Journal of Information Systems Engineering and Management
2026, 11(2s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

From Monolith to Microservices: Architectural Strategies for High-Availability Database Re-platforming

Siva Prakash

Bharathidasan University, India

ARTICLE INFO

Received: 11 Feb 2026

Revised: 15 Feb 2026

Accepted: 25 Feb 2026

ABSTRACT

The transition from monolithic architectures to microservices-based systems represents a fundamental paradigm shift in enterprise software development, particularly for organizations managing complex financial applications and database infrastructures. This article examines comprehensive architectural strategies for migrating legacy monolithic applications to distributed microservices architectures while simultaneously modernizing database platforms to achieve high availability and operational resilience. The article explores domain-driven decomposition methodologies that identify natural business boundaries and bounded contexts within existing systems, enabling effective service granularity decisions that balance independence benefits against distributed system complexity. Containerization and cloud-native technologies emerge as critical enablers, providing lightweight deployment mechanisms, orchestration capabilities, and dynamic resource allocation that significantly enhance system scalability and operational efficiency. Performance optimization techniques including query optimization, automated indexing strategies, polyglot persistence patterns, and infrastructure-as-code practices are analyzed to demonstrate how organizations can maintain optimal database performance across distributed microservices while managing data consistency and service dependencies. The article synthesizes research findings and empirical evidence to provide practitioners with actionable strategies for navigating the complexities of architectural transformation, addressing technical challenges in service orchestration, data architecture redesign, and resource management that are essential for successful microservices adoption in production environments.

Keywords: Microservices Architecture, Database Re-Platforming, Domain-Driven Design, Containerization, Cloud-Native Infrastructure

Introduction

Enterprise software architecture has developed into one of the greatest challenges for modern financial systems. Existing monolithic applications are sometimes reliable and understood, but they are often not suited to modern digital transformation programs. Microservices architecture has shown that customary monolithic applications have intrinsic limitations in scaling and maintaining codebases that are growing in size and complexity over the application life cycles [1]. Migrating from a tightly-coupled monolithic architecture to a distributed microservices architecture with an updated database globally equips organizations with the agility, scalability and efficiency to operate.

Studies have shown that the microservices architecture is being adopted across domains and by organizations with different needs, especially where high feature velocity and continuous delivery are critical [1]. Organizations using microservices architecture have reported that it has improved their ability to scale services independently, allocate development resources across functional teams, and respond quickly to business needs. Research argues that for microservices, teams own the service boundaries and are decoupled from each other. This reduces coordination overhead and enables parallelized streams of work, as teams can make progress independently, which is not possible in

Journal of Information Systems Engineering and Management
2026, 11(2s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

monolithic architecture, where teams directly depend on each other. Thus, careful system-level planning, decomposition, and knowledge about technical and business domains are required.

Microservices architecture is more recent than other methodologies. From the literature review, it is possible to find that microservices emerged as responses to practical challenges, as an evolution of service-oriented architectures and domain-driven design [2]. Microservices architecture is an application architecture style which decomposes applications into independently deployable services in well-defined functional domains which communicate over a network, often using a lightweight protocol. For the successful application of microservices architecture, practitioners and researchers have identified multiple technical challenges such as service discovery, distributed data management, inter-service communication, and model for eventual consistency [2]. Furthermore, the authors identified that the adoption of decomposition patterns requires carefully designed service boundaries based on business capabilities to minimize the interdependencies of the resulting services.

In addition, regarding database modernization, existing database schemas in relational databases may need to be redesigned, especially if the relational database was designed to support a monolithic application that has a database schema design based on the database access pattern of a monolithic application. Data ownership models also need to be reconsidered as each microservice needs to own its own data store for greater autonomy and independent scaling [2]. Trade-offs between data consistency, availability and partition tolerance must be carefully evaluated in the migration of databases to microservices architecture as well as the functional and non-functional requirements of a modern financial application. The biggest challenge in developing a microservice-based database for financial services or other sectors is ensuring data integrity without creating tightly-coupled dependencies between services.

Query Optimization and Indexing

In a distributed environment, a database specialist needs to analyze the queries and tune the database indexes, having deep knowledge of the query execution plans, bottlenecks and access patterns. The scientific literature on auto-statistics collection in database management systems (DBMS) shows that the main task with regard to the query optimizer is to ensure that the metadata regarding the statistical data distribution characteristics is maintained correctly [3]. Query optimization in distributed microservices architectures is often more difficult than for a monolithic architecture as the query may span multiple services and data stores. This adds network latency, data consistency models and communication between services to the cost-based optimizations. Database systems maintain and aggregate statistics on the cardinalities of tables, distributions and correlations of columns, and other metrics. These are used by the optimizer to choose access and join methods [3].

Automated tools for query optimization can also identify candidates for optimizations by analyzing execution statistics for a query, its slow query log, and the database metrics for all nodes in the distributed system. Machine learning or heuristic algorithms identify candidate indexes to create or drop, or transformations to rewrite a query using database optimization features. For proactive measures, continuous performance monitoring may identify potential bottlenecks adequately early. This can be based on an automated alerting mechanism by tracking metrics such as query execution time, connection pool usage, and transaction throughput. In cloud-native databases, query performance tuning can also include partition key choice, data distribution, and number of read replicas, again tailored to the application's access patterns.

Database-as-a-service technologies possess an ability to adapt to the workload scenarios. The capabilities of elastic infrastructures allow for the automatic allocation of computing resources to workloads. Throughput improvement of up to ten percent has been observed with request admission control and scheduling in e-commerce systems, while avoiding instability during overload [4]. Scaling

Journal of Information Systems Engineering and Management
2026, 11(2s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

techniques automatically adjust computational and storage resources allocated to a system according to observed changing load, increasing these resources when demand increases and de-provisioning resources when demand decreases. Minimum thresholds are maintained to reserve enough resources to guarantee service. Empirically, response time can improve by a factor of fourteen or more with advanced scheduling algorithms, and larger compute jobs see only small penalties of around fifteen percent [4].

Infrastructure-as-code also allows to define configurations that are repeatable and reproducible in a disaster recovery scenario. For database infrastructure, this is generally done declaratively. In such cases, the database infrastructure (compute instances, storage volumes, networking configurations, backups, etc.) can be managed via infrastructure-as-code, easing version control, maintainability, testing, and deployments of database infrastructure between development, testing, staging, and production environments. Besides being able to handle changing user demand, modern DBaaS systems apply resource-allocating algorithms for admission control. When resource requests exceed the estimated capacity, additional requests are queued until the capacity is released, preventing overloading and thrashing while providing maximum performance even when demand is many times higher than capacity [4]. For example, one study observed an admission control system achieving 941 interactions per minute compared to 589 for non-admitted requests.

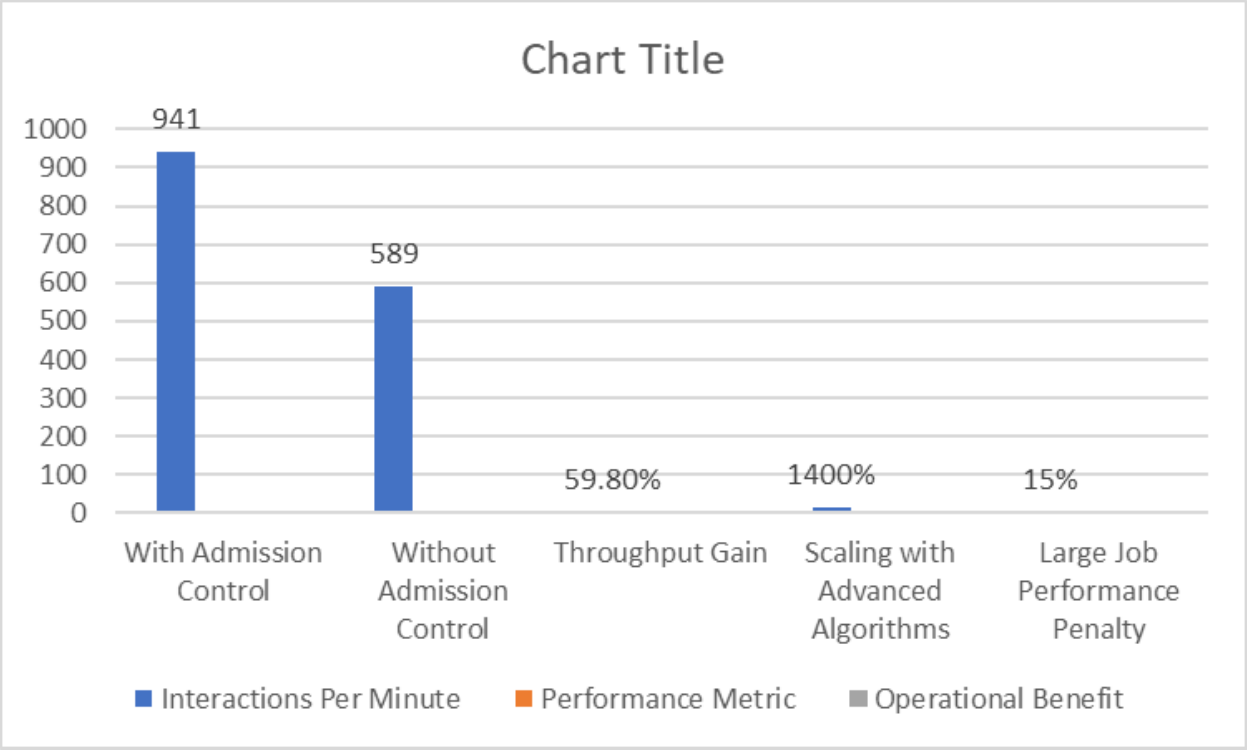

Fig 1: Resource Management and Admission Control Impact on Database Performance [3, 4]

Domain-Driven Decomposition Strategies

Identifying Bounded Contexts

The first step of a microservices migration is to identify clear business boundaries in monolithic applications. Decomposing a monolith by technical tiers is less effective than decomposing a system based on clearly identifiable business capabilities, as advocated by domain-driven design. Microservices migration research and architecture decision records state that the bounded context

Journal of Information Systems Engineering and Management
2026, 11(2s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

should be determined via an analysis of the currently existing system, and that organizations usually find between five and twenty bounded contexts while extracting a medium or large monolith. In this environment financial domains such as payment processing, account management and transaction reconciliation are examples of bounded contexts because they each have distinct responsibilities and a limited number of dependencies.

To identify bounded contexts it is possible to start by looking at the domain models and the ubiquitous language used in the different business areas, and the natural boundaries for aggregates in the business logic. Research shows that well-identified bounded contexts correlate with a reduced coupling between services and long-term maintainability [5]. Domain-driven design (DDD) recommends that service boundaries be defined by business capabilities rather than technical concerns. A bounded context encapsulates a coherent set of domain concepts and has few dependencies on other bounded contexts. If the software is monolithic, aggregates and value objects may be identified as microservices. Some studies have identified that organizations that published ADRs while undertaking decomposition had well-documented service boundaries, which helped in knowledge transfer and architectural knowledge between the developers involved in the transition from monolith to microservice [5].

Finding the right granularity is hard, as the more autonomy a granular domain brings, the more complexity it adds to a distributed system. Each microservice should be a business capability with its own domain model and business logic. This independence allows independent development, deployment, and scaling of services, without the coordination costs associated with large services, leading to an increase in the overall velocity. However, research about microservices architectural patterns and the quality attributes of a system indicated that operational complexity is strongly influenced by service granularity. Fine-grained services increase the network overhead. Coarse-grained services limit the opportunities to deploy and scale [6].

It turns out that the sweet spot is somewhere between a service too fine-grained causing too much network traffic, and a service too coarse-grained causing monolithic trade-offs again. Field studies of microservices in the wild across several domains uncovered a size between a bounded context with cohesive business logic and a set of related capabilities sharing data models and transaction boundaries [6]. This eases autonomous teams, technology independence, and independent deployment cycles, which combine to form environments suitable for continuous delivery. Empirical studies show that microservices architectures that stress autonomy of services show higher deployment frequency and lower change failure rates than those in tightly coupled distributed systems [6]. Service autonomy has non-technical aspects too. Services may have their own lifecycle, version policy, ownership model, and their teams are free to make architectural and implementation decisions locally. Along with less complex interfaces, high service autonomy also enables organizations to achieve higher developer productivity and coordination efficiency, as teams do not need to coordinate as much with others when iterating on their services [6].

Granularity Level	Network Communication	Deployment Flexibility	Scaling Independence	Coordination Overhead	Team Autonomy	System Complexity
Too Fine-grained	Very High Overhead	Very High	Very High	Excessive	High	Very Complex

Journal of Information Systems Engineering and Management
2026, 11(2s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

Optimal (Single Bounded Context)	Moderate	Very High	Very High	Low	Very High	Manageable
Optimal (Related Capabilities)	Moderate	High	High	Low	High	Moderate
Too Coarse-grained	Low Overhead	Limited	Limited	Moderate	Limited	Monolithic-like

Table 1: Service Granularity Trade-offs: Balancing Independence Against Distributed System Complexity [5, 6]

Containerization and Cloud-Native Architecture

Cloud-native development is a software development approach based on modern development frameworks. Thanks to their containerisation and/or microservice capabilities, Docker containers have been demonstrated to have lower overhead (in terms of startup time, memory requirements, security) than virtual machines. This means they can run much more efficiently. Startup time in a container can be reduced from minutes to milliseconds. Memory overhead per container can be reduced from gigabytes to 10-50MB [7]. This technology refresh allows enterprises to use contemporary software development practices with compatibility when migrating between techniques.

Modern containerization frameworks are built on a much more efficient computing model and allow for portable applications across heterogeneous computing environments. Studies on container clusters using Docker and Kubernetes show that enterprises are able to fit between 10 and 100 containers on each host server, compared to the range of between 5 and 10 virtual machines per host on the same server, drastically improving infrastructure utilization and costs [7]. Modern container virtualization stacks have several layers of filesystems, with lower layers (base images) shared between multiple containers. Only application-specific layers consume disk space. Therefore, resource consumption can be lower and the creation of new images can be done quickly in a distributed manner. In addition, container orchestration platforms, such as Kubernetes, have shown supporting scalable cluster sizes of more than 5000 nodes and more than 150,000 pods in one cluster [7]. Thanks to recent advances in container networking, overlay networks, service meshes, and ingress controllers are now available to implement complex microservice usage patterns with performance characteristics appropriate for production workloads.

In containerization, a service and its dependencies are encapsulated in lightweight containers that are then scheduled by orchestrators on pooled resources. Orchestrators also provide features such as auto-scaling, auto-healing, and rolling updates with zero downtime. Cloud computing containers Kubernetes supports various horizontal pod autoscaling algorithms. Applications can automatically scale from a single instance to hundreds of instances based on CPU utilization, custom metrics, or metrics supplied by third-party monitoring solutions. The Kubernetes scheduler makes scaling decisions within 30 to 60 seconds of a threshold being reached. [8] This massively improves service resiliency, which is the ability for systems to operate under varying load during instance updates or outages.

Container orchestration includes full lifecycle-management in which health checks are performed on containers and the health of the application is determined, by default, 10 seconds apart. If a

Journal of Information Systems Engineering and Management
2026, 11(2s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

container fails these checks, the orchestration manager should restart the container and deregister it from the service load balancers. One study of Kubernetes adoption found that Kubernetes rolling updates provided zero-downtime upgrades by gradually replacing old pod replicas with updated ones, using the `maxUnavailable` and `maxSurge` parameters (defaulting to 25% of the desired replica count) while still having a valid number of replicas on each update iteration [8]. Self-healing of orchestration failures can occur seconds to minutes after detection, with pod rescheduling generally occurring within 5 to 10 seconds of node failure detection, compared to manual failure remediation, which has a much larger mean time to recovery. Research shows that container orchestration systems support efficient bin packing and avoidance of resource contention via quality of service (QoS) classes, resource requests, and resource limits; for example, most production systems define CPU requests in units of millicores and memory requests in units of mebibytes to avoid differences in performance characteristics for different workloads [8].

Deployment Strategy	Update Speed	Availability During Update	Resource Overhead	Rollback Capability
Rolling Update (25% unavailable)	Gradual	75% minimum available	+25% surge capacity	Automatic
Zero-Downtime Deployment	Controlled	100% maintained	Variable surge	Built-in
Health-Based Deployment	Adaptive	Continuous availability	Minimal	Immediate
Manual Intervention	Slow	Variable	High	Manual

Table 2: Container Deployment Strategy Performance Metrics [7, 8]

Performance Optimization Techniques

Data Architecture Redesign

For cloud-native databases, new data modeling strategies might be required. For example, relational schemas optimized for relational database management systems running on a single server might need to be denormalized for distributed queries. Service relationship orchestration: Ensuring data consistency among microservices with large-scale Kubernetes deployments is considered one of the major challenges. For complex data-related scenarios, orchestration techniques can be leveraged with research on smart city platforms showing how orchestration mechanisms in distributed data-centric interactions preserve microservice isolation while guaranteeing final consistency [9]. Event sourcing and command query responsibility segregation patterns are another alternative to microservice data orchestration, they provide a mechanism to scale read and write operations independently.

Event-driven architecture patterns in cloud native applications require the identification of service dependencies and data flows to avoid cascading failures and to ensure reliability. As systems are built and scaled on Kubernetes-based platforms, managing service relationships becomes more important to the overall reliability of these systems. Platforms based on hundreds of microservices need to manage their dependencies and health states explicitly [9]. Refactoring data architecture based on distribution may involve adopting asynchronous communication patterns, such as adopting a practice in which services publish domain events to a message broker instead of making synchronous API service calls, thereby reducing coupling and allowing services to process events at their own pace without blocking upstream producers. One paper studying production microservices systems observed

Journal of Information Systems Engineering and Management
2026, 11(2s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

that adding service orchestration patterns like circuit breakers, timeouts and exponential backoff retries could improve system resiliency against resource starvation during partial failures and service degradation [9]. The architectural transition of the monolithic database to distributed data management proceeds as the team clarifies boundaries for ownership of data (each microservice can own its own data store) and eliminates shared databases that couple teams and create coordination bottlenecks.

Database-as-a-service platforms can also dynamically scale computing and storage resources in accordance with the workload characteristics to increase capacity and decrease capacity respectively during peak load and off-peak load, respectively, to provide performance and cost efficiency. Research into polyglot persistence patterns has demonstrated that different data storage technologies can offer different performance characteristics and operational costs, and that using appropriate database technologies for a particular workload can provide performance and cost benefits to organizations [10]. Infrastructure-as-code is used to codify the database technologies and other metadata to allow reproducible deployments and simplified disaster recovery.

Polyglot persistence is an architecture level solution, allowing organizations to use various specially tailored data storage technologies for each use case. This allows for specialized storage for each data requirement, rather than forcing data into a single database model. Various design studies of polyglot persistences have shown that document databases are a good fit for semi-structured data with flexible schema, key-value databases are a good fit for key-based lookups with sub-millisecond latency, relational databases are a good fit for transactional workloads with strong consistency guarantees, and graph databases for highly-connected data with complex traversals [10]. Some studies argue that there is additional operational complexity to polyglot persistence. This includes provisioning, configuring, and managing multiple database platforms, acquiring and training staff with expertise in the different databases, and accommodating cross database queries for business intelligence and reporting across polyglot persistence systems. Infrastructure as code practice can reduce some operational complexity challenges by codifying database provisioning scripts, configuration management, and backup processes. This can help achieve consistency across heterogeneous databases [10]. The choice of database technology for each part of the system often depends upon the nature of the current workload, such as the read to write ratio, the nature of querying patterns, and consistency and scalability requirements.

Workload Characteristic	Analysis Requirement	Impact on Technology Selection	Optimization Focus
Read-to-Write Ratio	Critical	High (determines scaling needs)	Read replicas vs. write optimization
Query Complexity	Important	High (affects technology choice)	Query engine capabilities
Consistency Requirements	Essential	Very High (ACID vs. BASE)	Transaction support level
Scalability Needs	Critical	Very High (horizontal vs. vertical)	Distribution strategy
Data Structure	Important	High (schema flexibility)	Data model alignment

Table 3: Workload Characteristics for Database Technology Selection [9, 10]

Journal of Information Systems Engineering and Management
2026, 11(2s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

Conclusion

With the shift from monolith to microservices architectures and database modernisation, this article identifies a wide variety of challenges and opportunities for organisations that seek to introduce greater agility and scalability into their digital technology. This article shows that the microservice architecture approach is based upon applying the domain driven design (DDD) technique of bounded context principles to identify the boundaries and scope of a microservice, selecting the appropriate service granularity and ensuring the optimal deployment and orchestration via containerisation technologies. Central issues of managing distributed data, such as cross-service query optimization, elastic resource provisioning, polyglot persistence, and maintaining consistency while allowing service autonomy using event-driven microservices and asynchronous communication, need to be considered for database re-platforming. The article results indicate that organizations making holistic architecture decisions while re-platforming databases achieve meaningful improvements in deployment frequency, system reliability, resource utilization, and development team productivity compared to adopting customary monolithic architecture approaches. Critical success factors include data store ownership boundaries, service orchestration capabilities such as circuit breakers and retry patterns, infrastructure-as-code patterns for reproducible deployments, and database technology choice depending on workload characteristics and consistency models. In summary, the article identifies architectural modernization patterns and anti-patterns to shape the design practices of enterprise system architects who face the challenge of accepting the trend towards more distributed, cloud-native solutions. The articles are particularly useful for modernization navigators, who must balance architectural improvement with system reliability and continuity of business operations.

References

[1] Claus Pahl et al., "Microservices: A Systematic Mapping Study," Researchgate, January 2016. Available: https://www.researchgate.net/publication/302973857_Microservices_A_Systematic_Mapping_Study

[2] Nicola Dragoni et al., "Microservices: yesterday, today, and tomorrow," in Present and Ulterior Software Engineering, Springer International Publishing, Researchgate, February 2017. Available: https://www.researchgate.net/publication/312137215_Microservices_yesterday_today_and_tomorrow

[3] Ashraf Aboulnaga , "Automated Statistics Collection in DB2 UDB," IBM Almaden Research Center, Researchgate, January 2004. Available: https://www.researchgate.net/publication/221310887_Automated_Statistics_Collection_in_DB2_UDB

[4] Sameh Elnikety et al., "A Method for Transparent Admission Control and Request Scheduling in E-Commerce Web Sites," Researchgate, April 2004. Available: https://www.researchgate.net/publication/37421633_A_Method_for_Transparent_Admission_Control_and_Request_Scheduling_in_E-Commerce_Web_Sites

[5] Muhammed Waseem et al., " A Systematic Mapping Study on Microservices Architecture in DevOps," Sciencedirect, December 2020. Available: https://www.sciencedirect.com/science/article/abs/pii/S0164121220302053

[6] Tingshuo Miao, " Systematic Mapping Study of Test Generation for Microservices: Approaches, Challenges, and Impact on System Quality," MDPI, 21 January 2025. Available: https://www.mdpi.com/2079-9292/14/7/1397

[7] Raj Verma & Prasun Jain, "Containerization in Cloud: Docker, Kubernetes, and Beyond," International Journal of Advanced Research in Computer Science and Software Engineering, Researchgate, Researchgate, October 2024. Available: https://www.researchgate.net/publication/385002203_Containerization_in_Cloud_Docker_Kubernetes_and_Beyond

[8] David Manor et al., "Containerization in Cloud Computing: A Review," Researchgate, February 2025. Available: https://www.researchgate.net/publication/389597107_Containerization_in_Cloud_Computing_A_Review

[9] Merkin Sebrechts et al., "Service Relationship Orchestration: Lessons Learned From Running Large Scale Smart City Platforms on Kubernetes," Researchgate, September 2021. Available: https://www.researchgate.net/publication/354805483_Service_Relationship_Orchestration_Lessons_Learned_From_Running_Large_Scale_Smart_City_Platforms_on_Kubernetes

[10] Abhishek Tiwari, "Polyglot Persistence Patterns," International Journal of Computer Applications, Researchgate, January 2012. Available: https://www.researchgate.net/publication/395241815_Polyglot_Persistence_Patterns