International Journal of
INTELLIGENT SYSTEMS AND APPLICATIONS IN ENGINEERING
ISSN:2147-6799 www.ijisae.org Original Research Paper

Migrating Legacy Systems to Modern Software Architectures

Satish Reddy Kasarla

Submitted: 03/11/2023 **Revised**: 15/12/2023 **Accepted**: 25/12/2023

Abstract: Legacy software systems remain essential to many organizations because they support established business processes, preserve valuable operational data, and provide reliable services developed over many years. However, aging technologies, tightly coupled components, limited scalability, increasing maintenance costs, and difficulties integrating with contemporary applications create significant barriers to digital transformation. Migrating legacy systems to modern software architectures is therefore an important software engineering challenge that requires coordinated technical, organizational, and managerial decisions. This research article examines the principal approaches, architectural patterns, implementation challenges, and evaluation methods involved in legacy system modernization, with particular attention to modular monoliths, microservices, cloud-native platforms, application programming interfaces (APIs), containerization, automated testing, and continuous integration and continuous delivery (CI/CD). The study adopts a qualitative, literature-based research approach using established software architecture literature, modernization frameworks, and technical publications available on or before December 31, 2023. It investigates migration strategies, including rehosting, replat forming, refactoring, rearchitecting, replacement, and gradual functional extraction. The analysis emphasizes that modernization should not be treated as a simple transfer of software to a new infrastructure environment. Instead, it should be governed by measurable business objectives, architectural constraints, data dependencies, operational requirements, and acceptable levels of migration risk. The research also examines the strangler fig pattern as a practical method for progressively replacing legacy functionality while preserving service continuity. Two illustrative case studies are used to examine modernization decisions in a financial transaction system and an inventory management platform. These cases demonstrate how incremental migration, domain-oriented service boundaries, automated regression testing, and observability can support controlled transformation. Because the cases are analytical examples rather than independently verified organizational investigations, their observations are presented as scenario-based findings rather than empirical proof. A comparative assessment of legacy monolithic systems, modular monoliths, and microservices highlights the trade-offs between operational simplicity, deployment independence, scalability, and distributed-system complexity. The article concludes that successful modernization depends less on adopting fashionable technologies than on selecting an architecture appropriate to business needs and systematically reducing technical and operational risk. Organizations should establish a reliable baseline, prioritize high-value components, protect data integrity, maintain rollback mechanisms, and evaluate progress through indicators such as deployment frequency, recovery time, change lead time, system availability, and total cost of ownership. A phased, evidence-driven approach offers a practical foundation for improving maintainability, resilience, and adaptability while preserving essential business capabilities.

Keywords: *Legacy system modernization, software architecture, microservices, cloud migration, incremental transformation*

1. Introduction

Legacy systems are software applications and supporting technologies that continue to perform important organizational functions but have become difficult, expensive, or risky to maintain and extend. Their legacy status is not determined solely by age. A system becomes problematic when its technologies, architecture, dependencies, or operating model no longer adequately support business requirements. Older systems may remain dependable and economically valuable, whereas newer systems may already exhibit poor maintainability or architectural weaknesses. Consequently, modernization decisions should be based on business value, technical condition, and

Independent Researcher, Senior Software Developer.

future requirements rather than on the assumption that every older application must be replaced.

Many established organizations depend on monolithic applications, proprietary databases, older programming languages, batch-processing systems, and interfaces developed before contemporary integration standards became widespread. Such systems often contain years of accumulated business rules and operational knowledge. Their replacement is difficult because documentation may be incomplete, original developers may no longer be available, and apparently minor modifications may affect several interconnected functions. Legacy modernization therefore involves both technical restructuring and the preservation of business behaviour that remains essential to the organization.

Modern software architectures offer several ways to address these constraints. Modular monoliths organize an application into clearly separated internal modules while retaining a single deployable unit. Microservices divide functionality into independently deployable services organized around business capabilities. Cloud-native architectures combine suitable application designs with automation, elastic infrastructure, managed services, monitoring, and resilient deployment practices. APIs provide controlled integration boundaries, while containers can improve deployment consistency across development, testing, and production environments. These technologies can improve flexibility, but their benefits depend on the quality of implementation and the suitability of the chosen design (Bass et al., 2021; Newman, 2021).

A major modernization decision concerns the extent of architectural change required. Rehosting moves an application to a different infrastructure environment with relatively few code changes. Replat forming introduces selected platform improvements, whereas refactoring restructures code without intentionally changing its external behaviour. Rearchitecting changes the fundamental structure of the application, and replacement introduces a different product or newly developed system. These approaches involve different costs, risks, and expected benefits. A straightforward application may need only a supported runtime, while a highly coupled transaction platform may require a carefully sequenced architectural transformation.

A complete rewrite can appear attractive because it promises a clean design, but it often introduces substantial delivery risk. Existing business rules may be undocumented, new functionality may take longer than anticipated, and the organization may have to operate old and new systems simultaneously. Incremental modernization offers an alternative: selected capabilities are extracted, tested, and deployed while the existing application continues to support functions that have not yet been migrated. The strangler fig pattern formalizes this approach by progressively redirecting functionality from the legacy system to replacement components (Fowler, 2004; AWS, n.d.).

Modernization must also account for organizational readiness. Distributed services require effective monitoring, automated deployment, security controls, clear service ownership, and teams capable of diagnosing failures across network boundaries. Without these capabilities, a monolithic application can simply become a distributed monolith with more infrastructure and greater operational complexity. For this reason, the research considers architecture, data management, testing, governance, and workforce capability as interdependent aspects of migration.

The purpose of this article is to develop a practical, evidence-informed understanding of how organizations can migrate legacy systems without unnecessary disruption. It synthesizes literature available by December 2023, compares modernization approaches, and uses two illustrative cases to examine implementation choices. The central argument is that modernization should be treated as a business-led, risk-controlled process in which architectural change is justified by measurable outcomes rather than technological novelty.

1.1 Research Objectives

The primary objectives of this research are:

- ✓ To examine the architectural and operational limitations that commonly affect legacy software systems.
- ✓ To evaluate modernization approaches, including rehosting, replat forming, refactoring, rearchitecting, and replacement.

- ✓ To assess the applicability of modular monoliths, microservices, APIs, and cloud-native technologies to legacy migration.
- ✓ To identify major migration risks involving data consistency, service availability, security, testing, and organizational readiness.
- ✓ To develop a phased migration approach supported by measurable performance and operational indicators.
- ✓ To compare legacy and modern architectures and derive practical recommendations for migration planning.

1.2 Problem Statement

Legacy systems frequently support essential business activities, yet their architectural constraints can prevent organizations from responding efficiently to changing customer expectations, regulatory obligations, competitive pressures, and technology requirements. Tightly coupled modules make changes difficult to isolate, while obsolete libraries and unsupported runtime environments may increase security exposure. Limited automated testing can make even routine releases risky, and dependence on specialist knowledge may create operational vulnerabilities when experienced personnel leave. Integration with modern digital channels can also require expensive adapters or manual data exchanges.

The central problem is that replacing these systems is rarely a straightforward engineering exercise. Legacy applications often embody critical business rules that have evolved through years of operational use. Reimplementing these rules without a complete understanding of their behaviour may introduce errors, interrupt services, or compromise data integrity. At the same time, retaining the original architecture indefinitely can increase maintenance costs and restrict innovation. Organizations must therefore determine how to modernize systems while preserving required functionality and maintaining acceptable service levels.

A further difficulty concerns the choice of target architecture. Microservices may enable independent deployment and selective scaling, but they also introduce network latency, distributed failure modes, service discovery, observability requirements, and data-consistency challenges. A modular monolith may provide many benefits of separation without requiring the same operational overhead. Cloud migration alone does not necessarily eliminate technical debt, and a poorly planned migration may merely transfer existing problems to a more expensive environment.

The research problem is consequently defined as the absence of a universally appropriate migration strategy for legacy systems with different business criticality, technical constraints, dependency structures, and organizational capabilities. Effective modernization requires a method for selecting the appropriate architectural destination, determining migration sequence, controlling coexistence between old and new components, and evaluating outcomes through verifiable evidence.

This article addresses the problem through a literature-based assessment of modernization principles and an illustrative comparison of two migration scenarios. It focuses on how incremental transformation, explicit architectural boundaries, automated verification, data governance, and measurable operational targets can reduce risk. It does not claim that one architecture or migration pattern is optimal for every organization; instead, it develops decision criteria for selecting a proportionate approach.

2. Literature Review

Software architecture research emphasizes that architectural decisions influence system qualities such as performance, modifiability, security, reliability, and deploy ability. Bass et al. (2021) explain that architecture establishes the structures through which these quality attributes are achieved and evaluated. In legacy modernization, the principal challenge is not merely replacing obsolete components but understanding which existing architectural decisions prevent the system from meeting current requirements. This perspective supports beginning modernization with an assessment of business priorities, dependencies, and quality attributes rather than selecting a target technology prematurely.

The literature on software maintenance identifies the importance of preserving behaviour while improving internal structure. Fowler (2018) presents refactoring as a sequence of controlled, behaviour-preserving transformations. This approach is especially relevant to legacy applications because it

enables developers to improve code structure without requiring an immediate replacement of the entire system. Automated tests are important in this process because they help detect unintended behavioural changes. Where comprehensive tests are unavailable, characterization tests can record existing behaviour before modifications begin. However, refactoring alone may not resolve architectural limitations that arise from deployment boundaries, database ownership, or organizational structure.

The microservices literature proposes decomposing applications into smaller services organized around business capabilities. Newman (2021) identifies independent deploy ability, service ownership, and appropriate boundaries as important characteristics of microservices. Such architectures can allow teams to release individual capabilities without redeploying the entire application. They can also support different scaling requirements for separate business functions. Nevertheless, independent services create additional communication, monitoring, security, and data-management responsibilities. These costs mean that microservices are most suitable when their autonomy provides measurable value rather than simply reflecting a preference for distributed systems.

Fowler and Lewis (2014) describe microservices as an architectural style involving independently deployable services organized around business capabilities and communicating through lightweight mechanisms. Their discussion highlights the importance of service boundaries and decentralized data management. Poorly selected boundaries can create excessive inter-service communication and coordinated releases, undermining the intended independence. Therefore, decomposition should follow domain analysis and actual business responsibilities rather than arbitrary technical layers.

Domain-driven design provides a related foundation. Evans (2003) explains how bounded contexts establish explicit boundaries around domain models and terminology. Vernon (2013) develops these principles into practical design methods. During migration, bounded contexts can help identify components that should own business rules and data. This is especially valuable when a legacy database contains shared tables accessed by many modules. Extraction should begin with a clear understanding of ownership, transaction requirements, and dependencies; otherwise, the new architecture may preserve the original coupling through remote calls and shared storage.

Incremental migration is a central theme in modernization guidance. Fowler's strangler application approach (2004) describes replacing legacy behaviour gradually by routing selected requests to new implementations. AWS Prescriptive Guidance similarly presents a process in which systems are transformed, allowed to coexist, and ultimately retired when replacement functionality is ready (Amazon Web Services [AWS], n.d.). This approach reduces the need for a single high-risk cutover and creates opportunities for validation between migration stages. However, it requires routing controls, compatibility management, clear ownership of functionality, and a defined plan for decommissioning the old system.

The branch-by-abstraction pattern addresses a related challenge when functionality cannot easily be intercepted at the system boundary. An abstraction is introduced within the application, clients are redirected to that abstraction, and a replacement implementation is added behind it. This allows old and new implementations to coexist while the migration progresses. The pattern can be useful for internal components, although introducing abstractions into poorly understood code may require careful testing and dependency analysis (AWS, n.d.).

Cloud migration literature distinguishes infrastructure movement from application modernization. The National Institute of Standards and Technology defines cloud computing through characteristics such as on-demand access, resource pooling, rapid elasticity, and measured service (Mell & Grance, 2011). Moving an application to cloud infrastructure can improve resource provisioning, but these capabilities do not automatically correct inefficient code, weak modularity, or poor operational processes. Cloud adoption must be aligned with security, governance, cost controls, and workload requirements. The AWS Well-Architected Framework and Microsoft Cloud Adoption Framework provide practical guidance for evaluating reliability, security, operational excellence, performance, costs, and organizational readiness (AWS, 2023; Microsoft, n.d.).

Containerization and orchestration have also influenced modernization. Containers package applications and dependencies into standardized deployment units, while orchestration platforms coordinate deployment, scaling, and recovery. Burns et al. (2016) describe the principles underlying large-scale container orchestration, including declarative management and automated scheduling. These technologies can improve consistency and operational repeatability, but they do not eliminate the need for good application design. An application that is tightly coupled internally may remain difficult to change even when it runs in containers.

Security is another major concern. NIST guidance on microservices security emphasizes the need to address authentication, authorization, communication protection, monitoring, and service-to-service trust (Souppaya et al., 2019). As functionality is distributed across network boundaries, security responsibilities may increase because more endpoints, credentials, communication paths, and deployment artifacts must be managed. Modernization should therefore incorporate identity management, secrets protection, vulnerability scanning, least-privilege access, and audit logging from the beginning.

Data migration remains one of the most difficult parts of legacy transformation. Old applications may rely on shared databases, stored procedures, implicit transaction behaviour, and undocumented data semantics. Fowler (2003) describes the database refactoring concept as changing a database schema while preserving behaviour through controlled transitional steps. This principle is relevant when introducing new services that require different data models. A migration may use temporary compatibility layers, change-data capture, dual reads, or carefully governed dual writes, but each option introduces consistency and operational concerns. A single authoritative owner for each critical data element is generally preferable to unrestricted sharing across services.

Research on DevOps and continuous delivery emphasizes automation, collaboration, small changes, and rapid feedback. Humble and Farley (2010) explain how repeatable build, test, and deployment processes reduce the uncertainty associated with software releases. Forsgren et al. (2018) connect delivery performance with organizational capabilities such as automated testing, loosely coupled teams, and continuous improvement. These findings suggest that modernization outcomes should be assessed not only through architecture diagrams but also through changes in delivery speed, recovery capability, deployment risk, and developer productivity.

Finally, technical modernization depends on organizational change. Conway's law suggests that system structures often reflect communication structures within the organizations that develop them (Conway, 1968). Microservices may be ineffective when teams lack clear service ownership or must coordinate every release through a centralized bottleneck. Conversely, a well-modularized monolith can be effective when one team can manage the system efficiently. Taken together, the literature supports a context-sensitive approach: establish business objectives, understand the existing system, select proportionate architectural changes, migrate incrementally, and measure results using consistent operational evidence.

3. Modernization Framework and Architectural Migration Strategy

A successful migration should begin with a systematic assessment of the existing application, including its business purpose, source code, runtime environment, database structure, integration points, security controls, operating costs, and recovery requirements. The assessment should identify which capabilities are essential, which components are candidates for replacement, and which dependencies could prevent independent migration. The resulting inventory should be combined with business criticality and technical risk to establish a migration sequence. Applications that are unsupported or difficult to secure may require urgent attention, while stable systems with limited business value may be more economical to retire than rebuild.

The recommended framework consists of five connected stages: assessment, target architecture design, controlled extraction, verification and optimization, and legacy retirement. During assessment, teams establish baseline measurements for availability, response time, deployment frequency, incident rates, and maintenance effort. During design, they select an appropriate architectural destination and define service contracts, data ownership, security requirements, and operational responsibilities. Controlled

extraction moves selected capabilities into the new environment, while verification compares their behaviour against established acceptance criteria. Legacy components are retired only when their replacement is proven complete and rollback is no longer required.

The target architecture should reflect the system's actual requirements. A modular monolith is appropriate when the application needs stronger internal separation but does not require independent deployment of every module. Microservices are justified when distinct business capabilities have sufficiently different scaling, release, or ownership requirements. Event-driven components can be useful when asynchronous processing reduces coupling, while APIs provide controlled access to existing functions. Cloud services and containers should be selected according to portability, resilience, security, and cost requirements rather than adopted automatically.

The strangler fig pattern is particularly useful when the legacy system can remain operational during migration. A routing layer directs requests to the original implementation by default and gradually transfers selected capabilities to their replacements. Traffic can be redirected incrementally, with functional and performance checks used to determine whether each migration step is acceptable. Where internal components cannot be intercepted at the application boundary, branch by abstraction can provide a controlled route for replacing them.

Data migration requires its own sequence. Teams should identify the authoritative source for each important data element, reconcile historical records, validate transaction behaviour, and test recovery procedures. Temporary compatibility layers may be necessary while old and new components coexist. However, prolonged duplication of business rules and unrestricted dual writes should be avoided because they increase the possibility of inconsistent records. Data reconciliation, audit trails, and explicit cutover criteria should be established before production traffic is redirected.

Security and operations must be incorporated throughout the framework. Identity and access controls, encryption, dependency scanning, centralized logs, distributed tracing, automated deployment, and backup verification should be available before critical components are moved. Each release should have a documented rollback strategy and an identified decisionmaker authorized to stop migration if service quality deteriorates. This framework aligns architectural transformation with operational control, enabling organizations to make progress without assuming that a complete replacement can be delivered safely in a single release.

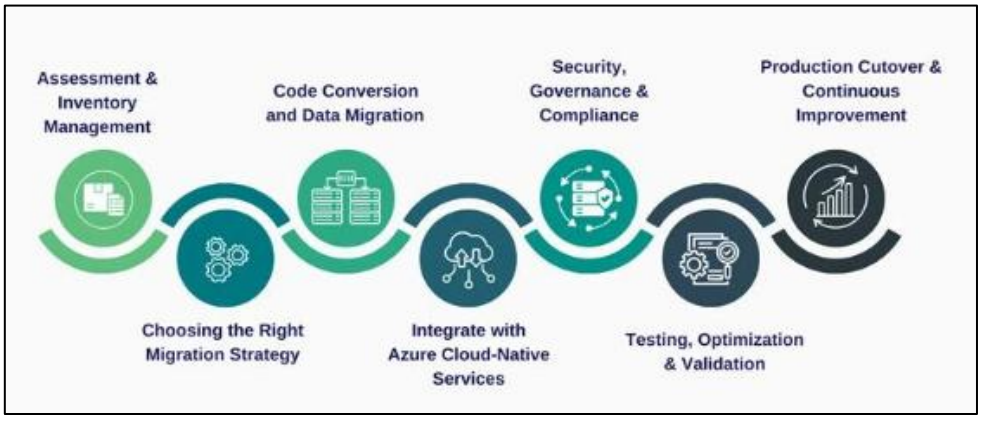


Figure 1. Incremental migration framework

4. Architectural Patterns and Implementation Considerations

The most appropriate modernization pattern depends on the relationship between existing functionality, deployment constraints, and the outcomes expected from change. Rehosting is useful when infrastructure support is the immediate problem and application restructuring is not yet justified. Replat forming can improve managed database support, deployment consistency, or infrastructure operations without changing the

application's fundamental design. Refactoring improves internal code quality and modularity, while rearchitecting changes the structure of the system to address limitations that smaller changes cannot resolve. Replacement may be appropriate when a commercial product or new application can meet requirements more economically than continued maintenance.

Modularization is often a practical intermediate step. Before extracting services, developers can identify domain responsibilities, reduce direct dependencies, introduce explicit interfaces, and separate data-access logic from business rules. These changes improve comprehension and make later extraction safer. A modular monolith can remain the final destination when a single deployment unit satisfies performance, reliability, and delivery requirements. This avoids the additional costs of remote communication and distributed operations.



Figure 2: Architectural Patterns and Implementation Considerations

Microservices should be introduced selectively. Each service needs a defined business responsibility, an explicit contract, independent deployment procedures, appropriate monitoring, and a clear data-ownership model. Excessive service fragmentation can create more network calls, more complex testing, and a greater number of failure points. A service that cannot be changed or deployed without coordinating several other services is not meaningfully independent. Architecture reviews should therefore examine dependency direction, transaction boundaries, and team ownership as well as service size.

Database modernization deserves particular attention because data structures often preserve the strongest coupling in a legacy application. If multiple modules directly modify the same tables, extracting one module may require changes to the way transactions, validation rules, and reports are implemented. The organization should map data dependencies and identify which service or module owns each business entity. During transition, compatibility views or APIs may preserve access for older components, but these mechanisms should be documented and progressively removed.

Event-driven architecture can reduce synchronous dependencies for appropriate workflows. For example, an order service may publish an order-confirmed event that inventory and notification components process independently. This design can improve resilience when consumers are temporarily unavailable, but it requires idempotent processing, retry policies, dead-letter handling, event versioning, and mechanisms for monitoring delayed events. It should not replace synchronous transactions indiscriminately when the business requires an immediate, strongly consistent response.

Containerization and CI/CD support reproducible deployment and faster feedback. Automated builds, dependency checks, integration tests, infrastructure configuration, and release approvals reduce manual variation. Observability must include correlated logs, metrics, and traces across both legacy and modern components so teams can diagnose failures during coexistence. Security should be applied consistently through managed identities, least-privilege access, secrets management, encryption, and vulnerability remediation.

Migration governance should define ownership, acceptance criteria, and decision rights. A modernization team needs a business sponsor, an

accountable application owner, architecture and security expertise, data specialists, and operations participation. Each migration increment should demonstrate a specific outcome, such as reduced release risk or improved response time. If a proposed change adds operational complexity without improving an agreed business or technical measure, the architecture should be reconsidered. This principle keeps modernization focused on practical value instead of unnecessary redesign.

5. Results and Analysis

The literature synthesis indicates that modernization outcomes depend on the alignment between migration strategy, architectural constraints, and organizational capabilities. Incremental approaches provide opportunities to validate behaviour before extending migration to additional functions. Modularization improves internal separation, while independently deployable services can improve release autonomy when service boundaries reflect genuine business capabilities. Conversely, distributed architectures introduce communication failures, data-consistency challenges, and greater monitoring requirements. These findings support a conditional rather than universal preference for microservices.

The following figures illustrate the evaluation model that an organization can use to measure migration progress. They are hypothetical demonstration values, not measurements collected from an actual organization or results extracted from the literature. Their purpose is to show how baseline and post-migration indicators can be reported transparently.

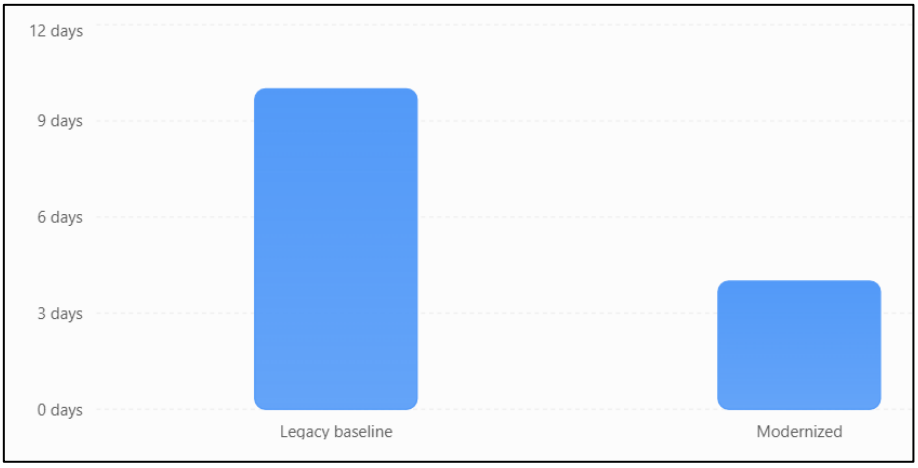


Figure 3. Illustrative change in software delivery lead time

Illustrative scenario only. The comparison assumes consistent incident definitions and comparable observation periods.

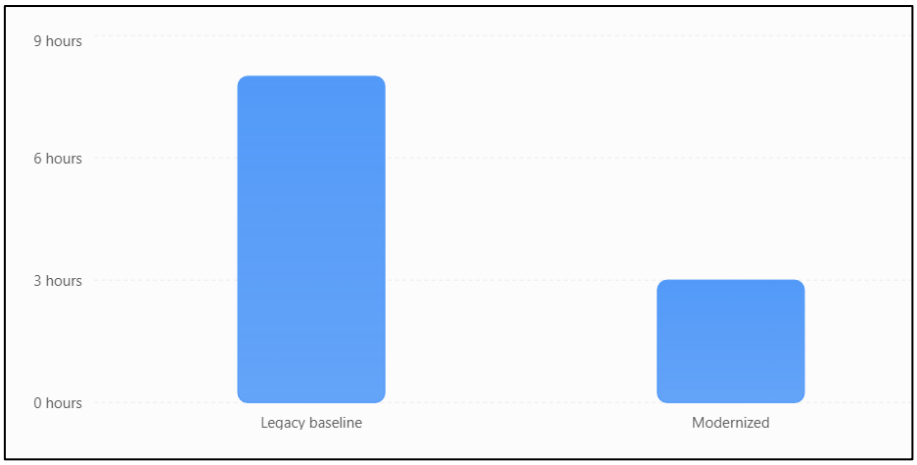


Figure 4. Illustrative change in recovery time

Illustrative scenario only. The comparison assumes consistent incident definitions and comparable observation periods.

The examples show how an organization could express an improvement as a percentage. A decline in lead time from 10 days to 4 days represents a 60% reduction, while a decline in recovery time from 8 hours to 3 hours represents a 62.5% reduction. These calculations describe the hypothetical values only; they do not establish that modernization generally produces improvements of these magnitudes. A valid evaluation would compare multiple releases and incidents, account for workload changes, and document the measurement period.

The assessment should also include system availability, transaction correctness, defect escape rate, deployment frequency, infrastructure expenditure, security findings, and user satisfaction. No single metric can adequately represent modernization success. For example, faster deployments are not beneficial if they increase production defects, while lower infrastructure expenditure may conceal higher support costs. A balanced evaluation should examine reliability, delivery performance, maintainability, and total cost of ownership together.

5.1 Case Study 1: Incremental Modernization of a Financial Transaction System

This illustrative case examines a hypothetical financial institution operating a long-established transaction-processing application. The system supports customer accounts, transaction validation, reporting, and settlement-related workflows. Its central application and shared database have accumulated years of business rules, and changes to one function can affect several others. The institution wants to introduce digital services and improve release flexibility without interrupting essential transactions.

A complete rewrite would expose the organization to substantial behavioural and data migration risk. The proposed approach therefore retains the original transaction engine initially and introduces an API gateway to control access to selected functions. The first migration candidate is a lower-risk customer notification capability because it has a clearly defined input, output, and limited authority over financial records. Once the replacement service passes functional, security, and load tests, requests can be routed progressively to the new implementation.

The next stage separates customer profile access from the transaction engine. The migration team defines ownership rules, documents the data fields exposed through the interface, and establishes compatibility tests for older consumers. Financial posting remains within the established system until the team has verified transaction boundaries and reconciliation requirements. This sequence avoids prematurely distributing operations that depend on atomic updates.

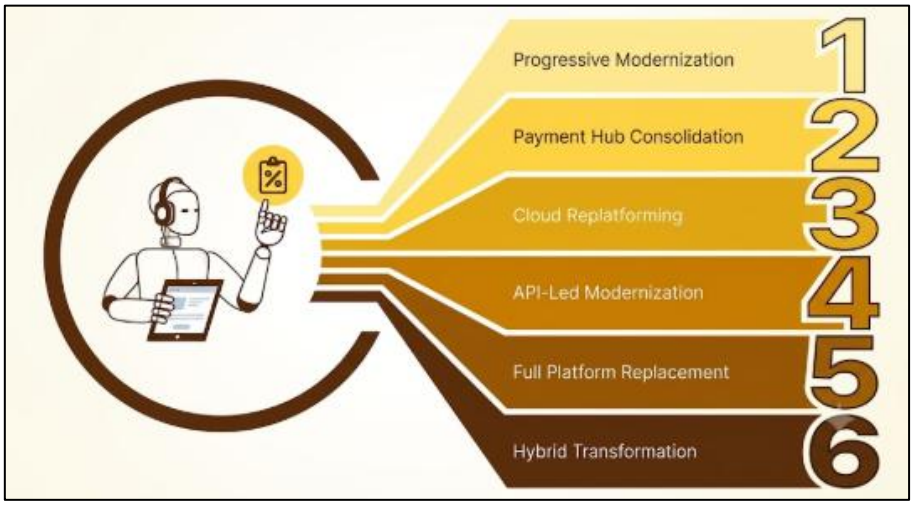


Figure 5: Incremental Modernization of a Financial Transaction System

The primary controls are automated regression tests, request tracing, access auditing, data reconciliation, and a tested rollback procedure. During coexistence, the team monitors error rates, response times, duplicate requests, and differences between expected and actual results. A failed migration

increment can be redirected to the original implementation while the issue is investigated. Legacy functions are retired only after dependent systems have been identified and the replacement meets the institution's acceptance criteria.

The analysis suggests that incremental extraction is appropriate when transaction correctness and continuity are more important than achieving architectural purity quickly. It also shows that not every capability should become an independent service. Core financial operations may remain in a cohesive module until transaction and data boundaries can be changed safely. This case is a constructed scenario based on established modernization patterns, not a verified report of a particular bank or a claim of measured financial-sector outcomes.

Case Study 2: Modernizing a Legacy Inventory Management Platform

This illustrative case considers a hypothetical retailer using an inventory application originally designed for periodic batch updates and centralized processing. The platform manages product records, stock quantities, warehouse movements, and replenishment reports. New digital sales channels require more timely stock information, but direct database access and tightly coupled business rules make integration difficult. Seasonal demand also creates uneven workloads that the existing platform handles inefficiently.

The proposed modernization begins with an assessment of stock updates, order reservations, warehouse interfaces, and reporting dependencies. The team then introduces a documented API for inventory queries while leaving stock adjustments under the authority of the original application. This provides a controlled integration point for the online sales channel without immediately replacing the rules governing stock ownership.



Figure 6: Modernizing a Legacy Inventory Management Platform

A subsequent phase introduces an event-driven notification mechanism for inventory changes. When a valid stock update occurs, an event informs downstream reporting and replenishment components. Consumers must tolerate duplicate events and delayed delivery, so each event includes a stable identifier and a versioned schema. The inventory system remains the authoritative source until the team has demonstrated that the new processing flow preserves the required business invariants.

The platform is deployed in a controlled environment with automated tests, centralized monitoring, and capacity tests that simulate peak shopping periods. The team compares inventory accuracy, API response times, failed requests, event-processing delays, and infrastructure costs against the original baseline. Rollback is retained during early releases, and the team verifies that reports and warehouse integrations continue to operate correctly.

This scenario illustrates how API-based integration and selective asynchronous processing can improve the adaptability of a legacy platform without requiring an immediate full rewrite. It also highlights a key limitation: asynchronous events do not guarantee that every sales channel sees an instantaneous, identical stock value. The application must define reservation rules and acceptable consistency delays to prevent overselling. As in Case Study 1, the case is an analytical illustration rather than independently verified empirical evidence.

6. Discussion

The analysis supports a central conclusion: modernization is an ongoing process of managing architectural change, not a one-time replacement of old technology. Incremental migration can reduce the size of individual changes and create opportunities for validation, but it does not remove the need for careful planning. During coexistence, organizations may temporarily operate duplicate infrastructure, maintain compatibility code, and support both old and new interfaces. The migration plan should therefore include explicit milestones for removing transitional components rather than allowing them to become permanent sources of complexity.

The selection of an architectural destination should be based on the problem being solved. Microservices can be advantageous when teams need independent deployment, separate scaling, or clear ownership of distinct business capabilities. However, they impose distributed-system costs, including network latency, partial failure, observability requirements, and coordination across data boundaries. A modular monolith is often a sensible alternative when one team can deliver the required changes efficiently and the application does not need independent scaling at the module level. Modernization should minimize unnecessary change while removing the constraints that prevent required business outcomes.

Data integrity is particularly important because architectural decomposition can alter transaction behaviour. A monolithic system may enforce business rules through a single database transaction, whereas a distributed workflow may involve multiple services and asynchronous events. Such changes require explicit decisions about consistency, retries, duplicate messages, compensating actions, and auditability. For financial or inventory operations, the migration team must define which invariants cannot be violated and verify them through tests and reconciliation. Merely demonstrating that APIs respond successfully is not sufficient evidence of correctness.

Security and operational readiness are equally important. Each new service can introduce additional endpoints, identities, secrets, dependencies, and communication paths. Teams need consistent authentication and authorization, encryption, vulnerability management, logging, and incident-response procedures. A modern deployment platform without reliable monitoring and recovery processes may increase operational risk instead of reducing it. The organization should establish these capabilities before expanding the number of independently deployed components.

The human and organizational dimensions also influence results. Legacy knowledge is often concentrated among a small number of specialists, making documentation and knowledge transfer essential. Developers, operators, business analysts, and security personnel should participate in migration planning because each group understands different dependencies and failure conditions. Service ownership must be clear, and teams should have the skills and authority to maintain the systems they inherit. Otherwise, technical decomposition may increase coordination overhead without improving delivery performance.

Evaluation criterion	Legacy monolith	Modular monolith	Microservices
Deployment	Usually, one application release	Usually, one application release	Independent releases when boundaries are well designed
Internal separation	May be tightly coupled	Explicit module boundaries	Separate services with network contracts
Scaling	Generally, scales as a whole	Generally, scales as a whole	Individual services can scale independently
Operational complexity	Often lower initially	Low to moderate	Higher due to distributed operations

Data management	Often shared database	Shared database with modular ownership	Service-owned data is common
Testing	Regression testing across broad functionality	Module and integration testing	Contract, integration, and end-to-end testing
Failure behaviour	Internal failures may affect the whole application	Depends on process and module isolation	Partial failures and network errors must be managed
Team structure	Often centralized	Suitable for one team or coordinated teams	Benefits from clear service and team ownership
Migration suitability	Existing state	Useful intermediate or final architecture	Suitable when independent deployment or scaling justifies cost
Principal risk	Accumulating technical debt	Weak boundaries may erode over time	Excessive fragmentation and distributed-system complexity

Source: Author's synthesis based on Bass et al. (2021), Newman (2021), and Fowler and Lewis (2014). These are qualitative comparisons, not universal performance measurements.

The comparison indicates that microservices are not automatically superior across all criteria. They provide stronger deployment independence but require additional engineering and operational maturity. A modular monolith can improve maintainability at lower complexity, while an unchanged legacy application may remain appropriate when it continues to meet business requirements and presents acceptable risk. The preferred option is the least complex architecture that satisfies the system's justified requirements.

Modernization outcomes should be evaluated using a consistent measurement framework. Delivery lead time and deployment frequency indicate how quickly changes reach users; change failure rate and recovery time indicate operational stability; availability and latency indicate service quality; and total cost of ownership captures infrastructure, licensing, migration, training, and support expenses. Measurements should be collected before and after each migration stage, with definitions kept consistent. Where no reliable baseline exists, the first phase should establish one instead of reporting unsupported percentage improvements.

The two illustrative case studies reinforce these conclusions. The financial transaction scenario prioritizes correctness, controlled interfaces, and rollback because errors can affect critical records. The inventory scenario uses APIs and asynchronous notifications to support integration but explicitly manages the risk of delayed or duplicated events. The appropriate architecture therefore depends on domain-specific invariants and operating conditions. Both scenarios Favor selective extraction over a simultaneous rewrite, but neither implies that every legacy system should ultimately become a large collection of microservices.

A limitation of this article is that its results are based on literature synthesis and constructed scenarios rather than a systematic review with formal screening statistics or original measurements from participating organizations. The illustrative graphs demonstrate how performance indicators can be reported, not what organizations should expect to achieve. Future empirical research could evaluate completed migrations across multiple sectors, compare total lifecycle costs, and examine how team structure and system criticality affect outcomes. Such studies would help quantify the circumstances in which modular monoliths, microservices, or other modernization patterns provide the strongest value.

7. Conclusion

Migrating legacy systems to modern software architectures is a complex process that combines technical restructuring, business continuity, data management, security, and organizational change. Legacy applications may contain valuable business rules and dependable operational capabilities even when their technologies and architectures have become difficult to maintain. Modernization should

therefore preserve useful behaviour while addressing the specific constraints that prevent systems from meeting present and future requirements. The literature reviewed in this article indicates that no single migration strategy is appropriate for every application. Rehosting and replat forming can be sufficient when infrastructure or platform limitations are the primary concern. Refactoring can improve internal code quality, while rearchitecting and service extraction are appropriate when structural constraints prevent the required level of adaptability. Replacement may be preferable when an existing product or a new application provides a more sustainable solution. The choice should be guided by business value, technical risk, system dependencies, and the cost of operating the proposed architecture. Incremental migration provides a practical means of reducing risk in systems that cannot be interrupted for a complete replacement. The strangler fig pattern and branch by abstraction enable old and new implementations to coexist while selected functionality is validated and transferred. Their effectiveness depends on clear routing, explicit interfaces, comprehensive testing, controlled data ownership, monitoring, and rollback procedures. Legacy functionality should be retired only after replacement behaviour and dependent integrations have been verified. The analysis also emphasizes that microservices introduce trade-offs. Independent deployment and scaling can be valuable, but they require disciplined service boundaries, distributed observability, security controls, and data-consistency mechanisms. Modular monoliths can offer a more economical destination when independent deployment is unnecessary. Architectural modernization should consequently prioritize simplicity, maintainability, and measurable business outcomes rather than the adoption of technologies. The two illustrative cases demonstrate how modernization priorities differ by domain. Financial transaction systems require strict attention to correctness, auditability, and continuity, whereas inventory platforms may benefit from controlled APIs and asynchronous processing when consistency requirements are explicitly managed. Both scenarios support phased migration, baseline measurement, automated verification, and operational readiness. Ultimately, modernization success should be demonstrated through evidence. Organizations should compare availability, response times, deployment lead time, recovery time, defect rates, security posture, and total cost of ownership against documented baselines. They should also invest in knowledge transfer, clear ownership, and the operational skills needed to maintain the target architecture. The central recommendation is to modernize incrementally, select architecture according to genuine requirements, and retire legacy components only when the replacement is proven. This approach offers a balanced route toward resilient, maintainable, and adaptable software systems while reducing unnecessary cost and disruption.

References

[1] Amazon Web Services. (n.d.). Branch by abstraction pattern. AWS Prescriptive Guidance. https://docs.aws.amazon.com/prescriptive-guidance/latest/modernization-decomposing-monoliths/branch-by-abstraction.html

[2] Amazon Web Services. (n.d.). The strangler fig pattern. AWS Prescriptive Guidance. https://docs.aws.amazon.com/prescriptive-guidance/latest/modernization-decomposing-monoliths/strangler-fig.html

[3] Amazon Web Services. (2023). AWS Well-Architected Framework. https://docs.aws.amazon.com/wellarchitected/latest/framework/welcome.html

[4] Bass, L., Clements, P., & Kazman, R. (2021). Software architecture in practice (4th ed.). Addison-Wesley Professional.

[5] Burns, B., Grant, B., Oppenheimer, D., Brewer, E., & Wilkes, J. (2016). Borg, Omega, and Kubernetes. ACM Queue, 14(1), 70–93. https://doi.org/10.1145/2898442.2898444

[6] Conway, M. E. (1968). How do committees invent? Datamation, 14(4), 28–31.

[7] Evans, E. (2003). Domain-driven design: Tackling complexity in the heart of software. Addison-Wesley.

[8] Fowler, M. (2003). Evolutionary database design. https://martinfowler.com/articles/evodb.html

[9] Fowler, M. (2004). Strangler application. https://martinfowler.com/bliki/StranglerFigApplication.html

[10] Fowler, M. (2018). Refactoring: Improving the design of existing code (2nd ed.). Addison-Wesley Professional.

[11] Fowler, M., & Lewis, J. (2014). Microservices: A definition of this new architectural term.

https://martinfowler.com/articles/microservices.html
[12] Forsgren, N., Humble, J., & Kim, G. (2018). Accelerate: The science of lean software and DevOps: Building and scaling high performing technology organizations. IT Revolution Press.
[13] Hohpe, G., & Woolf, B. (2003). Enterprise integration patterns: Designing, building, and deploying messaging solutions. Addison-Wesley.
[14] Humble, J., & Farley, D. (2010). Continuous delivery: Reliable software releases through build, test, and deployment automation. Addison-Wesley.
[15] ISO/IEC/IEEE. (2022). ISO/IEC/IEEE 42010:2022 systems and software engineering—Architecture description. International Organization for Standardization.
[16] Kim, G., Humble, J., Debois, P., & Willis, J. (2016). The DevOps handbook: How to create world-class agility, reliability, and security in technology organizations. IT Revolution Press.
[17] Lewis, J., & Fowler, M. (2014). Microservices. Martin Fowler. https://martinfowler.com/articles/microservices.html
[18] Microsoft. (n.d.). Microsoft Azure Well-Architected Framework. Microsoft Learn. https://learn.microsoft.com/en-us/azure/well-architected/
[19] Microsoft. (n.d.). Cloud Adoption Framework. Microsoft Learn. https://learn.microsoft.com/en-us/azure/cloud-adoption-framework/
[20] Mell, P., & Grance, T. (2011). The NIST definition of cloud computing (NIST Special Publication 800-145). National Institute of Standards and Technology. https://doi.org/10.6028/NIST.SP.800-145
[21] Newman, S. (2021). Building microservices: Designing fine-grained systems (2nd ed.). O'Reilly Media.
[22] Richards, M. (2015). Software architecture patterns. O'Reilly Media.
[23] Shaw, M., & Garlan, D. (1996). Software architecture: Perspectives on an emerging discipline. Prentice Hall.
[24] Souppaya, M., Morello, J., & Scarfone, K. (2019). Application container security guide (NIST Special Publication 800-190). National Institute of Standards and Technology. https://doi.org/10.6028/NIST.SP.800-190
[25] Vernon, V. (2013). Implementing domain-driven design. Addison-Wesley Professional.