VAWKUM Transactions on Computer Sciences
http://vfast.org/journals/index.php/VTCS@ 2024, ISSN(e): 2308-8168, ISSN(p): 2411-6335
Volume 12, Number 1, January-June 2024 *pp:94-110*

Transition Strategies from Monolithic to Microservices Architectures: A Domain-Driven Approach and Case Study

Momil Seedat [1], **Qaisar Abbas** [1], **Nadeem Ahmad** [2], **Irum Feroz** [2,3], **Affefah Qureshi** [2], **Alessia Amelio** [4]

[1]Faculty of Information Technology, University of Central Punjab, Lahore, Pakistan; [2]Department of Computing and Technology, Iqra University Islamabad, Pakistan; [3]Department of Computing, University of Portsmouth, United Kingdom; [4]Department of Engineering and Geology, Università degli Studi "G. d'Annunzio" Chieti - Pescara, Italy

Keywords: *Software Architecture, Domain Driven Design, System Migration, Microservices, Monolithic Applications, Bounded Context*

Journal Info:
Submitted:
April 15, 2024
Accepted:
May 16, 2024
Published:
June 3, 2024

Abstract This paper focuses on the systematic mapping of monolithic applications to microservices architecture, a popular alternative in the software development industry. The research examines the advantages of microservices and the challenges organizations encounter during the transition from monolithic systems. A case study of a financial application is presented, along with proposed techniques for identifying microservices within monolithic systems using domain-driven development concepts.The study highlights the difficulties and benefits of migrating from monolithic to microservices architecture, offering valuable insights for software architects and developers. Practical implications include a technique for identifying microservices on monolithic systems using domain-driven concepts and various communication protocols for service interaction. The findings suggest that this proposed technique can enhance work performance and establish clear models, particularly for complex systems. However, it may have limited effectiveness in less complex systems. The paper contributes to the field of software development by providing practical solutions for companies considering a shift to microservices architecture and comparing the two architectural styles.

***Correspondence author email address:** affefah.qureshi@iqraisb.edu.pk
DOI: 10.21015/vtcs.v12i1.1808

1 INTRODUCTION

In the modern business landscape, numerous organizations rely on large enterprise applications to run their operations effectively. However, with the continuous growth of businesses, these organizations are increasingly facing challenges when it comes to managing and scaling their applications. To address this issue, Martin Fowler

This work is licensed under a Creative Commons Attribution 3.0 License.

introduced the concept of Microservices Architecture as a solution [1], [2]. This architectural approach involves the development of a collection of small, independent services that collaborate as a unified application. These services communicate with each other using lightweight mechanisms, such as HTTP.

1.1 Microservices Architecture

Microservices architecture (MA) is an approach to building distributed applications, where the application is composed of individual modules known as microservices [1]. In MA, each microservice possesses its unique functionality, ensuring that a failure in one service does not have a detrimental impact on the entire application. This architecture promotes the encapsulation of related modules within a service, fostering high cohesion internally and loose coupling externally [3]. The benefits offered by this approach have prompted numerous companies, including Google, Amazon, IBM, and Netflix, to migrate from a monolithic architecture to a microservices architecture.

1.2 Monolithic Architecture

A monolithic architecture refers to an application consisting of a single code base where all components are tightly coupled with one another [4]. Developing, testing, and deploying monolithic applications are relatively easier [3]. Therefore, considering this architecture for initial project development can be a favorable choice. However, as the application grows, it becomes increasingly challenging to comprehend and modify the system, leading to slower development [5]. Issues such as locating errors in the code, complex code structure, and difficulties faced by developers working in the same environment may arise.

According to the study [6] both architectures have advantages and disadvantages, depending on the problem to be solved. Load tests indicate that Microservice architecture is more efficient for handling a high number of requests, offering scalability, reliability, and long-term maintainability, while Monolithic architecture is more efficient under lower loads and easier to develop and integrate. The choice should be guided by business goals to meet investor expectations.

1.3 Architecture Decomposition

Microservices architecture has become increasingly popular in the IT industry to address the challenges posed by monolithic systems [3]. However, building a new application solely with microservices can be costly and time-consuming due to the separate management of its components. To overcome this, a more efficient approach involves extracting small components from the existing monolithic architecture and developing new functionality as microservices. In this research article, we present an innovative technique for decomposing a monolithic application into microservices. The migration process presented various challenges, but ultimately, we successfully applied the technique to a real-world financial application with a substantial user base of over 1.2 million users.

2 LITERATURE REVIEW

Typically, the need to transition from a monolithic to microservices architecture arises when the codebase and company scale increase. As a result, new challenges related to refactoring and system structure emerge, aligning with the microservice design. Maintaining the monolithic system becomes increasingly challenging in such cases [7].

2.1 Background and Motivation

In 2008, the rise of microservices gained prominence after a critical data corruption incident and prolonged outages caused by a single mistake. Netflix responded by undertaking the migration of their entire application from a monolithic design to AWS cloud-based microservices. The primary goal was to enhance accessibility, scalability, and speed, ensuring seamless availability and swift operations around the clock. Similarly, as Uber expanded its

services to multiple cities, introducing new products resulted in rapid system growth, posing challenges in maintaining the monolithic architecture. Deploying the entire codebase at once hindered continuous integration, and long-term developers faced difficulties modifying the system due to dependencies between different application modules. Consequently, Uber opted to split the monolith into several codebases, adopting a service-oriented architecture (SOA) or, more specifically, a microservices architecture [8].

2.2 Migration of Architecture

The shift from monolithic architecture to microservices has gained popularity as companies aim to modernize their software applications with more flexible and scalable architectures. However, this transition poses several challenges. It requires careful planning, communication between teams, and continuous testing to ensure that the new architecture operates effectively [9]. Additional challenges include managing multiple services, handling the complexity of distributed systems, and managing potential impacts on application performance. It is crucial to implement proper security measures, such as securing communication channels between services and implementing authentication and authorization mechanisms, to ensure the safety of the application [10]. The monolithic architecture is well-suited for small-scale applications that have straightforward requirements. However, microservices architecture is better suited for complex applications that have multiple functionalities. Choosing an architecture style depends on several factors, including project complexity, scalability needs, and the development team's experience level. These findings highlight the importance of selecting the appropriate architecture style to ensure that it meets the project's unique requirements [11].

In their research, Hippchen conducted a case study focusing on the development of a microservices-based application using a domain-driven model [12]. The authors emphasized the value of utilizing existing functions, highlighting it as a significant advantage of employing domain-driven development alongside microservices. However, they also identified that the process of separating the domain model into multiple diagrams with domain views requires considerable effort and expertise. Rud in 2019 adopted a heterogeneous graph [8] for mapping software elements, i.e. programs and resources, and representing the different relationships among them, i.e. function calls, inheritance, etc., and employed a constraint-based clustering procedure on a novel heterogeneous Graph Neural Network (GNN). The performed experiments proved that the proposed approach is successful on different types of monoliths. However, the GNN needs a pre-training phase for encoder and decoder which can be computationally expensive.

Shifting from monolithic systems to microservices architectures can significantly improve the effectiveness of health informatics and mobile health (mHealth) applications. In the case of the Neonatal, Infant, and Maternal Death E-surveillance System, breaking down the system into microservices aligned with specific reporting and data collection tasks allowed for a more user-centered design, which facilitated real-time data gathering and enhanced the usability for technologically illiterate health workers [13]. Similarly, the development of a Usability-Based Rating Scale (UBRS) for evaluating mHealth applications underscores the importance of modularity in addressing usability barriers. By adopting a domain-driven approach, each microservice can be tailored to specific usability parameters, ensuring that the system as a whole meets the diverse needs of its users. This strategy not only improves data accuracy and user satisfaction but also allows for the incremental deployment of new features and services, ensuring continuous improvement and adaptability in dynamic healthcare environments [14].

A researcher Kalske [4], proposed a dataflow-driven semi-automatic decomposition approach characterized by four main steps: (i) generation of use case and business logic specification; (ii) construction of the fine-grained Data Flow Diagrams (DFD) and the process-datastore version of DFD (DFDPS) representing the business logics; (iii) extraction of the relationships between processes and datastores into decomposable sentence sets; and (iv) clustering of processes and their closely related datastores into single modules from the decomposable sentence sets for identification of potential microservices. One of the limitations of the proposed approach is the efficiency

of design.

Another researcher Newman considered the source code of the source application as the input and computed the similarities and relationships between all the system classes from their interactions and the domain terminology adopted within the code [15]. Then, they used a variant of a density-based clustering algorithm on the similarity values to create a hierarchical structure of the candidate microservices, together with potential outlier classes. However, such a solution is highly dependent on the selection of the DBSCAN hyperparameters.

Languric proposed a static approach for detecting microservices in a legacy software system based on topic models [16]. They used Latent Dirichlet Allocation (LDA) to identify the systems' topics, corresponding to domain terms, and representing the microservices implemented by that legacy system. A clustering method is then employed on the graph created from the combined topics for detecting the microservices. The limitation is that LDA needs to know the number of topics beforehand.

Raj and Ravichandra introduced an approach for extracting microservices from a Service-Oriented Architecture (SOA) based on graphs [17]. More specifically, they created four procedures: (i) building the Service Graph (SG), (ii) building the Task Graph (TG) for each service of the SOA, (iii) detecting potential microservices using the SG of SOA, and (iv) building a SG for a microservices application to keep the dependencies between the generated microservices. Although the proposed approach extracts the microservices, it is suitable for SOA architectures instead of monoliths.

Kalia et al. [18] proposed the Mono2Micro system, an AI-based toolchain that generates recommendations for splitting legacy web applications into microservice partitions. Static and runtime information are collected from a monolithic application and processed using a tempo-spatial clustering method to produce recommendations for splitting the application classes. Partitions are generated according to business functionalities and data dependencies. Although the framework is dynamic, it is only adopted for migrating legacy Java Enterprise Edition (JEE) applications toward a microservice architecture.

Transitioning from monolithic to microservices architectures can greatly enhance system scalability and maintainability, particularly in complex domains. A domain-driven approach emphasizes aligning software architecture with business needs, ensuring each microservice corresponds to a specific domain function. For instance, in the development of user-centered interfaces for deaf and functionally illiterate users, services were modularized to enhance accessibility and usability through specialized components such as the Italian Sign language dictionary and virtual character-based interfaces [19]. Similarly, in the usability analysis of educational information systems during the COVID-19 pandemic, a microservices architecture could allow for modular development and deployment of features such as online assistance, multilingual support, and interactive virtual classrooms. By decomposing these complex systems into manageable, domain-specific microservices, organizations can better address user requirements, improve system resilience, and adapt swiftly to changing needs. This case study underscores the importance of iterative development, continuous user feedback, and the integration of domain-specific knowledge in successfully transitioning to a microservices architecture [20].

In another study [21] four steps are proposed as a structured and iterative approach to migrating from monolithic to microservices architecture. The first step is analysis, in which the monolithic application is analyzed to identify its components and dependencies. The second step is extraction, in which each component is extracted into a separate service while preserving dependencies. The third step is refactoring, in which the extracted services are refactored to ensure adherence to microservices architecture principles, such as loose coupling and single responsibility. The fourth and final step is orchestration, in which an orchestration layer is implemented to manage communication between the microservices, such as through an API gateway or service mesh.

Haugeland presents a migration process that enables the transformation of monolithic applications into cloud-native microservices-based applications [22]. The authors recommend a gradual migration approach, starting

with the most critical functionality and adding more microservices over time. To demonstrate the feasibility of this approach, the authors presented a case study of a healthcare system that underwent this migration process. The new system is a multi-tenant cloud-native application that provides customizable healthcare solutions for different organizations.

The migration process from monoliths to microservices can bring performance benefits such as reduced latency, improved throughput, and better resource utilization. However, there are trade-offs between performance and modularity. The migration process can lead to increased network overhead, more complex deployment, and reduced fault tolerance. In 2020, Santos and Silva have given a complexity metric for migrating monolithic applications to microservices-based architecture [23]. The metric considers the number of microservices, the complexity of the communication between the microservices, and the complexity of the data flow between the microservices. The authors argue that this metric can help organizations obtain a quantitative measure of the complexity of the migration process and allow them to assess the potential risks and benefits of the migration.

The study conducted by the authors Blinowski, Ojdowska and Przybyłek in 2022 involves testing the performance and scalability of a monolithic and microservice architecture using a benchmark application [24]. The results showed that the microservice architecture outperformed the monolithic architecture in terms of response time and throughput. The microservice architecture was also found to be more scalable, as it was able to handle a higher number of requests per second. However, the authors note that implementing a microservice architecture can be more complex and require more effort than a monolithic architecture. Additionally, they caution that a poorly designed microservice architecture can result in worse performance and scalability than a well-designed monolithic architecture.

The study [25] provides a primary checklist to guide practitioners in the DDD4M application process and also highlights that there is a need for enhanced methodological support.

Our approach to microservice decomposition is based on data flow diagrams derived from business logic. By aligning with the desired operations and extracting relevant data from real-world applications, our approach ensures the delivery of a well-defined and optimized architecture.

3 TECHNICAL CHALLENGES

In the 1990s, internet companies typically operated by running a single, large monolithic program on a server, aiming to provide convenience to end customers. To handle increased traffic, these companies would simply add more instances of the monolith. A monolith consolidates the codebase, enabling engineers to debug the application by stepping through any part of the code. Additionally, a monolith can handle user requests entirely within its own system, minimizing the need for network calls and reducing the risk of network failures. Many software companies employ the monolithic pattern for their code [26]. However, when issues arise in these monoliths, fixing them in a centralized location leads to tightly coupled components that are challenging to modify, posing various difficulties for teams working in the same environment [27]. If the program becomes too large, running it on a typical traditional machine becomes impractical.

3.1 Monolithic and Microservices Architecture's Comparison

In enterprise applications, the decision to break up a monolithic application involves various considerations beyond its size. Microservice architecture offers a more practical approach for software development teams to take ownership of different code components when they are in separate areas and lack the need for extensive communication [15]. This decentralization of responsibilities streamlines the development process, as teams in different geographic locations no longer have to make changes to common software components. Assigning specific services to individual teams ensures cleaner code and faster resolution of technical issues. By adopting a microservices approach, the complexities associated with managing a monolithic application's growing size can

be effectively mitigated, reducing confusion for developers, and minimizing the likelihood of encountering bugs and errors in uncertain situations [28]. For small software applications where there is no requirement for multiple software development teams, a monolithic approach may be the most suitable option [5]. This is especially true when the application is small in scale and its future is predictable. Fig. 1 depicts the three-layer architecture commonly seen in enterprise software applications. It is evident that there is typically only one database, making it challenging to accommodate data normalization and scaling across different databases. As a result, this architecture restricts the ability of teams to meet growing demands and firm requirements. On the other hand, when utilizing microservices, as illustrated in 2, it becomes possible to choose either a relational database (SQL) or a non-relational database (NoSQL) for each service. This grants software development teams more flexibility in selecting the appropriate tools for their implementations [1]. Table 1 provides a comparison of these two architectural types, highlighting their respective advantages and disadvantages. It is evident that both architectures have their merits and drawbacks. However, when it comes to managing large enterprise applications, the microservice architecture style emerges as a more compelling choice.

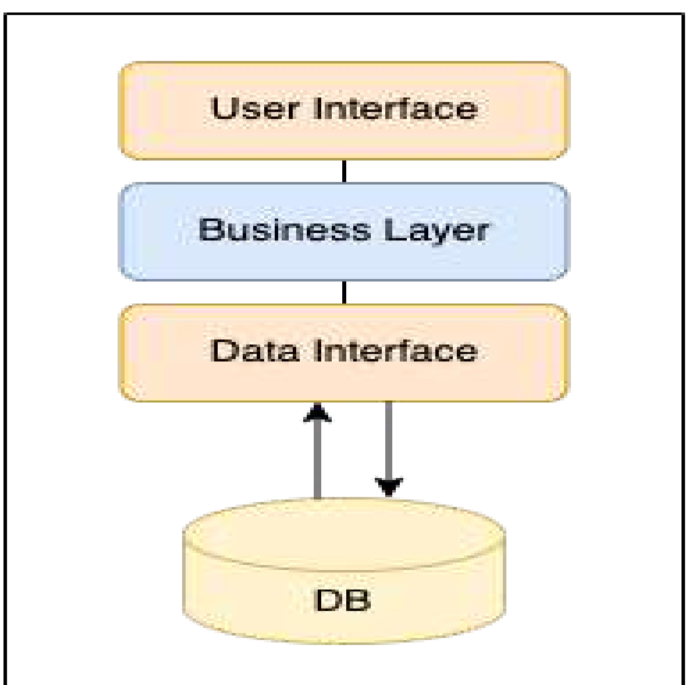

Figure 1. Typical Monolithic Application Consisting of Three Layers

-4mm

4 THE PROPOSED MIGRATION TECHNIQUE

Our proposed mechanism for reducing the complexity of microservice decomposition using domain-driven design involves the following steps:

Step I. Identification of Use Cases

- Identify the set of use cases U=u1, u2, u3, ...
- Create a data flow diagram for each use case in U

Step II. Identification of Bounded Contexts

- Identify the set of bounded contexts BC with the help of domain experts. If a bounded context has sub-contexts, identify the inner bounded contexts.

Step III. Identification of Entities, Aggregates, and Domain Services

- For each bounded context B in BC, identify the set of entities E, aggregates A, and domain services D.
- For each E, A, and D in B, identify the set of business processes BP that use them to get data from other processes.

Step IV. Reduction of Complexity using Combination Functions

- Define a set of systems S,(a set of entities E, a set of aggregates and a set of domain services D). Additionally, define a set of business processes BP.
- Represent the relationships between these sets using a bipartite graph G = (V, E), where V = S BP T and E is a set of edges connecting systems to processes and processes to tables.
- Define the functions CombineSystems: $S \times BP \rightarrow S'$, CombineProcesses: $BP \times T \rightarrow BP'$, CombineSystemsAnd-Processes: $S \times BP \rightarrow S'$
- Identify nodes in G that share edges and can be combined into a single node.
- Use the CombineSystems and CombineProcesses functions to merge nodes into a single node.
- Update G with the new nodes and edges.
- Repeat steps 3-5 until no further nodes can be combined.
- Return the resulting graph as the optimized system. The result would be the microservices.
- Repeat the same process for each bounded context.

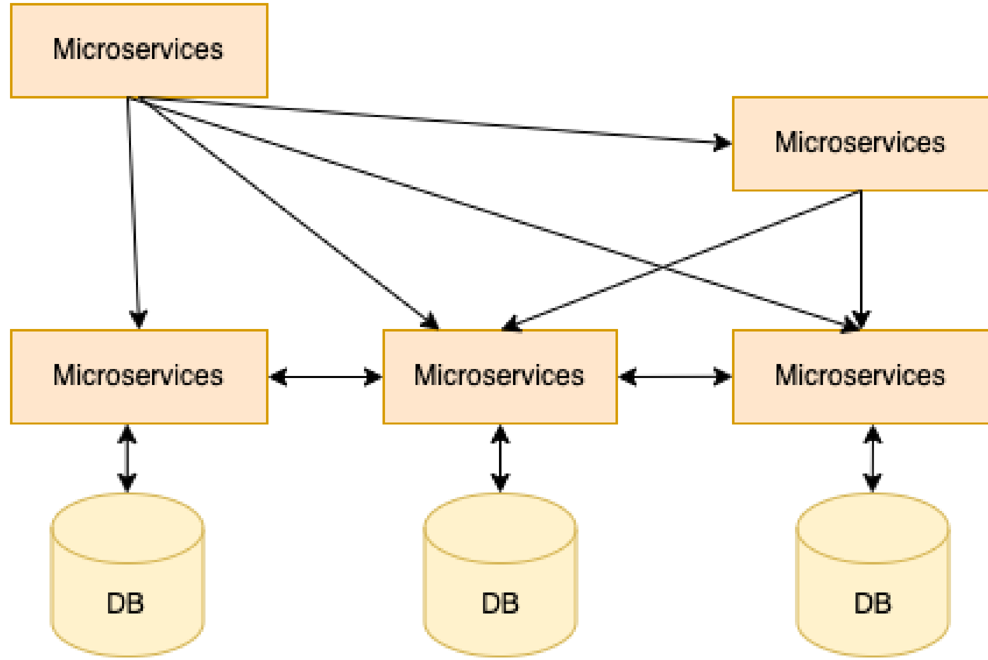

Figure 2. Representation of Microservices Architecture

Step V. Use of Aggregator Service

- If business use cases are independently dependent on more than 1 bounded context, use an aggregator service that will interact with multiple bounded contexts at the same time.
- Use with caution, as it can increase the number of service calls.

Step VI. Use of Anticorruption Layer

- Use an Anticorruption Layer if you need to interact with the older system.

VII. Use of API Gateway

- Use an API Gateway to make the migration to microservices transparent to clients.

5 CASE STUDY

This research presents a practical implementation and adoption of a domain driven Microservice architecture in a fintech application. Initially, the application operated on a monolithic architecture, where all components were integrated into a single-tiered system. However, as the system requirements evolved, the application's complexity and size grew, leading to challenges for developers in understanding and modifying the code effectively. During peak hours, system crashes occurred, making it difficult to trace exceptions in the code. To address these issues, a strategic decision was made to migrate certain parts of the application into microservices.

The case study focuses on an online financial application that offers loan services to verified customers. After completing the verification process successfully, customers become eligible to submit loan requests. The system evaluates the customer's credit score and assigns a specific limit based on the score and requested amount. Subsequently, customers can perform various transactions, including bill payments, fund transfers, and cash withdrawals.

5.1 Decomposition by Domain Driven Method

To facilitate the decomposition of the architecture, the approach of Domain-driven design (DDD) is utilized. DDD is a software development strategy that emphasizes modeling and designing software to closely align with a specific business domain [29], [29]. DDD aids in breaking down complex problems into smaller, manageable pieces. Within the context of application development, DDD refers to these problems as domains, which represent distinct business areas or processes within a company [16]. Each domain is further divided into Bounded Contexts, which align with individual microservices [17]. Organizations may have multiple domains, each with its own set of subdomains. Subdomains are groups of interconnected business rules and responsibilities. Bounded Contexts serve as boundaries around these groups of subdomains and provide a framework for defining microservices.

5.2 Domain Analysis

To begin, we conducted an analysis of the system's domain by identifying the business use cases that could be transformed into microservices. The application domain use cases are described in table 2. Once the functional requirements are defined, a Data Flow Diagram (DFD) is utilized to illustrate the business function of our monolithic application. Fig. 3 displays the DFD of the financial application, which consists of four components.

Process Notation. This symbolizes a process that transforms data flows. On a DFD, all processes must have inputs and outputs as they convert incoming data into outgoing data. The process is depicted by a circular symbol. Datastore Notation. This represents a collection of data items that are stored within the system. Data Flow Notation This indicates the direction of data flow and is represented by arrows, illustrating the movement of data. External Entity Notation. This demonstrates the data flow between external systems and identifies the source and destination systems. The external entity is represented by a rectangular symbol.

Table 1. Comparing Monolithic vs Microservices Architecture

Description	Monolithic	Microservices
Scalability	Difficult to upgrade	Easy to scale per service
Architecture	Single build of a unified code	Collection of small services
Programing language	Hard to change, take a lot of time to rewrite code	Language can be selected per service
Development	Depend on the team to perform the parallel operation in same application code	Team don't have to work parallel because each service can be delivered independently
Reliability	If service fails, the entire application goes down	Very reliable. If service fails, the application will not go down as a whole
Maintainability	Large code base intimidating to new developers	Small code base easier to manage
Agility	Not flexible and impossible to adopt new tech, language or frameworks	Integrate with new technologies to solve business purposes
Testing	End-to-end testing	Independent components need to be tested individually
Resiliency	One bug or issue can affect the whole system	A failure in one microservice does not affect other services
Price	Higher once the project scales	Higher at the first development stages

Table 2. Application domain use cases

No	Use Cases
1	For onboarding, customer submit kyc
2	Customer data verification by Agent
3	Agent approves customers if data verified, else reject
4	Loan request is submitted by customer
5	Load in repaied by customer
6	Customer performs bill payment
7	Funds transfer by customer
8	Money withdraws by customer from account
9	Money deposit in account by customer

5.3 Identification of Bounded Context

The term "bounded context" refers to a logical concept where specific terms, definitions, and rules are consistently applied within a software system [30]. By consulting domain experts, we identified three boundaries in our monolithic system: customer onboarding, loan, and transactions, as depicted in Fig. 4. The boundary line is represented by a solid red line.

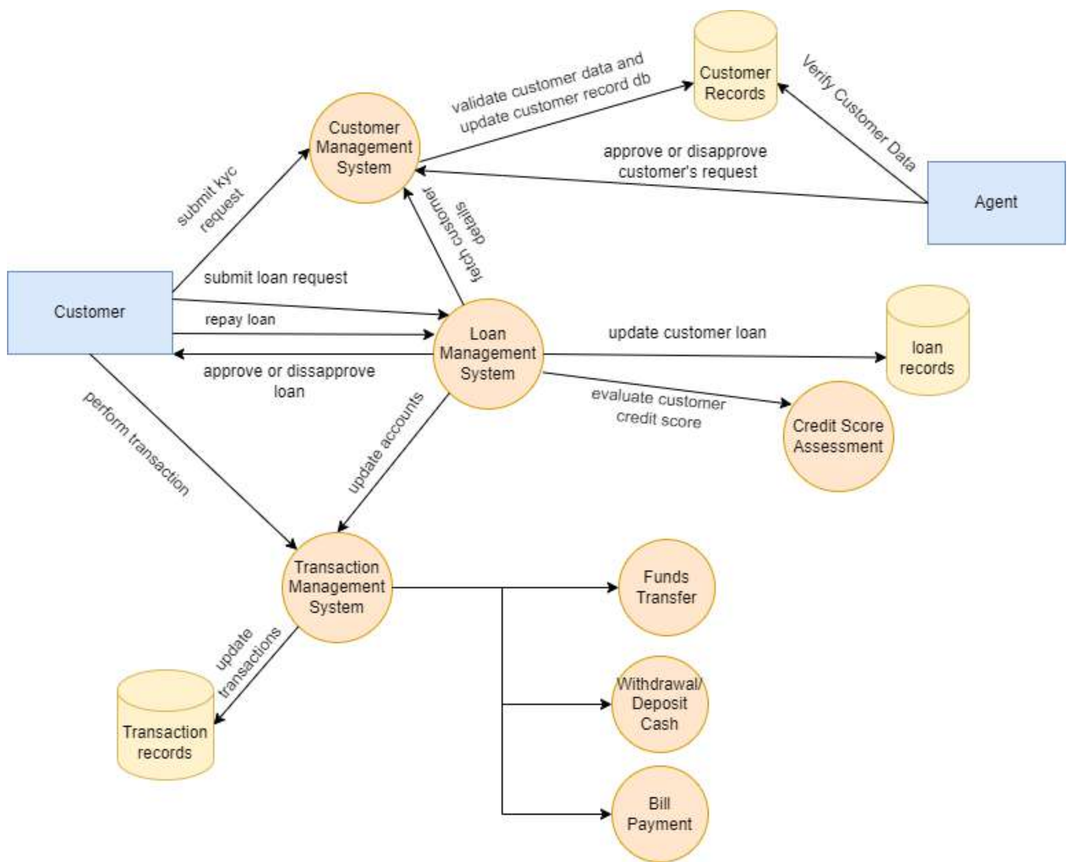

Figure 3. Data Flow Diagram of Financial Application

5.4 Define Aggregates, Entities, and Domain Service

Following the completion of the identification process, the next step involves categorizing aggregates, entities, and domain services within each bounded context.

5.4.1 Entities and Aggregates

Entities are objects with unique identities that persist over time. They may have attributes that can change, such as a person's address, while their identity remains constant. On the other hand, aggregates are groups of inter-connected objects treated as a single entity [30]. An aggregate defines a consistency boundary around one or more entities, with the root of the aggregate always being the same entity.

5.4.2 Domain Services

A domain service is described as "a standalone operation within the context of your domain, where a service object collects one or more services into an object." [29]. Domain services play a vital role in modeling behavior involving multiple entities. After identifying the bounded contexts, the next step is to identify entities, aggregates, and domain services within each context. Though detailed descriptions of entities and business functionalities cannot be disclosed for security reasons, here is a brief overview of how services were created.

- **Customers Onboard Bounded Context.** Within this context, entities like Customer and Agent represent onboard boundaries. The approval status of the customer is a child entity of the agent. A domain service

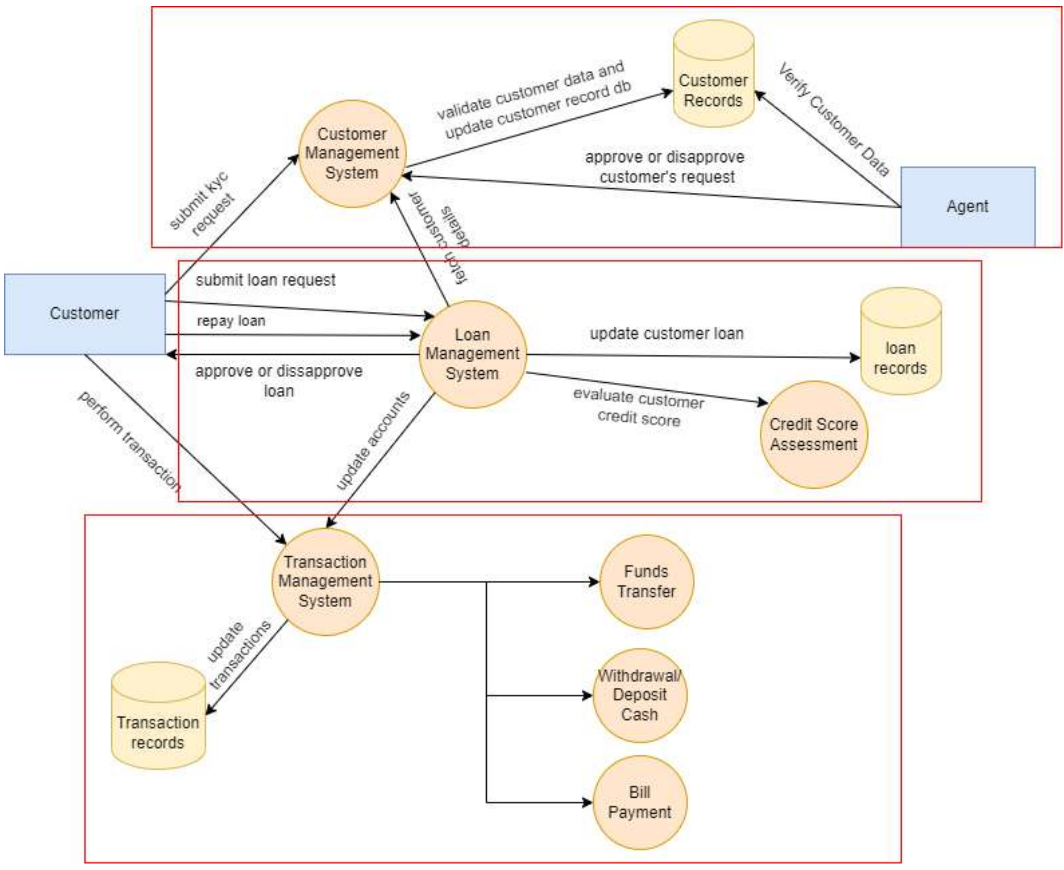

Figure 4. Identification of Bounded Context

named onboarding service is identified, and responsible for various business operations, including validating customer data and assigning requests to agents.

- **Loan Bounded Context.** In the loan-bounded context, customers can choose loan plans and submit loan requests. The loan management system evaluates the customer's credit score to determine loan approval or disbursement. Entities and aggregates in this context include loans, loan plans, loan repayments, and loan disbursement. A domain service called credit assessment evaluates a customer's credit score based on their history.
- **Transactional Bounded Context.** In this context, the system involves aggregates like loan repayment, funds transfer, bill payment, and cash withdrawal/deposit, each representing distinct and essential functions within the financial application. These aggregates are collections of related entities that ensure consistency and encapsulate business rules specific to their respective domains, thereby maintaining the integrity of business operations.

A central domain service named the transaction service handles routing requests to different processes, ensuring that each request is directed to the appropriate aggregate for processing. This service is crucial for managing the workflow and interactions between various aggregates. After each transaction, the transaction service updates records to reflect the changes, maintaining accurate and up-to-date information.

Figure 5 illustrates the identified sets of entities, aggregates, and domain services. This figure likely includes detailed diagrams or charts that map out how each component interacts within the system. These visual

Table 3. Identified microservices in bounded context

Onboard	Loans	Transactions
Customer Management Service	Loan Management Service	Transaction Management Service
Support Service	Loan Repayment Service	Bill Payment Service
Customer Onboard Service	Loan Disbursement Service	Funds Transfer Service
Credit Score	Credit Assessment Service	Cash Withdraw and Deposit Service

representations help in understanding the modular breakdown of the monolithic application into discrete, manageable services, highlighting the relationships and dependencies between different parts of the system.

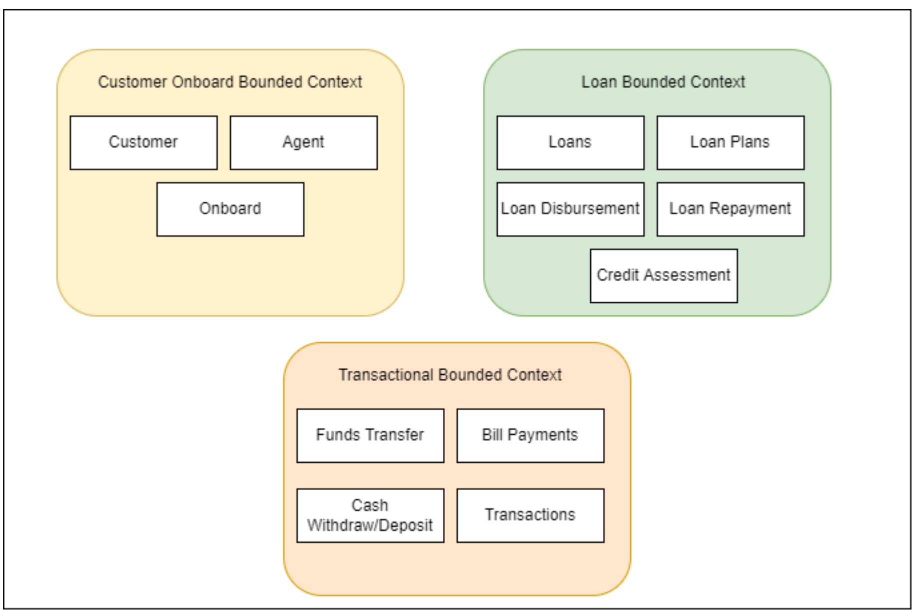

Figure 5. Entities, Aggregates, and Domain Services in Financial Application

5.5 Identification of Microservices

After conducting a thorough comparison of the system functionalities, the following systems have been recognized as microservices based on their specific contexts. To ensure that these domains are fully dedicated to individual teams, services have been consolidated into a new modified domain. The microservices identified within each bounded context are provided in table 7.

In Fig. 6, the rounded white rectangles in yellow, green, and orange colors depict the services within the bounded context. Additionally, white rounded rectangles represent additional services that are not part of any bounded context. A comprehensive description of these services is provided below. Aggregation Service. This newly created service receives customer requests, calls other services to gather their responses, and then consolidates the responses into a combined response, which is sent back to the customer. For example, when a customer wants to view their loan details and transactions, this service invokes both the loan and transaction services and returns the aggregated response to the customer. Gateway Routing. Requests are directed to multiple microservices through a single endpoint, eliminating the need for the client application to manage multiple

endpoints. This simplifies the client's interaction as it can continue sending requests to the gateway, and any modifications to internal services won't require client updates. Only the routing within the gateway needs to be adjusted. Anti-Corruption Layer. When migrating from legacy systems to microservices, there may be a need for the new services to interact with the legacy system resources. To facilitate this communication, an anti-corruption layer is introduced in Fig. 6. This layer acts as a translator, enabling seamless communication between the new services and the legacy system. Please refer to Fig. 6 for a visual representation of these services and their interactions within the system.

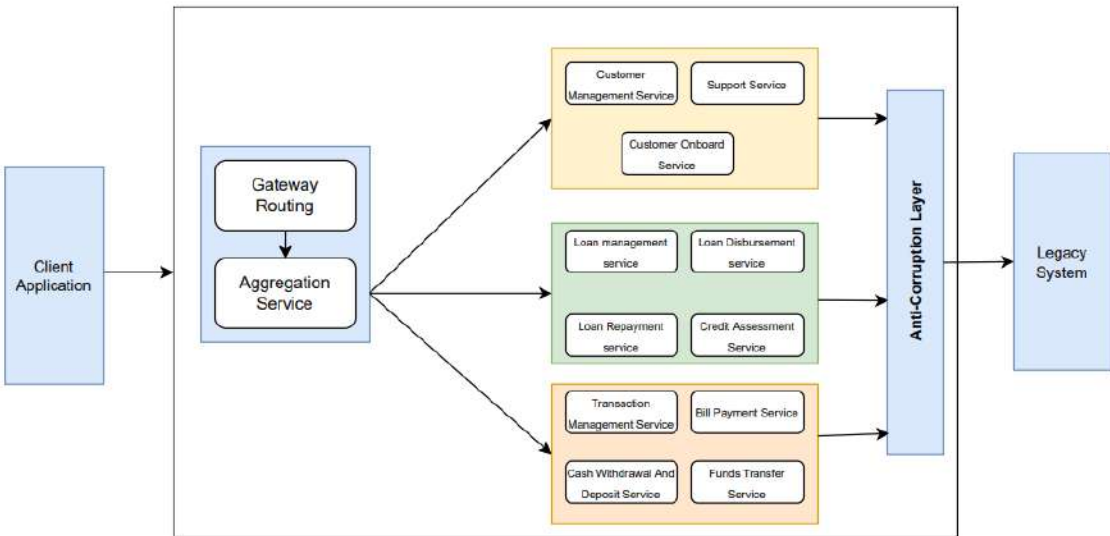

Figure 6. Migration Results of Monolithic to Microservice Architecture

5.6 Communication

Communication among microservices is a significant challenge due to their distributed nature and interactions on a network level. The extensive communication between services can lead to network overhead. To address this, specific communication protocols such as HTTP, gRPC (a modern framework based on Remote Procedure Call), or AMQP are utilized for service interaction.

For synchronized communication between microservices, HTTP and gRPC protocols are employed. HTTP calls between microservices are straightforward to implement and block the operation until a response is received or a timeout occurs. On the other hand, gRPC is a binary framing protocol designed for efficient data transport, distinguishing it from HTTP 1.1. Notably, gRPC exhibits exceptional performance, achieving speeds up to 8 times faster than JSON serialization while generating smaller messages by 60 to 80

Asynchronous communication between the loan management system and the transaction management system is established through an event bus. Whenever a loan settlement or disbursement occurs, the loan service initiates an event. The transaction service actively monitors these events and updates the transaction table accordingly. This decoupled approach enables efficient and real-time synchronization of data between the two systems.

6 Advantages of Our Migration Technique

Comparing our proposed migration technique with existing research, some advantages of our technique include:

1. Domain-driven design approach: Our technique incorporates domain-driven design principles, which involves working closely with domain experts to identify bounded contexts, entities, aggregates, and domain services. This ensures that the microservices are aligned with the business domain, leading to better maintainability, scalability, and extensibility of the resulting microservices architecture.

2. Reduction of complexity using combination functions: Our technique proposes the use of combination functions to merge nodes in a bipartite graph, resulting in optimized systems and microservices. This approach helps in reducing the complexity of the migration process by automating the identification and merging of related systems, processes, and tables, resulting in a more efficient and optimized microservices architecture.

3. Use of aggregator service and anticorruption layer: Our technique suggests the use of aggregator service and anticorruption layer to handle dependencies between bounded contexts and older systems. This helps in managing dependencies and isolating interactions between microservices, leading to better separation of concerns and maintainability.

4. Use of API gateway: Our technique recommends the use of an API gateway to make the migration to microservices transparent to clients. This helps in managing the communication between clients and microservices, providing a single entry point for API requests, and enabling features such as authentication, caching, and logging at the gateway level.

5. Comprehensive approach: Our technique covers various aspects of microservices migration, including identification of use cases, bounded contexts, entities, aggregates, and domain services, reduction of complexity using combination functions, handling dependencies between bounded contexts and older systems using aggregator service and anticorruption layer, and managing API communication using an API gateway. This comprehensive approach provides a well-structured and systematic method for migrating from monolithic applications to microservices architecture.

Overall, our proposed migration technique combines domain-driven design principles, automated complexity reduction using combination functions, and appropriate handling of dependencies and communication between microservices, making it a potentially advantageous approach for migrating applications from monolithic applications to microservices. However, it is important to thoroughly evaluate and validate the proposed technique in the context of specific applications and organizational requirements before implementation.

7 Summary and Findings

This research adopts a domain-driven design (DDD) approach to transform a real-time monolithic application into microservices. Given the complexity of modern business environments, DDD proves to be a valuable methodology for effectively addressing intricate business functions and establishing clear domain models that can be divided into smaller services or subsystems. The implementation of DDD has demonstrated improvements in team communication throughout the development cycle, as each team can focus on their specific domain, leading to well-defined individual roles.

However, one drawback of this approach is the need for collaboration between technical and domain experts to create an application model that effectively addresses domain problems. This collaboration may pose challenges, particularly for business domain experts dealing with applications of high technical complexity, leading to constraints that not all team members may be able to overcome. Nonetheless, DDD remains a powerful approach for successfully migrating from a monolithic to a microservices architecture, bringing benefits such as improved communication, modular design, and easier management of complex business processes.

One study [31] also reveals the complex factors driving decision-making in Microservices Architecture (MSA) adoption which emphasizes the roles of re-usability, scalability, extensibility, and maintainability, and highlight the monolithic approach as a critical strategy in transitioning from monolithic systems to MSA.

8 Conclusion

Microservices offer an effective solution for breaking down large applications into independent and self-contained services. This research paper introduces a technique for transforming monolithic application features into microservices, with a focus on a real-time application. The migration strategy primarily relies on domain-driven design principles, encompassing key steps like domain analysis using Data Flow Diagram, identification of bounded contexts, selection of aggregates, events, and domain services, as well as identification of microservices. The paper also addresses the communication approach among services to enhance application performance. However, it is essential to acknowledge that application decomposition is a time-consuming process that requires expert guidance. Additionally, this approach may not be suitable for applications with minimal complexities. Looking ahead, the research aims to extend the conversion of other application components in diverse domains, transitioning them from monolithic to microservices architecture. By leveraging the power of microservices, applications can achieve greater flexibility, scalability, and maintainability, enabling them to meet the dynamic demands of modern business environments.

Furthermore, the transition to microservices significantly enhances maintainability by isolating each service, allowing for easier updates and debugging without impacting the entire system. This isolation also bolsters security, as vulnerabilities in one service do not necessarily compromise others. However, potential performance bottlenecks can arise due to the overhead of inter-service communication, particularly if not well-optimized. Therefore, careful consideration and implementation of efficient communication protocols are crucial. These factors underscore the importance of a well-thought-out migration strategy to fully realize the benefits of microservices while mitigating associated challenges.

9 Future Work

In the future, this research can extend the application of the proposed microservices identification technique to other domains and types of applications beyond the financial sector. This includes exploring the decomposition of various complex monolithic systems into microservices and assessing the scalability and performance improvements in different industry contexts. Additionally, the research will investigate the integration of advanced tools and automation techniques to streamline the migration process, making it more efficient and accessible for organizations with varying levels of expertise. The goal is to refine the approach to ensure it is adaptable and effective across a wide range of application complexities.

Author Contributions

Momil Seedat: Conceptualization, Methodology, **Qaisar Abbas**:Software, Data curation **Nadeem Ahmad**: Writing- Original draft preparation. **Irum Feroz**: Supervision, Validation..**Affefah Qureshi**: Visualization. **Alessia Amelio**: Writing- Reviewing and Editing

Compliance with Ethical Standards

It is declared that all authors don't have any conflict of interest. Furthermore, informed consent was obtained from all individual participants included in the study.

References

[1] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: yesterday, today, and tomorrow," *Present and ulterior software engineering*, pp. 195–216, 2017.

[2] F. A. Hollanda, R. P. de Oliveira, F. F. F. da Silva, M. A. von Krüger, and W. C. de Albuquerque Pereira, "Platform for automated acquisition of ultrasonic signals in acoustic tank for tissue characterization," in *XXVI Brazilian Congress on Biomedical Engineering: CBEB 2018, Armação de Buzios, RJ, Brazil, 21-25 October 2018 (Vol. 1)*, pp. 525–529, Springer, 2019.

[3] J. Fritzsch, J. Bogner, A. Zimmermann, and S. Wagner, "From monolith to microservices: A classification of refactoring approaches," in *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment: First International Workshop, DEVOPS 2018, Chateau de Villebrumier, France, March 5-6, 2018, Revised Selected Papers 1*, pp. 128–141, Springer, 2019.

[4] M. Kalske, N. Mäkitalo, and T. Mikkonen, "Challenges when moving from monolith to microservice architecture," in *Current Trends in Web Engineering: ICWE 2017 International Workshops, Liquid Multi-Device Software and EnWoT, practi-O-web, NLPIT, SoWeMine, Rome, Italy, June 5-8, 2017, Revised Selected Papers 17*, pp. 32–47, Springer, 2018.

[5] F. Ponce, G. Márquez, and H. Astudillo, "Migrating from monolithic architecture to microservices: A rapid review," in *2019 38th International Conference of the Chilean Computer Science Society (SCCC)*, pp. 1–7, IEEE, 2019.

[6] K. Gos and W. Zabierowski, "The comparison of microservice and monolithic architecture," in *2020 IEEE XVIth International Conference on the Perspective Technologies and Methods in MEMS Design (MEMSTECH)*, pp. 150–153, IEEE, 2020.

[7] N. Gonçalves, D. Faustino, A. R. Silva, and M. Portela, "Monolith modularization towards microservices: Refactoring and performance trade-offs," in *2021 IEEE 18th International Conference on Software Architecture Companion (ICSA-C)*, pp. 1–8, IEEE, 2021.

[8] A. Rud, "Why and how netflix, amazon, and uber migrated to microservices: Learn from their experience," *URL: https://www. hys-enterprise. com/blog/whyand-how-netflix-amazon-and-uber-migrated-to-microservices-learnfrom-their-experience*, 2019.

[9] E. Axelsson and E. Karlkvist, "Extracting microservices from a monolithic application," 2019.

[10] V. Velepucha and P. Flores, "Monoliths to microservices-migration problems and challenges: A sms," in *2021 Second International Conference on Information Systems and Software Technologies (ICI2ST)*, pp. 135–142, IEEE, 2021.

[11] K. Gos and W. Zabierowski, "The comparison of microservice and monolithic architecture," in *2020 IEEE XVIth International Conference on the Perspective Technologies and Methods in MEMS Design (MEMSTECH)*, pp. 150–153, IEEE, 2020.

[12] B. Hippchen, P. Giessler, R. Steinegger, M. Schneider, and S. Abeck, "Designing microservice-based applications by using a domain-driven design approach," *International Journal on Advances in Software*, vol. 10, no. 3&4, pp. 432–445, 2017.

[13] M. I. A. Durrani, N. S. Qureshi, N. Ahmad, T. Naz, and A. Amelio, "A health informatics reporting system for technology illiterate workforce using mobile phone," *Applied clinical informatics*, vol. 10, no. 02, pp. 348–357, 2019.

[14] I. Feroz and N. Ahmad, "Usability based rating scale (ubrs) for evaluation of mobile health (mhealth) applications. hci and beyond: advances towards smart and interconnected environments. vol. 2," 2022.

[15] S. Newman, *Building microservices*. " O'Reilly Media, Inc.", 2021.

[16] M. Languric and L. Zaki, "Migrating monolithic system to domain-driven microservices: Developing a generalized migration strategy for an architecture built on microservices," 2022.

[17] M. Boyle, *Domain-Driven Design with Golang: Use Golang to create simple, maintainable systems to solve complex business problems*. Packt Publishing Ltd, 2022.

[18] R. J. Petrasch and R. R. Petrasch, "Data integration and interoperability: Towards a model-driven and pattern-oriented approach," *Modelling*, vol. 3, no. 1, pp. 105–126, 2022.

[19] N. Ahmad, *People centered HMI's for deaf and functionally illiterate users*. PhD thesis, Universität Potsdam, 2014.

[20] N. Ahmad, I. Feroz, and A. Anjum, "Usability analysis of educational information systems from student's perspective," in *Proceedings of the 2020 International Conference on Big Data in Management*, pp. 130–135, 2020.

[21] D. Kuryazov, D. Jabborov, and B. Khujamuratov, "Towards decomposing monolithic applications into microservices," in *2020 IEEE 14th International Conference on Application of Information and Communication Technologies (AICT)*, pp. 1–4, IEEE, 2020.

[22] S. G. Haugeland, P. H. Nguyen, H. Song, and F. Chauvel, "Migrating monoliths to microservices-based customizable multi-tenant cloud-native apps," in *2021 47th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pp. 170–177, IEEE, 2021.

[23] N. Santos and A. R. Silva, "A complexity metric for microservices architecture migration," in *2020 IEEE international conference on software architecture (ICSA)*, pp. 169–178, IEEE, 2020.

[24] G. Blinowski, A. Ojdowska, and A. Przybyłek, "Monolithic vs. microservice architecture: A performance and scalability evaluation," *IEEE Access*, vol. 10, pp. 20357–20374, 2022.

[25] C. Zhong, S. Li, H. Huang, X. Liu, Z. Chen, Y. Zhang, and H. Zhang, "Domain-driven design for microservices: An evidence-based investigation," *IEEE Transactions on Software Engineering*, 2024.

[26] C. Richardson, "Pattern: microservice architecture," *URL: http://microservices. io/patterns/microservices. html*, 2017.

[27] O. Al-Debagy and P. Martinek, "A comparative review of microservices and monolithic architectures," in *2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI)*, pp. 000149–000154, IEEE, 2018.

[28] A. Levcovitz, R. Terra, and M. T. Valente, "Towards a technique for extracting microservices from monolithic enterprise systems," *arXiv preprint arXiv:1605.03175*, 2016.

[29] E. Evans, *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004.

[30] H. Vural and M. Koyuncu, "Does domain-driven design lead to finding the optimal modularity of a microservice?," *IEEE Access*, vol. 9, pp. 32721–32733, 2021.

[31] M. AIT SAID, A. EZZATI, S. MIHI, and L. BELOUADDANE, "Microservices adoption: An industrial inquiry into factors influencing decisions and implementation strategies," *International Journal of Computing and Digital Systems*, vol. 15, no. 1, pp. 1417–1432, 2024.