. Environment-specific overrides are applied through values files or external configuration so that development, staging, and production can diverge safely while sharing a single authoritative template [30]. This model reduces manual variability, simplifies governance reviews, and supports repeatable, auditable releases in regulated domains [30].

6. PHASED MIGRATION USING THE STRANGLER PATTERN

A phased migration based on the Strangler pattern enables organizations to incrementally modernize a legacy monolith while maintaining continuous production availability [13], [24]. Instead of a disruptive big bang replacement, new functionality is gradually introduced around the edges of the existing system, progressively reducing the monolith's scope until it can be safely retired [3], [24]. This strategy minimizes operational

risk, preserves business continuity, and provides measurable value at each stage of transformation while allowing architectural learning to inform subsequent phases [2].

6.1 Phase 1: Routing Layer Introduction

The initial phase introduces an API gateway as a stable facade between external consumers and backend systems [2], [24]. The gateway becomes the single entry point for all traffic, routing most requests to the existing WebLogic application while selectively directing others to newly created Spring Boot services [30]. This abstraction shields clients from internal change while centralizing security, rate limiting, and observability [2]. As routing rules accumulate, however, the gateway can evolve into a configuration monolith that mirrors the system it replaces [3]. To avoid this, routing must remain coarse-grained, business logic should reside in services, and gateway policies should be treated as versioned infrastructure rather than ad hoc configuration [30].

6.2 Phase 2: Service Extraction Strategy

Service extraction prioritizes domains that are either low complexity or high business value to deliver early benefits with limited risk [3], [13]. Bounded contexts are refined and explicit ownership boundaries are established before code extraction begins [13], [23]. Business logic is refactored into independent microservices, while an anti-corruption layer translates legacy models to protect new services from monolithic coupling [13], [23]. Without disciplined pruning of migrated code, the legacy application can become a zombie monolith that retains high memory and startup costs despite reduced functionality [3], [30]. Active removal of obsolete modules is therefore essential to realize architectural gains [3].

6.3 Phase 3: Shared Database Coexistence

During intermediate stages, extracted services may temporarily operate against the legacy shared schema under controlled ownership rules [12], [3]. Each table or data domain is assigned to a single owning service as the authoritative writer, while broader read access may remain [13]. Validation, optimistic locking, and controlled schema changes reduce the risk of corruption during coexistence [3]. As boundaries shift, application-side joins may replace database joins, increasing network chatter and latency [13], [3]. This reinforces the need to extract domains with limited data coupling first [3].

6.4 Phase 4: Event Driven Integration

As service boundaries stabilize, integration shifts toward an event-driven model centered on Kafka [15], [16]. Topics are designed around domain events rather than technical messages, ensuring meaningful business semantics [25]. Simple workflows favor decentralized choreography, while complex or compliance-sensitive processes benefit from centralized Saga orchestration [5], [6], [26]. Event propagation introduces a temporal lag that can create short-lived read-your-writes inconsistencies [6], [25]. Mitigation relies on idempotent consumers, cautious caching, and user experience designs that tolerate brief delays rather than assuming immediate consistency [6].

6.5 Phase 5: Database Per Service Transition

Once services demonstrate stable behavior under shared schema coexistence, data is incrementally partitioned toward a database-per-service model [13], [9]. Data replication or synchronization mechanisms are introduced only where necessary to preserve cross-service access while eliminating direct schema sharing [3]. Cutover planning emphasizes staged migration, allowing subsets of data or workloads to move first while maintaining compatibility with remaining legacy components [3]. This gradual transition reduces operational risk and preserves rollback options if inconsistencies emerge, ultimately

granting each service independent schema evolution [9].

6.6 Phase 6: Monolith Decommissioning

The final phase focuses on systematically retiring modules that have been fully replaced by microservices [3]. Redundant code paths are removed, legacy interfaces are disabled, and residual dependencies are resolved to prevent hidden coupling [30]. Infrastructure components such as WebLogic clusters, associated storage, and legacy messaging providers are gradually shut down once workloads are fully migrated [3]. Compliance and archival requirements are addressed through secure data retention policies, audit trails, and formal documentation to satisfy regulatory obligations in financial environments [2], [30]. Completion of this phase eliminates the operational costs and maintenance burden of the legacy Java EE stack [30].

7. CHALLENGES IN MIGRATION

Migrating WebLogic based monoliths to cloud native Spring Boot microservices introduces technical, operational, and organizational challenges that extend beyond code refactoring [2], [3]. Many difficulties only become visible once functionality is distributed across services, data is partitioned, and execution shifts from a single JVM to a containerized platform [3]. This section synthesizes recurring challenges observed in real-world modernization efforts and outlines mitigation approaches that are consistent with the architectural patterns described earlier in the paper [2].

7.1 Legacy Component Dependencies and Boundary Discovery

Hidden coupling inside the monolith is one of the most persistent barriers to safe decomposition [27], [12], [3]. Deeply embedded EJB lifecycles, JNDI lookups, and shared utility classes create dependencies that are rarely visible until services are extracted [3], [30]. Stateless EJB logic frequently relies on instance variables that conflict with Spring's default singleton semantics, while stateful EJBs require a complete redesign to externalize conversational state into distributed storage rather than in-memory server state [3], [8]. Similarly, scattered JNDI lookups often fail only at runtime, making migration unpredictable without systematic dependency analysis [3]. Mitigation requires disciplined use of bounded contexts, explicit ownership boundaries, and anti corruption layers to prevent legacy assumptions from leaking into new services while extraction proceeds [13], [23].

7.2 Transactional Semantics and Eventual Consistency

WebLogic historically provided strong ACID guarantees through JTA and two-phase commit across multiple resources [5], [26]. Cloud native architectures replace this model with Saga based coordination, which sacrifices global isolation in favor of scalability and resilience [5], [6]. Intermediate business states may become temporarily visible, and failures require compensating actions rather than database rollbacks [6]. This introduces conceptual complexity for developers accustomed to implicit transactional safety [26]. Mitigation depends on clear domain modeling, idempotent processing, explicit compensations, and reliable event publication patterns that align state changes with Kafka messages so that systems can converge consistently without distributed locking [6], [25].

7.3 Data Gravity, Schema Coupling, and Performance Risk

Large, intertwined legacy schemas exhibit strong data gravity that resists rapid partitioning [12], [3]. Premature separation of tightly coupled tables can degrade performance or create reconciliation gaps [13]. During shared database coexistence, loss of database level referential integrity shifts some

enforcement to the application layer, increasing the risk of race conditions [3]. Additionally, replacing database joins with application side joins increases network chatter and latency [13], [3]. A staged transition that begins with controlled table ownership and progresses toward database per service reduces risk while preserving functional correctness [3].

7.4 Security, Sessions, and Network Complexity

Microservices introduce east-west traffic that did not exist inside the monolith, increasing the need for secure and observable communication [2], [30]. Bridging stateful WebLogic sessions with stateless Spring Boot services is particularly problematic when both systems attempt to manage cookies such as JSESSIONID, leading to conflicts or user logouts [3], [30]. Serialization incompatibilities may also emerge when shared session stores are used [3]. Mitigation strategies include migrating authentication toward OIDC and JWT, externalizing session state, or introducing an authentication bridge that translates legacy sessions into tokens consumable by downstream services [7]. At the network level, standardized service identity, mutual TLS, and consistent Kubernetes policies are required to maintain a strong security posture [2], [30].

7.5 Runtime Mismatch, Tooling, and Organizational Load

WebLogic workloads were designed for long-running, memory-intensive JVM processes, whereas Kubernetes assumes lightweight, rapidly restarting pods [3], [22]. Containerizing legacy components without JVM tuning often triggers OOM terminations or readiness failures [22]. Long startup times can also cause restart loops if probes are misconfigured [3]. Addressing this requires container-aware JVM settings, calibrated liveness and readiness probes, and resource sizing aligned with Kubernetes scheduling constraints [22]. Beyond technical challenges, teams experience increased cognitive load as responsibilities expand to include distributed systems design, container tooling, and YAML-based infrastructure [3]. Standardized CI/CD pipelines, shared Helm templates, and clear service ownership reduce this burden while maintaining governance and reproducibility [30].

8. OPERATIONAL READINESS AND PRODUCTION HARDENING

8.1 Resilience Patterns

Distributed systems are inherently vulnerable to partial failures that can cascade across dependencies [18], [21]. To mitigate this risk, services incorporate circuit breakers that detect repeated failures and prevent further calls to unhealthy downstream components [21], [18]. Retry mechanisms with controlled backoff are applied selectively to transient errors while avoiding amplification of load during outages [18]. Bulkhead isolation further constrains failure impact by partitioning resources such as thread pools and connection pools so that degradation in one functional area does not exhaust shared capacity [21], [18]. Together, these patterns shift resilience from implicit platform behavior to explicit application-level control, promoting graceful degradation and preserving system availability under stress [21], [30].

8.2 Observability Framework

Once execution is distributed across services, messaging infrastructure, and data stores, observability becomes essential for both operational stability and auditability [2], [20]. Metrics, logs, and distributed traces provide complementary signals that enable diagnosis of performance degradation and functional failures [2], [20]. Metrics collection is centralized

using Prometheus, with visualization and alerting provided through Grafana [20]. Consistent trace propagation and correlation identifiers allow transaction flows to be reconstructed across service boundaries, supporting root cause analysis and compliance reporting in financial systems [2], [20]. Observability instrumentation is treated as part of the service contract rather than an auxiliary capability [30].

8.3 Deployment Safety

Independent deployment improves delivery velocity but increases the risk of propagating faulty changes without appropriate safeguards [28]. Progressive delivery techniques reduce this risk by decoupling deployment from full exposure [28], [30]. Canary releases limit blast radius by routing a small portion of traffic to new versions before wider rollout, while blue-green deployments maintain parallel production environments to enable rapid rollback [28], [30]. These approaches reduce mean time to recovery by enabling rapid detection and reversal of regressions while preserving system stability under real workloads [28].

8.4 Decommissioning Strategy

Production hardening concludes with a controlled approach to retiring legacy infrastructure [3], [30]. Criteria for sunsetting WebLogic include complete functional replacement, elimination of runtime dependencies, and successful validation of audit and reconciliation workflows within the new architecture [3]. Final data migration ensures that authoritative and historical records are preserved in accordance with retention requirements [2], [30]. Only after sustained operational stability, compliance validation, and rollback contingencies are confirmed is legacy infrastructure formally decommissioned, completing the modernization lifecycle [3].

9. CONCLUSION

This paper has examined an incremental and architecture driven approach for modernizing legacy Java EE monoliths into cloud native microservices while preserving operational stability and regulatory compliance [1], [3], [13], [24]. Rather than advocating wholesale replacement, the proposed methodology emphasizes disciplined decomposition grounded in domain boundaries, explicit data ownership, and controlled migration [9], [13], [23]. By systematically mapping legacy constructs such as JSP, Struts, EJB, MDB, Axis, and iBATIS to cloud native patterns, the work demonstrates how architectural debt can be reduced in a controlled manner without compromising functional correctness, auditability, or business continuity [1], [3], [8].

Several architectural lessons emerge from this analysis. Effective modernization depends more on boundary definition and persistence refactoring than on technology selection, as premature service extraction without addressing shared schema coupling leads to distributed monoliths [3], [8]. Event driven integration and Saga based coordination provide scalable alternatives to global transaction management but require explicit handling of consistency, idempotency, and failure recovery [5], [6], [15], [26], [18], [16]. For regulated industries, these practices deliver long term benefits beyond scalability, including improved traceability, stronger auditability through explicit data flows, and standardized operational controls that support secure and compliant evolution of legacy platforms without recurring large scale rewrites [2], [30], [20]. While this work concentrates on architectural decomposition and operational readiness, related concerns such as deep security enforcement, organizational governance, and platform specific optimizations are intentionally treated as complementary concerns rather than exhaustively addressed [3], [9].

REFERENCES

[1] Y. Abgaz *et al.*, "Decomposition of Monolith Applications Into Microservices Architectures: A Systematic Review," *IEEE Trans. Softw. Eng.*, vol. 49, no. 8, pp. 4213–4242, 2023, doi: 10.1109/TSE.2023.3287297.

[2] V. Velepucha and P. Flores, "A Survey on Microservices Architecture: Principles, Patterns and Migration Challenges," *IEEE Access*, vol. 11, pp. 88339–88358, 2023, doi: 10.1109/ACCESS.2023.3305687.

[3] M. Kalske, N. Mäkitalo, and T. Mikkonen, "Challenges When Moving from Monolith to Microservice Architecture," in *Current Trends in Web Engineering*, I. Garrigós and M. Wimmer, Eds., Cham: Springer International Publishing, 2018, pp. 32–47. doi: https://doi.org/10.1007/978-3-319-74433-9_3.

[4] Oracle Corporation, "Oracle WebLogic Server 14.1.1.0 documentation," Oracle Corporation. [Online]. Available: https://docs.oracle.com/en/middleware/standalone/weblogic-server/14.1.1.0/

[5] M. Djerou and O. Tibermacine, "SAGA Distributed Transactions Verification Using Maude," in *2022 International Symposium on iNnovative Informatics of Biskra (ISNIB)*, 2022, pp. 1–6. doi: 10.1109/ISNIB57382.2022.10076050.

[6] E. Daraghmi, C.-P. Zhang, and S.-M. Yuan, "Enhancing saga pattern for distributed transactions within a microservices architecture," *Appl. Sci.*, vol. 12, no. 12, p. 6242, 2022, doi: 10.3390/app12126242.

[7] Eclipse Foundation, "Jakarta Transactions specification (Version 2.0)," Eclipse Foundation, Inc. [Online]. Available: https://jakarta.ee/specifications/transactions/2.0/

[8] J. Fritzsch, J. Bogner, A. Zimmermann, and S. Wagner, "From Monolith to Microservices: A Classification of Refactoring Approaches," in *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment*, J.-M. Bruel, M. Mazzara, and B. Meyer, Eds., Cham: Springer International Publishing, 2019, pp. 128–141. doi: https://doi.org/10.1007/978-3-030-06019-0_10.

[9] J. Bogner, J. Fritzsch, S. Wagner, and A. Zimmermann, "Microservices in Industry: Insights into Technologies, Characteristics, and Software Quality," in *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*, 2019, pp. 187–195. doi: 10.1109/ICSA-C.2019.00041.

[10] VMware Inc, "Spring Boot reference documentation (Version 3.1)," VMware, Inc. [Online]. Available: https://docs.spring.io/spring-boot/docs/3.1.x/reference/html/

[11] Kubernetes, "Kubernetes documentation (Version 1.28)," Kubernetes. [Online]. Available: https://kubernetes.io/docs/home/

[12] S. Kumari and S. K. Rath, "Performance comparison of SOAP and REST based Web Services for Enterprise Application Integration," in *2015 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, 2015, pp. 1656–1660. doi: 10.1109/ICACCI.2015.7275851.

[13] S.-P. Ma, C.-Y. Li, W.-T. Lee, and S.-J. Lee, "Microservice Migration Using Strangler Fig Pattern and Domain-Driven Design.," *J. Inf. Sci. Eng.*, vol. 38, no. 6, p. 1285, 2022, doi: 10.6688/JISE.2022ll_38(6).0010.

[14] Apache Software Foundation, "Apache iBATIS Java implementation," Apache Software Foundation. [Online]. Available: https://ibatis.apache.org/java.html

[15] T. P. Raptis and A. Passarella, "A Survey on Networked Data Streaming With Apache Kafka," *IEEE Access*, vol. 11, pp. 85333–85350, 2023, doi: 10.1109/ACCESS.2023.3303810.

[16] H. Wu, Z. Shang, G. Peng, and K. Wolter, "A Reactive Batching Strategy of Apache Kafka for Reliable Stream Processing in Real-time," in *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, 2020, pp. 207–217. doi: 10.1109/ISSRE5003.2020.00028.

[17] Apache Software Foundation, "Apache Kafka documentation (Version 3.6)," Apache Software Foundation. [Online]. Available: https://kafka.apache.org/36/documentation.html

[18] P. Di Francesco, P. Lago, and I. Malavolta, "Architecting with microservices: A systematic mapping study," *J. Syst. Softw.*, vol. 150, pp. 77–97, 2019, doi: 10.1016/j.jss.2019.01.001.

[19] D. Ernst and S. Tai, "Offline Trace Generation for Microservice Observability," in *2021 IEEE 25th International Enterprise Distributed Object Computing Workshop (EDOCW)*, 2021, pp. 308–317. doi: 10.1109/EDOCW52865.2021.00062.

[20] S. Vyas, R. K. Tyagi, C. Jain, and S. Sahu, "Performance Evaluation of Apache Kafka – A Modern Platform for Real Time Data Streaming," in *2022 2nd International Conference on Innovative Practices in Technology and Management (ICIPTM)*, 2022, pp. 465–470. doi: 10.1109/ICIPTM54933.2022.9754154.

[21] A. Akbulut and H. G. Perros, "Performance Analysis of Microservice Design Patterns," *IEEE Internet Comput.*, vol. 23, no. 6, pp. 19–27, 2019, doi: 10.1109/MIC.2019.2951094.

[22] M. Fowler, "Strangler fig application," Martin Fowler. [Online]. Available: https://martinfowler.com/bliki/StranglerFigApplication.html

[23] R. A. Schmidt and M. Thiry, "Microservices identification strategies : A review focused on Model-Driven Engineering and Domain Driven Design approaches," in *2020 15th Iberian Conference on Information Systems and Technologies (CISTI)*, 2020, pp. 1–6. doi: 10.23919/CISTI49556.2020.9141150.

[24] C.-Y. Li, S.-P. Ma, and T.-W. Lu, "Microservice Migration Using Strangler Fig Pattern: A Case Study on the Green Button System," in *2020 International Computer Symposium (ICS)*, 2020, pp. 519–524. doi: 10.1109/ICS51289.2020.00107.

[25] C. K. Rudrabhatla, "Comparison of event choreography and orchestration techniques in microservice architecture," *Int. J. Adv. Comput. Sci. Appl.*, vol. 9, no. 8, 2018, doi: 10.14569/IJACSA.2018.090804.

[26] M. Štefanko, O. Chaloupka, and B. Rossi, "The Saga Pattern in a Reactive Microservices Environment," in *Proceedings*

of the 14th International Conference on Software Technologies, in ICSOFT 2019. Setubal, PRT: SCITEPRESS - Science and Technology Publications, Lda, 2019, pp. 483–490. doi: 10.5220/0007918704830490.

[27] A. Krause, C. Zirkelbach, W. Hasselbring, S. Lenga, and D. Kröger, "Microservice Decomposition via Static and Dynamic Analysis of the Monolith," in *2020 IEEE International Conference on Software Architecture Companion (ICSA-C)*, 2020, pp. 9–16. doi: 10.1109/ICSA-C50368.2020.00011.

[28] M. Waseem, P. Liang, and M. Shahin, "A Systematic Mapping Study on Microservices Architecture in DevOps," *J. Syst. Softw.*, vol. 170, p. 110798, 2020, doi: https://doi.org/10.1016/j.jss.2020.110798.