World Journal of Advanced Engineering Technology and Sciences
eISSN: 2582-8266
Cross Ref DOI: 10.30574/wjaets
Journal homepage: https://wjaets.com/

(REVIEW ARTICLE)

Check for updates

Migrating from monolithic to microservices architecture: A practical technical guide

Srinivas Sriram Mantrala *

Osmania University, India.

World Journal of Advanced Engineering Technology and Sciences, 2025, 15(02), 1892-1900

Publication history: Received on 04 April 2025; revised on 13 May 2025; accepted on 15 May 2025

Article DOI: https://doi.org/10.30574/wjaets.2025.15.2.0727

Abstract

This article presents a comprehensive guide to transforming software architecture from monolithic to microservices, delving into critical technical strategies and implementation methodologies. By addressing the essential considerations organizations must evaluate during migration, the content provides a detailed roadmap for breaking down complex systems effectively. The article emphasizes practical challenges and solutions, offering actionable insights for engineering teams navigating architectural modernization. Focusing on concrete implementation details rather than theoretical discussions, the guide delivers a pragmatic framework for transitioning to a more flexible and scalable software infrastructure.

Keywords: Microservices; Service architecture; Distributed Systems; Software Transformation; Technical migration

1. Introduction

Modern software systems face increasing demands for scalability, deployment speed, and technological flexibility. Migrating from monolithic to microservices architecture has emerged as a key strategy for addressing these needs, but the journey involves significant technical and organizational changes. A thorough understanding of both architectural paradigms is essential for successful migration planning. Monolithic while initially simpler to develop architectures, p, become increasingly difficult to maintain and evolve over time. Migration projects typically require careful planning over several months, with implementation timelines extending based on application complexity and team readiness [1].

A monolithic application consists of a single, unified codebase where all components—user interfaces, business logic, and data access—are tightly coupled. While this approach simplifies initial development, it creates substantial challenges as applications grow. The deployment of monolithic systems generally requires complete application rebuilding and redeployment, even for minor changes to a single component. This leads to longer release cycles and increased risk during each deployment, with larger monoliths often requiring deployment windows measured in hours rather than minutes [1]. The architectural limitations of monoliths become particularly evident when handling scalability requirements, as resource allocation cannot be targeted to specific high-demand components.

Microservices architecture, in contrast, breaks applications into small, independent services, each with a specific business capability and its own deployment lifecycle. This architectural approach provides technical benefits, including independent deployability, allowing teams to release updates to individual services without disrupting the entire system. When properly implemented with containerization technologies, microservices enable more efficient resource utilization by scaling components independently based on demand patterns. The technical implementation requires establishing clear service boundaries and communication protocols, with each service typically focusing on a single business capability [2].

* Corresponding author: Srinivas Sriram Mantrala

The practical implementation of microservices involves several key technical considerations. Service granularity represents a critical decision point, with effective services providing enough functionality to be useful while remaining focused enough to be maintained independently. Communication patterns between services must be carefully designed, with synchronous REST-based communication being common for request-response patterns and asynchronous messaging for events and long-running processes [2]. Database architecture decisions significantly impact migration success, with options ranging from completely separated databases for each service to transitional approaches using schema separation within shared databases.

This guide focuses on the practical aspects of migration—what companies must consider, technical steps required, specific challenges to anticipate, and essential measures to ensure success. The migration process typically begins with identifying service boundaries based on business capabilities, followed by implementing supporting infrastructure, including containerization, orchestration, and monitoring systems. Organizations generally benefit from incremental migration approaches, converting smaller, less critical services first to build expertise before tackling core business functionality [2]. The technical implementation requires establishing patterns for service communication, data consistency, and deployment automation that align with organizational capabilities and business requirements.

2. Technical foundation: architecture assessment

2.1. Evaluating Your Monolith

Before beginning migration, organizations must conduct a thorough technical assessment of the existing application to understand its structure, behavior, and limitations. The evaluation process requires dedicated time for comprehensive analysis, with the depth of assessment directly correlating to migration success rates. This investment in assessment significantly reduces migration risks and provides critical insights for planning.

Codebase analysis forms the foundation of architectural assessment, enabling teams to visualize and understand component relationships. Static analysis tools should be employed to generate dependency graphs that identify both direct and transitive dependencies between modules. According to cloud architecture principles, effective assessments must examine how well the current application adheres to core architectural principles such as loose coupling and high cohesion [3]. These principles directly impact the feasibility of extracting services from the monolith. Teams should measure cyclomatic complexity for each module, identifying overly complex components as candidates for refactoring before extraction.

Runtime behavior analysis provides critical insights beyond static code examination, revealing actual system behavior under load. Implementing distributed tracing across critical transaction paths enables teams to understand request flows and identify inter-component communication patterns. Observability is essential for understanding how the application functions in production environments, as the actual behavior often differs from the theoretical design [3]. Performance monitoring during peak loads reveals the primary performance bottlenecks that often indicate natural service boundaries, as they represent distinct functional areas with unique scaling requirements.

Database structure assessment represents a crucial aspect of migration planning. Entity-relationship mapping typically reveals distinct data domains that could be separated. Transaction boundary analysis is particularly important, as it identifies which data entities must maintain consistency within the same service. Foreign key relationship documentation is essential, with constraints that must be maintained or replaced with alternative consistency mechanisms during migration.

Infrastructure evaluation provides insights into operational aspects of the current system. Documentation of deployment processes reveals the steps required for production deployment, including any manual interventions needed. Resource utilization patterns often show significant inefficiencies that indicate opportunities for optimization through independent service scaling.

2.2. Defining Migration Goals

Establishing clear technical objectives provides essential direction for the migration effort and creates measurable success criteria. Technical planning documents should articulate specific goals across multiple dimensions to guide implementation decisions and evaluate progress.

Performance targets should be defined based on current system behavior and business requirements. Clean architecture approaches recommend establishing clear boundaries between system components, with interfaces that

hide implementation details [4]. This principle directly affects performance as it reduces unnecessary coupling that creates latency. Scalability requirements should specify the exact concurrent user or transaction volumes the system must support.

Deployment metrics establish expectations for the operational capabilities of the new architecture. The dependency rule is fundamental to defining these metrics, ensuring that source code dependencies only point inward toward higher-level policies [4]. This architectural principle enables independent deployment of components, significantly reducing deployment time and complexity. Rollback time requirements are particularly important, with effective microservices architectures enabling quick service restoration after failure detection.

Maintenance goals ensure the long-term sustainability of the new architecture. According to clean architecture principles, systems should be organized around business rules rather than frameworks or tools, making them more maintainable over time [4]. Developer onboarding targets typically aim to reduce time-to-productivity compared to monolithic codebases by limiting the scope of each service. Technical debt reduction goals should be explicitly defined, with focused efforts on improving architecture during the migration process.

Table 1 Architectural Assessment Components and Benefits [3,4]

Assessment Component	Key Benefit
Codebase Analysis	Identifies module dependencies and extraction feasibility
Runtime Behavior Analysis	Reveals natural service boundaries under actual load
Database Structure Assessment	Maps data domains for proper separation
Infrastructure Evaluation	Identifies scaling opportunities and inefficiencies
Clean Architecture Implementation	Reduces coupling and improves independence

3. Migration strategy planning

3.1. Identifying Service Boundaries

Establishing clear service boundaries is a critical technical prerequisite for successful microservices migration. The effectiveness of the resulting architecture depends directly on how well service boundaries align with business capabilities and technical considerations. A systematic approach to boundary identification significantly reduces the need for service refactoring during later migration phases.

Domain-Driven Design techniques provide a structured methodology for identifying service boundaries based on business domains. Event storming workshops bring together technical and business stakeholders to map business processes and identify domain events, commands, and aggregates that form natural service boundaries. This collaborative approach creates a shared understanding of the problem domain, leading to more appropriate service boundaries [5]. Bounded context mapping visualizes relationships between potential services, with particular attention to shared data and functionality. These maps reveal translation layers needed between domains and highlight areas where service boundaries may be problematic. The definition of ubiquitous language creates a shared vocabulary between technical and business teams, reducing misunderstandings that lead to inappropriate service boundaries. Domain invariants and consistency requirements documentation captures business rules that must be maintained during the migration.

Transaction flow analysis provides insight into how data and operations move through the system, revealing natural boundaries based on operational patterns. Tracing complete business transactions through the system identifies distinct processing stages that can form service boundaries. Transactional boundary identification determines which operations must maintain ACID properties within a single service to preserve data integrity. Cross-service communication requirements documentation captures the necessary interactions between proposed services, with interaction density serving as a key metric for boundary effectiveness. Mapping read versus write operations provides insight into data access patterns, with read-heavy functionality often more easily extracted into separate services than write-intensive components [5].

Change frequency mapping leverages version control history to identify natural service boundaries based on how code evolves over time. Code that frequently changes together should typically be encapsulated within the same service to minimize cross-service dependencies during updates. High-churn areas benefit significantly from independent deployment capabilities provided by microservices. Identifying stable interfaces that rarely change provides opportunities for well-defined service boundaries, as these interfaces can serve as natural separation points with minimal future disruption.

3.2. Decomposition Strategy Selection

Selecting an appropriate decomposition strategy is essential for minimizing risk and disruption during migration. Different technical approaches suit different application characteristics and organizational constraints, with successful migrations often employing multiple complementary strategies.

The Strangler Fig Pattern implementation provides a gradual migration approach that minimizes risk by incrementally replacing monolithic functionality with microservices. This pattern involves creating a façade that intercepts calls to the monolith and redirects them to new microservices as they are developed [6]. Setting up a proxy or API gateway to route requests enables transparent redirection between old and new implementations without client modifications. Creating interception points for specific functionality allows targeted migration of distinct features, with technical implementations typically intercepting at network, service, or library boundaries. Implementing routing rules directs traffic appropriately between monolithic and microservice implementations. Establishing comprehensive monitoring for both old and new implementations enables comparison of behavior and performance.

The Branch by Abstraction technique offers an alternative approach focused on internal codebase refactoring before external service extraction. This technique involves creating an abstraction layer over the functionality to be extracted, then building a new implementation behind this abstraction [6]. Creating abstraction interfaces for functionality to be extracted establishes a clear boundary between the target functionality and the rest of the monolith. Implementing the abstraction in the monolith first enables testing and validation while maintaining the existing system structure. Building a new service implementation of the interface allows for independent development and testing of the microservice version. Gradually switching clients to use the new implementation minimizes risk.

The Parallel Run approach emphasizes validation and verification during migration by running both implementations simultaneously. This strategy provides safety by ensuring the new implementation produces the same results as the old one before fully transitioning [6]. Implementing the new service alongside monolithic functionality enables direct comparison of behavior under identical conditions. Sending production traffic to both implementations provides real-world validation of the microservice implementation without risking customer impact. Comparing results ensures correctness by identifying discrepancies between implementations. Gradually shifting traffic from monolithic to microservice implementations manages risk effectively.

Table 2 Service Boundary Identification and Decomposition Methods [5,6]

Boundary Identification Method	Primary Purpose
Domain-Driven Design	Aligns services with business capabilities
Transaction Flow Analysis	Identifies processing stages and data integrity boundaries
Change Frequency Mapping	Groups code that evolves together
Strangler Fig Pattern	Enables incremental replacement with minimal disruption
Branch by Abstraction	Facilitates internal refactoring before extraction

4. Technical implementation

4.1. Service Design and Development

Effective service design and development establish the foundation for a successful microservices architecture. Service templates standardize development and reduce setup time across development teams. These templates should include standardized project structures with common components like health checks, logging configurations, and dependency management. Each service requires basic observability hooks, including health check endpoints that provide detailed status information. Implementing health checks as separate endpoints allows systems to monitor both liveness (is the

service running) and readiness (can it accept requests), enabling more sophisticated orchestration decisions [7]. Logging configurations should capture structured data with consistent formats, making patterns and anomalies easier to detect across services.

API design guidelines ensure consistency and interoperability across services. RESTful resource naming conventions should follow established patterns, with noun-based resource identifiers and clear hierarchical relationships. API versioning strategies are essential for maintaining backward compatibility during service evolution, with URL path versioning providing simplicity while header-based versioning offers more flexibility. When designing microservices APIs, teams should consider adopting API-first development practices where the interface is designed and documented before implementation begins, significantly reducing integration issues during later development phases [7].

Service-specific data modeling recognizes that each microservice may require different data structures optimized for its specific domain. Data models should be designed for service efficiency rather than enterprise-wide normalization, focusing on query patterns and functional requirements. Technology stack selection should be based on service-specific requirements rather than standardized across all services, with lightweight frameworks typically offering better performance while full-featured frameworks accelerate development.

4.2. Infrastructure and DevOps Implementation

Infrastructure and DevOps implementation provide the technical foundation that enables microservices to deliver on their architectural promises. Containerization has become the standard approach for packaging and deploying microservices, with standardized container configurations ensuring consistency across environments. Multi-stage builds separate build and runtime dependencies, significantly reducing final image sizes. Container orchestration platforms provide automated deployment, scaling, and failover capabilities essential for managing microservices at scale [7].

CI/CD pipeline development enables the frequent, reliable deployments that represent a key benefit of microservices architectures. Effective microservices pipelines implement continuous integration with automated testing at multiple levels—unit, integration, and end-to-end—providing confidence in service quality. Deployment automation reduces both deployment time and human error, with automated canary deployments enabling safer production releases by gradually shifting traffic to new versions [7].

Observability infrastructure addresses the increased monitoring challenges inherent in distributed systems. The three pillars of observability—logs, metrics, and traces—provide complementary views into system behavior. Distributed tracing enables end-to-end visibility into request flows across services, significantly reducing troubleshooting time in complex architectures.

4.3. Data Management Strategies

Data management represents one of the most challenging aspects of microservices implementation. Database decomposition techniques determine how data will be structured and accessed across services. While the database-per-service pattern provides maximum independence, it increases complexity for cross-service queries and transactions. Technical implementations must include clear data access layers that encapsulate database interactions, hiding implementation details and enabling future changes with minimal service disruption [8].

Data migration implementation handles the transition from monolithic to service-specific data stores. Migration strategies should consider both initial data migration and ongoing synchronization during transition periods. Change data capture technologies can monitor database transaction logs to identify changes, enabling real-time synchronization between systems with minimal performance impact [8].

Distributed transaction handling addresses the challenges of maintaining consistency across multiple services when traditional two-phase commits are no longer feasible. The Saga pattern breaks transactions into a sequence of local transactions, each with compensating actions that can be executed if later steps fail. Implementing this pattern requires careful design of both the transaction steps and the error handling mechanisms to ensure data remains consistent across services [8].

Table 3 Technical Implementation Components for Microservices [7,8]

Implementation Component	Key Focus
Service Templates	Standardized structure with health checks and logging
API Design	Resource naming and versioning strategies
CI/CD Pipelines	Automated testing and canary deployments
Observability Tools	Logs, metrics, and distributed tracing
Data Management	Database decomposition and saga patterns

5. Technical Challenges and Solutions

5.1. Network and Communication Challenges

Network and communication challenges represent fundamental technical hurdles in microservices architectures due to their distributed nature. Latency management becomes critical as network calls replace in-process function calls, introducing delays and potential failures. Implementing circuit breakers provides essential resilience by preventing cascading failures when downstream services experience problems. The circuit breaker pattern involves three states: closed (normal operation), open (failing, requests short-circuited), and half-open (testing recovery). This pattern is particularly important for synchronous communication, where service dependencies can create ripple effects throughout the application [9]. Retry mechanisms with exponential backoff help manage transient failures, automatically repeating failed requests with progressively longer delays between attempts.

Service discovery implementation enables services to locate and communicate with each other in dynamic environments where addresses change frequently. Implementation options include client-side discovery, where services query a registry directly, and server-side discovery, which uses an intermediary for routing. Health check endpoints enable automated monitoring of service status, with implementations providing both basic availability checks and deeper dependency verification [9]. API gateway configuration provides a unified entry point for external clients to access multiple microservices. Gateway implementations route requests to appropriate services, implement security controls, and provide rate limiting to protect backend services from excessive traffic.

5.2. Data Consistency Challenges

Data consistency challenges arise from distributing data across multiple services, each with its own data store. Transaction boundary management identifies which operations must maintain strict consistency and which can tolerate eventual consistency. The Saga pattern offers an alternative to distributed transactions, breaking complex operations into a sequence of local transactions with compensating actions for failures [9]. This approach maintains data consistency without the locking and coordination overhead of traditional two-phase commits.

Eventual consistency handling acknowledges that perfect consistency is often impractical in distributed systems. Event-based synchronization propagates changes between services asynchronously, using message brokers to distribute events reliably. Background reconciliation processes periodically check for and resolve inconsistencies, complementing event-based approaches to handle missed events or processing failures [9]. Caching strategy implementation balances performance against data freshness in distributed environments. Effective cache invalidation strategies ensure data accuracy when underlying information changes, using approaches like time-based expiration, explicit invalidation events, or write-through updates.

5.3. Operational Challenges

Operational challenges in microservices environments stem from increased system complexity. Debugging across services requires tracing requests through multiple components, significantly more complex than monolithic debugging. Correlation IDs provide a crucial foundation, attaching a unique identifier to each request that is propagated across all service boundaries. With properly implemented correlation, teams can track requests end-to-end through complex distributed systems [10]. Centralized logging with context preservation aggregates information from all services, making it possible to reconstruct transaction flows across system boundaries.

Deployment coordination addresses the challenges of updating interconnected services while maintaining system functionality. Feature flags control the activation of new functionality, allowing deployment separate from activation. Service contract testing verifies that service interfaces adhere to documented contracts, catching integration issues before deployment. Blue/green or canary deployment strategies reduce risk by enabling rapid rollback if issues arise [10].

Monitoring and alerting must account for increased complexity and dynamic behavior. The golden signals methodology focuses on four key metrics: latency (request processing time), traffic (load on the system), errors (failed requests), and saturation (system resource utilization). This approach provides comprehensive visibility while avoiding metric overload [10]. Anomaly detection identifies unusual patterns that might indicate problems, using statistical approaches to establish baselines and detect deviations. Dependency-aware alerting reduces noise by correlating alerts across services, suppressing downstream notifications when upstream services fail.

Table 4 Technical Solutions for Distributed System Challenges [9,10]

Challenge Area	Solution Approach
Service Resilience	Circuit Breakers and Retry Mechanisms
Service Discovery	Client/Server-side Registry Patterns
Data Consistency	Saga Pattern and Event-based Synchronization
Request Tracing	Correlation IDs and Centralized Logging
Deployment Safety	Feature Flags and Canary Deployments

6. Migration Execution Checklist

6.1. Pre-Migration Technical Preparation

Thorough pre-migration preparation establishes the technical foundation and organizational readiness necessary for successful microservices adoption. Environment readiness represents a critical first step, involving the deployment of essential infrastructure components that will support the microservices ecosystem. Containerization has become the predominant deployment technology for microservices, with Docker being the most frequently adopted solution according to systematic reviews of implementation patterns [11]. Orchestration environment setup enables automated deployment, scaling, and management of containerized applications, with Kubernetes emerging as the most widely adopted platform. CI/CD infrastructure configuration establishes the deployment pipelines necessary for independent service delivery, with automated testing and deployment capabilities essential for realizing the benefits of microservices architecture.

Team preparation focuses on the human aspects of migration, ensuring that development and operations personnel have the skills and processes necessary for microservices success. This preparation must address both technical skills and organizational structures, as successful microservices implementations typically require changes to team organization and communication patterns [11]. Service ownership documentation clearly defines responsibilities for each service, specifying which team owns development, operation, and evolution of each component, as clear ownership has been identified as a critical success factor in microservices implementations.

Technical validation through limited-scope implementations verifies architectural decisions before full-scale migration. Proof-of-concept development for critical services provides practical verification of design choices, with most successful migrations implementing prototype services before committing to large-scale conversion. This validation phase should focus particularly on inter-service communication patterns, as communication represents one of the most challenging aspects of microservices architecture [11].

6.2. During Migration Technical Steps

The migration execution phase requires careful sequencing and continuous validation to maintain system stability while transitioning to microservices. Incremental migration execution minimizes risk by converting small portions of functionality at a time, allowing for course correction as implementation progresses. Starting with non-critical, less complex services builds team experience while limiting business impact if issues arise. According to established

migration taxonomies, the "strangler application" pattern represents one of the most effective approaches for incremental migration, gradually replacing monolithic functionality while maintaining system stability [12].

Continuous validation throughout the migration process ensures that the evolving system maintains functional correctness and performance characteristics. This validation should address both functional equivalence and non-functional aspects such as performance and reliability. Data consistency verification across systems ensures that information remains accurate during and after migration, a particularly important consideration when implementing a database-per-service pattern as identified in microservices taxonomies [12].

Technical debt management during migration prevents accumulation of implementation issues that could undermine long-term success. Successful migrations typically establish standardized approaches for cross-cutting concerns like security, logging, and monitoring, reducing duplication and ensuring consistency across the microservices ecosystem. Documenting technical decisions captures the rationale behind architectural choices, providing essential context for future evolution.

6.3. Post-Migration Technical Verification

Post-migration verification confirms that the completed microservices architecture delivers expected benefits and maintains operational stability. Performance validation through comprehensive testing verifies that the new architecture meets or exceeds business requirements. Load testing of the complete system measures behavior under expected and peak conditions, while resilience testing verifies system behavior during component failures. According to established microservices taxonomies, resilience testing should address both infrastructure failures and application-level faults to ensure comprehensive verification [12].

Operational readiness verification ensures that the system can be effectively maintained in production. Monitoring capabilities should address both technical metrics and business-relevant indicators, providing visibility into system behavior at multiple levels. Disaster recovery procedure testing confirms that the system can be restored following major disruptions, an essential consideration given the increased complexity of distributed architectures.

Continuous improvement mechanisms ensure that the microservices architecture evolves to meet changing business needs. Regular architecture review sessions facilitate systematic evaluation of the implemented architecture, while feedback mechanisms capture enhancement opportunities from multiple stakeholders. These mechanisms support architectural evolution, which has been identified as a key success factor for long-term microservices sustainability [11].

7. Conclusion

Transitioning from monolithic to microservices architecture demands meticulous technical planning, incremental implementation, and continuous validation. By concentrating on practical implementation details—including precise service boundary identification, robust infrastructure establishment, effective data management, and comprehensive operational practices—organizations can achieve successful migration while minimizing potential risks. The journey toward microservices represents an ongoing architectural evolution, not a singular project. Each migration step should deliver tangible benefits aligned with business objectives, whether through enhanced deployment velocity, improved scalability, or greater technological adaptability. The technical guidance outlined in this article empowers engineering teams to navigate complex microservices challenges and establish a sustainable foundation for long-term architectural

References

[1] Nicola Dragoni et al., "Microservices: Yesterday, Today, and Tomorrow," arXiv, 2017. [Online]. Available: https://arxiv.org/pdf/1606.04036

[2] Ayukut Bulgu, "A Guide to Microservices Design Patterns for Java," Diffblue, 2023. [Online]. Available: https://www.diffblue.com/resources/a-guide-to-microservices-design-patterns-for-java/

[3] Tom Grey, "5 principles for cloud-native architecture: What it is and how to master it," Google Cloud, 2019. [Online]. Available: https://cloud.google.com/blog/products/application-development/5-principles-for-cloud-native-architecture-what-it-is-and-how-to-master-it

[4] Saiful Islam Rasel, "Book Review and Takeaways: Clean Architecture: A Craftsman's Guide to Software Structure and Design," LinkedIn, 2024. [Online]. Available: https://www.linkedin.com/pulse/book-review-takeaways-clean-architecture-craftsmans-guide-rasel-fefvc/

[5] Karthik Ramesh, "Domain Driven Design for Microservices: Complete Guide 2025," SayOne, 2023. [Online]. Available: https://www.sayonetech.com/blog/domain-driven-design-microservices/

[6] Sam Newman, "Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith," O'Reilly Media, 2020. [Online]. Available: https://dl.ebooksworld.ir/books/Monolith.to.Microservices.Sam.Newman.OReilly.9781492047841.EBooksWorld.ir.pdf

[7] Ronnie Mitra & Irakli Nadareishvili, "Microservices: Up and Running: A Step-by-Step Guide to Building a Microservices Architecture," O'Reilly Media, 2021. [Online]. Available: https://www.f5.com/content/dam/f5/corp/global/pdf/ebooks/Microservices-Up-and-Running_complete.pdf

[8] Unisys, "Microservices for the enterprise: Designing, developing and deploying," Unisys.com, 2022. [Online]. Available: https://www.unisys.com/blog-post/cis/microservices-for-the-enterprise-designing-developing-and-deploying/

[9] Sarah Neenan, "Resilient software strategies all developers should know," TechTarget, 2021. [Online]. Available: https://www.techtarget.com/searchapparchitecture/feature/Resilient-software-strategies-all-developers-should-know

[10] Cindy Sridharan, "Distributed Systems Observability: A Guide to Building Robust Systems," O'Reilly Media, 2018. [Online]. Available: https://unlimited.humio.com/rs/756-LMY-106/images/Distributed-Systems-Observability-eBook.pdf

[11] Paolo Di Francesco et al., "Architecting with microservices: A systematic mapping study," Journal of Systems and Software, Volume 150, Pages 77-97, 2019. [Online]. Available: https://www.ivanomalavolta.com/files/papers/JSS_MSA_2019.pdf

[12] Martin Garriga, "Towards a Taxonomy of Microservices Architectures," In book: Software Engineering and Formal Methods (pp.203-218), 2018. [Online]. Available: https://www.researchgate.net/publication/322878804_Towards_a_Taxonomy_of_Microservices_Architectures