IJARSCT

International Journal of Advanced Research in Science, Communication and Technology

International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal

ISSN: 2581-9429

Volume 5, Issue 12, April 2025

Impact Factor: 7.67

Restructuring Software Architecture: Moving From Monoliths to Microservices

Darshana Dadaji Ahire and Dr. Dipalee D. Rane
D. Y. Patil College of Engineering Akurdi, Pune, India
darshuahire@gmail.com

Abstract: *The shift from monolithic software architectures to microservices has become a key approach in contemporary software development, offering improvements in scalability, flexibility, and maintainability. This transformation addresses the limitations of tightly integrated systems, such as reduced agility and challenges in scaling individual components. In contrast, microservices advocate for a decentralized model, where independent services communicate via lightweight protocols, such as REST or message queues. This paper explores the key reasons for adopting microservices, including the ability to support rapid deployment cycles, enhance fault isolation, and optimize resource utilization. It delves into the core principles of microservices architecture, such as domain-driven design, bounded contexts, and continuous delivery. The paper also addresses the technical and organizational hurdles of migrating to microservices, including issues like data consistency, greater operational complexity, and the need for comprehensive monitoring and logging. It presents practical approaches for transitioning from a monolithic to a microservices-based system, such as incremental decomposition, implementing API gateways, and utilizing containerization technologies. The conclusion emphasizes the importance of aligning organizational structures with the new architectural approach, as highlighted by Conway's Law, to fully realize the advantages of this transformation.*

Keywords: monolithic software

I. INTRODUCTION

The problem definition for implementing a milk dairy software system using microservices architecture revolves around addressing the complexities and challenges faced by traditional monolithic applications used in dairy operations. This involves designing a system that can manage various functions—such as milk procurement, processing, inventory, billing, customer management, and supply chain logistics—while ensuring scalability, flexibility, and resilience.

Key Challenges

Scalability Issues in Monolithic Systems:

Dairy operations often deal with varying workloads, such as seasonal spikes in milk production, distribution, or customer demands. Traditional monolithic software systems struggle to scale efficiently across different functionalities. In a monolithic setup, scaling the entire system for one function (e.g., inventory management) means scaling other unrelated parts, which is inefficient and resource-heavy.

Lack of Flexibility and Agility:

Dairy operations require agility to respond to dynamic changes in customer demand, supply chain disruptions, or regulatory requirements. A monolithic system can become cumbersome and slow to update as all components are tightly coupled. Making a change in one part of the system may require testing and deploying the entire application.

As the dairy business evolves, adding new features such as new payment methods, AI for demand prediction, or a mobile app for delivery tracking becomes cumbersome without breaking the whole system.

Operational Complexity:

Dairy systems require robust data management capabilities to handle large volumes of transactions across various channels, including inventory tracking, order processing, and customer management. Microservices can help by offering independent services for each function, making it easier to manage and optimize different operations separately (e.g., one service for customer orders and another for inventory control).

Data Integrity and Consistency:

Monolithic systems struggle with data consistency when distributed over a network or across multiple locations (e.g., across dairy farms, processing plants, and distribution centers). Microservices architectures allow for better data consistency models, such as eventual consistency or Saga patterns, making the management of distributed transactions more reliable.

Objectives

The primary objective of this study is to investigate and understand the process of transitioning from monolithic software architecture to microservices. The research aims to identify the limitations of monolithic systems and highlight how microservices offer better modularity, scalability, and maintainability. Another key objective is to explore best practices, strategies, and tools that can facilitate this architectural shift while minimizing risk.

II. PROPOSED SYSTEM

The purpose of this report is to outline a proposed system for transitioning from a monolithic architecture to a microservices architecture. As organizations evolve and their software needs become more complex, the limitations of monolithic applications—such as scalability issues, difficulty in deployment, and slow response to changing business requirements—become apparent. This report details the proposed system architecture, the transition strategy, and the benefits expected from this transition.

Current System Overview

The current system operates as a monolithic application, where all components are tightly coupled and share a single codebase. This structure leads to challenges such as:

- Limited Scalability: Scaling requires duplicating the entire application, leading to inefficient resource usage.
- Slower Development: Coordination among teams is necessary for changes, which slows down the release of new features.
- Difficult Maintenance**:** Any modification can affect the entire system, making debugging and upgrades cumbersome.

Proposed Microservices Architecture

The proposed architecture will decompose the monolithic application into distinct microservices based on business capabilities. Each service will operate independently, communicate through APIs, and can be developed and deployed by separate teams.

Key Components of the Proposed Architecture:

- User Service: Manages user authentication, profile information, and user-related operations.
- Registration Service: Handles the user registration process, including input validation and account creation.
- Product Service: Manages product catalog, inventory, and related functionalities.

III. TRANSITION STRATEGY

The transition from a monolithic architecture to microservices will be carried out in a phased approach to minimize disruption and ensure a smooth migration.

Phase 1: Assessment and Planning

- Identify Business Capabilities: Conduct workshops to understand the existing functionalities and group them into services.
- Define Service Boundaries: Establish clear boundaries for each microservice based on the identified business capabilities.

Phase 2: Development of Microservices

- Incremental Development: Start with less critical services to allow teams to gain experience. For example, begin with the User Service and Registration Service.
- API Design: Create well-defined RESTful APIs for communication between services. Utilize tools like Swagger for documentation.

Phase 3: Implementation of Infrastructure

- Containerization: Use Docker to package microservices for consistency across development and production environments.
- Orchestration: Implement Kubernetes for managing containerized applications, automating deployment, scaling, and operations.

Phase 4: Deployment and Testing

- CI/CD Pipeline: Establish a continuous integration and continuous deployment pipeline using tools like Jenkins or GitLab CI/CD for automated testing and deployment.
- Monitoring and Logging: Integrate monitoring tools such as Prometheus and logging tools like ELK Stack to monitor microservices performance.

Phase 5: Gradual Transition

- **Strangler Fig Pattern**: Gradually replace parts of the monolithic application with microservices. Route new functionality to the microservices while maintaining the legacy system until full transition is achieved.

Benefits of Transition

The transition to microservices architecture is expected to yield several benefits:

- **Improved Scalability**: Each service can be scaled independently, allowing for optimized resource usage.
- **Faster Time to Market**: Development teams can work on different services simultaneously, accelerating the delivery of new features.
- **Enhanced Resilience**: Fault isolation ensures that issues in one service do not bring down the entire application.
- **Technological Flexibility**: Teams can choose the best technologies suited for each microservice, fostering innovation.

IV. SYSTEM DESIGN AND SPECIFICATIONS

Data Flow Diagram

The data flow diagram is basically a diagram which describes how the data is flow and processed inside a system. The data flow diagram (DFD) is a visual representation of data flow and processing of a system. The sequence as well as timing process in the system is represented by the sequence diagram and the control flow is represented by the flow chart.

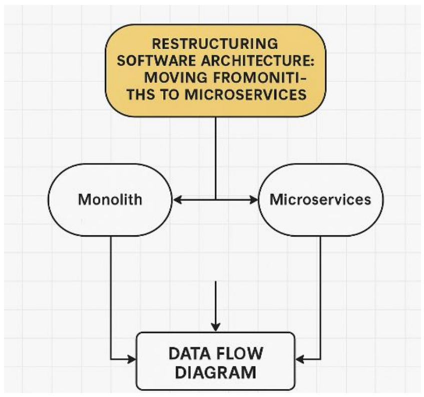

RESTRUCTURING SOFTWARE ARCHITECTURE: MOVING FROMONITI-THS TO MICROSERVICES
Monolith
Microservices
DATA FLOW DIAGRAM

Fig.: DFD Diagram

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 5, Issue 12, April 2025
Impact Factor: 7.67

The DFD is focuses on data processing in a system.
The DFD is very important for systems, which communicates with multiple systems. Because of high level user (might be non technical) can understand easily how the system interact with other
systems. In what manner the data comes in and how they are processed in the system and what is the output.

Class Diagram

In software engineering, a class diagram in the Unified Modeling Language (UML) is a type of static structure diagram that describes the structure of a system by showing the system's classes, their attributes, operations (or methods), and the relationships among objects.

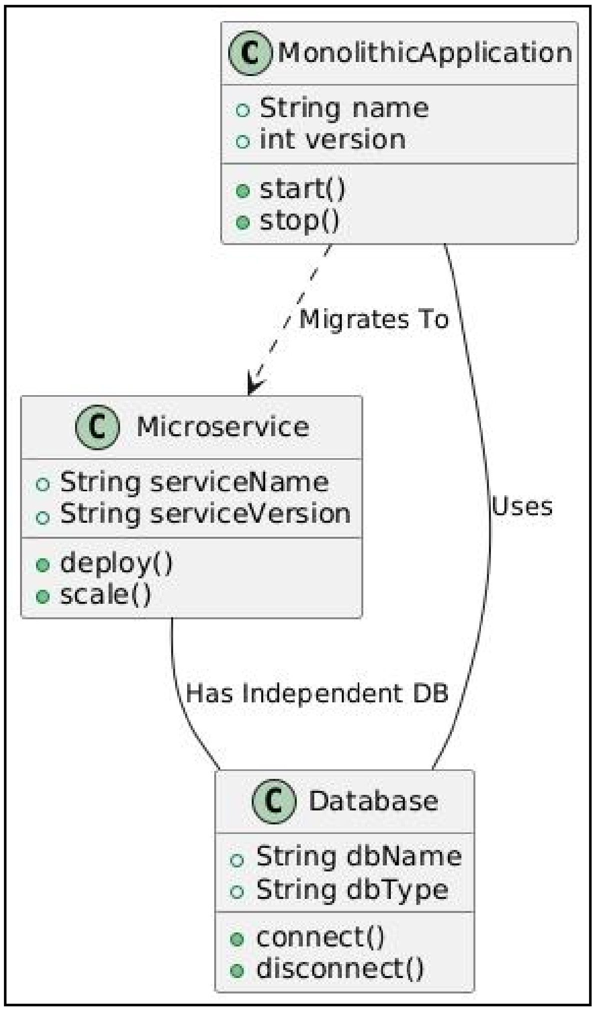


Fig: Class Diagram

Activity Diagram:

Activity diagrams are graphical representations of work flows of stepwise activities and actions with support for choice, iteration and concurrency. In the Unified Modeling Language, activity diagrams are intended to model both computational and organizational processes (i.e. work flows). Activity diagrams show the overall flow of control. Sequence diagrams are typically associated with use case realizations in the Logical View of the system under development

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
Volume 5, Issue 12, April 2025
ISSN: 2581-9429
Impact Factor: 7.67

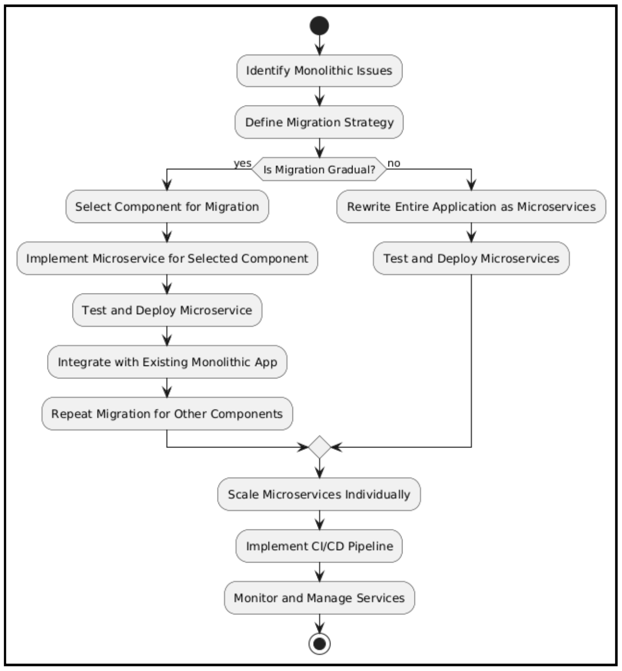
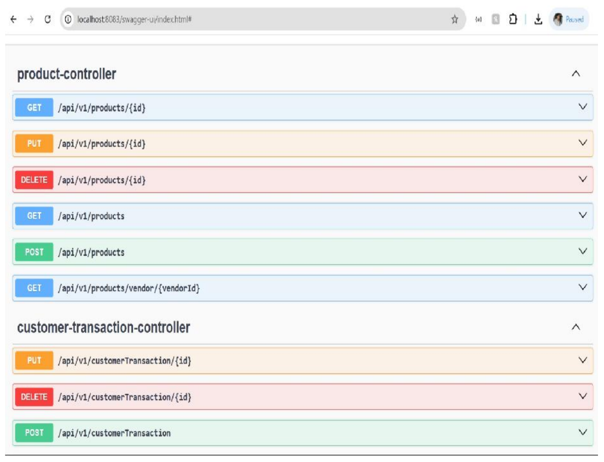

Identify Monolithic Issues
Define Migration Strategy
Is Migration Gradual?
yes
no
Select Component for Migration
Rewrite Entire Application as Microservices
Implement Microservice for Selected Component
Test and Deploy Microservices
Test and Deploy Microservice
Integrate with Existing Monolithic App
Repeat Migration for Other Components
Scale Microservices Individually
Implement CI/CD Pipeline
Monitor and Manage Services

Fig.Activity Diagram

V. RESULTS

Product microservices endpoint list.


localhost:8083/swagger-ui/index.html#

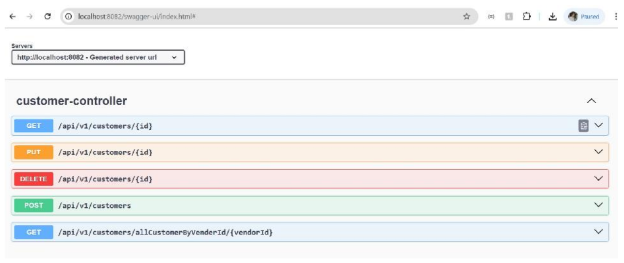
product-controller

GET /api/v1/products/{id}
PUT /api/v1/products/{id}
DELETE /api/v1/products/{id}
GET /api/v1/products
POST /api/v1/products
GET /api/v1/products/vendor/{vendorId}

customer-transaction-controller

PUT /api/v1/customerTransaction/{id}
DELETE /api/v1/customerTransaction/{id}
POST /api/v1/customerTransaction

Customer microservices endpoint list.

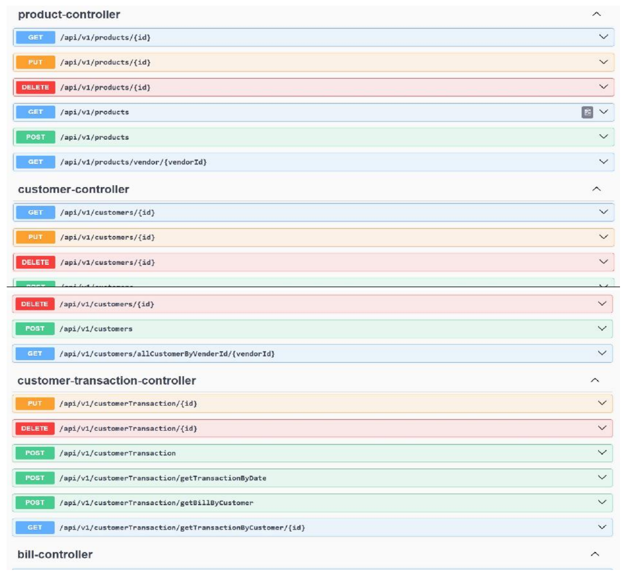

localhost:8082/swagger-ui/index.html#

Servers
http://localhost:8082 - Generated server url

customer-controller

GET /api/v1/customers/{id}
PUT /api/v1/customers/{id}
DELETE /api/v1/customers/{id}
POST /api/v1/customers
GET /api/v1/customers/allCustomerByVenderId/{vendorId}

Monolithic application endpoint list.



product-controller

- GET /api/v1/products/{id}
- PUT /api/v1/products/{id}
- DELETE /api/v1/products/{id}
- GET /api/v1/products
- POST /api/v1/products
- GET /api/v1/products/vendor/{vendorId}

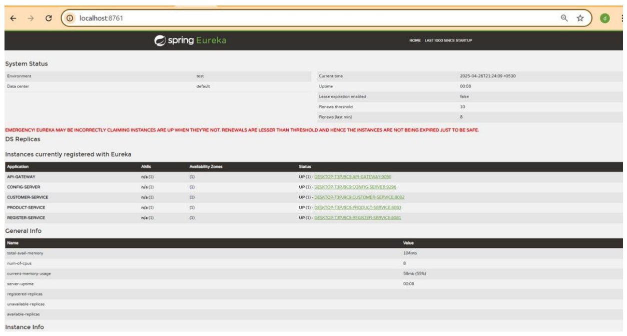
customer-controller

- GET /api/v1/customers/{id}
- PUT /api/v1/customers/{id}
- DELETE /api/v1/customers/{id}
- DELETE /api/v1/customers/{id}
- POST /api/v1/customers
- GET /api/v1/customers/allCustomerByVenderId/{vendorId}

customer-transaction-controller

- PUT /api/v1/customerTransaction/{id}
- DELETE /api/v1/customerTransaction/{id}
- POST /api/v1/customerTransaction
- POST /api/v1/customerTransaction/getTransactionByDate
- POST /api/v1/customerTransaction/getBillByCustomer
- GET /api/v1/customerTransaction/getTransactionByCustomer/{id}

bill-controller



localhost:8761

spring Eureka — HOME LAST 1000 SINCE STARTUP

System Status

Environment	test	Current time	2025-04-26T21:24:09 +0530
Data center	default	Uptime	00:08
		Lease expiration enabled	false
		Renews threshold	10
		Renews (last min)	8

EMERGENCY! EUREKA MAY BE INCORRECTLY CLAIMING INSTANCES ARE UP WHEN THEY'RE NOT. RENEWALS ARE LESSER THAN THRESHOLD AND HENCE THE INSTANCES ARE NOT BEING EXPIRED JUST TO BE SAFE.

DS Replicas

Instances currently registered with Eureka

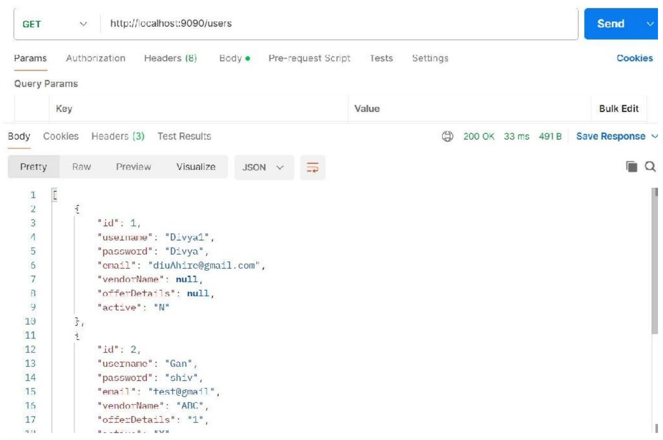
Application	AMIs	Availability Zones	Status
API-GATEWAY	n/a (1)	(1)	UP (1) - DESKTOP-T3PJ9C9:API-GATEWAY:9090
CONFIG-SERVER	n/a (1)	(1)	UP (1) - DESKTOP-T3PJ9C9:CONFIG-SERVER:9296
CUSTOMER-SERVICE	n/a (1)	(1)	UP (1) - DESKTOP-T3PJ9C9:CUSTOMER-SERVICE:8082
PRODUCT-SERVICE	n/a (1)	(1)	UP (1) - DESKTOP-T3PJ9C9:PRODUCT-SERVICE:8083
REGISTER-SERVICE	n/a (1)	(1)	UP (1) - DESKTOP-T3PJ9C9:REGISTER-SERVICE:8081

General Info

Name	Value
total-avail-memory	104mb
num-of-cpus	8
current-memory-usage	58mb (55%)
server-uptime	00:08
registered-replicas	
unavailable-replicas	
available-replicas	

Instance Info

Fig. Eureka dashboard



GET http://localhost:9090/users Send

Params Authorization Headers (8) Body Pre-request Script Tests Settings Cookies

Query Params

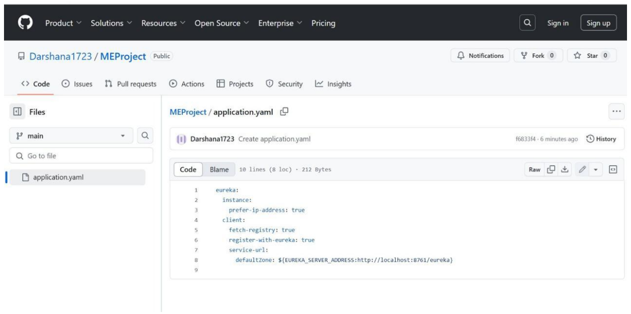
Key Value Bulk Edit

Body Cookies Headers (3) Test Results 200 OK 33 ms 491 B Save Response

Pretty Raw Preview Visualize JSON

```
[
    {
        "id": 1,
        "username": "Divya1",
        "password": "Divya",
        "email": "diuAhire@gmail.com",
        "vendorName": null,
        "offerDetails": null,
        "active": "N"
    },
    {
        "id": 2,
        "username": "Gan",
        "password": "shiv",
        "email": "test@gmail",
        "vendorName": "ABC",
        "offerDetails": "1",
```

Fig. Gateway port 9090



```yaml
eureka:
  instance:
    prefer-ip-address: true
  client:
    fetch-registry: true
    register-with-eureka: true
    service-url:
      defaultZone: ${EUREKA_SERVER_ADDRESS:http://localhost:8761/eureka}

```

Fig. Config repository

VI. SOFTWARE & HARDWARE SPECIFICATION

1. Development Frameworks
   - Spring Boot: A popular framework for building Java microservices with built-in support for RESTful APIs, security, and various data access technologies.
2. API Management
   - Swagger / OpenAPI: Tools for designing, documenting, and consuming RESTful APIs. They allow for automatic generation of API documentation and client SDKs.
3. Databases
   - Relational Databases: Options like PostgreSQL, MySQL, or Microsoft SQL Server for structured data management.

NoSQL Databases: Solutions such as MongoDB, Cassandra, or Redis for unstructured data and flexible schema designs.

4. Testing Frameworks
   - JUnit / TestNG: Frameworks for unit and integration testing in Java applications.
   - Mocha / Chai: Testing frameworks for JavaScript, often used in conjunction with Node.js microservices.
   - pytest: A testing framework for Python applications, useful for unit and functional testing of microservices.

Hardware Platform

An evaluation criterion for machine learning algorithms is the execution time. However, the execution time may vary depending on the performance of the computer being used. Because of this, the technical specifications of the computer used in the application are shared. The technical characteristics of the computer used in the implementation phase are

Central Processing Unit	Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz 2.90 GHz
Random Access Memory	8 GB (7.74 GB usable)
Operating System	Windows 10 Pro 64-bit

Space Complexity**:**

The space complexity depends on Presentation and visualization of discovered anomalies. More the storage of data more is the space complexity.

Time Complexity:

Check No. of anomalies available in the network= n

If (n>1) then retrieving of information can be time consuming.

Hardware Specification:

Processor: Intel Pentium 3 or above

RAM : 2 GB or above

Hard Disk : 200 GB or above

Other Hardware : Monitor , Keyboard , Mouse etc.

VII. FUTURE WORK

Future work in microservices architecture should focus on developing advanced tools for efficient service orchestration, automation, and monitoring to address the growing complexity of managing multiple services. Research into enhanced security protocols is essential to protect against the broader attack surface created by distributed systems. Additionally, improving data consistency mechanisms across microservices, especially in real-time synchronization, remains a critical area for optimization. Moreover, exploring strategies for integrating microservices with legacy systems and developing hybrid architectures can facilitate smoother transitions for organizations migrating from monolithic systems. These advancements will help businesses maximize the potential of microservices, ensuring scalability, flexibility, and security in increasingly complex environments.

VIII. CONCLUSION

Defining the feature scope in microservices is crucial for the success of microservices architecture. It ensures that each service remains focused, manageable, and aligned with business goals. By following the principles outlined above, teams can effectively design and implement microservices that provide robust and scalable functionalities. Microservice components collectively support the principles of microservices architecture, promoting independence, scalability, resilience, and agility in application development and deployment

REFERENCES

[1]. N. Dragoni et al., “Microservices: Yesterday, today, and tomorrow,” in Present and Ulterior Software Engineering, M. Mazzara and B. Meyer, Eds., Cham, Switzerland: Springer, 2017, pp. 195–216. [Online]. Available: https://doi.org/10.1007/978--3-319-67425- 4_12

[2]. M. Kalske, N. Mäkitalo, and T. Mikkonen, “Challenges when moving from monolith to microservice architecture,” in Proc. Int. Conf. Web Eng., I. Garrigós and M. Wimmer, Eds., Cham, Switzerland: Springer, 2018, pp. 32–47. [Online]. Available: https://doi.org/10.1007/978--3-319-74433-9_3

[3]. N. C. Mendonça, C. Box, C. Manolache, and L. Ryan, “The monolith strikes back: Why Istio migrated from microservices to a monolithic architecture,” IEEE Softw., vol. 38, no. 5, pp. 17–22, Sep./Oct. 2021.

[4]. [Online]. Available: https://doi.org/10.1109/MS.2021.3080335

[5]. Balalaie, A. Heydarnoori, P. Jamshidi, D. A. Tamburri, and T. Lynn, “Microservices migration patterns,” Softw.: Pract. Experience, vol. 48, no. 11, pp. 2019–2042, 2018. [Online]. Available: https://doi.org/10. 1002/spe.2608

[6]. Lewis, J., & Fowler, M. (2014). Microservices: Decomposing applications for deployability and scalability. https://martinfowler.com/articles/microservices.html

[7]. Alur, D., & Basu, A. (2017). Microservices and the journey from monolithic to distributed architectures. Springer.

[8]. Dragoni, N., Giallorenzo, S., Lenzini, L., & Mazzara, M. (2017). Microservices: A survey of recent advances. Software Engineering Conference (ICSE), 47-56.

[9]. Smith, B. (2019). The evolution of software architecture: From monolithic to microservices. Journal of Software Engineering, 10(3), 120-130.