International Journal of Science and Research (IJSR)
ISSN: 2319-7064
Impact Factor 2025: 7.089

Theoretical Foundations of Integrating Microservices Architecture into Monolithic Enterprise Systems

Abdul Nadeem Mohammed

Full Stack Developer, Techlance Solutions Inc, Michigan, United States

Abstract: ***The article examines the integration of microservices architecture into monolithic enterprise systems as a problem of enterprise architectural evolution. The purpose of the article is to develop a theoretical foundation for integrating microservices into a legacy monolith. The relevance of the topic is determined by the growth of the structural complexity of large systems, the intensification of intermodule dependencies, the rising cost of change, and the misalignment between software structure, data, and domain boundaries. The scientific novelty lies in the synthesis of the principles of modularity, loose coupling, bounded context, and evolutionary architecture into a unified model of controlled transformation, as well as in the proposed roadmap for integrating microservices into a monolithic system. The conclusion states that the outcome of integration depends on the precision of domain boundaries, data autonomy, contract-based interaction, and the alignment of architecture with team structure. It is shown that the greatest value belongs to a phased hybrid trajectory: architectural audit, identification of service candidates, construction of an integration layer, and pilot extraction of business services. The article will be useful to researchers, software architects, technical leaders, and enterprise digital transformation teams.***

Keywords: microservices architecture, monolithic systems, enterprise information systems, architectural evolution

1. Introduction

The growth of enterprise information systems has led to a situation in which the original architectural scheme has ceased to contain complexity within acceptable boundaries [1]. A large monolith is characterized by a unified change delivery cycle, dense module dependencies, a high cost of local modifications, and the accumulation of architectural debt. As the domain expands, each new function affects an increasing number of code areas and strengthens the dependency between application logic, data, and integration mechanisms [2]. Studies over the past few years associate the turn toward decomposition with the need for manageability, maintainability, and alignment of architecture with domain boundaries [3]. This context makes the integration of microservice architecture into existing enterprise monoliths a subject of theoretical analysis, beyond mere engineering practice.

The problems with monolithic enterprise systems stem from the fact that a unified codebase gradually becomes an environment where any local task intersects with system-wide constraints. In such systems, delineating responsibility boundaries becomes more difficult, the burden of coordinating changes across teams grows, and the data structure entrenches past design decisions at the platform level [2]. For an enterprise, this means increased costs for development, testing, and new-version releases. An additional layer of difficulty is created by the system's history. Obsolete links between subsystems, duplication of functions, and gaps between the business model and software implementation remain embedded within it. Empirical migration studies show that the transition from a monolith affects organizational structure, responsibility distribution, and modes of architectural decision-making [4]. The question of system transformation extends beyond technology selection and enters the domain of enterprise systemic evolution.

Interest in microservices architecture arises from the aspiration to restore manageable boundaries within the system and to connect architecture with distinct business capabilities. Scholarly works describe this approach as decomposing into domain contexts, ensuring data autonomy and delivery independence, and achieving greater resilience to local failures [5]. At the same time, the literature records the other side of the issue. Decomposition requires a precise selection of granularity. Otherwise, the system acquires new forms of connectivity and becomes a distributed variant of the former monolith [6]. For this reason, the topic of integrating microservices into an enterprise monolith requires a theoretical foundation that binds together modularity, domain boundaries, architectural qualities, and organizational structure.

2. Literature Review

The literature frames the shift from monolithic enterprise systems to microservices as a response to accumulated structural strain. Recent migration studies describe the monolith as an architecture whose growth concentrates dependencies, expands the blast radius of change, and ties release management to a single delivery pipeline [1, 2]. This diagnosis is further developed by research linking decomposition to enterprise architecture and domain structure. In that line of work, the key issue is the fit between software boundaries and business capabilities [3, 4]. Studies on microservice transition treat bounded contexts, service autonomy, and granularity as core analytical categories, since decomposition quality depends on whether a service encloses a coherent slice of domain logic and a stable set of responsibilities [5, 17]. The same argument appears in work on requirements and architectural granularity, where partitioning is tied to the clarity of functional units and the system's future capacity to evolve under changing demands [9]. Research on automated identification of microservices extends these ideas through clustering, semantic analysis, and

pattern-based migration logic, using cohesion and coupling as operational criteria for service extraction [10, 11]. Within this body of literature, decomposition emerges as a disciplined architectural act shaped by domain meaning, dependency structure, and the cost profile of change.

A second stream of research addresses the conditions under which decomposition remains viable in enterprise practice. Reviews of modular monoliths and hybrid migration strategies present an intermediate architectural form that preserves operational continuity while preparing service boundaries and integration contracts [6, 12, 19]. This perspective is reinforced by studies of architectural change, which show that service systems drift as interfaces multiply, teams make local decisions, and original design assumptions lose force over time [7, 13]. Empirical work on ownership and organizational coupling places team structure inside the architecture problem itself, since communication paths and responsibility allocation shape the density of interservice dependence [18]. The literature also identifies a set of recurrent risks. Architectural smells, unstable API evolution, weak observability, and unresolved data ownership can turn service migration into a distributed reproduction of monolithic coupling [16, 20, 21]. Performance studies add another layer, showing that the testing and debugging model changes once a single execution contour is replaced by networked interactions across services [14]. Taken together, these studies support a theoretical view of integration as controlled architectural evolution. In that view, success depends on boundary precision, contract-based interaction, data autonomy, and a phased migration path grounded in architectural audit and pilot extraction [1, 6, 19].

3. Problem Definition

Enterprise monolithic systems face a growing mismatch between business and software structures as functional scope expands, dependencies thicken, and shared data models lock in past design decisions in current operations. Under these conditions, even limited changes can trigger system-wide coordination costs, weaken maintainability, and reduce the capacity to isolate failures, scale specific functions, or assign responsibility along clear domain boundaries. The central problem is determining how to introduce service-oriented decomposition into such environments without reproducing inherited coupling, fragmenting the domain model, or undermining the integrity of the existing platform.

In view of the identified theoretical and practical contradictions associated with the integration of microservices architecture into monolithic enterprise systems, it can be assumed that the success of such transformation depends on the technical decomposition of the system and on the correct alignment of architectural, domain, data, and organizational boundaries. Proceeding from this, the study puts forward the following hypotheses:

H1: The effectiveness of integrating microservices into a monolithic enterprise system is determined by the alignment of service boundaries with domain contexts, the autonomy of data ownership, and the use of contract-based interaction mechanisms.

H2: A phased hybrid approach to architectural transformation provides the most theoretically grounded and practically viable path for integrating microservices into a legacy monolith, since it allows the enterprise to preserve platform integrity while gradually reducing inherited coupling.

4. Materials and Methodology

The study is based on an analysis of 21 sources, including studies on migration of monolithic systems, works on architectural decomposition, and empirical observations of enterprise platform transformation [1–6]. The theoretical foundation consists of studies linking architectural change with the growth of systemic complexity, the accumulation of dependencies, and the misalignment between software structure and domain [2, 3]. Works on microservice architecture [5, 6] were used to identify the principles of decomposition, service autonomy, and granularity issues, whereas studies of organizational influence and architectural evolution [4] enabled consideration of integration as part of the transformation of the enterprise system as a whole. Additional significance belongs to studies of modularity and architectural practices [11, 12], where the properties of cohesion and coupling are applied to identify service boundaries and assess transformation quality.

The methodological part is grounded in a combination of comparative analysis of architectural approaches, a systematic review of migration studies, and content analysis of empirical cases. A comparative analysis was conducted to examine the characteristics of monolithic and microservice architectures with respect to structure, dependency management, and change processes [10, 15]. The review of migration studies [1, 6] identified recurring integration patterns, including phased service extraction and hybrid architectures. Content analysis of empirical studies [18, 19, 21] was used to reveal the influence of organizational structure, data ownership distribution, and legacy repositories on the integration process, enabling connections between architectural decisions and the actual conditions of the enterprise environment.

5. Results and Discussion

The architecture of a software system is the stable organization of components, relationships, constraints, and development rules through which responsibility boundaries and permissible modes of system change are defined. Such a view occupies a central place in current studies on architectural decisions and their auditability, where architecture is regarded as an object of observation, assessment, and revision throughout the system life cycle [7]. In this sense, architecture establishes the form of technical order within the enterprise and connects software implementation with the arrangement of processes, data, and work roles, which accords with the systemic representation of the enterprise as a unified engineering formation [8].

In architectural design, architectural styles serve as repeatable schemes for ordering a system, through which permissible forms of interaction between application parts, mechanisms for distributing responsibility, and the nature of the dependency between computation and data are selected. Studies on the alignment of requirements and architectural granularity show that style selection influences the manner of

Paper ID: SR26421181425 DOI: https://dx.doi.org/10.21275/SR26421181425

functional partitioning, the depth of intermodule ties, and the limits of further system decomposition [9]. The transition from a monolith to microservices requires a change in architectural logic, where a unified system with a common execution contour yields to a multiplicity of service units with their own boundaries and separate cycles of change [10].

The principles of modularity, cohesion, and loose coupling form the theoretical framework without which the quality of architectural decomposition cannot be assessed. High internal cohesion of a module indicates the semantic unity of its functions. Loose coupling between modules reduces the cost of change and lowers the risk of cascading disruptions in neighboring parts of the system. Works on automated migration of monoliths to microservices employ these properties as a basis for identifying service boundaries and clustering components [11]. The same set of features is applied in studies of the modular monolith, which is regarded as an intermediate form between an integral application and a distributed system, capable of retaining domain boundaries within a single deployment [12].

Evolutionary architecture views a system as a mutable structure that must remain operable as new requirements, team practices, and integration dependencies accumulate. In the context of enterprise platforms, this means rejecting the representation of completed architecture as a once-and-for-all fixed construction. Studies of architectural change in microservice systems show that deviations from the original design emerge as the number of services, interfaces, and local decisions adopted by different teams increases [13]. For this reason, integrating microservices architecture into a monolithic environment should be approached through a controlled transformation trajectory, in which each stage of change must preserve the integrity of the domain model and the controllability of dependencies.

Within this sequence, monolithic architecture represents an application with a unified delivery process, a shared codebase, a common execution contour, and a high density of internal dependencies. This form remained a foundational solution for enterprise systems for a long time, as it simplified development in early stages, eased end-to-end debugging, and enabled preservation of transactional integrity within a single application. Studies of testing practices show that many qualities of the monolith are tied to execution predictability and fewer network interactions than in distributed systems [14]. As enterprise scale expands, these advantages begin to generate a countervailing effect. A shared database entrenches rigid ties between subsystems. A unified version release raises the cost of even a minor change. Maintenance and scaling require touching the whole system when the source of load or defect is localized in a single domain [10].
Microservices architecture proceeds from a different logic. The system is divided into a set of services, each implementing a bounded segment of the domain, owning its own logic, and participating in data exchange through formalized interfaces. Current reviews describe this approach as characterized by fine-grained services, relative team autonomy, independent delivery cycles, distributed data ownership, and a growing role for interservice interaction [15]. Such a scheme opens the way to selective scaling, failure localization, and a tighter link between architecture and the boundaries of business functions. At the same time, microservice adoption introduces measurable costs associated with distributed complexity and architectural degradation. In an empirical study of open-source microservice systems, Li et al. analyzed 38 projects comprising 379 microservices, with an average of 10 microservices per project, and recorded more than 25,000 same-day switches between microservices by developers [18]. The same study found large variation in yearly organizational coupling, ranging from 28.28 in one project to 7782.27 in another. These figures illustrate the growth of coordination overhead and the managerial burden that emerges when architectural responsibility is spread across many services and contributors. In parallel, Zhong et al. identified six typical architectural smells in an industrial microservice system and grouped their causes into five aspects: technology, project, organization, business, and professional [16]. They report that these smells reduce modularity, modifiability, analyzability, and testability, and they are associated with extra cross-team communication and with change-prone and fault-prone microservices. The difference between a monolith and microservices lies in dependency management, change management, and data management.The difference between a monolith and microservices lies in dependency management, change management, and data management.

The costs also appear in observability, performance diagnosis, and the risk of reproducing monolithic coupling in distributed form. Eismann et al. showed that under identical loads of 700, 800, and 900 requests per second, repeated test runs could lead to different execution environments because autoscaling provisioned different numbers of service instances. For the Auth microservice at 700 requests per second, one set of runs used 3 instances while another used 7 instances, which complicates stable tracing and interpretation of performance behavior. Faustino et al. provide a direct example of distributed-monolith-type penalties during migration. In their case study, average latency for Source Listing increased from 22 ms to 982 ms and then to 8384 ms, while Fragment Listing increased from 61 ms to 4845 ms and then to 32767 ms as the system moved from the monolith toward microservices. The same study reports that inconsistent states during migration could cause failed requests in functions such as Virtual Edition Listing and Game Listing, showing that the cost of decomposition may extend from infrastructure and latency into user-visible service instability.

Comparison of monolithic and microservice architectures is shown in Table 1.

Table 1: Comparison of Monolithic and Microservice Architectures

Aspect	Monolithic Architecture	Microservice Architecture
Structure	A unified application with a shared codebase and a common execution environment	A set of autonomous services with clearly defined responsibility boundaries
Change and release process	A single release cycle, with even local modifications affecting the system as a whole	Independent delivery cycles, with changes localized within individual services

Data and dependencies	A shared database and tightly coupled subsystems	Formalized interfaces and distributed ownership of data
Scalability and failures	The application is typically scaled as a whole, while failures more readily affect the entire system	Individual services can be scaled selectively, and failures are more often localized
Principal effect	Simplicity of development and debugging at early stages	Greater flexibility, accompanied by higher distributed complexity

The transition to integrating service decomposition into a monolith follows the logic of architectural evolution in information systems. As an enterprise platform grows, the distribution of functions changes, dependencies between parts of the application become more complex, and the cost of each architectural decision related to data, interfaces, and version releases increases. A systematic study of monolithic system migration shows that this transformation is associated with the accumulation of structural complexity, difficulty in identifying service boundaries, and the need for methods of phased system change [1]. Within this logic, the concept of bounded context serves as a theoretical mechanism that enables correlating domain boundaries with those of software components. Studies of microservice decomposition relate the quality of such partitioning to the principle of minimal knowledge sharing between system components and to the identification of segments within which a unified meaning of models and operations is preserved [17]. In a monolithic enterprise environment, this proposition has special value, since an attempt to split the system without relying on domain boundaries results in the transfer of old dependencies into a new form and entrenches hidden connectedness at the level of service interactions.

The enterprise's organizational structure influences the system's architectural form through stable communication channels, responsibility distribution, and team coordination. An empirical study of microservice ownership shows that organizational connectedness between developers and services affects the level of interservice dependency and the quality of architectural separation [18]. It follows that integrating microservices into a monolith requires aligning technical boundaries with team and work-process boundaries. Such a formulation necessitates hybrid architectural models, in which some functions remain within the monolith, whereas others are extracted into separate services during a phased transformation. A study of the transition from a monolith to a service system with a hybrid data storage scheme shows that it is precisely the intermediate architectural form that helps preserve the integrity of the existing platform and reduce the risk of rupture between the old and new parts of the system [19]. In this case, the hybrid model serves as a theoretically grounded approach to managing architectural change through bounded steps, in which domain contexts, organizational boundaries, and integration mechanisms converge into a unified transformation trajectory.

After considering hybrid architecture as a controlled transition toward service decomposition, the question arises: how should new services be integrated into an operating monolith? Systematic mapping of migration studies shows that the transition to microservices involves many different techniques; a single universal approach is absent, and the number of tools supporting this transition remains small, concentrated around a limited circle of technologies [1]. From this follows the basic principle of phased migration, in which the transformation is divided into a series of local steps while preserving the operability of the entire system. Within this logic, the stepwise displacement pattern occupies a special place, since it enables extracting individual functions from the monolith into external services as domain boundaries and integration contracts mature.

In this scheme, the anticorruption layer serves as an intermediary stratum, preventing the new model from inheriting old dependencies and transforming data, commands, and responses into a form aligned with the service boundary. The connection between the monolith and services is built through software interfaces and events. A study of interface evolution in service systems shows that, under synchronous and event-driven exchanges, the main difficulties arise around version compatibility, change-impact analysis, and organizational connectivity between teams [20]. For this reason, the choice between synchronous invocation and event-driven interaction is determined by latency and consistency requirements. In reported migration cases, synchronous remote interaction increased average latency from 22 ms to 982 ms and up to 8384 ms for Source Listing, and from 61 ms to 4845 ms and up to 32767 ms for Fragment Listing, while even a 4 ms delay could correspond to about 10% of the system's base response time [6]. In terms of consistency, event-driven interaction must account for at least nine event types affecting cross-service data alignment, with the highest-impact events propagating across three databases and, under inconsistent states, causing failure in functions such as Virtual Edition Listing and Game Listing.

A practical example of this integration logic can be illustrated through the extraction of an Order Management function from a monolithic enterprise platform. In the initial state, order registration, payment processing, inventory update, and customer notification are implemented inside one codebase and use a shared relational database. During phased decomposition, the Order Management module is isolated as a separate service packaged in Docker, which gives it an independent runtime unit and a stable deployment format across development, testing, and production environments. The service is then deployed in Kubernetes, where scaling rules, health checks, and service discovery support its operation as an independent component within the wider enterprise platform.

The connection between the monolith and the new service can be organized through an integration layer that exposes REST or gRPC contracts for synchronous requests and uses Apache Kafka for domain events such as OrderCreated, PaymentConfirmed, and StockReserved. In this configuration, the monolith retains control over legacy business functions, while the extracted service takes responsibility for its own logic and for a bounded subset of data. Kafka supports asynchronous propagation of state changes across related components and reduces direct temporal dependency between the order, payment, and inventory domains. REST and gRPC define stable service

contracts and make version control of interfaces part of the architectural boundary itself.

This example shows how the theoretical principles discussed in the article are translated into enterprise implementation practice. Docker provides the technical unit for service isolation. Kubernetes supports orchestration, recovery, and scaling at the infrastructure level. Kafka supports event exchange between bounded contexts and helps preserve service autonomy during the transition period. Under such a model, the hybrid trajectory described in the article receives a concrete implementation path in which architectural audit identifies the Order Management domain as a candidate for extraction, the integration layer protects the new service boundary, and pilot deployment creates the basis for further decomposition of the monolith.

During the migration process, the user continues to interact with the system through the same business functions, yet the internal route of request processing may gradually shift from the monolith to newly extracted services. For the user, the main requirement is the preservation of functional continuity, transaction reliability, response predictability, and the integrity of visible data across all interaction channels. If part of the operation is processed within the monolith and another part within a separate service, the platform must preserve a unified entry point, consistent interface behavior, and stable handling of user actions such as order creation, payment submission, status tracking, or document access. User interaction therefore becomes an important criterion for migration design, since each extraction step must be coordinated with interface stability, session continuity, error transparency, and the prevention of situations in which the architectural transition appears to the user as a loss of service integrity.

The problems of monolith decomposition begin when architectural separation is replaced by technical fragmentation into layers, libraries, or transport mechanisms. An error in boundary selection leads to services with unclear purpose, a high number of calls between them, and a constant need to coordinate changes. A shared database creates a particular threat. A study of migrating a monolithic repository into a service environment indicates that moving toward a service architecture without separating data ownership preserves the former connectedness and blocks genuine component autonomy [21]. The same study presents an indicator showing that 71 percent of the largest corporations retain systems based on universal mainframe-type computing platforms, underscoring the high significance of legacy repositories in enterprise migration. If services continue to share a single data schema while change releases require common coordination, a distributed monolith emerges. Such a form combines network complexity with inherited connectedness, depriving the system of the qualities for which microservice integration is undertaken. Challenges in integrating service decomposition into a monolith are illustrated in Figure 1.

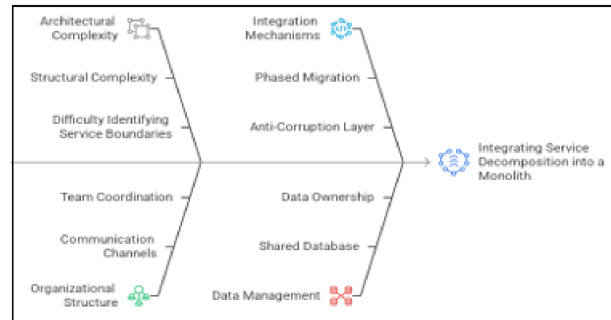

Figure 1: Challenges in Integrating Service Decomposition into a Monolith

To resolve these difficulties, the following roadmap is proposed. It proceeds on the premise that integrating service architecture into a monolithic system requires a controlled sequence of actions, where each change builds on knowledge of the system's current state. The starting point is an architectural audit of the monolith. Its task is to identify the application's actual structure, hidden module dependencies, nodes of business-logic concentration, data-exchange routes, and the areas where the greatest technical debt has accumulated. Within such an analysis, it is important to identify which parts of the system already exhibit internal integrity, which subsystems are overloaded with incidental responsibilities, and which components retain an excessive number of connections. In this model, the audit is understood as a means of transition from a formal description of the system to its real architectural map, without which subsequent decomposition loses its domain foundation.

The next step involves identifying domain boundaries and selecting candidates for extraction. Here, the domain logic of the system acquires key significance, since the boundaries of future services must arise from stable business functions, the composition of operations, and the nature of data. An approach based on domain boundaries enables the identification of segments within which a unified meaning for objects, rules, and processing scenarios is preserved. Modules with high internal cohesion, a pronounced business function, and relative independence in data processing should be considered the first candidates. At the same time, the design of an integration layer is required; this layer will ensure alignment between the new service and the monolith without directly transferring old dependencies. Such a layer assumes responsibility for request transformation, call routing, data model isolation, and control over exchange rules between the old and new parts of the system [22].

After preparation of boundaries and the integration layer, the pilot extraction of one business service begins. The choice of the pilot module has methodological significance, since it must be sufficiently important for testing the architectural decision and, at the same time, sufficiently localized to keep risk within acceptable limits. The pilot service enables verifying the stability of new contracts, identifying hidden dependencies, assessing the burden on maintenance teams, and refining requirements for monitoring, logging, and testing. At this stage, special attention is given to the transition to event-driven interaction and data autonomy. If the new service retains a direct dependency on the shared storage schema, it continues to reproduce the monolith's logic in another form. For this reason, service extraction must be accompanied by a gradual transfer of responsibility for data

within its boundaries and by replacing part of the direct calls with event exchange when this corresponds to the nature of the business process.

Physical migration also has a direct economic dimension, since the transfer of functions, data, and integration routes from a monolithic environment into a hybrid or service-based architecture may require temporary service suspension, restricted transaction processing, or scheduled maintenance windows. For an enterprise platform, such interruptions mean lost revenue, delayed operations, increased workload for support teams, and a rise in coordination costs across business and technical units. The scale of these losses depends on the criticality of the migrated domain, the duration of downtime, the volume of dependent operations, and the degree to which the current business process relies on continuous system availability. For this reason, the migration trajectory must include an assessment of downtime costs, fallback procedures, data synchronization plans, and the acceptable limits of operational disruption.

At the same time, the cost of temporary disruption may be justified when the long-term gains of decomposition exceed the losses incurred during the migration interval. If service extraction reduces the cost of change, limits failure propagation, improves scalability of high-load functions, and shortens the release cycle for business-critical modules, the enterprise receives a structural effect that compensates for the initial interruption. In such conditions, migration should be evaluated as an investment in architectural capacity, where short-term operational losses are correlated with future savings in maintenance, incident isolation, delivery speed, and organizational manageability.

The concluding contour of the roadmap involves scaling the acquired practice to other system modules. After pilot implementation, the organization obtains more than a separate service; it receives a validated architectural transformation scheme that can be transferred to new domains, with due regard for accumulated experience. In this model, scaling is built as a repeatable cycle. Module analysis is carried out, domain boundaries are refined, an integration layer is created, service extraction is performed, and then the autonomy of its data and interactions is consolidated. As the number of such transformations increases, the system's architectural nature changes. The monolith loses its status as the sole center of execution and becomes the core of a hybrid environment, where the distribution of functions follows the enterprise's domain structure. The value of this roadmap lies in connecting an architectural decision to a sequence of verifiable steps, thereby shifting service integration from the realm of isolated experiments into the realm of controlled, systemic development. The roadmap for integrating a monolith into a service architecture is illustrated in Figure 2.

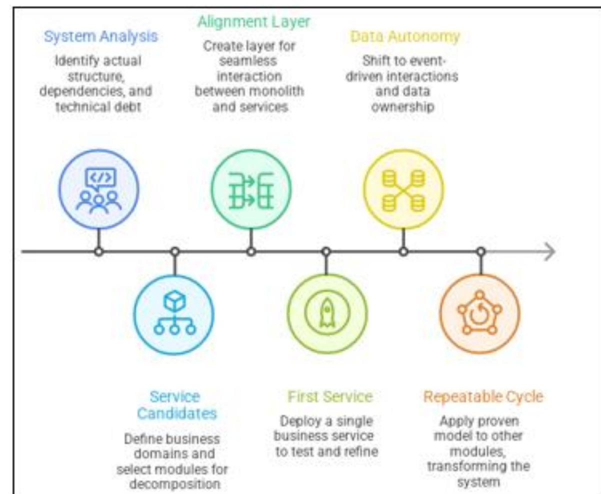

Figure 2: Roadmap for Monolith to Service Architecture Integration

The integration of service architecture into a monolithic environment increases distributed complexity, as a unified execution logic is split across a multiplicity of connected nodes, each with its own area of responsibility, change cycle, and failure conditions. In such a system, it becomes more difficult to retain an integral representation of call routes, data state, and the consequences of a local failure for the whole platform. A separate group of risks is formed by data consistency and transactional integrity. After functions are divided between services, the unified processing contour disappears, within whose limits the system could rely on common state-fixation mechanisms. Because of this, the number of situations increases in which data in different parts of the platform temporarily diverge in meaning and update moment. Additionally, monitoring, testing, and debugging become more difficult. The reason is that the source of a defect may lie at the intersection of several services, message queues, routing rules, and storage schemas. Organizational risk arises when teams retain the former coordination model within a new architectural form. Then services lose independence, decisions accumulate in a single coordination center, and the distributed system reproduces the logic of the former monolith through a network of external calls and dependencies.

Successful integration requires a methodological foundation that establishes rules for selecting components for extraction, defining boundaries, and implementing change management. Candidates for extraction into service form should be regarded as parts of the system with a pronounced domain function, high internal cohesion, a stable data set, and a limited number of intersections with neighboring modules. Interaction between services must be built on a contract-based approach, in which interfaces, message formats, and conditions for changing connections are described as autonomous architectural objects. Such a scheme reduces the risk of hidden dependencies and facilitates system evolution across multiple teams. To preserve manageability, end-to-end observability is required, encompassing event logs, request tracing, state indicators, and mechanisms of early failure detection. At the same time, the architectural autonomy of services requires alignment with unified security, versioning, error-handling, and data-management rules. The balance between team independence and common architectural norms

determines the stability of the entire integration model. Without it, the service environment loses integrity and becomes a set of loosely coordinated decisions.

6. Conclusion

The conducted analysis shows that integrating microservice architecture into monolithic enterprise systems falls within the class of tasks in enterprise systemic evolution. The source of this transformation is the growth in the monolith's structural complexity, the intensification of internal dependencies, the rising cost of change, and the divergence between domain boundaries, data, and software implementation. Under these conditions, service decomposition becomes an architectural mechanism that enables the system to establish manageable responsibility boundaries, align the structure of software components with business functions, and lay a foundation for phased platform development.

The theoretical foundation of such integration is built on the principles of modularity, cohesion, loose coupling, bounded context, and evolutionary architecture. The quality of transformation is determined by the extent to which service boundaries align with the domain model, the distribution of data, and the assignment of team responsibilities. An error in granularity selection, preservation of a shared storage schema, and the transfer of old dependencies into the service environment reproduce monolithic connectedness in a distributed form and deprive architectural transformation of its substantive result.

The first hypothesis is supported by the theoretical analysis showing that the outcome of microservice integration depends on the precision of domain boundaries, the separation of data ownership, and the use of formalized contracts between services. The article demonstrates that without these conditions, legacy dependencies are reproduced in a distributed form, leading to the emergence of a distributed monolith.

The second hypothesis is confirmed by the synthesis of migration studies and architectural principles, which indicates that the greatest value belongs to a phased hybrid trajectory. Architectural audit, identification of service candidates, construction of an integration layer, and pilot service extraction make it possible to preserve the integrity of the existing platform, control dependencies, and implement architectural transformation through verifiable and manageable steps.

The resulting conclusions confirm that phased integration through a hybrid architectural model, architectural audit, identification of domain boundaries, construction of an integration layer, and pilot extraction of individual business services possesses the greatest theoretical and practical value. Such a trajectory preserves the integrity of the operating platform, enables dependency control, and connects architectural decisions to a sequence of verifiable steps. In this way, integrating microservices into a monolith appears as a controlled process of architectural transformation in which system stability depends on the alignment of domain contexts, interface contracts, data autonomy, and the enterprise's organizational structure.

7. Future Scope

Future research can expand this line of inquiry through longitudinal studies of hybrid enterprise architectures, with attention to the way service boundaries, data ownership, and interface contracts change across extended transformation cycles. Comparative work across industries would also be valuable, since the pace and form of architectural decomposition differ under distinct operational and regulatory conditions. Another promising direction is the design of boundary-identation models that combine domain analysis with architectural indicators such as cohesion, coupling, and dependency density. This would sharpen the criteria for selecting service candidates and strengthen methods for evaluating decomposition quality in enterprise settings.

The scope of the present study is defined by its focus on theoretical synthesis and general architectural patterns. This gives the section a broad explanatory range and makes it useful for interpreting transformation across different enterprise contexts. In contrast to earlier research, which often focused on single migration cases, tooling decisions, or isolated performance questions, this approach offers a broader analytical frame. It integrates domain structure, organizational coupling, data autonomy, and evolutionary change into a single model. That broader frame creates a solid basis for future work on microservice integration as a question of enterprise architectural evolution.

References

[1] Martínez Saucedo A, Rodríguez G, Gomes Rocha F, Santos RP dos. Migration of monolithic systems to microservices: A systematic mapping study. Information and Software Technology. 2024 Oct 1;177:107590.

[2] Michael Ayas H, Leitner P, Hebig R. An empirical study of the systemic and technical migration towards microservices. Empirical Software Engineering. 2023 May 22;28(4).

[3] Abd-Elwahab AM, Mohamed AG, Shaaban EM. MicroServices-driven enterprise architecture model for infrastructure optimization. Future Business Journal. 2023;9(1):90.

[4] Özkan O, Babur Ö, van den Brand M. Refactoring with domain-driven design in an industrial context. Empirical Software Engineering. 2023 Jun 15;28(4):94.

[5] Hassan S, Bahsoon R, Kazman R. Microservice transition and its granularity problem: A systematic mapping study. Software: Practice and Experience. 2020 Jun 25;50(9):1651–81.

[6] Faustino D, Gonçalves N, Portela M, Rito Silva A. Stepwise migration of a monolith to a microservice architecture: Performance and migration effort evaluation. Performance Evaluation. 2024 May;164:102411.

[7] Cruz P, Solar M, Astudillo H. Towards Software Architecture as an Auditable Practice. Applied Sciences. 2026 Mar 20;16(6):3020.

[8] Nori KV, Natarajan S, Kumar A, Lokku DS. A Systems Engineering Approach to Modelling Enterprises. Handbook of Systems Sciences. 2020;1–33.
[9] Spijkman T, Molenaar S, Dalpiaz F, Brinkkemper S. Alignment and granularity of requirements and architecture in agile development: A functional perspective. Information and Software Technology. 2021 May;133:106535.
[10] Trabelsi I, Abdellatif M, Abubaker A, Moha N, Mosser S, Ebrahimi-Kahou S, 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. 2022 Sep 4;35(10):e2503.
[11] Hassan H, Abdel-Fattah MA, Mohamed W. A Pattern-Based Framework for Automated Migration of Monolithic Applications to Microservices. Big Data and Cognitive Computing. 2025 Oct 6;9(10):253.
[12] Al-Qora'n LF, Ahmad AAS. Modular Monolith Architecture in Cloud Environments: A Systematic Literature Review. Future Internet. 2025 Oct 29;17(11):496.
[13] Rao AE, Vagavolu D, Chimalakonda S. AC2: Towards understanding architectural changes in Python projects. Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2021 Aug 18;1555–9.
[14] Eismann S, Bezemer CP, Shang W, Okanović D, Hoorn A van . Microservices: A Performance Tester's Dream or Nightmare? ICPE '20: Proceedings of the ACM/SPEC International Conference on Performance Engineering. 2020 Apr 20;138–49.
[15] Abdelfattah AS, Cerny T. Roadmap to Reasoning in Microservice Systems: A Rapid Review. Applied Sciences. 2023 Jan 31;13(3):1838.
[16] Zhong C, Huang H, Zhang H, Li S. Impacts, causes, and solutions of architectural smells in microservices: An industrial investigation. Software: Practice and Experience. 2022 Aug 18;52(12):2574–97.
[17] Camilli M, Colarusso C, Russo B, Zimeo E. Actor-driven Decomposition of Microservices through Multi-level Scalability Assessment. ACM Transactions on Software Engineering and Methodology. 2023 Feb 15;32(5):1–46.
[18] Li X, d'Aragona DA, Cerny T, Lenarduzzi V, Janes A. Exploring microservice ownership and organizational coupling in open-source projects: an empirical study. Computing. 2025 Apr 1;107(4).
[19] Ng T, Bin Rawi AA, Sum CS, Tso E, Yau PC, Wong D. Migrating from Monolithic to Microservices with Hybrid Database Design Architecture. Proceedings of the 2024 9th International Conference on Intelligent Information Technology. 2024 Feb 23;536–41.
[20] Lercher A, Glock J, Macho C, Pinzger M. Microservice API Evolution in Practice: A Study on Strategies and Challenges. Journal of systems and software. 2024 Sep 1;215:112110.
[21] Kazanavičius J, Mažeika D, Kalibatienė D. An Approach to Migrate a Monolith Database into Multi-Model Polyglot Persistence Based on Microservice Architecture: A Case Study for Mainframe Database. Applied Sciences. 2022 Jun 17;12(12):6189.
[22] Nadeem MA. Trust Boundaries and Data Isolation in Virtualized Cloud Environments: A Case Study on Encrypted Compute Models. Proceedings of the 2025 International Conference on Electrical, Electronics, and Computer Science with Advance Power Technologies - A Future Trends (ICE2CPT). 2025 Oct 29.