International Journal of Computer (IJC)
ISSN 2307-4523 (Print & Online)
https://ijcjournal.org/index.php/InternationalJournalOfComputer/index

Analysis of Migration Strategies for Legacy Software Complexes to Modern Technology Stacks

Abdul Nadeem Mohammed*

Full Stack Developer, Techlance Solutions Inc, Michigan, United States
Email: abdulnadeemms@gmail.com

Abstract

The article examines strategies for migrating legacy software systems to modern technology stacks, accounting for architectural constraints, technical debt, hidden domain logic, and organizational dependencies. The aim of the study is to compare the principal transition trajectories, identify the factors shaping strategy selection, and determine the conditions under which migration preserves the operability of critical functions and establishes a foundation for the development of the enterprise digital environment. The study's relevance lies in the growing requirements for integration, scaling, independent component deployment, and the manageability of the engineering lifecycle, which legacy systems often struggle with. The novelty of the article lies in considering migration as a unified architectural, technological, and organizational process in which the choice of strategy is determined by the condition of the source complex, the density of integrations, the value of accumulated logic, and the level of acceptable risk. It is concluded that no universal migration strategy exists and that the greatest resilience is provided by a phased transition based on architectural audit, inventory of data and interfaces, sequential transfer of functions, and control of the integrity of application scenarios. The article will be useful to researchers, software system architects, modernization engineers, and IT transformation managers.

Keywords: legacy software complexes; system migration; technology stack; microservice architecture; software modernization.

Received: 2/30/2026
Accepted: 4/30/2026
Published: 5/8/2026

---

* *Corresponding author.*

1. Introduction

The modernization of legacy software complexes has become one of the central tasks of applied engineering, since such systems continue to retain the core of computational, accounting, production, and administrative processes in large organizations, while their architectural form is becoming less and less aligned with the requirements of integration, scaling, and frequent functional change [1]. Studies over the past few years have linked the renewal of such complexes to broader processes of digital transformation and show that legacy solutions impose direct constraints on intersystem interaction, data reuse, and the restructuring of services around user needs Reference [2]. For this reason, the topic of migration extends beyond narrow engineering optimization and affects the resilience of organizational processes, the manageability of architecture, and an enterprise's ability to sustain the long-term development of the application landscape [3].

The problematic nature of legacy complexes stems from the accumulation of structural complexity, hidden domain logic, and dependence on languages, platforms, and deployment methods that are poorly aligned with current development and maintenance requirements [1]. A systematic review of approaches to service extraction from legacy systems shows that the choice of transformation trajectory depends on modernization goals, the composition of available artifacts, the organization of the source development process, and the expected outcome. For this reason, no single migration scheme exists for all classes of systems [4]. The same review notes that many approaches still lack robust tool support and limited validation in industrial systems, increasing the risk of errors when transferring critical logic, data, and integration links. Such complexes are also characterized by a high cost of change, as the loss of specialists on the source platform, the scarcity of documentation, and the dense coupling of modules complicate local rework even when the business requirement appears limited in scope [5].

The relevance of modernization is reinforced by the fact that the transition to distributed architectures, container environments, and cloud infrastructure requires a different organization of system boundaries, data, and component interactions [6]. A systematic mapping study on the transformation of legacy systems into service architecture showed a steady increase in interest in phased migration scenarios, structured around a sequence of macro-actions. This reflects a practical need to reduce the risk of interrupting critical processes [7]. Within a similar logic, architecture-oriented studies of migration to service form point to the lengthy and costly character of such a transition and to the shortage of systemic guidelines for decision-making on decomposition and the planning of work stages [8]. This understanding underscores the need to analyze migration strategies that preserve the system core's operability and pave the way for a modern technology stack. The study aims to prove the following hypothesis:

H1: The selection of a migration strategy for legacy software complexes depends on the architectural condition of the source system, the density of integrations, the value of embedded domain logic, and the acceptable level of organizational risk.

H2: Phased migration based on architectural audit, inventory of data and interfaces, sequential transfer of functions, and verification of application scenario integrity provides the most resilient path for preserving critical operability during transition to a modern technology stack.

2. Materials and Methodology

The study of migration strategies for legacy software complexes to modern technology stacks is based on the analysis of 15 sources covering systematic reviews, literature mappings, architectural studies, reengineering works, and publications devoted to the cloud, service-oriented, and microservice transformation of software systems [1, 4, 6, 7, 10]. The material base consisted of studies that considered legacy complexes as carriers of critical domain logic, accumulated technical debt, organizational dependency, and architectural inertia [1, 2, 3, 9]. The theoretical foundation included works on service identification and monolith decomposition, since they provide the conceptual framework for analyzing the depth of transformation, component boundaries, and variants of phased transition [4, 6, 15]. An additional layer of materials was formed by studies linking migration with the transition to cloud infrastructure, container environments, orchestration, DevOps practices, and a new logic of the engineering lifecycle, where the architectural solution is considered together with the mode of delivery, maintenance, and observability of the system [11, 12, 14].

The research methodology combined comparative analysis of migration strategies, thematic comparison of architectural approaches, and interpretation of risk factors identified in academic literature [1, 7, 8, 10]. In the comparative analysis, scenarios of transfer with minor changes, partial platform adaptation, deep architectural rework, phased service extraction, and complete rebuilding of the system on a new technological basis were correlated [1, 8, 10]. Thematic analysis identified recurring categories associated with component coupling, hidden dependencies, documentation scarcity, decomposition complexity, data transfer, and the influence of the organizational environment on modernization [5, 7, 9, 13]. The method of interpretation was applied to establish the relationship between the condition of the source complex and the choice of migration trajectory, including cases where architectural wear, high criticality of business functions, and the density of integrations require a phased transition with control over the integrity of application logic and preservation of system operability during the transition period [2, 8, 11, 12].

3. Results

The body of prior research outlines several converging trajectories of inquiry that frame the present study. Hasan and his colleagues examine cloud migration through an architectural lens and treat legacy complexes as mission-critical monoliths whose internal coupling determines the feasible transformation envelope [1], while Irani and his colleagues extend this view into the public administration domain and trace how legacy dependencies propagate through inter-system data exchange [2]. Abu Bakar and his colleagues formulate a conceptual model of citizen-centric modernization and connect architectural renewal with service delivery objectives [3]. Abdellatif and his colleagues construct a taxonomy of service identification approaches and reveal the fragmentation of decomposition methods across industrial contexts [4], a gap that Abgaz and his colleagues address through a systematic review of monolith-to-microservices transitions [6]. Fávero and his colleagues map the field of legacy modernization toward microservice architecture and document the rise of phased migration patterns [7], while Fritzsch and his colleaguesargue for an architecture-centric methodology that anchors decomposition in explicit decision criteria [8].

Rinta-Kahila and his colleagues shift the analytical frame toward sociotechnical dynamics and demonstrate how technical debt persists through replacement attempts [9]. Martínez Saucedo and his colleagues catalogue the drivers and obstacles of microservice migration [10], da Silva and his colleagues integrate reengineering with DevOps practices in mission-critical settings [11], and Nascimento and his colleagues address the migration from container-based to cloud-native deployments under availability and security constraints [12]. Freire and his colleagues apply aspect-oriented programming to production monolith decomposition [13], Al-Qora'n and Ahmad investigate modular monolith architecture in cloud environments [14], and Trabelsi and his colleagues introduce type-based machine learning techniques for microservice identification [15]. Lano and his colleagues contribute agile model-driven reengineering as a complementary path [5], and Nadeem analyzes trust boundaries in virtualized environments that host migrated workloads [16]. The convergence of these works around architecture, organizational coupling, data integrity, and engineering practice motivates the integrated perspective adopted in this article.

A software complex should be understood as a system in long-term operation that supports the basic functions of an organization, stores accumulated domain logic, and relies on technological solutions of earlier generations. In a review of architectural approaches to migrating such systems to the cloud, they are described as mission-critical systems that contain the key functions of an organization, are usually built in a monolithic form, and become more complex as changes accumulate, turning maintenance into a source of growing costs and constraints on development [1]. Legacy status is determined by the age of the code taken in isolation, but by the combination of architectural inertia, strong dependence on past design decisions, and weak adaptability to new environmental requirements. A study of the influence of such systems on digital transformation in public administration shows that the complexity of transformation grows as systems are integrated into a broader circuit of data and service exchange, where each connection intensifies dependence on the source architecture [2].

Legacy architectures are characterized by dense component coupling, large modules with blurred boundaries of responsibility, hidden dependencies between code, data, and maintenance procedures, and weak partitioning into parts suitable for separate modification and deployment. A systematic mapping of works on transforming legacy systems into service architecture records that such systems are difficult to modify, scale, and maintain, while typical problems include coupling, maintenance cost, lengthy release cycles, and declining team productivity [7]. Organizational constraints have the same nature. They manifest as dependence on a narrow circle of specialists, weak documentation, divergence between the formal process model and the system's actual logic, and a high risk of knowledge loss when individual components are replaced. A study of the sociotechnical dynamics of legacy system replacement shows that technical debt in such cases is linked to the structure of the organizational environment, the distribution of responsibility, and the sequence of managerial decisions, as a result of which the modernization attempt itself may reproduce previous constraints in a new form [9].

The reasons for migration to modern technology stacks derive from this aggregate of characteristics. The growth of maintenance costs is associated with increasing system complexity, expansion of the functional scope, and intensification of module dependencies, resulting in each modification affecting an ever larger number of code sections and infrastructure [1]. Technical debt and architectural constraints restrain scaling and integration, since a system created for an isolated operating mode is poorly aligned with requirements for data exchange, flexible

deployment, and the separate evolution of parts of the software complex. An analysis of the migration from monolithic systems to microservice architecture shows that the key incentives for such a transition are scalability, maintainability, and the partitioning of the system into manageable elements, although the process itself remains complex and variable [10]. The choice of migration trajectory is also influenced by requirements for security, reliability, and the speed of releasing changes. A study of reengineering processes in mission-critical public systems shows that transitioning to a service architecture and incorporating continuous delivery practices are seen as responses to high technical debt, high maintenance costs, and the need to preserve system quality during transformation [11]. These reasons link migration with digital transformation as a long-term change in architecture, processes, and methods of governing the software landscape. Table 1 illustrates the key characteristics of legacy systems and their implications for migration.

Table 1: Key Characteristics of Legacy Systems and Their Migration Implications

Dimension	Legacy system characteristics	Migration implications
Architecture	Monolithic structure, strong dependence on past design decisions, accumulated domain logic	Requires a shift toward a more modular architecture
Coupling	Tight coupling and hidden dependencies between code, data, and operations	Calls for decomposition into more independent components
Maintenance	Blurred module boundaries, long release cycles, high maintenance costs	Migration aims to improve maintainability and delivery speed
Organization	Reliance on a small number of experts and weak documentation	Migration must address organizational as well as technical constraints
Integration	Poor fit for data exchange, scalability, and independent development	Encourages adoption of service-based architectural approaches
Drivers	Technical debt, growing complexity, and higher demands for reliability and security	Makes migration a strategic step in digital transformation

The target migration environment is connected with the transition from a closed software contour to a computing environment in which resources are allocated on demand, components are deployed independently, and architectural boundaries become the subject of separate design. A review of architectural approaches to migrating legacy systems to the cloud environment shows that the greatest benefits are achieved by solutions whose architecture permits the division of components into isolated parts and the transfer of computational load to distributed infrastructure [1]. Container packaging and orchestration tools in this environment serve as mechanisms for reproducible deployment, dependency coordination, and component lifecycle management. A study of migration of containerized applications to cloud-oriented orchestrated environments links this form of deployment with requirements for availability, scalability, and protection that are difficult to ensure within older application placement schemes [12]. The transition to microservice and service architecture complements this shift with a new principle of system composition, in which functions are cut along business boundaries and obtain autonomous interaction interfaces. A systematic study of migrating production monoliths to microservices shows

that this transition aims to increase maintainability and scalability, though it intensifies requirements for data coordination, call tracing, and architectural control [13].

The modern technology stack also includes a different mode of engineering work, in which chains of build, verification, delivery, and observability are embedded in the everyday cycle of system change. A study of reengineering mission-critical legacy systems into service-oriented systems links migration success to the adoption of continuous integration, continuous delivery, and collaboration between development and operations teams, since without these mechanisms, the new architectural arrangement loses a considerable part of its manageability [11]. Such an environment requires robust observability, enabling connections between the behavior of individual services and flows of user operations and infrastructural events. The technological basis of the application layer also changes. Modern languages, development frameworks, and data management systems are selected based on modularity, test automation, support for distributed processing, and integration with the cloud environment, which expands the design space compared with the rigidly coupled stacks of earlier generations [14].

From these properties of the target environment follow the basic migration strategies. The first strategy is to replace the system with a new infrastructure while preserving the application's main structure. Such a move reduces dependence on obsolete hardware and opens access to cloud resources, while preserving the source architectural constraints [1]. The second strategy presupposes transfer with partial adaptation of platform components, when storage, deployment, or execution means change, while the core of application logic retains its previous form. The third strategy involves reworking the system's internal structure, leading to the extraction of services, the creation of new interfaces, and the revision of domain boundaries. Works on microservice identification in legacy systems show that this path requires specialized decomposition methods, as errors in boundary definition create new dependencies and complicate maintenance [15]. The fourth strategy involves recreating the system on a new technological basis by transferring domain logic into a newly designed architecture. The fifth strategy is built around replacing the existing complex with a ready-made product or platform solution. A comparison of these trajectories in studies of migration to microservice architecture shows that the choice depends on the depth of technical debt, the density of integrations, requirements concerning timeframes, and the acceptable level of organizational risk [10]. This is why the analysis of migration strategies requires consideration of the technical goal, the architectural form, and the mode of change as a unified object of study.

Legacy migration scenarios become more concrete when the source platforms and data stores are specified. A hypothetical public insurance system may run on an IBM zSeries mainframe with COBOL transaction programs, CICS middleware, and a DB2 for z/OS database. In such a case, the first migration step may move reporting and document-generation functions to containerized Java or C# services deployed in Kubernetes, while the transactional core remains on the mainframe. The target data layer may combine PostgreSQL for operational services and Apache Kafka for event exchange with the remaining legacy modules [16]. This configuration supports gradual separation of business capabilities without immediate replacement of the core transaction engine.

A different trajectory can be illustrated by a manufacturing complex deployed on an older Unisys ClearPath environment with batch processing modules and hierarchical data storage. Its production planning logic may be

transferred to a modular monolith built in Spring Boot or .NET, while machine telemetry and scheduling services are redesigned as independent microservices. The legacy data model may be mapped to PostgreSQL for structured production records and to a time-series platform such as InfluxDB for equipment signals. Such a migration path is suitable where plant operations depend on stable planning routines and where the extraction of isolated services must follow the real boundaries of workshop processes, inventory flows, and maintenance cycles.

Banking and logistics environments often involve another class of migration problem. A hypothetical settlement platform may use an IBM System z mainframe, IMS TM for transaction management, and IMS DB for high-volume operational records. Migration in this setting may direct customer profile management, notification functions, and case tracking to cloud-native services written in Go or Java, while payment clearing remains in the source environment for a defined transition period. Oracle Database or PostgreSQL may serve the new service layer, and Redis may support session data and high-frequency access patterns. The movement from IMS-based structures to relational or document-oriented models requires precise reconstruction of record semantics, validation rules, and temporal dependencies embedded in transaction chains.

A retail or public sector system may also originate from an HP NonStop platform or from an ICL/Fujitsu mainframe used for uninterrupted processing of orders, registries, or account updates. A hypothetical migration route may replace terminal-based front ends with web applications, move integration logic to API gateways, and relocate selected domain services to Linux-based environments managed through containers. The target persistence layer may include Microsoft SQL Server for reporting and coordination functions, MongoDB for flexible document records, and a data lake for historical archives and audit traces. This type of example shows that migration is shaped by the interaction between source hardware, execution model, transaction guarantees, and the structure of accumulated business data.

4. Discussion

Comparing migration strategies requires a set of criteria that reflect the technical, organizational, and economic dimensions of transition. Such criteria include the volume of required changes, the duration of the work, the cost of transformation, the burden on maintenance teams, the probability of failures during transition, the depth of architectural intervention, and the degree of influence on current business processes. Strategies with a small volume of change usually provide a faster launch and a lower initial cost, while preserving many of the constraints of the source system. Strategies with deep reworking require a larger budget, a longer design cycle, and broad involvement of specialists in architecture, data, and application logic. The difference between them is also manifested in the nature of the result. Some approaches transfer the system to a new environment with little revision of its internal arrangement. Others create conditions for the partitioning of functions, extraction of autonomous components, and restructuring of data flows. For this reason, the evaluation of timeframes and cost cannot be separated from the question of which properties the system must acquire after migration is completed.

The level of risk changes together with the depth of architectural intervention and the degree of organizational dependence on the source complex. A transfer that preserves the previous logic reduces the risk of losing hidden functions and shortens the user-adaptation period, while leaving the dependence on accumulated technical debt

intact. Architectural rework opens the way to better scalability, simplified maintenance, and accelerated release delivery, though it increases the risk of errors in component boundaries, interaction schemes, and data processing rules. Complete rebuilding of the system is associated with the highest risk of divergence between the new implementation and the accumulated practice of organizational work, since some critical rules are embedded in the old complex in an implicit form. Strategy selection depends on the condition of the legacy complex. If the architectural framework preserves coherence and permits local changes, a phased transformation can serve as a basis for it. If the system is overloaded with dependencies, poorly documented, and restrains the development of key processes, a deeper migration scenario is required. A comparison of migration strategies based on key evaluation criteria is presented in Table 2.

Table 2: Comparison of Migration Strategies by Key Evaluation Criteria

Evaluation criterion	**Low-change migration**	**Incremental transformation**	**Deep architectural refactoring**	**Full system rebuilding**
Scope of change	Minimal changes to the existing system	Selective and gradual changes to system components	Extensive redesign of components, data flows, and interfaces	Complete redevelopment of the system
Time and cost	Lower initial cost and shorter implementation time	Moderate cost and duration	Higher cost and longer implementation cycle	Highest cost and longest duration
Risk level	Lower transition risk, but legacy constraints remain	Balanced risk, depending on system complexity	Higher risk of design and integration errors	Highest risk of mismatch with existing organizational practices
Architectural impact	Limited architectural improvement	Enables partial architectural improvement	Significant improvement in modularity and maintainability	Creates a new architectural foundation
Business process impact	Minimal short-term disruption	Allows gradual adaptation of business processes	Requires substantial process adjustment	Strong impact on existing business processes

Phased migration is built around such a transformation of the system in which its functions are transferred in parts. One approach is based on the gradual displacement of the old contour by new components. The new parts assume responsibility for individual operations, after which dependence on the source core decreases. Such a path helps preserve the system's working state during the period of change.

Another approach involves extracting a stable layer of software interfaces. Through this layer, the rules of data exchange and function invocation are fixed. This simplifies internal restructuring, as external consumers continue to operate under a familiar scheme. A similar task is solved by decomposing the monolith into domain areas. It helps identify parts of the system with their own boundaries of responsibility and development cycles.

The transitional stage often requires parallel operation of the old and new systems. Some operations at this point are already transferred to the new contour, while others remain in the previous environment. The greatest difficulty in such a scheme is connected with data. It is necessary to coordinate formats, transformation rules, and synchronization order. Otherwise, divergence between the two contours will begin to destroy process integrity.

In a hypothetical migration of a regional tax administration system from an IBM zSeries mainframe to a cloud-based Java platform, cost and schedule pressure emerge from the data layer before application refactoring reaches its peak. The source system may store records in DB2 for z/OS and exchange files in EBCDIC, while the target platform may use PostgreSQL with UTF-8 encoding. A pilot transfer of 18 terabytes may reveal character conversion defects in personal names, address fields, and archival identifiers. This defect class may force the team to introduce an intermediate validation layer, rebuild parsing rules, and repeat test loads across several historical datasets. Under such conditions, a migration stage planned for four months may expand to seven months, while the data work package may grow from 420,000 to 690,000 US dollars due to reprocessing, audit support, and manual reconciliation of corrupted records.

A different hypothetical case may involve a public insurance platform built on an HP NonStop environment with transaction logic tied to proprietary batch utilities and terminal interfaces. The target architecture may rely on containerized .NET services deployed in Kubernetes. The migration team may discover that a claims validation module depends on a vendor-specific library for message formatting and queue control that has no direct equivalent in the container environment. Emulation may reproduce part of the runtime behavior, yet failure patterns under peak load may remain inconsistent with the source platform. This conflict may delay integration testing by eleven weeks and require a separate engineering stream devoted to adapter development and synthetic workload testing. A budget reserve of 160,000 US dollars may be consumed by this issue alone through contractor support, environment duplication, and repeated fault isolation sessions.

A third hypothetical example may concern a logistics platform moved from an IBM System z environment with IMS DB to a service-based architecture using Go services, Oracle Database, and Kafka. During migration, the team may find that shipment status codes and exception paths are embedded in transaction sequences. A narrow technical problem may arise when converted UTF-8 messages pass through a service chain that still receives fixed-length records shaped by mainframe field conventions. Field truncation may alter event semantics and generate duplicate exception cases in downstream services. Resolution may require redesign of message contracts, extension of regression test packs, and a frozen release window for six weeks. The direct cost of this delay may reach 240,000 US dollars, while the indirect cost may include postponed interface rollout for partner systems and continued parallel operation of the legacy queue layer.

The key migration risk is associated with hidden domain logic accumulated in old code, workaround mechanisms, and local processing rules. During transfer, such logic may fall out of the new system's model. The problem is intensified by errors in data transformation and the underestimation of integration links. Even a small change in one module can affect external services, reporting procedures, and internal regulations.

Organizational risk also has substantial significance. Employees perceive migration as interference in the habitual

order of work. This slows implementation and increases conflicts between technical and business units. Against this background, the strategy is chosen based on system criticality, the condition of the architecture, available competencies, resources, and the expected period of further operation.

If the system retains central processes and the cost of error is high, preference is given to a path involving gradual transfer and the possibility of rollback. If architectural wear has reached a high degree, maintenance requires excessive expenditures, and development on the old basis loses practical meaning, a deeper transformation is required. Economic evaluation in such a case involves comparing the cost of transition with the cost of preserving accumulated constraints. The phased system migration process is shown in Figure 1.

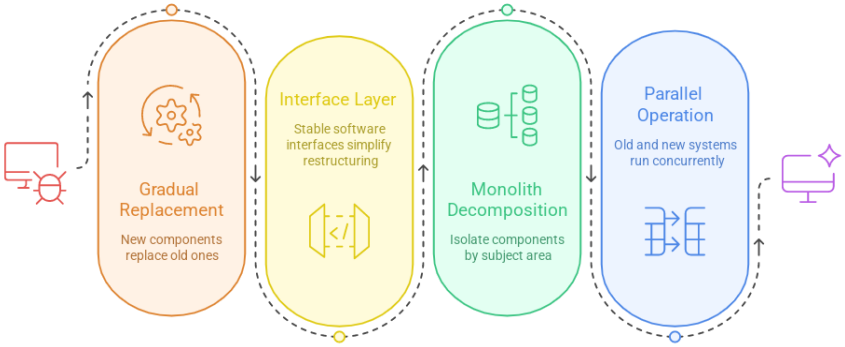

Figure 1: Phased System Migration Process

Successful migration requires preparation, combining technical analysis with organizational discipline and an understanding of the system's application logic. Work begins with an architectural and functional audit that enables identification of component composition, the true boundaries of modules, critical scenarios, obsolete code sections, and hidden dependencies between subsystems. The next step involves a complete inventory of data, interfaces, external links, and regulatory procedures, since it is precisely in these links that points of failure and sources of divergence between the old and new contours are most often concentrated. The transition should be built in stages, through the sequential transfer of functions and verification of each change in the working process, since simultaneous replacement of the entire system increases the risk of logic loss, integration failures, and long service interruptions.

Automated quality verification plays a substantial role, including testing application scenarios, ensuring data integrity, verifying interface compatibility, and tracking system behavior after each change. Separate significance belongs to the management of changes within the organization. Development, maintenance, analytics, and business teams must work within a common decision contour, understand the aims of migration, master new tools, and participate in coordinating transitional rules. Such an approach reduces the risk of errors, maintains process manageability, and creates conditions for the system's stable development after migration is complete.

The aggregated results indicate that migration outcomes diverge along two intersecting axes. The architectural depth of intervention and the density of organizational coupling around the source complex. Cases drawn from mainframe environments with COBOL transaction cores, IMS-based settlement layers, and HP NonStop

platforms reveal that the dominant cost driver shifts from code refactoring toward data semantics, character encoding, and the reconstruction of implicit transaction rules embedded in fixed-length record structures. The comparative table of strategies shows that low-change transfers contain transitional risk inside a narrow technical perimeter while leaving architectural debt intact, whereas deep refactoring redistributes risk toward boundary definition, message contracts, and the alignment of new service decompositions with established business workflows.

The hypothetical migration scenarios further clarify that schedule expansion and budget overruns concentrate around three recurring zones. Encoding conversion under heterogeneous character sets, emulation gaps for vendor-specific runtime libraries, and field-level semantic drift in cross-service message chains. These findings refine the interpretation of phased migration as an architectural sequence and as a discipline of controlled exposure to the heterogeneity of legacy artifacts, where each transferred function rebalances the load between technical correctness and the preservation of organizational continuity.

5. Conclusion

The study shows that migrating legacy software complexes to modern technology stacks is a multilayered task in which architectural constraints, accumulated domain logic, technical debt, and organizational dependence on established practices form a unified set of factors. The legacy nature of such systems is determined by the combination of a monolithic structure, dense coupling, weak modular partitioning, scarce documentation, and their inclusion in the organization's critical processes. Under these conditions, migration serves to preserve the manageability of the application landscape, improve maintainability, expand integration capabilities, and lay a foundation for further development of the digital environment.

Comparison of migration strategies shows that the choice of trajectory is determined by the depth of architectural wear, the density of integration links, the value of accumulated logic, the acceptable level of risk, and the organization’s resources. A transfer with minor changes reduces the burden during the transitional period while preserving a significant portion of the previous constraints. Deep architectural rework and complete rebuilding open a path toward modularity, independent component deployment, and accelerated change delivery, though they require more complex decomposition, revision of domain boundaries, control over data, and coordination of the new system with the actual practice of enterprise work.

The proposed hypotheses were confirmed by the results of the study. Hypothesis 1 was confirmed through the comparative analysis of migration strategies, which showed that strategy selection is shaped by architectural wear, integration density, hidden domain logic, documentation quality, system criticality, and organizational constraints. Hypothesis 2 was confirmed through the synthesis of the examined studies and the discussion of migration scenarios, which demonstrated that phased migration offers the strongest conditions for preserving critical functions, controlling data and interface consistency, reducing transition risk, and sustaining system operability throughout modernization.The greatest resilience is provided by phased migration based on architectural and functional audits, data and interface inventory, sequential transfer of functions, and continuous verification of the integrity of application scenarios. Management of hidden domain logic, control of transitional integrations, data

synchronization, and inclusion in the development, maintenance, analytics, and business teams are critical. The success of migration is connected with the quality of preparation and with the degree to which the chosen strategy is correlated with the condition of the source complex and with the horizon of its further operation.The scope of the present study is shaped by several boundary conditions that situate the obtained conclusions within a defined analytical perimeter. The argument rests on fifteen academic sources covering systematic reviews, architectural studies, and reengineering works, which establishes a coherent thematic field and opens a natural extension toward industrial reports, vendor case archives, and longitudinal observations of completed migration programs. The migration scenarios across mainframe, manufacturing, banking, and logistics settings are presented in hypothetical form to illustrate the interaction between source platforms, transaction guarantees, and target architectures, and their parametric calibration against measured production data marks a productive direction for subsequent inquiry. The comparative evaluation of strategies operates through qualitative criteria synthesized from the literature, and the construction of quantitative metrics tied to defect density, integration latency, and operational continuity represents a further analytical horizon. The interpretive framework treats architectural, organizational, and data dimensions as a unified object, and the disaggregation of these dimensions through dedicated empirical instruments offers a path toward more granular decision support for modernization programs.

References

[1] M. H. Hasan, M. H. Osman, N. I. Admodisastro, and M. S. Muhammad, “Legacy systems to cloud migration: A review from the architectural perspective,” Journal of Systems and Software, vol. 202, p. 111702, Aug. 2023, doi: https://doi.org/10.1016/j.jss.2023.111702.

[2] Z. Irani, R. M. Abril, V. Weerakkody, A. Omar, and U. Sivarajah, “The impact of legacy systems on digital transformation in European public administration: Lesson learned from a multi case analysis,” Government Information Quarterly, vol. 40, no. 1, p. 101784, Jan. 2023, doi: https://doi.org/10.1016/j.giq.2022.101784.

[3] H. Abu Bakar, R. Razali, and D. I. Jambari, “Legacy Systems Modernisation for Citizen-Centric Digital Government: A Conceptual Model,” Sustainability, vol. 13, no. 23, p. 13112, Nov. 2021, doi: https://doi.org/10.3390/su132313112.

[4] M. Abdellatif et al., “A taxonomy of service identification approaches for legacy software systems modernization,” Journal of Systems and Software, vol. 173, p. 110868, Mar. 2021, doi: https://doi.org/10.1016/j.jss.2020.110868.

[5] K. Lano, H. Haughton, Z. Yuan, and H. Alfraihi, “Agile model-driven re-engineering,” Innovations in Systems and Software Engineering, vol. 20, pp. 559–584, Jun. 2024, doi: https://doi.org/10.1007/s11334-024-00568-z.

[6] Y. M. Abgaz et al., “Decomposition of Monolith Applications Into Microservices Architectures: A Systematic Review,” IEEE Transactions on Software Engineering, vol. 49, no. 8, pp. 4213–4242, Jan. 2023, doi: https://doi.org/10.1109/tse.2023.3287297.

[7] L. F. Fávero, N. Rodrigues, and F. J. Affonso, “A Systematic Mapping Study on the Modernization of Legacy Systems to Microservice Architecture,” Applied System Innovation, vol. 8, no. 4, p. 86, Jun. 2025, doi: https://doi.org/10.3390/asi8040086.

[8] J. Fritzsch, J. Bogner, M. Haug, S. Wagner, and A. Zimmermann, “Towards an Architecture-Centric Methodology for Migrating to Microservices,” Lecture notes in business information processing, vol. 489, pp. 39–47, Dec. 2023, doi: https://doi.org/10.1007/978-3-031-48550-3_5.

[9] T. Rinta-Kahila, E. Penttinen, and K. Lyytinen, “Getting Trapped in Technical Debt: Sociotechnical Analysis of a Legacy System’s Replacement,” MIS Quarterly, vol. 47, no. 1, pp. 1–32, Mar. 2023, doi: https://doi.org/10.25300/misq/2022/16711.

[10] A. Martínez Saucedo, G. Rodríguez, F. Gomes Rocha, and R. P. dos Santos, “Migration of monolithic systems to microservices: A systematic mapping study,” Information and Software Technology, vol. 177, p. 107590, Oct. 2024, doi: https://doi.org/10.1016/j.infsof.2024.107590.

[11] C. E. da Silva, Y. de L. Justino, and E. Adachi, “SPReaD: service-oriented process for reengineering and DevOps,” Service Oriented Computing and Applications, vol. 16, pp. 1–16, Oct. 2021, doi: https://doi.org/10.1007/s11761-021-00329-x.

[12] B. Nascimento, R. Santos, J. Henriques, M. V. Bernardo, and F. Caldeira, “Availability, Scalability, and Security in the Migration from Container-Based to Cloud-Native Applications,” Computers, vol. 13, no. 8, p. 192, Aug. 2024, doi: https://doi.org/10.3390/computers13080192.

[13] A. F. A. A. Freire, A. F. Sampaio, L. H. L. Carvalho, O. Medeiros, and N. C. Mendonça, “Migrating production monolithic systems to microservices using aspect oriented programming,” Software: Practice and Experience, vol. 51, no. 6, pp. 1280–1307, Mar. 2021, doi: https://doi.org/10.1002/spe.2956.

[14] L. F. Al-Qora’n and A. A.-S. Ahmad, “Modular Monolith Architecture in Cloud Environments: A Systematic Literature Review,” Future Internet, vol. 17, no. 11, p. 496, Oct. 2025, doi: https://doi.org/10.3390/fi17110496.

[15] I. Trabelsi et al., “From legacy to microservices: A type-based approach for microservices identification using machine learning and semantic analysis,” Journal of Software: Evolution and Process, vol. 35, no. 10, p. e2503, Sep. 2022, doi: https://doi.org/10.1002/smr.2503.

[16] M. A. Nadeem, “Trust Boundaries and Data Isolation in Virtualized Cloud Environments: A Case Study on Encrypted Compute Models,” Proceedings of 2025 International Conference on Electrical, Electronics, and Computer Science with Advance Power Technologies - A Future Trends, Oct. 2025, doi: https://doi.org/10.1109/ICE2CPT66440.2025.11340326.