International Journal of Advanced Engineering, Management and Science (IJAEMS)
Peer-Reviewed Journal
ISSN: 2454-1311 | Vol-12, Issue-3; May-Jun, 2026
Journal Home Page: https://ijaems.com/
DOI: https://dx.doi.org/10.22161/ijaems.123.5

A Review of Modern Strategies for Migrating Monolithic Enterprise Applications to Microservices Architecture: Challenges, Patterns, and Best Practices

Sai Sruthi Puchakayala

Software Development Engineer III, Walmart, Centerton, Arkansas

Received: 02 Apr 2026; Received in revised form: 01 May 2026; Accepted: 05 May 2026; Available online: 08 May 2026

Abstract *— Migration restructures enterprise applications through redefinition of service boundaries, data ownership, and coordination routines under distributed execution. Tightly coupled components constrain scaling because shared schemas and execution paths bind functionality across layers and prevent direct isolation. Decomposition, communication, and data separation evolve simultaneously when engineers transform such systems. Structural dependencies interact with runtime behavior and domain semantics during boundary formation, which produces instability at early stages and requires iterative refinement. Coordination overhead increases when network communication replaces local execution and introduces synchronization and fault-handling mechanisms. The work sets a task to explain how these processes interact under conditions of structural resistance and distributed deployment. Analytical review combined with conceptual synthesis is used to reconstruct relationships between decomposition, interaction, and coordination. Heterogeneous studies provide evidence of recurring mechanisms and inconsistent interpretation of their interaction. Migration redistributes complexity across architectural layers and requires continuous adjustment of system configuration. The article will be useful for researchers and practitioners involved in distributed system design and architectural transformation.*

Keywords *— **microservices architecture, monolithic systems, system decomposition, distributed systems, service boundaries, architectural patterns, system migration, software architecture***

I. INTRODUCTION

Shared schemas and execution paths bind components across layers and prevent direct separation during system restructuring. Scaling pressure forces redistribution of functionality across independent services, yet embedded dependencies propagate through data structures and control flows and resist isolation. Migration develops under these conditions as continuous redefinition of boundaries, communication, and coordination. Stability does not appear immediately.

Structural dependencies, runtime interaction patterns, and coordination mechanisms jointly shape migration outcomes because each element influences boundary formation and system behavior under deployment conditions. Alignment between these elements stabilizes service structure, while divergence produces fragmentation and communication overhead.

The purpose is to explain how interaction between structural dependencies, runtime behavior, and coordination mechanisms determines stability and performance of microservices architectures under migration conditions.

The objectives define analytical steps. Identify how dependency structures constrain boundary formation. Analyze how runtime execution and domain semantics modify service definitions during iteration. Evaluate how coordination mechanisms and

architectural patterns influence stability and performance in distributed environments.

Stable migration emerges when decomposition logic reflects runtime interaction and aligns with coordination mechanisms that regulate communication and consistency. Misalignment produces boundary instability, increased latency, and structural reversal when coordination overhead exceeds performance gains.

Existing research isolates decomposition methods, architectural patterns, and performance evaluation. Interaction between these elements remains insufficiently explained. Analytical reconstruction of migration as a connected process addresses this limitation and clarifies how structural, behavioral, and operational mechanisms interact.The novelty of the study lies in conceptualizing migration as a coupled structural, behavioral, and operational process. Existing studies examine decomposition, coordination, and performance separately. The present work integrates these elements into a unified analytical model explaining instability mechanisms and architectural reversibility.

II. METHODS AND MATERIALS

The study applies a structured analytical review combined with conceptual synthesis. The methodology reconstructs relationships between decomposition mechanisms, runtime interaction patterns, and coordination models. The review follows three stages: source identification, analytical filtering, and conceptual integration. Recent studies indexed in Scopus, Web of Science, and IEEE Xplore provide analytical descriptions of system decomposition, distributed coordination, and architectural transformation within a five-year publication window. Search logic combines keyword clusters such as “microservices AND migration,” “monolith decomposition OR service boundaries,” and “distributed systems AND coordination,” which enables identification of works describing structural dependencies, runtime interaction, and operational constraints.

Initial retrieval yields approximately forty sources. Filtering based on analytical depth, relevance to system-level mechanisms, and presence of implementation detail reduces the dataset to fourteen studies. Selection emphasizes descriptions of relationships between components, execution behavior, and coordination processes rather than isolated technical solutions.

Clustering-based decomposition models group components according to interaction density and structural proximity, while runtime-oriented analyses reveal hidden communication paths formed under execution conditions. Pattern-driven approaches coordinate service extraction, interface transformation, and data isolation, and distributed coordination models describe synchronization, consistency management, and fault-handling processes. These mechanisms generate observable effects: divergence of service structures, increase in coordination overhead, variability of performance across deployment conditions, and emergence of intermediate architectural forms.

Heterogeneity appears in variation of system scale, analytical depth, and evaluation methods. Controlled experiments describe isolated mechanisms under simplified conditions, while industrial observations expose complex interactions with incomplete visibility. This variation produces inconsistent explanations of how structural decomposition interacts with runtime behavior and coordination mechanisms.

Comparison of selected studies reveals fragmentation. Structural, behavioral, and operational elements are examined separately, while their interaction remains insufficiently articulated. Interdependence between decomposition, execution, and coordination requires reinterpretation as a connected process.

III. RESULTS

Migration restructures internal system logic through continuous redefinition of service boundaries, data ownership, and coordination routines. Accumulated coupling in legacy components binds modules through shared schemas and execution paths, which prevents direct separation when engineers attempt decomposition. This constraint emerges in systems where historical dependencies span multiple layers and remain embedded in data structures and control flows.

Engineers identify candidate services by combining static code inspection, runtime tracing, and domain modeling, since each source captures only a fragment of interaction structure [1]. Static analysis reveals explicit dependencies encoded in code, runtime observation exposes hidden communication formed during execution, and domain abstractions align components with business processes under conditions of semantic consistency. These layers interact during iterative refinement cycles where engineers adjust boundaries after observing system behavior under deployment conditions. Each iteration introduces new dependency patterns when workload characteristics change. Service definitions shift.

Analytical procedures group components using clustering and graph-based representations that approximate interaction intensity and structural proximity. These methods operate on heterogeneous datasets, where variations in input structure and metric definitions alter grouping outcomes [1]. Under such conditions identical systems produce different service configurations when processed through distinct analytical pipelines. Variability arises from weighting differences between structural and behavioral signals. Engineers interpret clustering outputs as provisional structures subject to validation. The systematization of migration mechanisms is presented below (Table 1).

Table 1. Classification of migration mechanisms and analytical approaches in monolith-to-microservices transformation (compiled by the author based on [1-3])

Analytical Dimension	***Mechanism Type***	***Structural Function***	***Operational Effect***
Decomposition logic	Static analysis	Extract structural dependencies	Reveals code-level coupling
Decomposition logic	Dynamic analysis	Capture runtime interactions	Detects hidden dependencies
Decomposition logic	Domain-driven modeling	Align services with business processes	Improves semantic cohesion
Optimization layer	Clustering algorithms	Group related components	Forms candidate services
Optimization layer	Evolutionary algorithms	Optimize service boundaries	Balances cohesion and coupling
Architectural patterns	Strangler pattern	Gradual replacement of modules	Reduces migration risk
Architectural patterns	Adapter services	Transform interfaces	Enables compatibility
Data architecture	Database per service	Isolate data ownership	Reduces shared-state conflicts
Control layer	AI-driven monitoring	Analyze system behavior	Supports adaptive optimization

Engineers apply structural patterns to coordinate service extraction, interface transformation, and data isolation, since isolated execution of these operations creates inconsistency between communication protocols and ownership boundaries [2]. Incremental replacement maintains system availability by preserving execution continuity while new services absorb functionality. Abrupt restructuring introduces failure points when unresolved dependencies propagate across newly defined interfaces. Patterns align architectural layers under conditions of partial system transformation. Stability increases.

Distributed deployment replaces local calls with network communication, which introduces latency and requires explicit synchronization mechanisms. Engineers configure messaging protocols, transaction coordination, and fault-handling routines to maintain consistency across nodes. Coordination overhead increases with the number of services because each interaction requires

validation and state alignment. These processes transform implicit execution into explicit distributed coordination. System behavior changes. The layered transformation process is illustrated below (Figure 1).

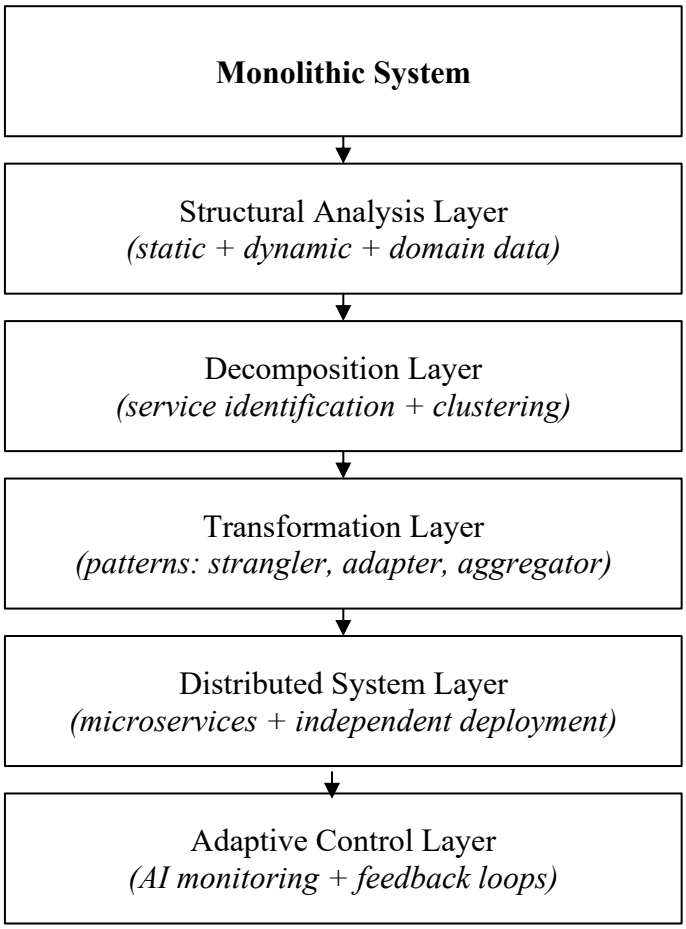

Fig.1. Scheme of layered migration from monolithic architecture to microservices (compiled by the author based on [1,2])

Systems revert to centralized structures when coordination overhead, latency, or operational cost exceed acceptable thresholds [4]. This transition occurs in environments where workload distribution does not benefit from horizontal scaling or where organizational constraints limit effective service management. Engineers evaluate architectural configuration against deployment conditions, including latency sensitivity and infrastructure complexity. Architectural selection shifts.

Machine learning models analyze dependency graphs and execution traces to propose service boundaries through clustering and optimization procedures. These models operate under conflicting objectives: reducing coupling increases fragmentation, while increasing cohesion constrains flexibility [2]. Engineers compare model outputs with operational constraints and adjust boundaries when trade-offs produce instability. Automated analysis accelerates exploration of candidate structures. Ambiguity remains.

Monitoring systems process runtime metrics and detect anomalies through embedded feedback loops that trigger reconfiguration of resource allocation and communication patterns. These mechanisms operate under variable load conditions where static configurations degrade performance [3]. Engineers adjust scaling policies and service placement based on observed deviations in system behavior.

Engineers introduce modular monoliths during early stages to retain centralized execution while enforcing internal boundaries that approximate service decomposition. This configuration limits

communication overhead and simplifies consistency management under controlled conditions [2]. Gradual transition allows validation of structural assumptions before distributed deployment. Risk decreases.

Performance divergence appears when computational load redistributes between local execution and network communication. Monolithic systems maintain efficiency in single-node environments where communication overhead remains minimal, while microservices scale under high concurrency through distributed processing [2]. This relationship depends on workload intensity, deployment topology, and latency constraints. Performance shifts. The comparative behavior of architectures is presented below (Figure 2).

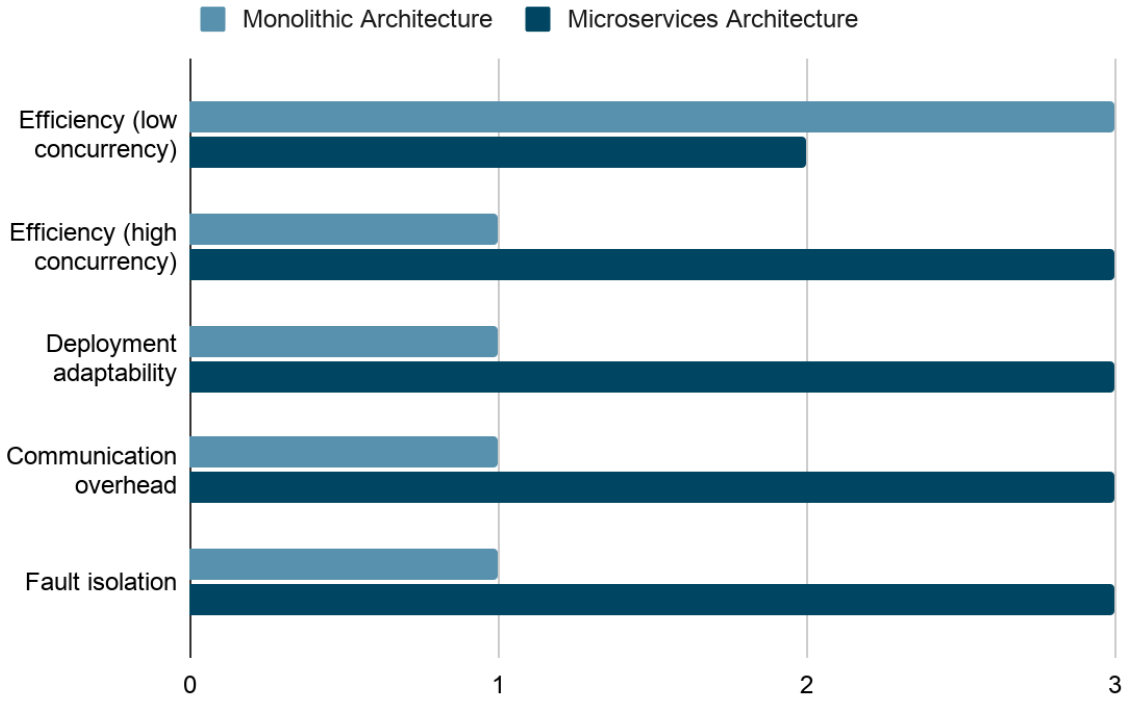

Fig.2. Comparative operational profile of monolithic and microservices architectures under different system conditions (compiled by the author based on [2, 5–8])

Engineers evaluate systems using metrics such as coupling, cohesion, latency, and throughput, yet measurement practices differ across studies. Variability in metric definitions and measurement levels produces inconsistent results and limits comparability. Benchmarking lacks standardized datasets and unified evaluation procedures.

Migration restructures development workflows alongside architecture through redistribution of responsibilities, modification of deployment pipelines, and adaptation of coordination practices. Engineers reorganize teams around service boundaries and adjust operational processes to distributed environments. Limited expertise constrains decision quality because incorrect boundary definitions propagate through infrastructure layers [2]. Errors accumulate.

The study relies on heterogeneous sources with different evaluation designs and experimental conditions, which restricts direct comparison of results. Many approaches are validated in controlled environments rather than large-scale deployments, which limits external validity. Tool ecosystems remain incomplete and integrated analytical frameworks are not fully established [1]. Constraints remain.

Microservices redistribute complexity across operational layers rather than eliminating it.

IV. DISCUSSION

Structural resistance constrains decomposition when shared schemas and execution paths bind components across layers and prevent direct separation. Coupling persists because historical dependencies remain embedded in data models, transaction flows, and infrastructure bindings that are not removed by code-level partitioning. This condition appears in systems with long development cycles and incremental extensions where cross-module interactions accumulate. Boundary isolation breaks. Dependencies re-emerge during integration.

Static inspection exposes explicit links between modules, while runtime tracing reveals additional communication paths formed under execution load, and domain modeling aligns components with business processes only when behavioral data is reconciled with structural representation. These mechanisms operate simultaneously and produce conflicting signals when structural grouping diverges from runtime interaction patterns. Engineers resolve these conflicts by prioritizing observed execution behavior under deployment conditions. Boundary definitions shift after each iteration. Stability remains provisional.

Clustering procedures group components based on interaction density and structural proximity, yet grouping outcomes depend on dataset composition and metric interpretation. Variations in input graphs, weighting strategies, and similarity thresholds alter cluster formation and lead to divergent service candidates. This variability emerges because optimization targets structural cohesion without incorporating execution constraints that appear at runtime. Identical systems yield different decompositions when processed through distinct analytical pipelines. Results diverge. Engineers validate clusters through deployment feedback rather than static evaluation.

Pattern-driven transformation coordinates service extraction with interface mediation and data isolation because independent execution produces inconsistencies between communication protocols and ownership boundaries. Incremental replacement preserves execution continuity by allowing legacy and new components to coexist while responsibilities shift gradually. Abrupt restructuring introduces failure points when unresolved dependencies propagate across interfaces and disrupt transaction flows. Controlled sequencing reduces instability. Transition stabilizes under partial decomposition.

Distributed deployment replaces local procedure calls with network communication and introduces latency, synchronization requirements, and fault-handling routines. Message exchange requires validation and state alignment across nodes, which increases coordination overhead as the number of services grows [6]. This effect appears under conditions of distributed execution where network variability and partial failures influence interaction timing. Engineers configure messaging protocols and consistency models to mitigate these effects.

Reverse transitions occur when coordination overhead, latency, and operational cost exceed performance gains. Systems revert to centralized configurations in environments where workload distribution does not require horizontal scaling or where coordination complexity outweighs benefits [4]. This behavior emerges when service interactions intensify and network communication dominates execution time. Architectural form changes. Distributed structure contracts.

Machine learning models process dependency graphs and execution traces to generate candidate service boundaries through clustering and optimization routines. Optimization objectives conflict: reduction of coupling increases fragmentation, while enforcement of cohesion constrains modular flexibility. These conflicts appear when structural similarity does not align with runtime interaction frequency. Engineers adjust model outputs using deployment constraints and operational requirements. Automation accelerates exploration. Ambiguity persists.

Adaptive monitoring systems process runtime metrics and detect anomalies through feedback loops that trigger reconfiguration of resource allocation and communication patterns. These mechanisms operate under variable load where static configurations degrade performance and require continuous adjustment [3]. Engineers modify scaling policies and service placement based on observed deviations in latency and throughput.

System configuration evolves. Control remains dynamic.

Modular monoliths retain centralized execution while introducing internal boundaries that approximate service decomposition. This configuration limits network overhead and simplifies consistency management under conditions where distributed coordination would introduce excessive complexity. Engineers use this structure to validate boundary assumptions before distributing services across nodes. Transition proceeds incrementally. Risk decreases.

Performance divergence emerges when computational load shifts between local execution and network communication. Local execution maintains efficiency when interaction remains internal and communication cost remains minimal, while distributed execution improves scalability under high concurrency when workload spreads across nodes. This relationship depends on workload intensity, deployment topology, and latency sensitivity. Performance varies across conditions. No configuration dominates.

Evaluation practices differ across studies because metrics such as coupling, latency, and throughput are defined and measured at different system levels. Variation in measurement granularity and experimental setup produces inconsistent results and prevents direct comparison. This inconsistency emerges when benchmarks rely on controlled environments that do not reproduce production conditions. Engineers interpret results within local constraints. Generalization weakens.

Organizational restructuring accompanies architectural transformation when teams redistribute responsibilities around service boundaries and adapt deployment pipelines to distributed environments. Coordination mechanisms between teams replace centralized control structures, which introduces additional communication overhead and dependency management requirements [6]. This effect appears when team structure does not align with service boundaries. Expertise gaps amplify instability. Errors propagate through infrastructure layers.

Source heterogeneity constrains interpretation because studies employ different datasets, evaluation criteria, and experimental designs. Variability in reported results reflects differences in system scale, workload characteristics, and validation environments. Controlled experiments dominate available evidence, while large-scale industrial observations remain limited. External validity decreases. Comparability weakens.

Reliance on secondary analysis introduces methodological constraints because conclusions depend on reported findings rather than direct observation of systems. Differences in metric definitions and reporting practices influence interpretation of mechanisms and obscure causal relationships. Absence of unified evaluation frameworks prevents consistent synthesis across studies. Precision declines. Uncertainty remains.

Microservices migration redistributes complexity across architectural and operational layers through interaction between structural dependencies, runtime behavior, and organizational constraints. Continuous adjustment replaces static design. Stability remains conditional.

V. CONCLUSION

Shared schemas and execution paths bind components across layers and prevent direct boundary isolation during decomposition. Dependency persistence emerges because historical interactions remain embedded in data models, transaction flows, and infrastructure bindings that are not removed by code partitioning. This condition appears in systems with accumulated cross-module interactions and long development cycles. Engineers refine boundaries iteratively after deployment. Dependencies reappear. Stability does not hold.

Runtime execution exposes interaction patterns that differ from static structure, while domain modeling aligns functionality only when behavioral data is incorporated into boundary definition. These mechanisms operate together under changing workloads and produce conflicting signals when structural grouping diverges from observed execution. Engineers prioritize runtime behavior when inconsistencies emerge. Boundaries shift after each iteration. Service structure remains unstable.

Network communication replaces local execution in distributed environments and introduces latency, synchronization requirements, and fault-

handling routines. Each service interaction requires validation and state alignment across nodes, which increases coordination overhead as system scale grows. This effect appears under distributed deployment where interaction frequency intensifies. Engineers configure messaging protocols and consistency models to manage these constraints. Overhead accumulates. Performance degrades under misconfiguration.

Coordination mechanisms align service interaction, data consistency, and resource allocation when decomposition logic reflects runtime behavior and communication patterns. Misalignment emerges when structural separation ignores execution dependencies or coordination requirements, which produces fragmentation and excessive communication cost. Systems revert to centralized structures when coordination overhead exceeds performance gains. Architectural form shifts. Distributed configuration contracts.

The study confirms the hypothesis that migration stability depends on alignment between structural decomposition, runtime interaction patterns, and coordination mechanisms. Misalignment produces boundary instability, increased latency, and architectural reversibility. The findings demonstrate that microservices migration redistributes complexity rather than reducing it.

The practical implication is that migration planning must integrate dependency analysis, runtime observation, and coordination design. Iterative validation of service boundaries reduces instability during distributed deployment. The proposed conceptual model supports architectural decision-making in enterprise system transformation.

REFERENCES

[1] Abgaz, Y., Yalemisew, A., McCarren, A., Elger, P., Solan, D., Lapuz, N., Bivol, M., Jackson, G., Yilmaz, M., Buckley, J., & Clarke, P. (2023). Decomposition of monolith applications into microservices architectures: A systematic review. *IEEE Transactions on Software Engineering*, 1–32. https://doi.org/10.1109/TSE.2023.3287297

[2] Hassan, H., Abdel-Fattah, M., & Mohamed, W. (2024). Migrating from monolithic to microservice architectures: A systematic literature review. *International Journal of Advanced Computer Science and Applications*, 15, 104–116. https://doi.org/10.14569/IJACSA.2024.0151013

[3] Moreschini, S., Pour, S., Lanese, I., et al. (2025). AI techniques in the microservices life-cycle: A systematic mapping study. *Computing, 107*, 100. https://doi.org/10.1007/s00607-025-01432-z

[4] Su, R., Li, X., & Taibi, D. (2024). From microservice to monolith: A multivocal literature review. *Electronics, 13*(8), 1452. https://doi.org/10.3390/electronics13081452

[5] Martínez Saucedo, A., Rodríguez, G., Gomes Rocha, F., & Pereira dos Santos, R. (2025). Migration of monolithic systems to microservices: A systematic mapping study. *Information and Software Technology, 177*, 107590. https://doi.org/10.1016/j.infsof.2024.107590

[6] Zhou, X., Li, S., Cao, L., Zhang, H., Jia, Z., Zhong, C., Shan, Z., & Babar, M. A. (2023). Revisiting the practices and pains of microservice architecture in reality: An industrial inquiry. *Journal of Systems and Software, 195*, 111521. https://doi.org/10.1016/j.jss.2022.111521

[7] Willard, J., & Hutson, J. (2025). The evolution and future of microservices architecture with AI-driven enhancements. *International Journal of Recent Engineering Science (IJRES), 12*(1), 16–22. https://doi.org/10.14445/23497157/IJRES-V12I1P103

[8] Hassan, H., Abdel-Fattah, M. A., & Mohamed, W. (2025). A pattern-based framework for automated migration of monolithic applications to microservices. *Big Data and Cognitive Computing, 9*(10), 253. https://doi.org/10.3390/bdcc9100253