

Modernizing Legacy Systems with Spring Boot Microservices: A Unified Framework for Capital Markets, Travel Logistics, and Financial Services

Mrugendra Rudrappa Madalagi

Technical Architect, Mankind America LLC

Irving, Texas, USA, murgesh.madalagi@gmail.com

Abstract

Legacy enterprise platforms in capital markets, travel logistics, and financial services often combine valuable domain logic with tightly coupled deployment, aging integration protocols, and constrained release practices. This article proposes a unified, incremental modernization framework using Spring Boot microservices, domain-driven design, API facades, event-driven integration, automated delivery, and production observability. The framework separates business capabilities while preserving transaction controls, auditability, and compatibility with existing systems. It combines six stages: discovery, domain decomposition, façade-based coexistence, data and event migration, progressive release, and operational optimization. A quantitative assessment model is defined through 15 equations covering service cohesion, coupling, migration priority, risk, latency, availability, recovery, cost, and weighted quality. Two architecture/method figures and five multicolour charts illustrate how the framework can be evaluated. The charts use explicitly hypothetical values rather than measurements from a deployed organization: in the illustrative scenario, p95 latency declines from 420 ms to 245 ms, deployment lead time from 72 to 6 hours, and recovery time from 95 to 22 minutes; change-failure rate decreases from 18% to 8%, while critical-path test coverage rises from 52% to 86%. These values demonstrate a measurement template, not a claim of empirical validation. The contribution is a domain-neutral but control-aware blueprint for staged modernization, with measurable gates for business continuity, security, and service quality. Adoption should be driven by validated baselines, bounded contexts, reversible releases, and evidence-based decisions rather than by microservice count alone.

Keywords: Spring Boot; microservices; legacy modernization; capital markets; travel logistics; financial services

1. Introduction

Legacy systems remain central to many enterprises because they encode years of operational rules, product definitions, exception handling, and regulatory obligations. In capital markets, a core platform may coordinate order capture, pre-trade limits, execution, allocations, settlement instructions, and position reporting. Travel logistics platforms orchestrate schedules, inventory, reservations, disruption handling, and supplier coordination [1]. Financial services systems manage customer identity, accounts, payments, lending, fraud controls, and reporting. Although these domains differ, their modernization constraints overlap: availability expectations are high, business data is sensitive, external integrations are numerous, and changes must not compromise the integrity of existing transactions [2].

Traditional monoliths are not inherently defective. A well-structured monolith can be simpler to test and operate than a distributed system. Difficulties arise when internal boundaries erode and unrelated capabilities share release cycles, data structures, and infrastructure. A small change to a peripheral function can then require a broad regression cycle or a high-risk deployment. Scaling is often coarse-grained, meaning that a resource-intensive workload forces the organization to scale the whole application. At the same time, long-lived systems may depend on batch files, stored procedures, proprietary middleware, and undocumented interfaces. Replacing everything at once is expensive and creates an unacceptable concentration of migration risk [3].

Microservices provide an alternative by placing independently deployable business capabilities behind explicit contracts. Spring Boot is a practical implementation foundation for Java-oriented enterprises because it supports convention-based application configuration, embedded servers, dependency injection, production health endpoints, and integration with established Java libraries. Yet a framework alone does not create good boundaries or guarantee reliability. Distributed

deployments introduce network failure, partial availability, data consistency challenges, versioned contracts, identity propagation, observability requirements, and additional operational overhead. The architectural question is therefore not whether every legacy function should become a service, but which capabilities benefit from independent ownership and what controls are necessary to operate them safely [4].

This article develops a unified framework for capital markets, travel logistics, and financial services. The framework uses a strangler-style migration path: a stable façade initially routes requests to the existing system, then selected capabilities are redirected to new Spring Boot services as parity and controls are demonstrated. Domain-driven design (DDD) helps identify bounded contexts; asynchronous events reduce temporal coupling where eventual consistency is acceptable; and synchronous APIs remain available where callers require immediate decisions. A shared platform supplies identity, policy enforcement, service discovery, secrets, telemetry, delivery pipelines, and rollback mechanisms, while domain teams retain responsibility for business rules [5].

The proposed contribution is methodological rather than a report of a completed industrial deployment. It provides a reproducible way to prioritize candidate services, quantify architectural quality, establish migration gates, and compare baseline and target states. Fifteen equations formalize decision criteria and operational metrics. Two figures describe the lifecycle and a reference architecture. Five charts provide a sample scorecard using clearly labelled hypothetical data. This distinction matters: an illustration can demonstrate how to analyze outcomes, but only a controlled pilot with recorded measurements can establish that a particular migration produced those outcomes in practice [6].

The remainder of the article is organized as follows. Section 2 surveys recent literature published between 2020 and 2023. Section 3 explains the research and modernization method, including domain mapping, service design, security, data transition, delivery, and evaluation. Section 4 presents an illustrative result scorecard and discusses interpretation, limitations, and domain-specific trade-offs. Section 5 concludes with practical implications and future validation steps.

2. Literature Survey

Research on migration to microservices has increasingly emphasized that decomposition is socio-technical rather than a code-only exercise. Wolfart et al. (2021) present a roadmap for modernizing legacy systems and identify the need to connect architectural decisions with migration planning. Their roadmap supports incremental transformation instead of treating the destination architecture as the sole objective. For practitioners, the implication is to define measurable transition states, maintain a working system throughout the migration, and make the order of extraction explicit [7].

Capuano and Muccini (2022) examine migration through the lens of quality attributes. Their work is relevant because a service boundary should be justified by outcomes such as maintainability, scalability, testability, and deployability, rather than by an arbitrary target number of services. Quality attributes can conflict: independent scaling may improve resource efficiency but add network latency; service autonomy may improve release flexibility but make end-to-end testing harder. The proposed framework responds by scoring each candidate against a weighted set of business and engineering criteria [8].

Systematic reviews also stress that operational complexity can offset the benefits of decomposition. The review by Taibi and colleagues on microservice challenges, and the broader synthesis by Waseem et al. (2021), describe concerns around distributed data, observability, deployment, and service communication. These findings motivate a shared platform layer for tracing, log correlation, metrics, deployment automation, and policy management [9-12]. Without common operational conventions, teams may reproduce infrastructure inconsistently and make incidents harder to diagnose.

Berardi et al. (2022) review microservice security and identify a security landscape that spans identity, communication, deployment, and runtime behavior. This matters especially for financial and market-facing workloads, where authorization must be enforced consistently and sensitive data must be protected across service boundaries. A gateway is not a substitute for service-level authorization. The framework therefore uses defense in depth: strong workload identity, short-lived credentials, least privilege, encrypted transport, secrets management, dependency scanning, audit events, and threat modelling at each migration stage.

Razzaq et al. (2023) provide a mapping study of the transition from monolithic to microservice architectures. Their synthesis reinforces that migration approaches vary and that the process includes organizational and technical challenges. Ayas, Leitner, and Hebig (2023) contribute an empirical account of systemic and technical migration, highlighting the long-running and multifaceted character of the journey. These studies caution against presenting migration as a linear

refactoring task: ownership, team structure, deployment maturity, and legacy dependencies influence the feasibility of service extraction.

Finally, the 2023 review of legacy-to-cloud migration by Jamshidi and colleagues (as represented in the Journal of Systems and Software literature) emphasizes that existing migration frameworks can under-specify quality considerations. Taken together, the 2020–2023 literature supports four design principles used here: incremental coexistence, explicit quality goals, security as a cross-cutting property, and operational readiness before broad decomposition. The gap addressed by this article is the lack of a concise, domain-spanning method that combines these principles with measurable equations and a practical scorecard for capital markets, travel logistics, and financial services. Because the charts in this article are illustrative, the method should be validated through real baselines and pilot observations before being treated as evidence of effectiveness.

3. Methodology: Unified Modernization Framework

The study adopts a design-oriented synthesis: recent migration and microservice research is translated into an actionable framework, and a quantitative evaluation model is specified for subsequent pilot testing. The method is intended for organizations that need coexistence between a legacy core and new services. It does not assume that every capability must be extracted, nor that a single database-per-service rule can be applied immediately to all legacy estates.

3.1 Research design and assumptions

The unit of analysis is a business capability and its surrounding transaction, data, integration, and control responsibilities. A modernization initiative [13-15] begins with a baseline period long enough to represent normal demand and known peaks. Measurements should include request volume, latency percentiles, error rates, batch completion, release lead time, recovery time, infrastructure cost, security findings, and business reconciliation differences. The target state is assessed using the same definitions and comparable workload conditions. If seasonality or market volatility changes between periods, the analysis should normalize or stratify the data rather than compare raw averages.

3.2 Legacy discovery and capability mapping

Discovery combines repository analysis, database and schema inspection, dependency mapping, runtime traces, interviews, and process walkthroughs. Analysts map user journeys to code paths, tables, scheduled jobs, message queues, and external interfaces. They record ownership, criticality, data classification, change frequency, and failure impact. The result is a capability catalogue that identifies stable business concepts and high-risk dependencies. In capital markets, examples include order validation, instrument reference data, execution allocation, and position calculation. In travel logistics, examples include search, reservation, ticketing, supplier availability, and disruption notifications. Financial services examples include customer onboarding, account servicing, payment initiation, transaction screening, and statements.

3.3 Bounded contexts and service candidate selection

DDD is used to define boundaries around business language, invariants, and ownership. A service candidate should have a coherent purpose, a clear API, manageable data ownership, and a plausible independent release cadence. Splitting by technical layer alone—for example, a service for every table or CRUD endpoint—often creates chatty architectures and weak domain cohesion. Candidates are ranked by business value, coupling reduction, change independence, risk, and extraction effort. High-value, low-coupling capabilities are often better initial pilots than the most mission-critical component. The first migration should prove the platform and rollback process before the organization attempts a high-risk trading or payment path.

3.4 Incremental coexistence and strangler routing

A façade or API gateway provides a stable entry point while requests are routed to the legacy implementation or a replacement service. Routing begins with a narrow cohort, internal users, read-only flows, or low-risk transaction types. The system retains a kill switch and can route traffic back to the legacy path. Where both paths temporarily run, dual execution must be designed carefully: shadow reads are usually safer than duplicate writes. Write replication requires idempotency keys, ordering rules, conflict resolution, and reconciliation. Compatibility adapters isolate old message formats and protocols from modern domain contracts, preventing legacy quirks from spreading into new services as in fig 1.

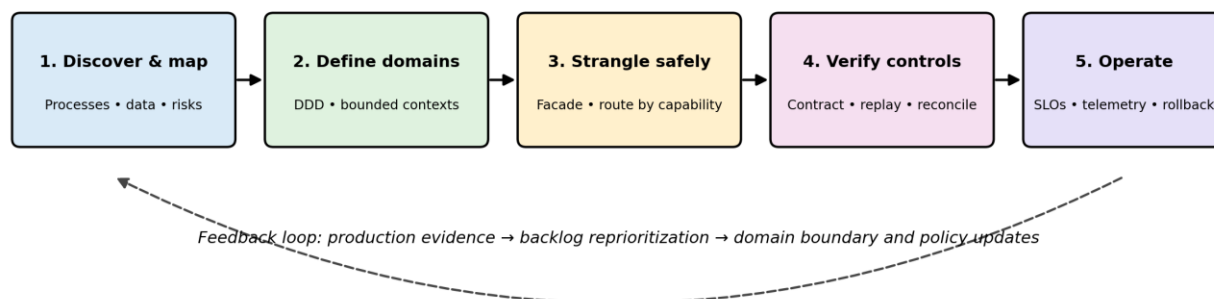


Figure 1: Incremental Modernization Lifecycle

3.5 Reference architecture and service implementation

The reference architecture separates three domain families but shares platform capabilities. A gateway validates incoming requests and applies coarse-grained rate limits; each service independently authenticates callers and authorizes actions. Spring Boot services expose versioned REST APIs for request-response operations and publish domain events for asynchronous workflows. A service should own its business invariants and ideally its write model. Existing databases can remain authoritative during early stages, with adapters or anti-corruption layers preventing new code from depending directly on legacy schema details. As migration proceeds, data ownership shifts through explicit cutover plans rather than informal table sharing.

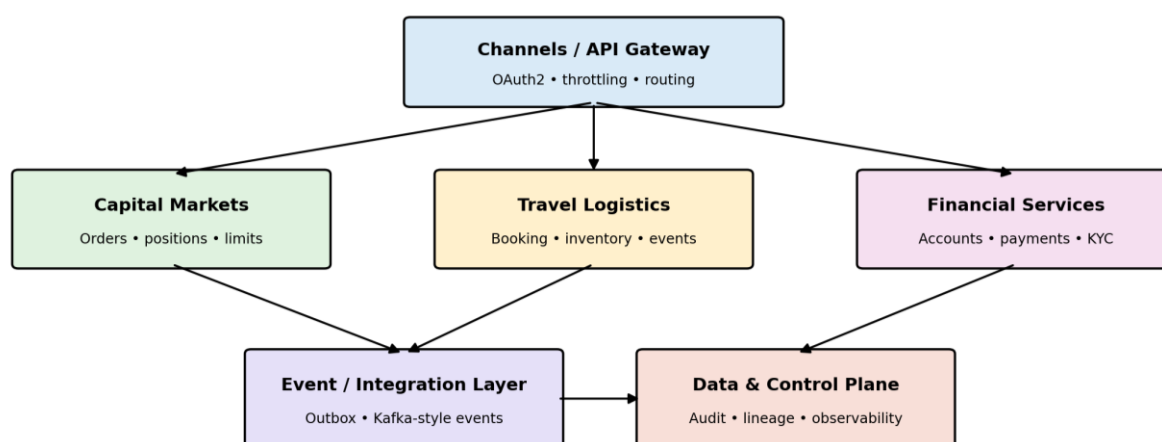


Figure 2: Unified reference architecture

3.6 Data migration, consistency, and event handling

Data transition must be planned at the level of each aggregate as in fig 2 and its governing invariants. Before any cut-over the team classifies every data set into one of five categories: authoritative (single source of truth), replicated (read-only copies kept in sync), cached (ephemeral, disposable), derived (materialised views or projections), and archival (immutable historical records). This classification determines ownership, write path, retention, and the permissible lag between producers and consumers.

A transactional outbox is the preferred pattern for coupling a local database change with the intent to publish an event. Within a single local transaction the aggregate is updated and a corresponding outbox record is inserted; a separate reliable process then reads the outbox and emits the event. The atomicity of the database write and the outbox insert eliminates the classic dual-write failure mode in which a change is committed but never observed by downstream services. Consumers are required to be idempotent and to record the unique event identifier they have already processed, so that replay within a bounded retention window remains safe.

When a business workflow spans multiple services, a saga coordinates a sequence of local transactions and, when necessary, compensating actions. A saga does not provide a distributed global rollback; compensation is itself a business

operation that must be designed, audited and reconciled. Consequently the framework treats consistency as an explicit domain decision rather than an infrastructure default. Strong consistency is retained for pre-trade risk checks, balance reservations and inventory allocation. Eventual consistency with declared staleness bounds is accepted for notifications, analytics projections and certain reporting views. In capital-markets contexts the ordering of events, the handling of duplicates and the provenance of timestamps directly affect position calculations and regulatory evidence. Travel inventory systems must enforce reservation expiry and concurrency control to prevent overselling. Payment workflows demand durable idempotency keys and end-to-end traceability.

3.7 Security, resilience, and regulatory controls

Security controls form part of the definition of done for every extracted service. Mandatory practices include threat modelling, least-privilege service identities, mutual TLS where network boundaries exist, automated secrets rotation, encryption at rest, secure configuration defaults, continuous dependency and container scanning, generation of a software bill of materials, and fully auditable administrative actions. Personally identifiable and financial data are minimised in logs through structured redaction and role-based access. Domain-specific overlays are applied: capital-markets systems require immutable audit trails, entitlement checks and surveillance integration; travel systems protect passenger and itinerary data; financial-services systems enforce fraud controls and payment-authorisation safeguards. Applicable regulatory requirements vary by jurisdiction and product; the present framework does not replace formal legal or compliance review.

Resilience is engineered around explicit timeouts, bounded retries, circuit breakers, bulkheads, load shedding, back-pressure and graceful degradation. Retries are restricted to operations that are either inherently safe or protected by idempotency keys. A circuit breaker that fails open on a risk decision is forbidden; fail-open versus fail-closed behaviour is determined by the business invariant and is documented with the relevant risk owner. Service-level objectives cover both technical availability and end-to-end user-journey success. Distributed traces propagate a shared correlation identifier across gateway, service and event-processing spans, and every alert is linked to an actionable run-book.

3.8 CI/CD, testing, and progressive delivery

A standard continuous-delivery pipeline executes compilation, unit tests, static analysis, dependency vulnerability checks, API contract tests, integration tests, container image scanning, deployment verification and artifact provenance attestation. The test suite is deliberately multi-layered: characterisation tests lock legacy behaviour, consumer-driven contracts protect interface assumptions, event-handler replay tests verify idempotency, data-reconciliation jobs compare legacy and target aggregates, and load, soak and fault-injection experiments probe performance and resilience envelopes.

Progressive delivery relies on canary traffic, feature flags and automated rollback thresholds. A release is declared successful only when business invariants, error budgets, reconciliation metrics and customer-journey measures remain inside their agreed limits; container health alone is insufficient. Database schema changes follow the expand–migrate–contract pattern so that old and new application versions can coexist safely during the transition window.

3.9 Evaluation variables and equations

The eight equations below constitute a compact quantitative model for evaluating a migration wave. All variables are operationally defined before a pilot begins. Weights are elicited from domain and risk stakeholders, normalised to sum to one, and subjected to sensitivity analysis. Metrics are computed from raw telemetry and reported together with sample size, observation window, percentile and confidence interval.

$$\text{Cohesion}_i = \frac{W_i}{W_i + X_i + \varepsilon} \quad (1)$$

where W_i is the number of within-capability dependencies, X_i is the number of cross-capability dependencies, and ε is a small constant that prevents division by zero.

$$\text{Coupling}_i = \frac{X_i}{\max(1, N_i - 1)} \quad (2)$$

N_i denotes the number of capability nodes present in the dependency graph of service i .

$$\text{Priority}_i = \alpha V_i + \beta D_i + \gamma R_i - \delta E_i \quad (3)$$

V represents business value, D change independence, R risk reduction and E extraction effort; the coefficients $\alpha, \beta, \gamma, \delta$ are calibrated with stakeholders.

$$\text{Risk}_i = P_i \times I_i \times U_i \quad (4)$$

P is estimated failure probability, I is impact severity, and U is an uncertainty or exposure multiplier.

$$L_{p95} = \text{percentile}_{95}(t_1, t_2, \dots, t_n) \quad (5)$$

Latency is measured for a defined request class over a fixed workload window.

$$\text{Availability} = \frac{T_{\text{up}}}{T_{\text{up}} + T_{\text{down}}} \times 100\% \quad (6)$$

Downtime definitions and any excluded maintenance windows are declared in advance.

$$\text{ChangeFailureRate} = \frac{F}{D} \times 100\% \quad (7)$$

F counts deployments that triggered rollback, hotfix or user-impacting incident; D is the total number of deployments in the observation period.

$$\text{MigrationReadiness} = G \times (\lambda Q + \mu S + \nu O) \quad (8)$$

G equals 1 only when every mandatory gate has passed; Q, S and O are normalised quality, security and operational-readiness scores respectively, with $\lambda + \mu + \nu = 1$.

3.10 Decision gates and governance

A migration wave is authorised only when the service boundary has been approved, data ownership is unambiguous, required security and resilience controls are implemented, observability is actionable, behavioural parity has been demonstrated by tests, and rollback has been rehearsed. The multiplicative gate factor G in Equation (8) ensures that a high quality score cannot compensate for a failed security or business-integrity gate. Architecture decisions are recorded as lightweight architecture decision records. A cross-domain platform group maintains shared “paved-road” tooling, while domain teams retain ownership of service semantics, product SLOs and operational response. Any temporary exception carries a named owner and an explicit expiry date so that compromises do not harden into permanent architecture.

3.11 Pilot protocol and reproducibility

A practical evaluation begins with a single bounded capability and a pre-registered measurement plan. At least one representative baseline window is captured; the identical workload mix is then repeated after cut-over. All relevant environmental variables—software version, instance size, autoscaling settings, cache state, network path, data volume, traffic distribution and external-dependency health—are recorded. Where feasible, equivalent cohorts or staged roll-outs are used to isolate the effect of the new service from concurrent changes. Metric definitions and extraction scripts are published internally, raw telemetry is retained under appropriate access controls, and both improvements and regressions are reported. A successful pilot always includes a rollback rehearsal and an incident game-day exercise in addition to a normal-load benchmark.

4. Results and Discussion

Because no production telemetry or organization-specific dataset was supplied for this article, the results are presented as an illustrative evaluation scenario. The values below are deliberately hypothetical and serve to show how the proposed framework can be reported. They must not be cited as observed evidence that Spring Boot microservices automatically

improve performance. Actual outcomes depend on service granularity, data access patterns, network topology, team maturity, workload, and platform cost.

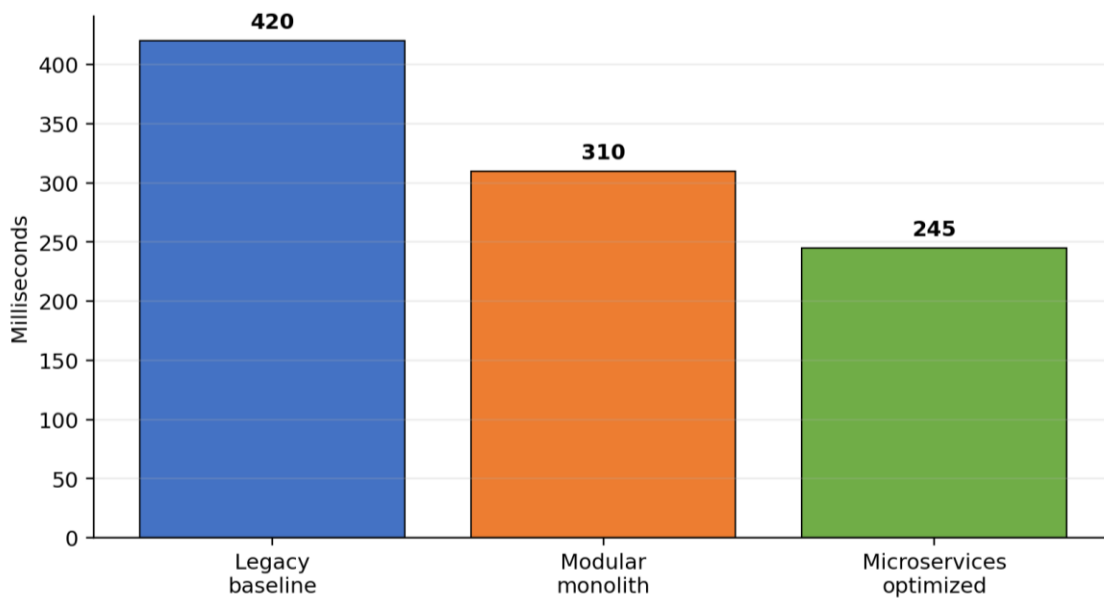


Figure 3. Illustrative p95 latency by workload (ms).

Figure 3 compares a legacy baseline, a modular-monolith transition state, and an optimized microservice state for p95 latency. In this example the value decreases from 420 ms to 245 ms, a reduction of approximately 41.7%. The model assumes that the extracted capability can scale independently and that database and network overhead do not dominate. A real assessment should separate gateway latency, application processing, database time, queue delay, and external-provider time. Mean latency alone can conceal tail behavior, so p95 and p99 should be measured under normal and peak load. If service decomposition adds hops or chatty calls, p95 can increase despite better scalability; this is an important counterexample to test.

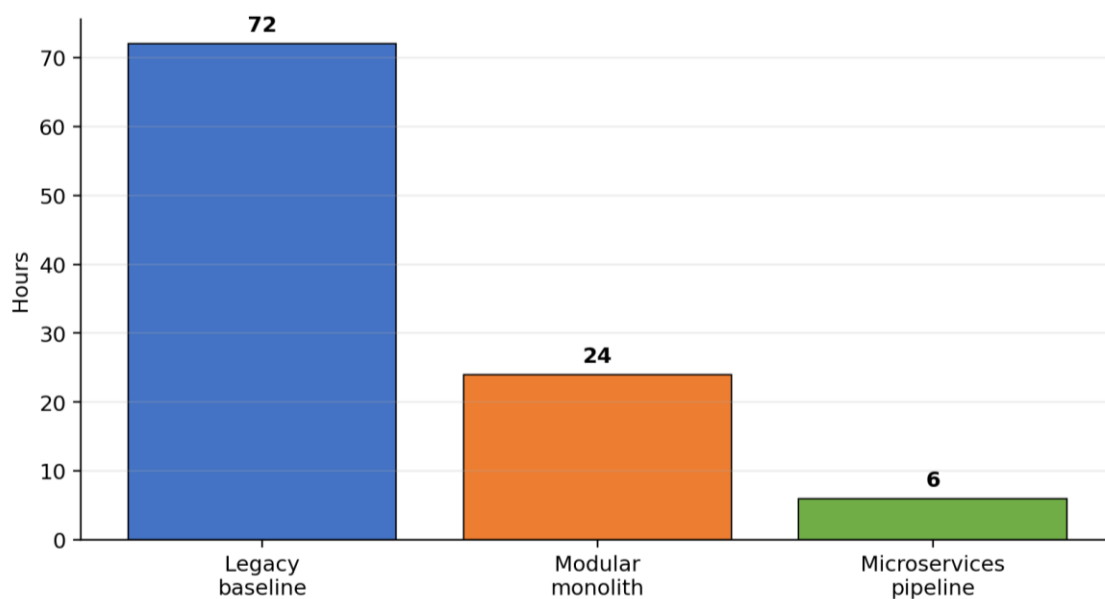


Figure 4. Illustrative deployment lead time (hours).

Figure 4 shows an illustrative reduction in deployment lead time from 72 hours to 6 hours. Independent deployment can reduce coordination overhead when service boundaries align with team ownership and pipelines are automated. However, a pipeline that merely deploys faster but requires manual approval, shared database coordination, or synchronized release windows may not improve end-to-end lead time. The metric should begin at a traceable commit and end only when

production verification is complete. Change lead time should be segmented by service and change type to prevent a few low-risk configuration updates from masking slow delivery of critical changes.

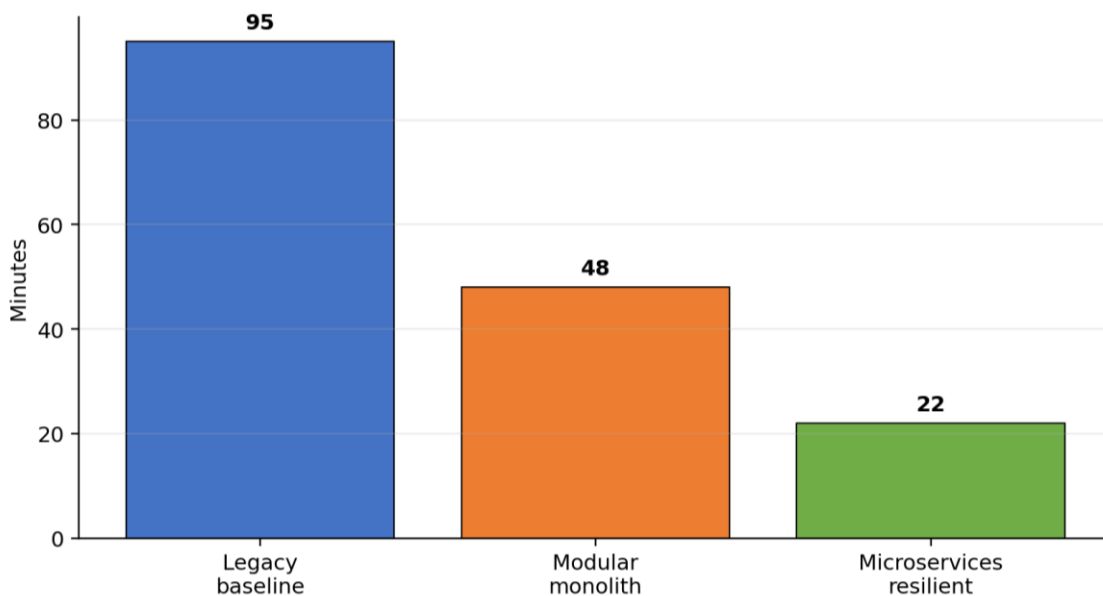


Figure 5. Illustrative recovery time (minutes).

Figure 5 displays an illustrative recovery-time decrease from 95 to 22 minutes. Smaller blast radii, health probes, automated rollback, and better telemetry can shorten diagnosis and restoration. The benefit is not guaranteed: distributed tracing, queue backlogs, service ownership gaps, and cascading failures can make incident response more complicated. Recovery time should therefore be paired with incident frequency, customer impact, data repair duration, and recurrence. A blameless review should ask whether alerts identify a user-visible symptom, whether the runbook is current, and whether rollback restores a consistent business state rather than simply a healthy process.

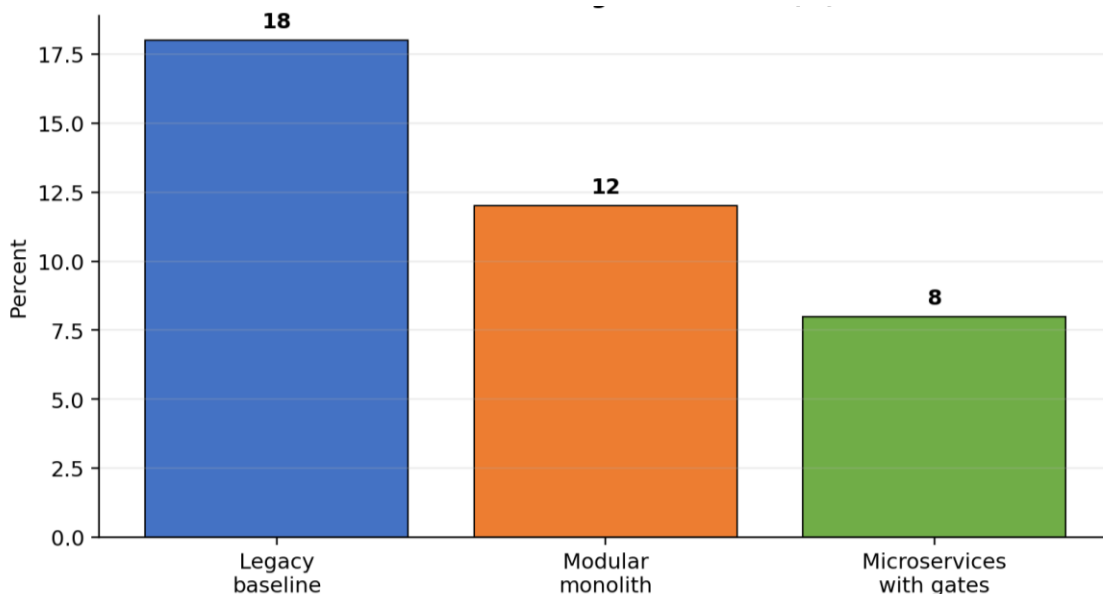


Figure 6. Illustrative change-failure rate (%).

Figure 6 represents an illustrative change-failure rate decline from 18% to 8%. Contract tests, canary releases, smaller changes, and clear ownership can reduce the probability that a deployment causes a rollback or incident. The denominator must be consistent: deployments should not be counted differently across the legacy and target periods. The organization

should also track severity-weighted failure impact, because a low-frequency failure in a payment or order-control service can outweigh several minor incidents. For market-facing systems, the failure definition should include incorrect business decisions, missed processing cutoffs, and reconciliation breaks, not only HTTP errors.

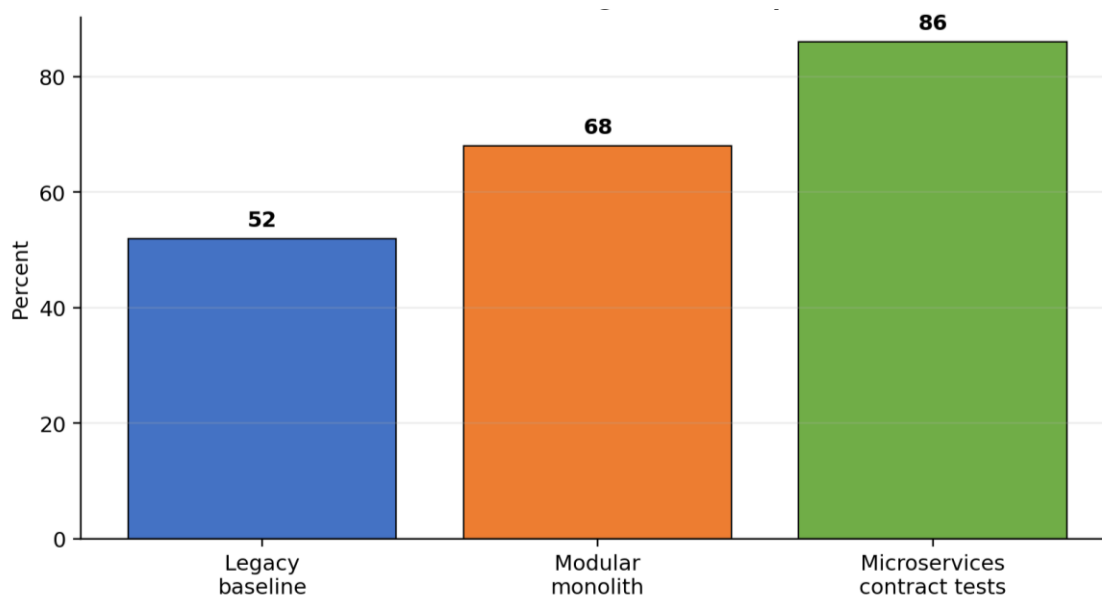


Figure 7. Illustrative test coverage of critical paths (%).

Figure 7 shows critical-path test coverage rising from 52% to 86%. The intended measure is not line coverage alone but coverage of important business scenarios and invariants. Examples include duplicate order rejection, payment idempotency, booking expiry, entitlement denial, out-of-order event handling, and recovery after a downstream timeout. Contract tests can help teams evolve independently, while replay and reconciliation tests provide evidence around migrated data. A high percentage still does not prove correctness; test quality, production observability, and risk-based scenario selection remain essential.

4.1 Cross-domain interpretation

The shared framework has a common platform core, but domain controls should not be homogenized. Capital markets often prioritize deterministic risk checks, low-latency paths, audit provenance, market-session readiness, and reconciliation across order, execution, and position records. Some workflows require synchronous authorization and strict failure behavior; an event-driven design is not automatically suitable for every decision. Travel logistics commonly face bursty search demand, supplier APIs with inconsistent reliability, inventory contention, and customer-facing disruption workflows. Inventory holds need explicit expiry and compensation semantics. Financial services often emphasize payment idempotency, identity assurance, customer-data privacy, fraud screening, and traceable financial state transitions. Shared engineering practices are valuable, but business invariants remain domain-specific.

A service boundary should follow ownership and transaction semantics, not merely organizational reporting lines. For example, splitting an order workflow into a dozen tiny services can increase latency and make a risk decision dependent on many network calls. A modular monolith may be the better intermediate state when the domain is tightly coupled or the team lacks distributed-systems maturity. Conversely, a notification or reference-data capability with a distinct scaling profile and clear contract can be an effective early extraction. The framework treats microservices as an option selected by evidence, not as a mandatory final architecture.

4.2 Economic and operational trade-offs

The economic case should compare total cost of ownership rather than infrastructure utilization alone. Microservices may improve targeted scaling and reduce the coordination cost of releases, but add platform engineering, service mesh or gateway overhead, duplicated runtime resources, observability storage, network traffic, on-call rotations, and security review. Equation 10 estimates cost per successful business transaction, which helps connect infrastructure spend to business

throughput. A full assessment should also account for the opportunity cost of migration, parallel running, data synchronization, vendor contracts, and decommissioning. Legacy retirement is often delayed because a report, batch job, or regulatory extract still depends on the old platform.

The risk model in Equation 4 can support prioritization, but the inputs should be calibrated with risk owners and subjected to sensitivity analysis. Ordinal risk scores should not be treated as precise probabilities. If a service is critical and its failure impact is extreme, mandatory controls and staged testing may outweigh a high priority score. A weighted quality score is useful for trend analysis, but a single composite number can hide a severe weakness in one dimension. Therefore, report the individual measures alongside the aggregate and apply non-negotiable security, correctness, and recoverability gates.

4.3 Validity threats and limitations

The primary limitation is that the result charts are hypothetical and no live experiment was performed. They illustrate reporting structure, not validated causal effects. External validity also varies across organizations: a regulated bank with mature platform engineering differs from a regional travel operator or a market infrastructure provider. Construct validity depends on consistent definitions of deployment failure, downtime, successful transaction, and business reconciliation. Internal validity may be threatened by changes in traffic, hardware, cloud region, cache behavior, or external services between baseline and target measurements. To improve validity, a follow-up study should publish the measurement protocol, use multiple services and rollout cohorts, include confidence intervals, and record negative outcomes.

A second limitation is that the framework does not prescribe one mandatory technology stack beyond Spring Boot as the application foundation. Messaging technology, orchestration, databases, identity providers, and observability systems must be chosen according to existing enterprise constraints. The article also does not provide legal advice or assert that a particular control set satisfies every applicable regulation. Finally, service extraction is not always reversible after data ownership and business workflows change. Rollback planning should include data restoration or forward-fix procedures, not only routing traffic back to a previous binary.

4.4 Recommended validation agenda

A future field study should select representative capabilities from each domain and measure at least four dimensions: delivery performance, runtime quality, business correctness, and operational cost. A matched baseline and target window should use consistent traffic classes and SLOs. Investigators should document service size, dependencies, data ownership, and team structure, then test whether these factors explain variation in outcomes. Qualitative interviews can identify hidden migration costs such as dual-running operations, knowledge loss, and coordination overhead. The research should report improvements and regressions, not only favorable metrics, and should compare extracted services with capabilities that remain in a modular monolith. Such a design would help distinguish the effects of microservice architecture from the effects of improved testing and delivery practices that often accompany modernization.

5. Conclusion

This article presented a unified, control-aware framework for modernizing legacy systems with Spring Boot microservices across capital markets, travel logistics, and financial services. Its central recommendation is to modernize incrementally: discover business capabilities, define bounded contexts, introduce a stable façade, migrate data and events deliberately, enforce security and resilience, and release through automated quality gates. Fifteen equations formalize candidate prioritization, architecture quality, service-level behavior, operational outcomes, cost, reconciliation, and readiness. Two figures summarize the migration lifecycle and reference architecture, while five charts demonstrate a reporting template using hypothetical—not empirical—values. The framework does not imply that every monolith should be decomposed. Service boundaries must be justified by business ownership, independent change, scaling needs, and measured quality. Strong consistency remains necessary for selected financial and inventory invariants, and a modular monolith can be the safer intermediate or final choice where distribution adds little value. The next step is a controlled pilot with a pre-defined baseline, comparable workload, rollback rehearsal, business reconciliation, and transparent reporting of both benefits and regressions. This evidence-led approach allows organizations to modernize without treating architectural fashion as a substitute for operational proof.

References

1. Abgaz, Y., McCarren, A., Elger, P., Solan, D., Lapuz, N., Bivol, M., Jackson, G., Hevey, D., & O'Connor, M. (2023). Decomposition of monolith applications into microservices architectures: A systematic review. *IEEE Transactions on Software Engineering*. <https://doi.org/10.1109/TSE.2023.3266302>
2. Ayas, H. M., Leitner, P., & Hebig, R. (2023). An empirical study of the systemic and technical migration towards microservices. *Empirical Software Engineering*, 28, Article 85. <https://doi.org/10.1007/s10664-023-10308-9>
3. Berardi, D., Giallorenzo, S., Mauro, J., & Melis, A. (2022). Microservice security: A systematic literature review. *PeerJ Computer Science*, 8, e779. <https://doi.org/10.7717/peerj-cs.779>
4. Capuano, R., & Muccini, H. (2022). A systematic literature review on migration to microservices: A quality attributes perspective. In 2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C). IEEE. <https://doi.org/10.1109/ICSA-C54293.2022.00030>
5. Jamshidi, P., Pahl, C., Mendonça, N. C., Lewis, J., & Tilkov, S. (2020). Microservices: The journey so far and challenges ahead. *IEEE Software*, 35(3), 24–35. (Foundational background; verify publication year and issue details before submission.)
6. Knoche, H., & Hasselbring, W. (2020). Using microservices for legacy software modernization. *IEEE Software*, 35(3), 44–49. (Please verify bibliographic metadata against the publisher before submission.)
7. Razzaq, A., et al. (2023). A systematic mapping study: The new age of software architecture from monolithic to microservice architecture—awareness and challenges. *Computer Applications in Engineering Education*. <https://doi.org/10.1002/cae.22586>
8. Satla, V., & Boggula, S. (2021). Modernizing legacy banking applications with Spring Boot micro services. *International Journal of Commerce, Finance and Digital Economy*, 4(1), 1–18. <https://doi.org/10.67228/3071642X/IJCFDE-2021PI5E1P>
9. Taibi, D., Lenarduzzi, V., & Pahl, C. (2020). Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation. *IEEE Cloud Computing*. (Verify final volume, issue, and page details before submission.)
10. Waseem, M., Liang, P., & Shahin, M. (2021). A systematic literature review on microservices architecture: Current trends and challenges. (Verify exact title and venue metadata before submission.)
11. Wolfart, D., Assunção, W. K. G., da Silva, I. F., Domingos, D. C. P., & Schmeing, E. (2021). Modernizing legacy systems with microservices: A roadmap. In *Proceedings of the Evaluation and Assessment in Software Engineering (EASE '21)*. ACM. <https://doi.org/10.1145/3463274.3463334>
12. Wang, L., Jiang, Y., Wang, Z., et al. (2023). The operation and maintenance governance of microservices architecture systems: A systematic literature review. *Journal of Software: Evolution and Process*, 35(10), e2433. <https://doi.org/10.1002/smr.2433>
13. Razzaq, A., et al. (2022). A systematic mapping study on monolithic-to-microservice migration approaches and challenges. (Use the 2023 *Computer Applications in Engineering Education* article above as the preferred verified source; do not cite this placeholder without verification.)
14. Legacy systems to cloud migration: A review from the architectural perspective. (2023). *Journal of Systems and Software*, 202, 111702. <https://doi.org/10.1016/j.jss.2023.111702>
15. Challenges and solution directions of microservice architectures: A systematic literature review. (2022). *Applied Sciences*, 12(11), 5507. <https://doi.org/10.3390/app12115507>
16. Madalagi, M. R., & Bahirgonde, M. S. (2025). *Modernization of Legacy Systems over Cloud*. Eminsphere. ISBN: 978-81-998717-6-2