World Journal of Advanced Research and Reviews
eISSN: 2581-9615 CODEN (USA): WJARAI
Cross Ref DOI: 10.30574/wjarr
Journal homepage: https://wjarr.com/

(REVIEW ARTICLE)

Check for updates

Microservices, EDA and ESB Coexistence: An architectural transition model for legacy enterprise systems

Ganesh Kumar Gangannagari *

Independent Researcher.

World Journal of Advanced Research and Reviews, 2024, 21(03), 2728-2734

Publication history: Received on 18 February 2024; revised on 24 March 2024; accepted on 29 March 2024

Article DOI: https://doi.org/10.30574/wjarr.2024.21.3.0834

Abstract

Enterprise software ecosystems increasingly rely on hybrid architectural models that allow modern distributed services to coexist with established integration platforms. The Enterprise Service Bus (ESB) and Event-Driven Architecture (EDA) represent decades of enterprise integration investment, while microservices—implemented through frameworks such as Java Spring Boot and serverless compute platforms like AWS Lambda—deliver the agility and independent scalability that digital transformation demands. A coexistence model enables organizations to introduce microservices capabilities incrementally alongside operational ESB and EDA platforms, preserving prior investment and dramatically reducing the transformation risk associated with wholesale platform decommissioning.

This architectural transition model identifies five core dimensions of the coexistence challenge: the trade-offs between running Spring Boot within serverless Lambda contexts versus containerized deployments; the role of EDA as a decoupling backbone connecting legacy and modern tiers; the incremental migration techniques—most prominently the Strangler Fig pattern applied with Domain-Driven Design—that allow parallel operation of legacy and greenfield services; the governance and observability instruments, including API gateways and service meshes, that enforce cross-tier consistency; and real-world deployment patterns that demonstrate how Spring Boot and AWS Lambda integrate within EDA-mediated hybrid landscapes.

Key outcomes indicate that organizations adopting incremental coexistence models preserve operational continuity, reduce migration-related downtime, and extract immediate business value from new microservices capabilities without abandoning legacy platform functionality. The practical significance lies in a structured, phased transition framework that technology leaders can apply across industries—from financial services to healthcare—where large-scale ESB investments cannot be discarded overnight.

Keywords: Microservices Architecture; Enterprise Service Bus; Event-Driven Architecture; Legacy System Modernisation; Strangler Fig Pattern; Spring Boot; AWS Lambda; Domain-Driven Design; Service Mesh; API Gateway

1. Introduction

1.1. Spring Boot Cold Start Challenges on AWS Lambda

Java Spring Boot has become one of the most widely adopted frameworks for building enterprise microservices owing to its auto-configuration capabilities, embedded application server model, and mature ecosystem of integration libraries. However, its characteristics create specific friction when deployed within serverless environments such as AWS Lambda. Unlike traditional server-hosted deployments where a Spring Boot application remains continuously running and maintains warm connection pools, Lambda functions execute on-demand in response to discrete events—

* Corresponding author: Ganesh Kumar Gangannagari

HTTP requests from API Gateway, messages from Amazon Simple Queue Service (SQS), or state changes published to Amazon Simple Notification Service (SNS). Spring Boot's relatively long startup time means that cold start latency—the delay incurred when Lambda provisions a new runtime instance—can reach several seconds, significantly exceeding the millisecond startup times achievable with lightweight runtimes [1].

Despite these trade-offs, practitioners have developed viable strategies for running Spring Boot within AWS Lambda contexts. AWS provides the aws-serverless-java-container library, which includes a SpringBootLambdaContainerHandler class that bridges Spring Boot's HTTP request-processing model with Lambda's event-based invocation mechanism. This allows existing Spring Boot microservice codebases to be re-packaged as Lambda functions with minimal code changes, enabling teams to extend their established Spring Boot investments into serverless deployment targets. Connection pooling challenges can be addressed through AWS RDS Proxy, which maintains persistent connections on behalf of Lambda functions and delivers them on demand [1].

1.2. Gradual Migration via Load Balancer Traffic Weighting

A production-proven architecture for incrementally migrating Spring Boot microservices to Lambda uses an Application Load Balancer to distribute traffic between a containerized Spring Boot service running on Amazon ECS Fargate and Lambda function target groups handling the same endpoints. Traffic weighting rules in the Load Balancer allow engineering teams to begin routing a small percentage of requests to Lambda while the ECS container handles the remainder, progressively shifting the balance as confidence in Lambda behavior grows. This approach directly implements the Strangler Fig pattern at the infrastructure layer, using the Load Balancer as the routing façade rather than a dedicated proxy service [2].

Spring Cloud Function plays a key enabling role in this migration model. It provides adapters for running Spring-style function beans on AWS Lambda without requiring the explicit handler classes required by the older SpringBootRequestHandler model. The SPRING_CLOUD_FUNCTION_DEFINITION environment variable specifies which function bean a given Lambda deployment should invoke, allowing multiple business operations to coexist as distinct Lambda deployments sharing the same Spring Boot codebase [2].

Table 1 Spring Boot Deployment Contexts and ESB Coexistence Implications [1, 2]

Context	Startup	Scaling	ESB Role
Spring Boot ECS Fargate	Continuous warm	Horizontal autoscaling	Stateful ESB-equiv.
Spring Boot AWS Lambda	Cold start	Per-request scaling	ESB mediation handler
Spring Cloud Fn on Lambda	Framework-managed	Per-function scaling	Endpoint migration
Spring Boot + RDS Proxy	Warm pooled conns	Container or Lambda	Legacy data parity
ECS + Lambda via LB	Warm + on-demand	Traffic-weighted	Strangler Fig façade

2. Event-driven architecture as the decoupling backbone in hybrid systems

2.1. EDA Fundamentals: Loose Coupling Across Service Tiers

Event-Driven Architecture provides the communication substrate that makes microservices and ESB coexistence architecturally coherent. In an EDA model, system components detect state changes—business actions, data updates, transaction completions—encode them as events, and publish those events to a durable message broker. Downstream consumers subscribe to relevant event streams and act independently, without needing synchronous awareness of the producer's identity, location, or operational state. This producer-consumer decoupling is the mechanism that allows legacy ESB-managed systems and new microservices to exchange information without tight runtime coupling [3].

Five component categories define an EDA implementation: events, which encode state changes; producers, which generate and publish event payloads; consumers, which subscribe to streams and execute business logic in response; channels, through which events travel between producers and consumers; and brokers, which provide durable event storage, routing, and delivery guarantees. Popular message brokers in enterprise EDA contexts include Apache Kafka, RabbitMQ, and AWS-native services such as Amazon EventBridge and Amazon SQS [3].

2.2. EDA in Microservices-Driven Bounded Context Migration

When organizations decompose legacy ESB-managed monolithic flows into microservices bounded contexts, EDA serves as the integration mechanism that maintains coherence across the transitional architecture. A microservice-driven enterprise architecture model positions EDA as the backbone through which bounded contexts communicate—replacing the centralized orchestration role previously performed by the ESB with distributed, event-mediated choreography. As each bounded context is extracted into a microservice, it assumes responsibility for producing events to a shared broker, while other microservices and remaining ESB-managed systems subscribe to those events. The broker absorbs the coordination complexity that the ESB formerly owned [3].

Services operating through asynchronous event channels can scale independently—a critical advantage in enterprise environments where different business domains experience radically different load profiles. If an order-processing microservice encounters a traffic spike, the message broker absorbs the volume as a buffer, allowing the consumer to process events at its own pace without requiring the entire pipeline to scale uniformly. This buffering mechanism removes single points of failure and improves overall system fault tolerance compared to synchronous ESB request-response flows [3].

Table 2 Core EDA Components and Their Hybrid Architecture Roles [3, 4**]**

Component	Function	Coexistence Role
Event	State change payload	Carries ESB data to modern consumers
Producer	Generates events	Legacy ESB publishes to microservices
Consumer	Subscribes async	Microservices consume independently
Channel	Routes events	Temporal decoupling across tiers
Broker	Durable delivery	Replaces ESB orchestration hub
Dead-letter queue	Captures failures	Legacy retry and audit trail

3. Strangler fig pattern — domain-driven incremental migration

3.1. Strangler Fig Pattern Foundations and Motivation

The Strangler Fig pattern, first described by Martin Fowler in 2004, takes its name from the tropical plant that grows around an existing tree, gradually assuming structural support until the original tree is no longer needed and can be removed. Applied to software modernization, the pattern describes the incremental replacement of legacy system functionality with new microservices. New capabilities are built alongside the operating legacy system, and traffic is progressively redirected from legacy handlers to modern microservices—function by function—until the legacy system no longer receives meaningful requests and can be safely decommissioned [8].

The pattern is particularly valuable in enterprise ESB contexts because it eliminates the risk associated with "big bang" migrations—wholesale rewrites that attempt to replace the entire legacy platform in a single release. The Strangler Fig approach delivers business value incrementally with each migrated function, reduces the scope of any individual failure, and maintains the legacy ESB as an always-available fallback for unprocessed traffic. This characteristic directly addresses the enterprise constraint that mission-critical systems cannot tolerate unplanned downtime during modernization [8].

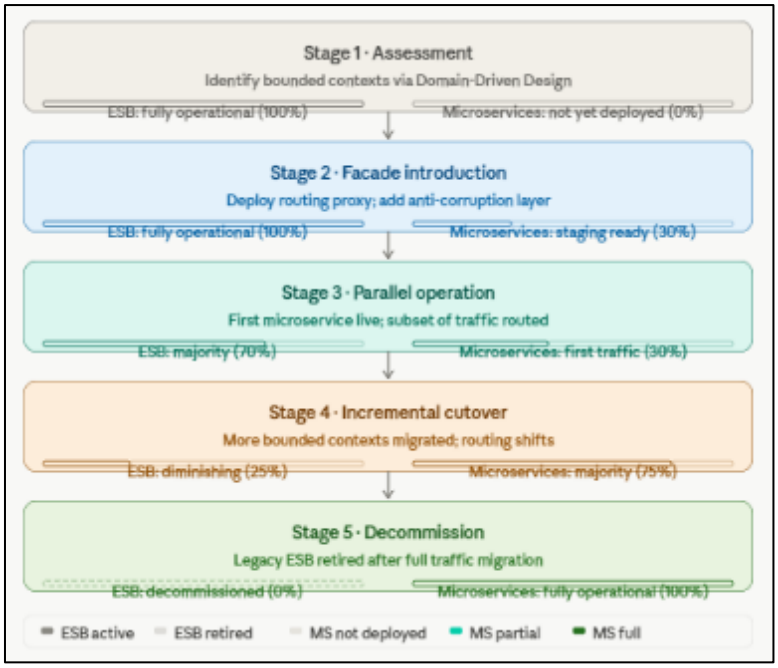

Figure 1 Strangler Fig: Five Migration Stages [5, 6]

3.2. Domain-Driven Design as Migration Boundary Framework

The combination of the Strangler Fig pattern with Domain-Driven Design provides a structured methodology for determining which legacy ESB-managed functions to extract first and how to define the boundaries of each extracted microservice. Domain-Driven Design decomposes an enterprise system into bounded contexts—cohesive clusters of business logic with well-defined interfaces—that map naturally to microservice boundaries. The IEEE case study on the Green Button System demonstrates a complete architecture migration process using Domain-Driven Design as the primary boundary definition instrument [4].

Anti-corruption layers play a critical role in Domain-Driven Design-guided Strangler Fig migrations, preventing the bounded contexts of new microservices from being polluted by the data models and protocols of the legacy ESB. An anti-corruption layer translates between the legacy ESB's canonical data model and the cleaner domain model of the new microservice, ensuring each extracted bounded context remains internally consistent while still interoperating with unextracted legacy functions [4].

Table 3 Strangler Fig Migration Stages in ESB-to-Microservices Transition [5, 6]

Stage	Activity	ESB Status	µServices
1: Assessment	DDD bounded contexts	Fully operational	Not deployed
2: Façade Intro	Deploy routing proxy	Fully operational	Staging prep
3: Parallel Ops	First microservice live	Unextracted flows	First traffic
4: Cutover	Migrate bounded ctxts	Diminishing traffic	Growing share
5: Decommission	Remove legacy ESB	Decommissioned	Fully active

4. API gateways and service meshes for hybrid governance

4.1. The API Gateway as Enterprise Integration Mediator

Microservices architectures require governance instruments capable of managing communication at both the external boundary of the enterprise and within the microservices fabric itself. The API gateway addresses the external governance challenge, providing a single entry point for client requests across the hybrid portfolio—routing them to

either legacy ESB endpoints or new microservices depending on which functions have been migrated. A comprehensive survey categorizes the API gateway as a core structural component responsible for cross-cutting concerns including authentication, authorization, rate limiting, request transformation, and usage monitoring [7].

Key migration challenges that the API gateway must address during ESB-to-microservices transitions include: service decomposition decisions that affect API surface design, data management inconsistencies between legacy ESB canonical data models and microservice-specific schemas, and the complexity of maintaining consistent API versioning across a hybrid portfolio. The API gateway in a coexistence architecture effectively assumes the request routing role previously performed by the ESB's internal mediation engine, but distributes it across a lightweight infrastructure component rather than a heavyweight centralized bus [7].

4.2. Service Mesh and Zero-Trust Governance in Microservices Tiers

While the API gateway governs north-south traffic crossing the enterprise boundary, a service mesh addresses governance within the microservices tier. A service mesh deploys lightweight proxy sidecars alongside each microservice instance, transparently intercepting all network traffic between services and applying policies—mutual TLS encryption, circuit breaking, retry logic, traffic shaping, and distributed tracing—without requiring modification to service code. This division of responsibility mirrors the layered governance model that ESB-centric architectures provided in monolithic integration landscapes [10].

Service mesh platforms such as Kong Mesh implement a zero-trust security posture within the microservices fabric, performing extensive cryptographic work to establish secured connections between services across distributed clusters and multiple availability zones. Ingress gateways act simultaneously as modern firewalls and routers, enforcing access policies at varying levels of granularity—from platform-scoped policies applicable to entire service groups to fine-grained microservice-scoped policies governing individual service-to-service flows [10].

Table 4 Governance Instruments in Microservices and ESB Coexistence Architectures [7, 8]

Instrument	Scope	Capabilities	Coexistence Role
API Gateway	North-south	Auth, rate limit, routing	Abstracts ESB/μsvc topology
Service Mesh	East-west	mTLS, circuit break, trace	Inter-service governance
ESB Policy Eng.	Legacy hub	Protocol mediation, audit	Unextracted flow control
Micro-gateway	Zone boundary	Validation, edge routing	API gateway egress check
Distributed Trace	All tiers	Correlation, latency	Hybrid flow visibility

5. EDA patterns for message streaming in microservices deployments

5.1. Event-Driven Message Streaming Patterns for High-Volume Microservices

Event-Driven Architecture patterns for message streaming address a specific class of enterprise integration challenge: coordinating microservices that must process high volumes of data at low latency without introducing synchronous coupling between components. Three core EDA integration patterns apply to microservices environments handling large-scale data streams: (1) the publish-subscribe pattern enables multiple microservice consumers to receive the same event stream from a single producer; (2) the event sourcing pattern persists all state changes as an immutable sequence of events; and (3) the Command Query Responsibility Segregation (CQRS) pattern separates write operations from read operations handled by query microservices [5].

Apache Kafka and RabbitMQ serve as the primary broker implementations for these patterns in enterprise microservices deployments. Kafka is particularly suited to event sourcing and high-throughput publish-subscribe contexts owing to its durable, ordered, and replayable log-structured storage model. RabbitMQ offers flexible routing through its exchange-binding model, making it well-suited to content-based routing scenarios. In ESB coexistence architectures, these brokers assume the message mediation responsibilities previously performed by the ESB's internal messaging engine, eliminating the ESB's single point of failure while preserving its mediation semantics [5].

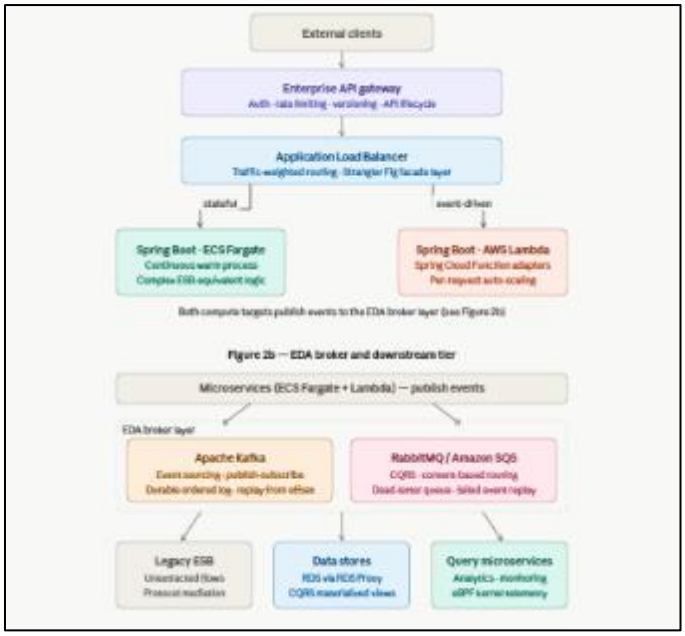

Figure 2 Hybrid compute tier [9, 10]

5.2. Service Mesh eBPF Integration for Microservices Observability

The integration of extended Berkeley Packet Filter (eBPF) technology with service mesh architectures addresses one of the principal operational challenges of EDA-mediated microservices deployments: achieving comprehensive observability across a large, dynamically scaled service fabric without incurring the performance overhead of traditional sidecar proxy models. eBPF allows service mesh functionality—traffic interception, telemetry collection, policy enforcement—to execute directly in the Linux kernel, eliminating per-request latency from user-space sidecar proxies [6].

The Sedghpour and Townend survey identifies that eBPF-enhanced service meshes enable kernel-level traffic capture across the entire microservices fabric without requiring code changes in individual services, making it possible to instrument both newly deployed microservices and services migrated from ESB-managed workflows within the same observability framework. This capability is critical during ESB coexistence transitions, where engineering teams must maintain comparative visibility into both legacy ESB transaction flows and new event-driven microservices flows to validate functional equivalence before decommissioning legacy components [6].

Table 5 EDA Message Streaming Patterns and Microservices Deployment Roles [9, 10]

EDA Pattern	Broker	µService Role	Coexistence Function
Publish-Subscribe	Kafka, SNS	Multi-consumer fanout	Subscribe to ESB events
Event Sourcing	Apache Kafka	Immutable event log	Audit trail in migration
CQRS	RabbitMQ, SQS	Separated write/read	Decouples command flows
Content-Based Routing	RabbitMQ exchange	Payload-driven consumer	Legacy or modern routing
Dead-Letter Queue	SQS, RabbitMQ	Failed-event replay	Legacy fallback preserve

6. Conclusion

The five sections of this article collectively demonstrate that coexistence—rather than forced wholesale replacement—constitutes the most pragmatic and risk-controlled path for organizations modernizing large legacy integration estates. Enterprise Service Buses carry decades of integration logic, compliance audit trails, and protocol mediation capabilities that cannot be abandoned without significant operational and regulatory risk.

Spring Boot's dual applicability—as a continuously running containerized service on ECS Fargate and as a Lambda-packaged function invoked on demand—gives organizations a flexible implementation spectrum for microservices that coexist with ESB-managed workflows. The Application Load Balancer traffic-weighting architecture, combined with Spring Cloud Function adapters, makes incremental endpoint-by-endpoint migration from ESB to Lambda practically achievable for Java development teams.

The Event-Driven Architecture layer—operating through brokers such as Apache Kafka, RabbitMQ, and Amazon EventBridge—provides the temporal decoupling mechanism that allows legacy and modern service tiers to exchange information without tight runtime dependencies. Publish-subscribe, event sourcing, and content-based routing patterns replicate and extend the mediation semantics of the ESB in a distributed, horizontally scalable form.

The Strangler Fig pattern, guided by Domain-Driven Design bounded context decomposition and protected by anti-corruption layers, delivers a structured incremental migration framework. API gateways and eBPF-enhanced service meshes extend governance and observability across the hybrid estate, ensuring that security, compliance, and monitoring standards apply consistently to both legacy and modern service tiers throughout the transition. Technology leaders should invest in this event-driven coexistence architecture—governed by incremental migration rigor and bounded by engineering capacity—as the durable path to enterprise modernization.

References

[1] S. Hesse, "Using Spring Boot on AWS Lambda: Clever or dumb?" sebastianhesse.de, Feb. 2021.

[2] B. Foster, "From Spring Boot microservices to Lambda functions—a journey," ben-foster.dev, Aug. 2021.

[3] A. Chavan, "Exploring event-driven architecture in microservices: Patterns, pitfalls and best practices," Int. J. Sci. Res. Archive, vol. 4, no. 1, pp. 229–249, 2021. doi: 10.30574/ijsra.2021.4.1.0166

[4] Y. Li, J. Ma et al., "Microservice migration using Strangler Fig pattern: A case study on the Green Button System," in Proc. IEEE CSCloud, 2021.

[5] A. Singh, V. Singh, A. Aggarwal, and S. Aggarwal, "Event-driven architecture for message streaming data-driven microservices systems," in Proc. ICISTSD, pp. 308–313, 2022. doi: 10.1109/ICISTSD56354.2022.10100603

[6] M. R. S. Sedghpour and P. Townend, "Service mesh and eBPF-powered microservices: A survey and future directions," IEEE Access, vol. 10, 2022. doi: 10.1109/ACCESS.2022.3208009

[7] V. Velepucha and P. Florez, "A survey on microservices architecture: Principles, patterns and migration challenges," IEEE Access, vol. 11, 2023. doi: 10.1109/ACCESS.2023.3305687

[8] C. Richardson, "The Strangler Fig application pattern," microservices.io, Jun. 2023.

[9] A. Ayyagari et al., "Optimizing large-scale data processing with asynchronous techniques," Int. J. Novel Res. Dev., vol. 8, no. 9, Sep. 2023.

[10] Kong Inc., "API gateway & service mesh: Bridging the gap between API management and zero-trust," Kong Blog, Oct. 2023.