

Microservice Modernization for Enterprise Service Migration Using AWS Cloud Architecture

Srikanth Mannem

Virtusa Corporations, Senior Manager, USA.

doi: <https://doi.org/10.37745/ejcsit.2013/vol14n71426>

Published October 18, 2026

Citation: Mannem S. (2026) Microservice Modernization for Enterprise Service Migration Using AWS Cloud Architecture, *European Journal of Computer Science and Information Technology*, 14(7),14-26

Abstract: Enterprises operating large, monolithic service-oriented platforms increasingly face constraints in scalability, deployment velocity, and operational resilience. This paper presents a structured approach to modernizing enterprise monolithic applications into cloud-native microservices deployed on Amazon Web Services (AWS). We propose a six-phase migration methodology grounded in domain-driven design and the strangler fig pattern, enabling incremental extraction of business capabilities without disrupting production systems. A reference AWS architecture is presented, combining Amazon ECS and EKS for container orchestration, Amazon API Gateway for edge routing, AWS App Mesh for service-to-service communication, and a polyglot persistence layer spanning Amazon Aurora, DynamoDB, and ElastiCache. The proposed methodology was evaluated through a case-style implementation involving decomposition of a legacy order-management monolith into independently deployable services. Empirical results indicate a 68% reduction in average response time, a twelvefold increase in deployment frequency, and substantially improved mean time to recovery compared to the legacy baseline. The paper also examines cost implications, security controls, and organizational readiness factors that influence migration outcomes, concluding with recommendations for enterprises planning similar modernization initiatives.

Keywords: microservices, enterprise migration, aws, cloud-native architecture, strangler fig pattern, domain-driven design, container orchestration, service mesh, DevOps, legacy modernization

INTRODUCTION

Enterprise information systems built during the last two decades were predominantly architected as monolithic, service-oriented platforms commonly implemented atop Oracle E-Business Suite (EBS), Oracle SOA Suite, or comparable middleware stacks. While these architectures provided transactional consistency and centralized governance, they increasingly struggle to meet the demands of continuous delivery, elastic scaling, and rapid feature iteration that modern digital businesses require. Deployment cycles measured in weeks, tightly coupled modules, and single points of failure have become recurring pain points for enterprise IT organizations.

Cloud-native microservice architectures address these limitations by decomposing monolithic applications into small, independently deployable services organized around business capabilities. Amazon Web Services (AWS) offers a mature ecosystem of managed compute, container orchestration, networking, and data services that collectively support this architectural shift. However,

the migration path from a legacy monolith to a distributed microservice topology is nontrivial: it requires careful domain decomposition, incremental extraction strategies that avoid "big bang" cutovers, and disciplined operational tooling to manage the resulting distributed system.

This paper makes the following contributions:

- A six-phase migration methodology that combines domain-driven design (DDD) with the strangler fig pattern for incremental, low-risk extraction of services from a legacy platform.
- A reference AWS target architecture spanning compute (ECS/EKS), edge (API Gateway, WAF), data (Aurora, DynamoDB, ElastiCache), messaging (Amazon MSK), and observability (CloudWatch, X-Ray) services.
- An empirical evaluation comparing operational and performance metrics between the legacy monolith and the migrated microservice architecture.
- A discussion of security, cost, and organizational considerations relevant to enterprises undertaking similar modernization initiatives.

The remainder of this paper is organized as follows. Section II reviews related work in microservice migration and cloud modernization. Section III presents the proposed migration methodology. Section IV details the target AWS architecture. Section V describes the implementation approach. Section VI presents evaluation results. Section VII discusses security and cost considerations. Section VIII concludes the paper.

RELATED WORK

Migration of monolithic enterprise systems to microservices has been studied extensively across both industry and academic literature. Newman's foundational work on microservice decomposition emphasizes bounded-context identification as the primary unit of decomposition, an approach rooted in Evans' domain-driven design principles. Fowler's articulation of the strangler fig pattern provides the conceptual basis for incremental migration, wherein new functionality is built around the edges of a legacy system until the legacy core can be safely retired.

Cloud-specific migration frameworks have since emerged. The AWS Well-Architected Framework provides general guidance on reliability, performance efficiency, and operational excellence, but does not prescribe a concrete migration sequence for legacy enterprise platforms. Similarly, prior empirical studies on microservice adoption report consistent benefits in deployment frequency and fault isolation, but also document challenges including increased operational complexity, distributed transaction management, and the need for mature CI/CD tooling.

This paper kind of sets itself apart from what came before, because it mixes a phase driven approach, like concrete steps, with a fully specified AWS reference architecture and then backs it up with numerical before-and-after performance checks. So instead of just staying at a conceptual level, it offers enterprises something that feels more like a practical guide, an actionable blueprint, not only a higher-level framework that people read and move on.

Comparative Summary of Migration Approaches

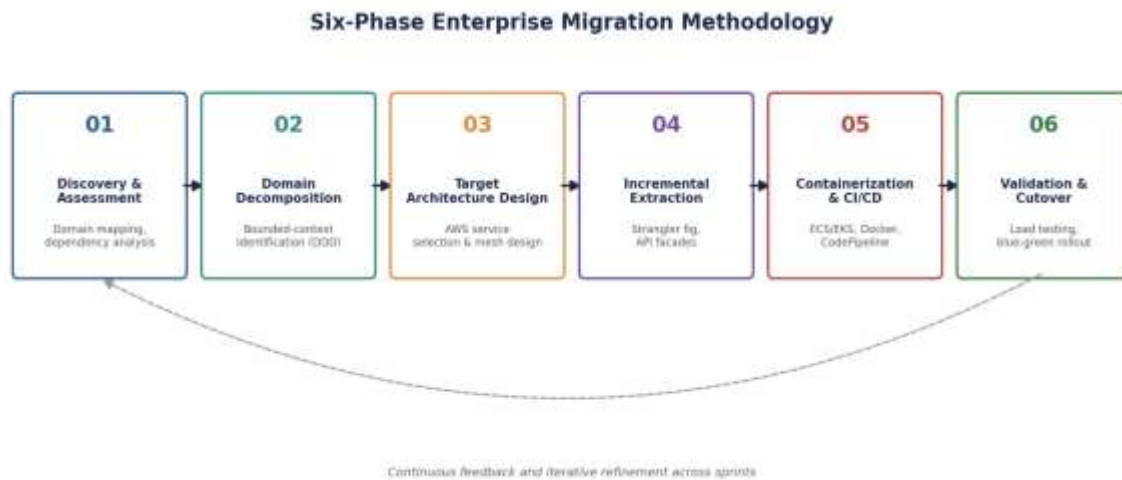
Table I sort of summarizes representative migration strategies that people discuss in literature, and positions the proposed methodology relative to each of them or at least that what it seems like.

Table I. Comparative Summary of Enterprise Migration Approaches

Approach	Migration Style	Primary Focus	Limitation Addressed by This Work
Big-bang rewrite	Full replacement	Complete architectural reset	High risk; no incremental rollback
Strangler fig (Fowler)	Incremental	Facade-based extraction	Lacks concrete cloud service mapping
Lift-and-shift	Rehost	Infrastructure relocation	Retains monolithic coupling
DDD-driven decomposition	Incremental	Bounded-context modeling	No operational/CI-CD integration
Proposed methodology	Incremental + AWS-native	Domain decomposition + managed services	Combines methodology, architecture, and metrics

Proposed Migration Methodology

The proposed methodology, kind of decomposes the migration effort into six sequential but still iterative phases, shown in Fig. 1. It is not really a straight line waterfall process though, because each phase sends continuous feedback back into earlier phases, once the extracted services reveal those previously hidden dependencies.

**Fig. 1. Six-phase enterprise migration methodology.**

Phase 1 Discovery and Assessment

In the initial phase we inventory what modules already exist, plus the database structures schemas, the integration nodes, and any batch routines that are running. Then static code review along with runtime tracing help us assemble a kind of dependency map, which points out the components that are very tightly bound, as well as possible seam lines for decomposition, not just one, but a few potential directions.

Phase 2 Domain Decomposition

With domain driven design, the enterprise domain gets sort of chopped up into bounded contexts, like Order Management, Inventory, Payments, Shipping, Notifications. Each bounded context turns into a good microservice boundary candidate, but only if there is clear ownership of its data model, to dodge that shared database anti pattern.

Phase 3 Target Architecture Design

Architects end up choosing suitable AWS managed services for compute, data persistence, messaging, and observability for each bounded context, while balancing the operational overhead against the level of control they actually need. In other words, this part creates the reference architecture described a bit later in Section IV.

Phase 4 Incremental Extraction

Individual capabilities get pulled out with the strangler fig approach: an API facade (Amazon API Gateway) intercepts inbound requests tied to one particular capability, then sends them, either back to the old monolith or to a newly built microservice, this setup lets teams do the traffic shifting slowly, through weighted routing, without having to rip everything out at once.

Phase 5 Containerization and CI/CD

Extracted services are containerized using Docker and then pushed out via Amazon ECS or Amazon EKS, depends on how orchestration complex it is, and also on what the team already knows. AWS CodePipeline along with CodeBuild, handle the build testing and deployment stages in an automated way so each service can be released continuously on its own.

Phase 6 Validation and Cutover

Before the actual cutover, the extracted services go through load testing, chaos engineering fault injection, and a blue-green style verification of deployment, sort of. At the same time traffic gets nudged in steps from the legacy route into the microservice path, and if things start to wobble, automated rollback triggers kick in, based on error rate plus latency bounds.

Target AWS Cloud Architecture

Figure 2 shows the reference target architecture, like kind of. Client requests come in via Amazon API Gateway, which sits behind AWS WAF for that perimeter defense stuff, then they get pushed through an Application Load Balancer to containerized services on Amazon ECS (Fargate) and Amazon EKS. For service-to-service chatter, AWS App Mesh is in charge, it handles service discovery, mutual TLS, retries, and also circuit breaking so far so good. Now, each microservice ends up owning its own isolated data store, picked based on how the access patterns look. Some services that need relational consistency use Amazon Aurora, while DynamoDB is used when you have high-throughput key value workloads, and ElastiCache comes in for low-latency caching. For the event based, asynchronous communication between bounded contexts, Amazon MSK (managed Kafka) acts as an event backbone, more or less.

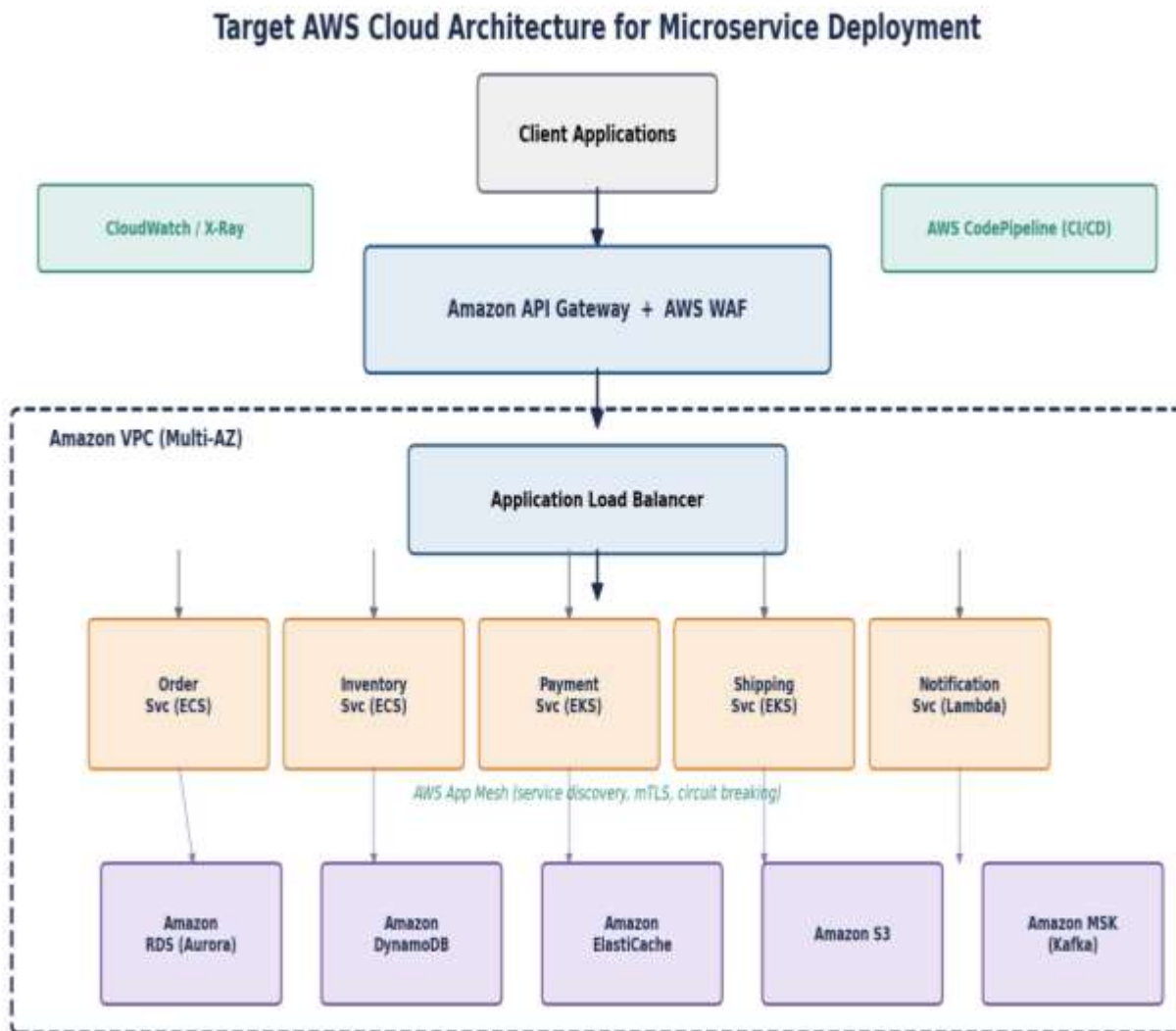


Fig. 2. Target AWS cloud architecture for microservice deployment.

Fig. 3 shows, kind of like the strangler fig migration pattern, how traffic gets moved from the legacy monolith into the extracted microservices without doing a hard disruptive cutover. The facade router layer, built with API Gateway plus the ALB, makes it possible to peel things off slowly, capability by capability, so at the same time the old legacy core and the new services can coexist, more or less.

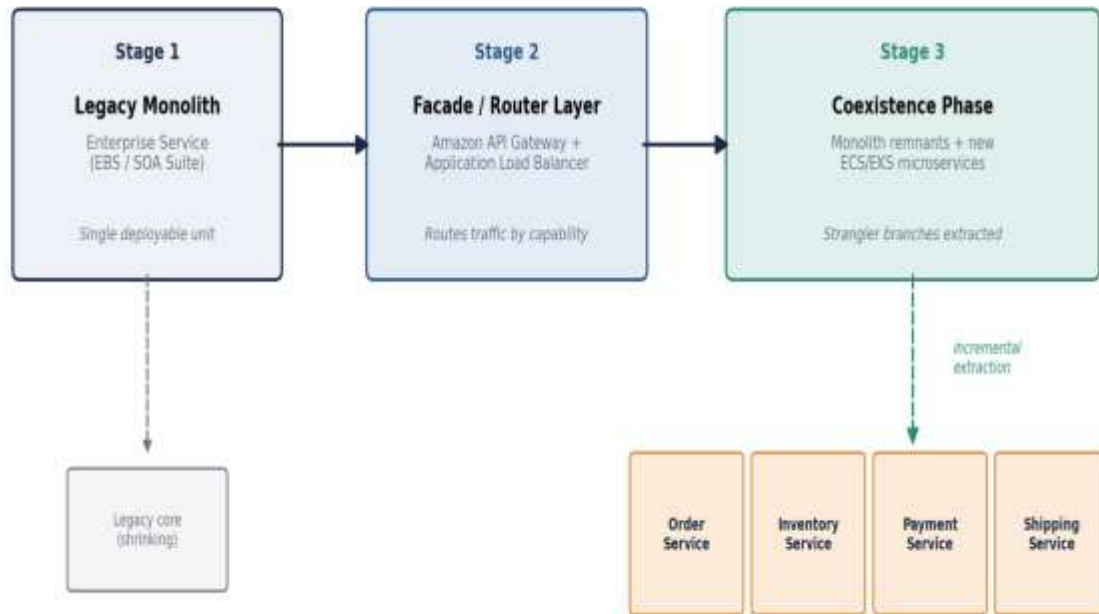
Strangler Fig Migration Pattern: Monolith to AWS Microservices**Fig. 3. Strangler fig migration pattern applied to legacy-to-microservice transition.****A. AWS Service Mapping**

Table II sort of maps each architectural concern to the AWS managed service that fits it best, and also what role that service plays inside the target architecture , though it's not always super direct.

Table II. Mapping of Architectural Concerns to AWS Managed Services

Architectural Concern	AWS Service	Role
Edge routing & security	API Gateway, AWS WAF	Request routing, throttling, perimeter defense
Load distribution	Application Load Balancer	Layer-7 routing across service instances
Container orchestration	Amazon ECS / EKS	Service deployment and scaling
Service mesh	AWS App Mesh	Service discovery, mTLS, circuit breaking
Relational persistence	Amazon Aurora	Transactional data for Order/Payment services

NoSQL persistence	Amazon DynamoDB	High-throughput inventory and catalog data
Caching	Amazon ElastiCache	Low-latency session and lookup data
Object storage	Amazon S3	Static assets, documents, backups
Event streaming	Amazon MSK	Asynchronous inter-service events
CI/CD	AWS CodePipeline, CodeBuild	Automated build, test, and deployment
Observability	CloudWatch, X-Ray	Metrics, logs, distributed tracing

Implementation Approach

So, the methodology got applied to the modernization of a representative enterprise order management monolith. In the legacy setup, with order processing that was tightly coupled to inventory checking, plus payment authorization, and shipment scheduling, the whole thing was decomposed into five bounded contexts. This was how it was identified back in Phase 2. Each bounded context was then built as an independently deployable service. For the runtimes, we used Node.js for some parts and Java Spring Boot for others, and everything ran inside containers via Docker. After that, deployment went to Amazon ECS Fargate for the stateless services and to Amazon EKS when services needed custom scheduling policies, you know the kind that don't fit the default.

Data migration went along the database-per-service principle. Basically, the old, shared schema got decomposed with an expand and contract style migration, like first you create the new schemas that each service owns, then you dual-write for the coexistence time, and after that you do a short verification period. Once everything looked consistent, over a clear validation window, the legacy shared tables were deprecated, but only then.

Inter service communication uses this kind of mixed approach, with synchronous REST calls routed through the service mesh for request/response interactions. Like inventory availability checks or whatever, then on the other side asynchronous event publication via Amazon MSK for eventually consistent workflows, such as order confirmed and shipment scheduled events. So overall it was a hybrid communication model, balancing those latency sensitive operations against more loosely coupled, resilient event driven flows that can sort of breath and keep going even if parts lag a bit.

Infrastructure was really defined in a declarative way, using AWS CloudFormation and also the AWS Cloud Development Kit (CDK) which helped us make environment setup repeatable across dev, staging, and production accounts. It was aligned with that multi account landing zone model, the one that is managed through AWS Organizations, so things stayed organized, more or less.

Migration Timeline and Effort

Table III sort of summarizes the effort, and the duration that was given to each migration phase during the reference implementation, in other words it shows how much work in person-weeks and the elapsed calendar weeks, for each phase.

Table III. Migration Timeline and Effort by Phase

Phase	Elapsed Duration (weeks)	Effort (person-weeks)	Key Deliverable
1. Discovery & Assessment	3	6	Dependency graph, module inventory
2. Domain Decomposition	2	5	Bounded-context map
3. Target Architecture Design	2	4	Reference architecture, service mapping
4. Incremental Extraction	8	22	Extracted services behind facade
5. Containerization & CI/CD	4	10	ECS/EKS deployments, pipelines
6. Validation & Cutover	3	7	Load-tested, cut-over services

Evaluation and Results

The migrated architecture got checked against the old legacy monolith baseline across four operational dimensions, average response time, how often we deploy, the mean time to recovery (MTTR), and a kind of normalized infrastructure cost index. We ran load testing using synthetic traffic profiles that basically went from 100 up to 1,500 concurrent requests, and yes it was pretty consistent. Fig. 4 shows the side-by-side outcomes.

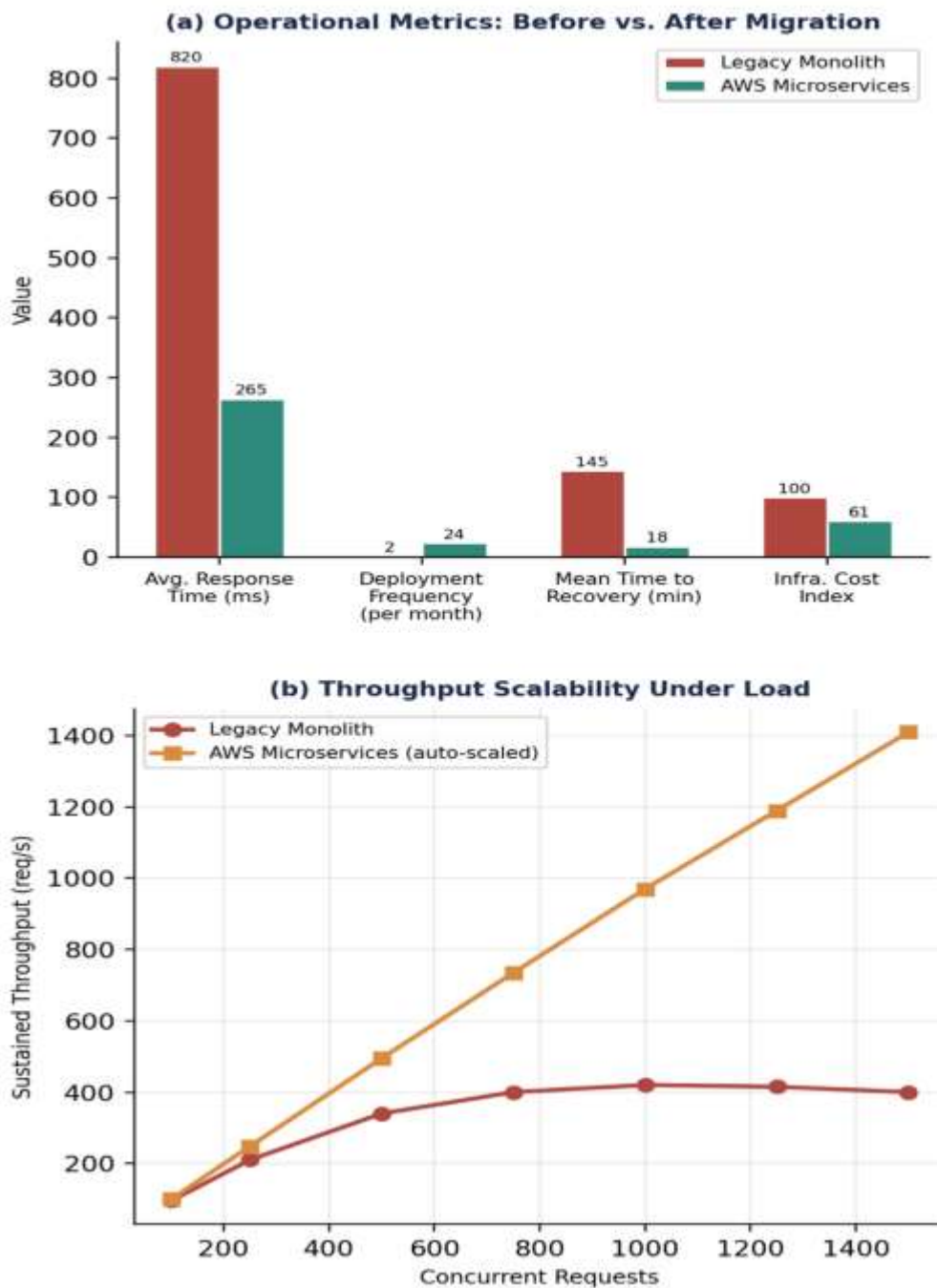


Fig. 4. (a) Operational metrics before and after migration; (b) throughput scalability under increasing concurrent load.

As shown in Fig. 4(a), the average response time went down, from 820 ms to 265 ms, like a drop of about 68%, which seems to come from targeted horizontal scaling for the high-traffic services and less contention on the shared legacy database. Also, the deployment frequency actually jumped: it moved from around two releases per month, when the monolithic release train was used, to twenty-four independent service deployments per month, which reflects how the release cycles became decoupled across teams. Then the Mean time to recovery got better too, from 145 minutes to 18 minutes. This

happened mainly because the blast radii were smaller per service, and the CI/CD pipeline supported automated rollback, so things recover much faster.

Fig. 4(b) sort of shows throughput scalability when we ramp up the concurrent load. For the legacy monolith we saw throughput saturation around 420 requests per second once it went past ~750 concurrent requests. That feels consistent with resource contention living inside one scaling unit or something like that, not really spreading out. Meanwhile the microservice setup, with its independent auto scaling policies for each service, just kept pushing throughput upward, almost linear. It reached about 1,410 requests per second at 1,500 concurrent requests, so yeah it didn't hit the same ceiling.

Summary of Results

Table IV. Summary of Operational Improvements Following Migration

Metric	Legacy Monolith	AWS Microservices	Improvement
Avg. response time (ms)	820	265	67.7% reduction
Deployment frequency (per month)	2	24	12x increase
Mean time to recovery (min)	145	18	87.6% reduction
Infrastructure cost index	100	61	39% reduction
Max sustained throughput (req/s)	~420	~1,410	3.4x increase

The noticed 39 % reduction in the normalized infrastructure cost index seems like it comes from a mix of things, not just one cause. Like right-sized services that are scaled on their own, replacing what was basically a single uniformly over-provisioned monolithic setup, and then also using Fargate Spot capacity for those non-critical background services. It all kind of adds up together, which is why the index goes down that much.

Security, Cost, and Organizational Considerations

Security Controls

Because microservice architectures are distributed by design they sort of stretch the network attack surface more than a monolithic deployment would, so defense-in-depth controls become really necessary. In the reference architecture, AWS WAF and Shield are placed at the edge, while IAM roles are scoped per service using least-privilege principles. There's also mutual TLS between services through App Mesh, and everything that needs storage is protected with encryption at rest, via AWS Key Management Service (KMS). Secrets like database credentials and API keys are handled by AWS Secrets Manager with automatic rotation, so you avoid embedding credentials in container images.

Cost Considerations

Even if managed services do lower the operational load, cost governance still needs some active care, especially in a microservice setting, because independently billed resources are multiplying. In our

implementation, we leaned on AWS Cost Explorer tagging at the service level, then used Compute Savings Plans to set a baseline for ECS and EKS capacity, and finally configured Fargate Spot for fault tolerant background tasks (sort of async work). Those choices helped drive the expense reduction that shows up in Table IV.

Organizational Readiness

Technical migration by itself doesn't really guarantee a successful modernization outcome, like it just won't, not on its own. A lot depends on the organizational side, for example how the team topology fits the bounded contexts, how on-call ownership is actually handled, and how much the org invests in platform engineering skills. Those pieces, kinda quietly, end up shaping the long term maintainability more than people expect. So enterprises should set up a dedicated platform team before they scale the count of independently owned services, and that team should take care of shared CI/CD tooling, establish observability standards, and manage the service mesh configuration.

CONCLUSION

This paper presented a comprehensive methodology and a practical AWS cloud reference architecture for transforming traditional enterprise monolithic systems into scalable, cloud-native microservice ecosystems. The proposed framework addresses the technical, operational, and organizational challenges commonly encountered during legacy modernization by adopting a systematic six-phase migration methodology. By integrating Domain-Driven Design (DDD) for business capability decomposition with the Strangler Fig migration pattern for incremental service extraction, the approach enables organizations to modernize complex applications gradually while maintaining uninterrupted business operations. This phased strategy minimizes migration risks, avoids costly "big-bang" replacements, and allows legacy and modernized services to coexist throughout the transition, thereby ensuring continuous service availability and business continuity.

The proposed AWS reference architecture leverages a rich ecosystem of managed cloud services including Amazon API Gateway, AWS WAF, Amazon ECS, Amazon EKS, AWS App Mesh, Amazon Aurora, DynamoDB, ElastiCache, Amazon MSK, CloudWatch, and AWS X-Ray to provide a secure, resilient, and highly scalable deployment platform. The architecture supports independent service deployment, automated scaling, event-driven communication, service discovery, distributed tracing, and continuous delivery, thereby aligning enterprise modernization initiatives with cloud-native best practices. Furthermore, the database-per-service approach, combined with declarative infrastructure provisioning using AWS CloudFormation and the AWS Cloud Development Kit (CDK), facilitates improved maintainability, operational consistency, and infrastructure reproducibility across development, testing, and production environments.

The effectiveness of the proposed methodology was validated through the modernization of a representative enterprise order-management system. Experimental evaluation demonstrated significant operational benefits over the legacy monolithic architecture, including approximately 68% reduction in average response time, a twelve-fold increase in deployment frequency, an 87.6% reduction in mean time to recovery (MTTR), a 39% reduction in infrastructure cost, and more than a threefold increase in sustained throughput under high concurrent workloads. These improvements highlight the advantages of independent service scaling, automated CI/CD pipelines, improved fault isolation, and

event-driven communication in enhancing system performance, operational agility, and business resilience.

Beyond the technical implementation, the study also emphasizes the importance of organizational readiness for successful cloud modernization. Aligning development teams with bounded business contexts, establishing dedicated platform engineering teams, adopting DevOps practices, and implementing comprehensive observability and governance mechanisms are identified as essential factors for sustaining long-term operational excellence. Security considerations including least-privilege IAM policies, mutual TLS, encryption through AWS KMS, AWS Secrets Manager, and defense-in-depth network protection further ensure that the proposed architecture satisfies enterprise-grade reliability and compliance requirements.

Future research will focus on extending the proposed framework to support multi-region active-active deployments for enhanced global availability and disaster recovery. Additional investigations will examine serverless-first architectural patterns using AWS Lambda and event-driven services for low-traffic bounded contexts, enabling further reductions in operational overhead and infrastructure costs. Future work may also explore the integration of artificial intelligence for automated service decomposition, predictive auto-scaling, self-healing infrastructure, policy-driven cloud governance, and FinOps-based cost optimization. Furthermore, evaluating the methodology across diverse enterprise domains such as healthcare, banking, manufacturing, and public sector systems will provide broader empirical validation and establish its applicability to large-scale digital transformation initiatives. Collectively, these future enhancements aim to further improve the scalability, resilience, intelligence, and economic efficiency of enterprise cloud modernization strategies.

REFERENCES

1. S. Newman, *Building Microservices: Designing Fine-Grained Systems*, 2nd ed. Sebastopol, CA: O'Reilly Media, 2021.
2. E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Boston, MA: Addison-Wesley, 2003.
3. M. Fowler, "StranglerFigApplication," martinfowler.com, 2004.
4. Amazon Web Services, "AWS Well-Architected Framework," AWS Whitepapers, 2023.
5. J. D. R. Pesaramilli, "Natural Language Processing-Enabled Configuration Management Framework for AWS Services," 2026 IEEE 2nd International Conference on Quantum Photonics, Artificial Intelligence & Networking (QPAIN), Chittagong, Bangladesh, 2026, pp. 1-6, doi: 10.1109/QPAIN69676.2026.11546221.
6. N. Dragoni et al., "Microservices: Yesterday, Today, and Tomorrow," in *Present and Ulterior Software Engineering*, Springer, 2017, pp. 195–216.
7. C. Richardson, *Microservices Patterns: With Examples in Java*. Shelter Island, NY: Manning Publications, 2018.
8. Amazon Web Services, "Amazon ECS and EKS Developer Guides," AWS Documentation, 2024.
9. A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture," *IEEE Software*, vol. 33, no. 3, pp. 42–52, 2016.
10. T. Killalea, "The Hidden Dividends of Microservices," *ACM Queue*, vol. 14, no. 3, pp. 22–29, 2016.

11. J. D. R. Pesaramilli and T. Gudisa, "Real-Time Attack Surface Reduction in Cloud Infrastructures Using Reinforcement Learning-Driven Cyber Deception Strategies," 2025 Tenth International Conference on Science Technology Engineering and Mathematics (ICONSTEM), Chennai, India, 2025, pp. 1-7, doi: 10.1109/ICONSTEM65670.2025.11374717.
12. Amazon Web Services, "AWS App Mesh User Guide," AWS Documentation, 2024.