Foreign Key Constraints to Maintain Referential Integrity in Distributed Database in Microservices Architecture

Shamsa Kanwal[1], Nauman Riaz Chaudhry[2], Reema Choudhary[3],
Younus Ahamad Shaik[4], Pankaj Yadav[5], Ayesha Rashid[6]
Department of Computer Science, University of Gujrat, Gujrat, Pakistan[1,2,6]
Department of Software Engineering, University of Gujrat, Gujrat, Pakistan[3]
Jawaharlal Nehru Technology University, Anantapur, India[4]
Data Center and Artificial Intelligence Group, Intel Technology India Pvt Ltd, Bangalore, India[5]

***Abstract*—In the world of modern software development, microservices architecture has become increasingly popular due to its ability to help developers to build large and complex applications that are more agile, faster and more scalable. In large scale applications (such as e-commerce, healthcare, finance, social media, inventory management, travel booking, content management, and customer relationship management systems etc.) with many interconnected services, it is tough to keep the data accurate and consistent. The concept of referential integrity is applied to validate the data. Referential integrity refers to the preservation of relationships between tables. In a monolithic architecture, where the application and database are closely linked and co-located on the same server, referential integrity via foreign key constraints makes it feasible to preserve consistent and accurate data. But in the microservices architecture, maintaining referential integrity across distributed databases poses significant challenges due to its decentralized nature of data management. This study utilizes a hybrid research methodology, combining empirical research and design science research to discover and address the challenges of maintaining referential integrity in distributed databases in microservice architecture and calculate the response time by comparing and analyzing with existing models. The results of the evolution in term of response time are presented in this work.**

Keywords*—Foreign key constraints; relational mapping; referential integrity; saga pattern; event driven architecture; APIs; microservice; distributed database

I. Introduction

In modern cloud-native applications, distributed databases are an important part of microservices architecture, as they offer availability, reliability and scalability [1]. In large application with many interconnected links, it's hard to keep the data correct and consistent. For that the concept of "Referential integrity Constraints" is used to keep the data correct.

In a monolithic architecture, where the application and database are closely linked and co-located on the same server, referential integrity via foreign key constraints makes it feasible to preserve consistent and accurate data [2]. But in the microservices architecture, maintaining referential integrity across distributed databases poses significant challenges due to its decentralized nature of data management.

A. Database in Monolithic Architecture

Monolithic architecture is an outdated software design method where whole application or system is made as a single, interconnected unit [3]. In this architectural style, all the components, modules, and functionalities of the application are tightly integrated and run as a single codebase within a single process [4]. In monolithic architecture, CRUD operations directly perform in centralized relational database [5]. When new data is Create into a table that contains a foreign key column, the referenced value should first be checked to ensure that it exists in the referenced table's primary key. If not, the data is not allowed. For Read operation, foreign keys don't affect the process directly, but they help to make sure the data is correct. When deleting data from a parent table that is referenced by a foreign key constraint in another table, database checks if related child record exist. It can either delete the child records or stop the deletion too, depending on the rules. When updating data in the parent table, database updates the related records according to rules or stop the change to avoid breaking the link.

A monolithic system is hard to grow and update quickly because all components and modules are tightly connected. This approach contrasts with modern microservice architecture for handling large applications which offer better availability, well organized use of resources and scalability, making them a better option.

B. Distributed Databases in Microservices Architecture

Large-scale applications can be developed by dividing them into smaller, autonomous services thanks to microservices architecture [6]. These services have their own unique business logic and database. Each service is updated, tested, deployed, and scaled independently of the others [7]. This method frequently makes use of databases with different schemas, shared database, and distributed databases, each of which fulfills a distinct purpose. Shared database and databases per service are the two primary categories of distributed database. When it comes to databases for individual services, every microservice has a unique database [8].

1) Shared databases in microservices architecture: In a shared database approach, all services use a single database.

This means that multiple services can write data to the same database, and each service has access to data written by the other services. Services can conveniently access and share data since they all contribute to the same database [9]. With all services operating within a single database for reading and writing, data sharing is simplified. This eliminates the need for complex data synchronization or replication procedures, as all services have direct access to the same dataset.

In a shared database model, multiple microservices access and function on the same database. Although this simplifies data consistency and relational integrity implementation, it basically violates key microservices principles such as service autonomy, scalability, and independent deployment.

Service autonomy is negotiated because microservices become tightly coupled through the shared schema, meaning that changes in one service's database tables can accidentally impact others. This dependency increases the risk of breaking changes and requires coordinated deployments, reducing the flexibility and agility that microservices are meant to provide. Scalability is also stuck, as the shared database can become a bottleneck; scaling services independently is more challenging when all trust on the same data store. Moreover, operational complexity grows when services must negotiate over shared resources, and fault isolation becomes difficult. If one service causes a database failure or locks essential tables, it can affect all other services connected to that database. These limitations run contrary to the core goals of microservices architecture, which emphasize loose coupling, independent scaling, resilience, and continuous delivery. Therefore, while the shared database approach may offer short-term gains in performance and development simplicity, it imposes long-term constraints on maintainability, scalability, and system robustness.

2) Databases per service in microservices architecture: In the database-per-service pattern within a distributed database architecture, each microservice has its own dedicated databases. These services manage their own data independently [1] without interference from other service. Each services can deploy, scale and update independently. Both synchronous and asynchronous [10] method used for communication between the services. Synchronous communication means that a service sending a request to another service and pausing its operation until response is received. Typically, this method is realized through protocols like HTTP/REST or gRPC [11]. While asynchronous communication occurs when a service sends a request or message to another service and continues its operation without waiting for an immediate response. This method is often facilitated by message brokers like RabbitMQ or Kafka. This procedure make it simple for various microservices to exchange and access data across many platforms.

In microservices architecture with distributed databases, maintaining data consistency particularly using foreign key constraints for referential integrity is a major challenge. Unlike traditional systems where databases impose these rules directly, microservices have separate databases, making this execution tough. The CAP theorem describes that distributed systems can only attain two out of three: Consistency, Availability, and Partition Tolerance. Meanwhile microservices must handle network issues (partition tolerance), they often choose between strong consistency or high availability.

Some researchers propose the BAC model (Backup, Availability, Consistency) [12] to better handle real-world failures. Systems that focus on consistency (CP) ensure correct data but may be slower or unavailable during issues, while systems that focus on availability (AP) respond faster but rely on eventual consistency, requiring extra logic to maintain data integrity. In such cases, referential integrity is often managed through custom validation, events, or shared read-only views instead of traditional foreign key constraints.

C. Background

The term "Referential integrity" and "Foreign key constraints" are the two central concepts for ensuring data consistency, accuracy and validity across related tables in relational database system. A relational database is a key element for storing, managing, and retrieving data for the entire program in a monolithic architecture. The architecture uses structured tables to store data [12] from multiple application modules in a single, centralized relational database. A centralized data schema defines the organization of the database by specifying tables, relationships, columns, data types, and constraints to ensure consistent data storage.

In relational database each database table has a PRIMARY KEY that uniquely identifies its records. A FOREIGN KEY serves as a database key designed, which consists of one or more columns in a table, references the primary key in another table, thereby creating relationships between the tables. The term "FOREIGN KEY CONSTRAINTS" is a column (or combination of columns) in a table whose values must match values of a column in some other table. This [12] foreign key constraint connects the child table's foreign key to the parent table's primary key, maintaining referential integrity.

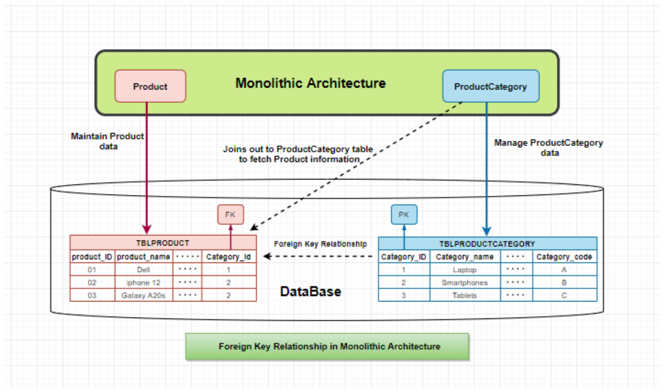

Monolithic Architecture
Product
ProductCategory
Maintain Product data
Joins out to ProductCategory table to fetch Product information
Manage ProductCategory data
FK
PK
TBLPRODUCT
product_ID | product_name | ····· | Category_Id
01 | Dell | ···· | 1
02 | iphone 12 | ···· | 2
03 | Galaxy A20s | ···· | 2
Foreign Key Relationship
TBLPRODUCTCATEGORY
Category_ID | Category_name | ···· | Category_code
1 | Laptop | ···· | A
2 | Smartphones | ···· | B
3 | Tablets | ···· | C
DataBase
Foreign Key Relationship in Monolithic Architecture

Fig. 1. Foreign key relationships in monolithic architecture.

Fig. 1 illustrates how foreign key constraints are openly imposed within a monolithic architecture, where all modules share a single centralized database. In this setup, relational integrity between entities (e.g., Product and Product-Category) is maintained through native database-level foreign keys.Distributed database is a type of database system where data is stored through multiple physical locations, frequently on different servers, regions, or even data centers, but appears to users as a single, unified database. Each location may manage part of the data independently, but the system ensures coordination so users can access and manipulate data as if it were stored in one place.

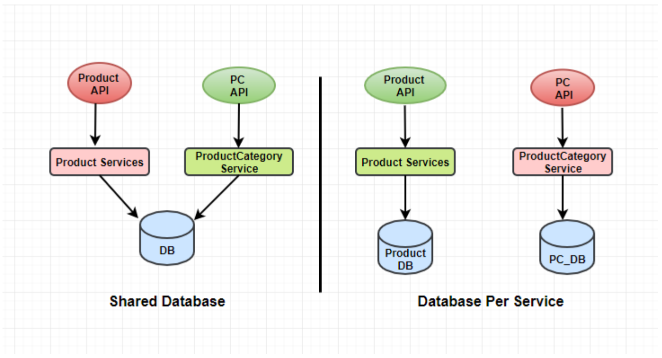
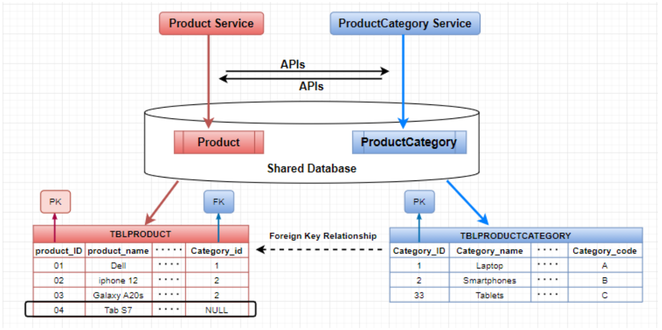


Fig. 2. Shared database and databases per service patterns.

Fig. 2 shows the difference between Shared Database and Database per Service. Shared Database use single database but doesn't scale well, while Database per Service gives each service its own data and needs extra steps to keep data linked.

Microservice architecture is a software development strategy [24] that differs from monolithic architectures. Rather than having one large application, it involves breaking down an application into several small, independent services that are not tightly integrated with one another. One of the key feature of this architecture is the use of multiple, independent services that communicate with each other over the network. Each service is created to perform a specific business task and interacts with other services through clearly defined APIs.



TBLPRODUCT

product_ID	product_name	·····	Category_id
01	Dell	····	1
02	iphone 12	····	2
03	Galaxy A20s	····	2
04	Tab S7	····	NULL

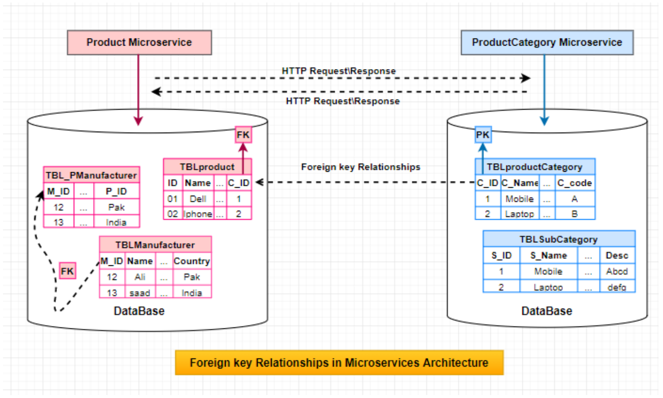
TBLPRODUCTCATEGORY

Category_ID	Category_name	····	Category_code
1	Laptop	····	A
2	Smartphones	····	B
33	Tablets	····	C

Fig. 3. Foreign key relationships in shared database using multiple microservices.

In Fig. 3 multiple services, such as ProductService and ProductCategoryService, interact with a single, unified database. ProductCategory service manage product category in the ProductCategory table, while ProductService manage individual products in the Product table. Both tables are linked through a foreign key relationship, ProductCategory table as the parent and the Product table as the child, ensuring referential integrity [12] and consistent data relationships across services. On the other hand, microservices also interact with each other via events. For instance, ProductService can produce the event exposed by ProductCategoryService to fetch category details.

For example in Fig. 4, consider two microservices: ProductService and ProductCategoryService. Each of these microservices manages its own separate database, but they are linked through a foreign key relationship to maintain referential integrity.


Product Microservice
ProductCategory Microservice
HTTP Request\Response
HTTP Request\Response
FK
Foreign key Relationships
PK
FK
DataBase
DataBase
Foreign key Relationships in Microservices Architecture

TBL_PManufacturer

M_ID	...	P_ID
12	...	Pak
13	...	India

TBLproduct

ID	Name	...	C_ID
01	Dell	...	1
02	Iphone	...	2

TBLManufacturer

M_ID	Name	...	Country
12	Ali	...	Pak
13	saad	...	India

TBLproductCategory

C_ID	C_Name	...	C_code
1	Mobile	...	A
2	Laptop	...	B

TBLSubCategory

S_ID	S_Name	...	Desc
1	Mobile	...	Abcd
2	Laptop	...	defg

Fig. 4. Database-per-service in microservices architecture.

Synchronous communication is ideal for real-time interactions requiring immediate responses, offering simplicity and directness. In contrast, asynchronous communication excels in scalability and resilience, making it suitable for tasks that can be handled independently over time. In large application with many interconnected links, it's hard to keep the data correct and consistence. For that the concept of "Referential integrity Constraints" is used to keep the data correct. Still, the complexity of handling inter-service communication and ensuring data consistency across multiple services increases.

D. Purpose of the Research

The purpose of this research is to discover how referential integrity can be sustained in distributed databases within microservices architecture, where traditional foreign key constraints cannot span across service boundaries. Using a case study with separate Product and ProductCategory services, the study evaluates and compares practical methods such as shared databases, API validation, event-driven communication, and saga pattern to preserve data relationships while balancing scalability, fault tolerance, and system complexity.

E. Research Objectives

- To handle foreign key constraints in distributed database in microservices architecture.
- To improve response time during CRUD operation in distributed database in microservices architecture.
- To analyze and compare the existing model in a distributed database in microservices architecture.

In the introduction section the paper addresses the challenges of maintaining referential integrity in distributed databases within microservices architecture, this research explores the application of foreign key constraints across different architectural patterns. It investigates how referential integrity is preserved in both shared databases and database-per-service designs using three distinct implementation methods: Shared Database Pattern using APIs, Saga Choreography Pattern with database-per-service, and an Event-Driven Architecture with database-per-service. Each method offers different trade-offs in terms of consistency, performance, and decoupling, which are critical factors in distributed environments.

The remainder of this paper is structured as follows: Section II presents a comprehensive literature review covering the evolution from monolithic systems to microservices, and analyzes the implications for database design and schema evolution. Section III outlines the research methodology, detailing the experimental setup for each of the three architectural patterns. Section IV presents the results, including a comparative analysis of referential integrity, response time, and behavior under CRUD operations. Finally, Section V concludes the findings and provides recommendations for selecting appropriate patterns to maintain referential integrity based on system requirements.

II. Literature Review

This chapter gives us a closer look at what's happening in research right now and what we already know about the topic. It shows us what different authors have to say about the subject based on what's already been written.

A. Evolution to Microservices and Distributed Databases

Microservice-based architecture, which is used for developing software systems with independent maintainability and scalability [13]. Many companies migrate their existing monolithic software systems towards microservice-based architectures [14]. The paper [15] proposes a data-centric process to identify microservices by decomposing the database (on the bases of foreign key relationship) into clusters and identifying topics that correspond to potential microservices. This process performs database schema analysis and clustering.

Migrating from a monolithic to microservice architecture with a shared database and then to multiple databases based on the database-per-microservice pattern. The migration process involves identifying microservice boundaries, decomposing the application into multiple entities. Framework called Architect, which is based on a domain-specific language for building dependable distributed systems [16], to ensure the data consistency of distributed transactions using the Saga pattern to reduce complexity and attain eventual consistency.

The saga pattern is a technique used to ensure data consistency across multiple microservices [17]. It uses sequential transactions that trigger subsequent ones and compensatory transaction when failures occur. However, it lacks isolation, allowing access to ongoing transactions. An improved version[18] of the saga pattern addresses this issue by using a quota cache and commit-sync service to rollback modifications in case of failures without affecting the database layer. The database is committed once all transactions are successful.

Event-driven architecture used for building a distributed database for microservices that is scale and flexible. In EDA, the communication between different components of the system happen through events. EDA can be used to ensure that change to the data are propagated to all the microservices that depend on it. By decoupling the components of the system through events which ensure that changes are propagated quickly [19] and efficiently, while also maintaining data consistency across the system . Identified multiple instances were enforcing foreign key constraints between microservices is essential to ensure accuracy.

Architectural trade-off analysis based on patterns to assist software architects in selecting appropriate pattern for foreign key relationship between different microservices using immutable event by conducted a study [20] to identify architectural patterns of microservices that affect structural design decision related to the size of services, database sharing, and level of service coupling. Software architects can better understand and choose the most appropriate pattern for foreign key relationship between different microservices by using Command Query Responsibility Segregation(CQRS), which can guide the architecture in the desired direction.

Adoption of microservice-based event-driven architecture for data-intensive systems, also known as big data system(BDS) in which microservices communicate through APIs. The authors conducted action research [21] which are Questionnaire to investigate the reasons for the adoption of this architecture and to document the challenges and lessons learned. The resulting architecture was found to enable easier maintenance and fault isolation.

Recovering microservices that use polyglot persistence, which means they use multiple data storage techniques. When backups are taken independently, it becomes difficult to recover them in a consistent state, and references across microservice boundaries may break after disaster recovery. The authors [22] propose a weak global consistency definition for microservice architectures and a recovery protocol that utilizes cached referenced data to reduce the time interval after the most recent backup, during which state changes may have been lost.

The widespread adoption [23] of microservices and event-driven software applications, which favor the separation of databases into independent silos of data owned by a single service. It mention that microservices communicate via synchronous REST API calls and introduces the concept of virtual actors, The only way for a microservice to get data from another microservice is to make a call on an endpoint.

In study [27] Paper offers a modest way to design and evaluate microservices that focus on data, keeping in mind the limitations of the CAP theorem, which applies to distributed systems like microservices. It helps developers and architects compare different design and database patterns to find the best mix of cost and performance. The CAP theorem is not used as a strict rule but as a monitor to group systems as CA (Consistency–Availability), CP (Consistency–Partition tolerance), or AP (Availability–Partition tolerance). The approach is explained with three real-world examples, each showing how different choices affect system design, making it easier to make smart architecture and database decisions. Comparison table is shown in Table I.

B. Database Schema Analysis

A dominant model for cloud based application, consist of integrated, independently deployed and occasionally distributed services that communicate with each other through API calls and maintain their own databases. While enabling software development is a key motivation for implementing microservices, latest surveys[25] highlight database schema evolution as an important challenge. The main aim of this thesis is to reduce the burden which evolve in cloud based application and provide developers automated database schema

TABLE I. COMPARISON TABLE

Paper Names and References	Isolation	Scalability	Consistency
Towards Migrating Legacy Software Systems to Microservice [15].	Yes	Yes	Yes
A Framework for the Migration to Microservices [16].		Yes	Yes
Enhancing Saga Pattern for Distributed Transactions within a Microservices [18]	Yes	Yes	Yes
A distributed database system for event-based microservices [19]	Yes	Yes	Yes
Data Management in Microservices	Yes	Yes	
A Method for Architectural Trade-off Analysis Based on Patterns [20].		Yes	
From Monolithic Big Data System to a Microservices Event-Drive [21].	Yes	Yes	Yes
Microservice Disaster Crash Recovery.		Yes	Yes
Operational Stream Processing [23].		Yes	Yes
Performance Engineering for Microservices: Challenges & Directions [24].		Yes	Yes

evolution with tools. This research[26] mainly focused on modeling, generation, evolution, visualization and monitoring of database schema in microservices.

Microservice-based architecture, which is used for developing software systems with independent maintainability and scalability. Many companies migrate their existing monolithic software systems towards microservice-based architectures. The paper [4] proposes a data-centric process to identify microservices by decomposing the database (on the bases of foreign key relationship) into clusters and identifying topics that correspond to potential microservices. This process performs database schema analysis and clustering.

The abstraction of software components from legacy system, is an active research area within microservices architecture. Domain driven business dataflow diagram have been proposed using different approaches such as static and dynamic code analysis. Legacy systems, which are typically monolithic in nature, usually come with inherited database schemas and data, resulting limited code opportunities for code reused. To identify the database object for microservices extraction, analyze these artifacts, types and usage are aim of this research. A systematic mapping study [27] in legacy system discovered using database artifacts to identify services or component, the result concentrate on using database schema, data state and dependencies to identify separate business unit that could be transformed to potential microservices.

Event-driven architecture(EDA) used for building a distributed database for microservices that is scale and flexible [28]. In EDA, the communication between different components of the system happens through events, which are messages that are generated when certain actions or changes occur. EDA can be used to ensure that change to the data are propagated to all the microservices that depend on it. By decoupling the components of the system [19]through events which ensure that changes are propagated quickly and efficiently, while also maintaining data consistency across the system.

A microservice architecture built an application which are independently deployable, loosely coupled services focused on business capabilities, need careful data storage consideration [29]. Evaluating RDBMS and document store data stores through prototypes and performance test using simplified Electronic Medical Record (EMR) studies [30] five different data storage pattern for microservices in this research such as document store microservice overtake RDBMS, shared database overtake databases per services, CQRS overtake shared database all pattern have minor resource impact. Regardless have conflicts with modularity, the CQRS with event sourcing pattern could alleviate issues. In future may use hybrid pattern to discover data related operation in shared database and databases per services.

When migrating from monolithic application to microservices architecture various challenges occur, mostly with splitting and sharing databases. In [31] this article show framework where managing database splitting among different microservices to achieve data consistency and independent deployment while addressing latency problems. The offered solution improve performance, improved response time and handle messaging in evaluation. This framework maintains constructing services with better database independence.

Demand for data driven features has led to complicated networks of services which communicate each other without integrated architecture in Modern web applications. For large scale web services, synapse helps easy to used strong semantic system [32], allowing isolated, scalable and independent services to share data across different databases. In MVC-based applications that force high level data models using scalable replication mechanism that synchronizes read only view of shared data in real time, enabling data replication using different databases such as MYSQL, MongoDB, Oracle and PostgreSQL. Data duplication among different databases SQL and NoSQL supports in this synapse. For data integration use different models and ORMs to guarantee compatibility between SQL and NoSQL, without compromising scalability to maintain consistency. Synapse determine good performance and scalability. Table II shows database schema analysis in microservices.

III. RESEARCH METHODOLOGY

This section describes the methodology we followed when lot of literature review has been discussed to support the hypothesis of this research study and gap in the literature is also identified. The primary goal to study literature review was to collect experiences on distributed databases in microservice and how foreign key constraints work in microservices architecture, reporting best practices, and identify the problems occur during development processes. This study utilizes a hybrid research methodology, combining empirical research and design science research to discover and address the challenges of maintaining referential integrity in distributed databases in microservice architecture. This dual approach allows us to first observe and analyze real-world systems (empirical) and then propose and evaluate a novel design framework (design

TABLE II. Database Schema Analysis in Microservices

Research Papers Name	Distributed Database
Automated Database Schema Evolution in Microservices [26]	Database-Per-Service
Towards Migrating Legacy Software Systems to Microservice-based Architectures [4]	Database-Per-Service
Using Database Schemas of Legacy Applications for Microservices Identification: A Mapping Study [27]	Database-Per-Service
A Distributed Database System for Event-based Microservices [19]	Database-Per-Service
Data Management in Microservices: State of the Practice, Challenges, and Research Directions [7]	Database-Per-Service & shared database
Evaluation of Data Storage Patterns in Microservices Architecture [30]	Database-Per-Service & shared database
Framework for Interaction Between Databases and Microservice Architecture [31]	Database-Per-Service & shared database
Synapse: A Microservices Architecture for Heterogeneous-Database Web Applications [32]	Database-Per-Service

science), ensuring both theoretical consistency and practical relevance.

A. Proposed Framework

Subsequent steps have been followed for the proposed Framework in Fig. 5.

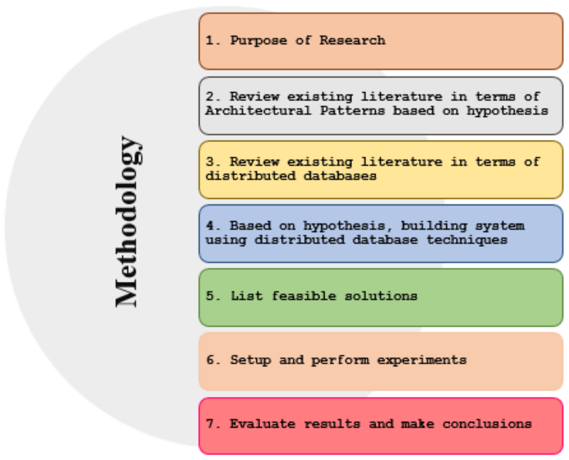

Methodology
1. Purpose of Research
2. Review existing literature in terms of Architectural Patterns based on hypothesis
3. Review existing literature in terms of distributed databases
4. Based on hypothesis, building system using distributed database techniques
5. List feasible solutions
6. Setup and perform experiments
7. Evaluate results and make conclusions

Fig. 5. Proposed framework.

1) Feasible solutions: This research is conducted using hybrid research methodology, combining empirical research and design science research supports the development of robust methods and techniques such as event-driven architecture, the saga choreography pattern, and APIs using a shared database technique which is based on hypothesis and evidences collected from the literature that address the challenges of enforcing foreign key constraints in distributed databases within microservices architecture.

B. Requirements for the Framework Implementation

This section provides an overview on which methods and tools were used.

1) Software requirements: Since this research is focused on foreign key constraints in distributed databases in microservices architecture the software used for the development of proposed framework is Spring tool Suite (STS) is an Eclipse-based development environment design for developing spring based application. This tool used for creating and managing spring projects, validation for spring configuration files and graphical boot dashboard to manage spring boot application. For validation Postman is an excellent tool that API developers and tester can use for guaranteeing API performance and functionality. In order to test RESTful APIs, create and run HTTP requests such as PUT, DELETE, GET, POST etc. In addition MYSQL is a common Relational database management system that helps for quick read/write operations and Xampp is the most commonly used free and open-source web server solution packages that work across multiple platforms. It is suitable for developers, when establishing a local web server environment for PHP development.

2) System requirements: Windows operating system is required for experimental purpose.

C. Implementation of Framework

After analysis the literature review we discuss some method which are used in this research.

1) Shared database pattern using APIs: In microservices design, a shared database is a designed pattern in which multiple microservice share the same central database, as opposed to databases per service pattern. All services interacts with a common set of tables or schema, usually within single database pattern. This structure makes it possible to store data centrally. In this pattern, multiple services shared the same database and each service get data from the same pool. Both services product and ProductCategory microservices communicate with each other through APIs and shared the same database.

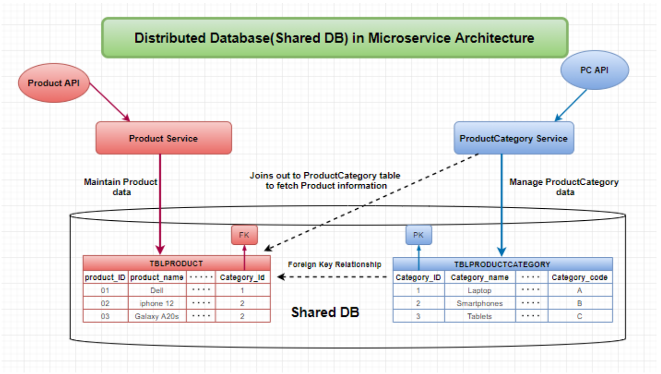
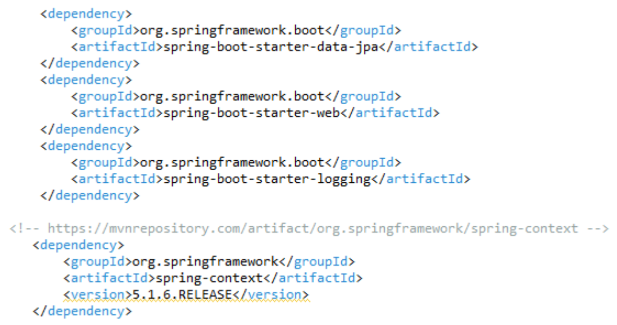

Distributed Database(Shared DB) in Microservice Architecture
Product API
PC API
Product Service
ProductCategory Service
Maintain Product data
Joins out to ProductCategory table to fetch Product information
Manage ProductCategory data
FK
PK
TBLPRODUCT
product_ID | product_name | ····· | Category_Id
01 | Dell | ···· | 1
02 | iphone 12 | ···· | 2
03 | Galaxy A20s | ···· | 2
Foreign Key Relationship
Shared DB
TBLPRODUCTCATEGORY
Category_ID | Category_name | ···· | Category_code
1 | Laptop | ···· | A
2 | Smartphones | ···· | B
3 | Tablets | ···· | C

Fig. 6. Shared database pattern using APIs.

MYSQL was used to create and implemented distributed database such as shared databases. For this model, first step to create product and ProductCategory microservices in spring tool suit(STS) IDE. The dependencies as shown in Fig. 7 were managed using maven and pom.xml file in project directory.

The simplified shared database pattern of product and ProductCategory is shown in above Fig. 6, where both mi-

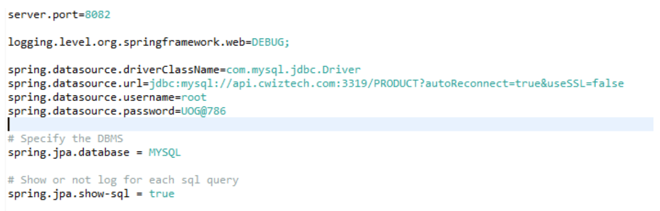


```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-logging</artifactId>
</dependency>

<!-- https://mvnrepository.com/artifact/org.springframework/spring-context -->
   <dependency>
       <groupId>org.springframework</groupId>
       <artifactId>spring-context</artifactId>
       <version>5.1.6.RELEASE</version>
   </dependency>
```

Fig. 7. Shared database dependencies.

croservices shared the same database. Java persistence API was used to connect to a MYSQL database such as shared database with product and ProductCategory microservices.



```
server.port=8082

logging.level.org.springframework.web=DEBUG;

spring.datasource.driverClassName=com.mysql.jdbc.Driver
spring.datasource.url=jdbc:mysql://api.cwiztech.com:3319/PRODUCT?autoReconnect=true&useSSL=false
spring.datasource.username=root
spring.datasource.password=UOG@786

# Specify the DBMS
spring.jpa.database = MYSQL

# Show or not log for each sql query
spring.jpa.show-sql = true
```

Fig. 8. Database connectivity.

Database connectivity in Fig. 8 discusses to the mechanisms and protocols that allow an application or service to establish and maintain a connection with a database system for executing operations such as data retrieval, insertion, updates, and deletion.

This pattern was implemented using STS and create product and product-category microservices. for each microservice, Create Entity class which is all about tables, Repository is an interface that define the presentation part of the database, Rest Controller classes define APIs so that request can be accept and provide the response. Microservices communicate with each other through APIs. Both microservices Perform CRUD operation and calculate response time. Product microservice send request to ProductCategory microservice to get data. Product-category microservices accept request, if required data available in product-category microservices using Request Mapping.

2) Saga choreography pattern using database per services: The Saga Choreography Pattern (Fig. 9) is a distributed coordination mechanism in microservices architecture that manage distributed communication across several services without depending on a central orchestrator. In this pattern, every service waits for particular events and having finished its activity, posts an event that other services respond to, carrying on the sequence until the whole business process is over. Microservices basically follow the pattern called database per services pattern. Where product service has its own database similarly product category microservices have its own database. so if you observe, there is two microservices and they are pointing to two different databases. If it is a single database, then the approach will be same as monolithic. In this pattern, microservices communicate with each other using asynchronous way such as events.

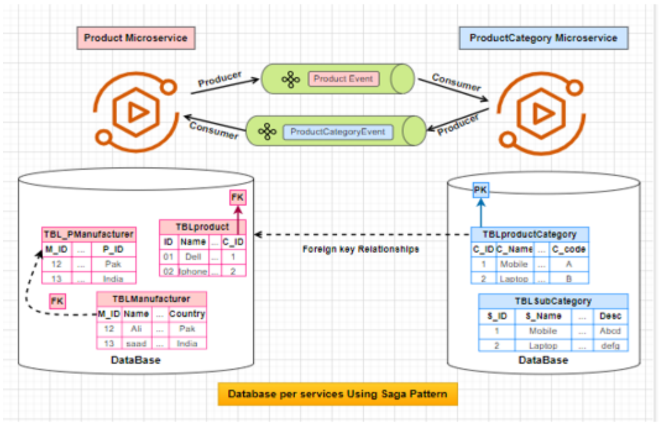
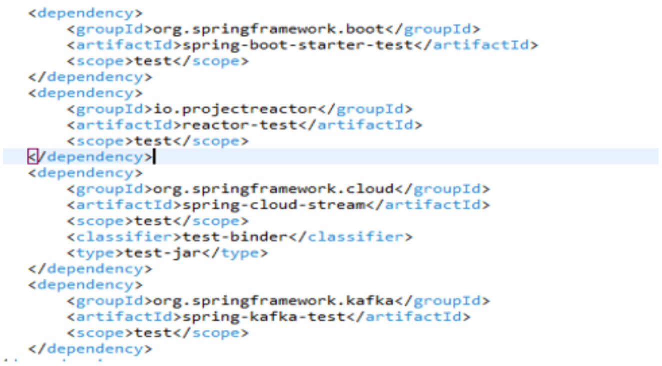


Fig. 9. Saga choreography pattern.

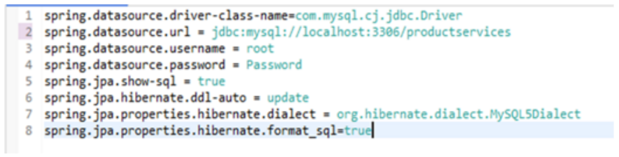
At the very first, to Setup apache kafka environment such as message broker to manage publish and subscribe the events. After that, create product and product-category microservices and added required dependencies in Fig. 10.



```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>io.projectreactor</groupId>
    <artifactId>reactor-test</artifactId>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-stream</artifactId>
    <scope>test</scope>
    <classifier>test-binder</classifier>
    <type>test-jar</type>
</dependency>
<dependency>
    <groupId>org.springframework.kafka</groupId>
    <artifactId>spring-kafka-test</artifactId>
    <scope>test</scope>
</dependency>
```

Fig. 10. Dependencies in saga choreography pattern.

For each microservices add Connection Properties in Application properties files to connect microservice with its own database in Fig. 11.



```
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url = jdbc:mysql://localhost:3306/productservices
spring.datasource.username = root
spring.datasource.password = Password
spring.jpa.show-sql = true
spring.jpa.hibernate.ddl-auto = update
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL5Dialect
spring.jpa.properties.hibernate.format_sql=true
```

Fig. 11. Database connectivity.

Product services will receive the request from the product-category microservice then again product service will redirect that request to the product-category services to validate whether that required category is available or not in Product-Category microservices. Microservice Communicate with each other using events. Any change occurs in microservices was handle using Event.

3) Event driven architecture using database per services: Event-driven architecture (EDA) using database per service is a paradigm for designing and building software systems where events are central to communication between different components or services. Event driven architecture using asynchronous way to communicate with each. Event driven architecture (EDA) using database per service, components react to events asynchronously, enabling loose coupling, scalability, and flexibility.

Event driven architecture use apache Kafka as a message broker for asynchronous way to communicate with each. A kafka Broker is a server that hosts Kafka partitions. It is responsible for managing the storage, retrieval, and replication of the data (messages) produced by Kafka producers and consumed by Kafka consumers is a server that hosts Kafka partitions. It is responsible for managing the storage, retrieval, and replication of the data (messages) produced by Kafka producers and consumed by Kafka consumers. A Kafka Topic is a logical channel to which producers publish messages and from which consumers read messages. A topic is like a category or feed name to which records are sent as shown in Fig. 12.

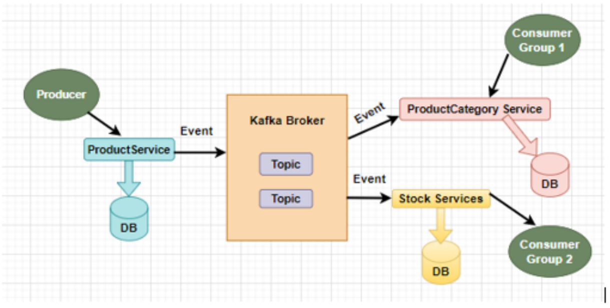
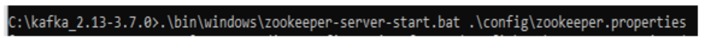


Fig. 12. Event driven architecture using apache kafka.

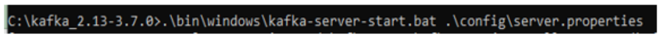
First step to Setup apache kafka environment such as message broker for asynchronously communication to manage publish and subscribe the events. Start zookeeper service using command.



```
C:\kafka_2.13-3.7.0>.\bin\windows\zookeeper-server-start.bat .\config\zookeeper.properties
```

Fig. 13. Start zookeeper service command.

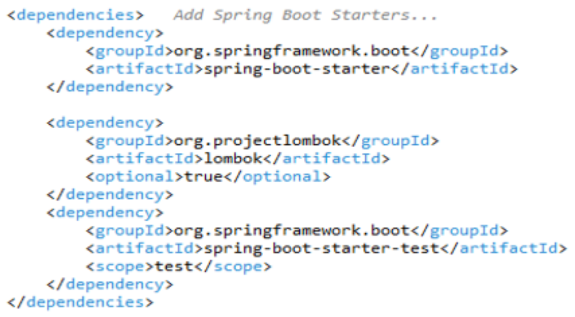
Start Kafka Broker using command (see Fig. 13 and Fig. 14)



```
C:\kafka_2.13-3.7.0>.\bin\windows\kafka-server-start.bat .\config\server.properties
```

Fig. 14. Start kafka broker command.

In EDA, product service has its own database which is responsible for managing the product similarly product category microservices has its own database which is responsible for managing product category. At the very first, create product



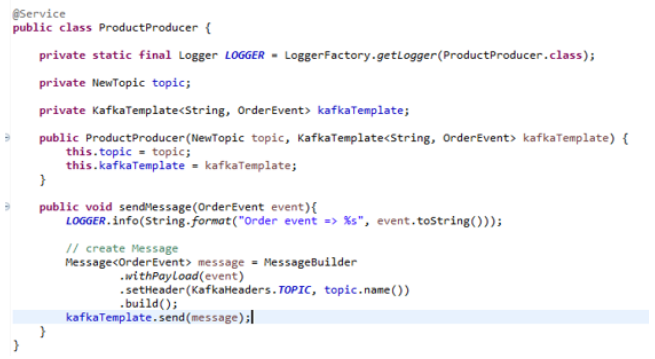
```
<dependencies>   Add Spring Boot Starters...
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter</artifactId>
    </dependency>

    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <optional>true</optional>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>
```

Fig. 15. Dependencies in event driven architecture.

and product-category microservices and added required dependencies.

Event driven architecture use event producer and event consumer while communication. Event Producer that produce event when state changes occur. Producers publish events to the event broker. while event consumer consumes event and perform action according to the request. Performing CRUD operations in event driven architecture (Fig. 15) using database per services pattern and validate foreign key relationship.



```
@Service
public class ProductProducer {

    private static final Logger LOGGER = LoggerFactory.getLogger(ProductProducer.class);

    private NewTopic topic;

    private KafkaTemplate<String, OrderEvent> kafkaTemplate;

    public ProductProducer(NewTopic topic, KafkaTemplate<String, OrderEvent> kafkaTemplate) {
        this.topic = topic;
        this.kafkaTemplate = kafkaTemplate;
    }

    public void sendMessage(OrderEvent event){
        LOGGER.info(String.format("Order event => %s", event.toString()));

        // create Message
        Message<OrderEvent> message = MessageBuilder
                .withPayload(event)
                .setHeader(KafkaHeaders.TOPIC, topic.name())
                .build();
        kafkaTemplate.send(message);
    }
}
```

Fig. 16. Event producer in EDA.

Product microservices create a new entity in its local database such as product. After that, product microservices publish an event with the entity data to Apache kafka. After that kafka broker publish an event and send event to the related microservice. Then productcategory microservices consume the event and creates a related entity in its local database such as Productcategory services with a foreign key referencing the product entity in product microservices database (Fig. 16).

In event-driven systems using the database-per-service pattern, enforcing foreign key constraints across services is not possible through traditional relational resources. Instead, referential integrity is maintained asynchronously using events. However, this approach must handle the realities of network partitioning, as defined in the CAP theorem, which states that in a distributed system, it is difficult to simultaneously guarantee Consistency, Availability, and Partition Tolerance. Event-driven architectures usually prioritize availability and partition tolerance, which means consistency is eventually achieved rather than immediately enforced. This trade-off affects CRUD operations, especially create and delete, where integrity checks must rely on event messages from the owning service (e.g., ensuring a ProductCategory exists before creating an Order). In such cases, idempotency becomes essential. Since events can be delayed, duplicated, or retried, systems must be designed to safely handle the same event more than once without corrupting data. This adds operational complexity but is crucial to ensure reliable referential integrity in the absence of direct foreign key enforcement. Thus, network partition handling and idempotent design are key pillars in using event-driven architectures to maintain data consistency across microservices.

IV. Results and Discussion

A. Overview

This section contains the experimentation of the proposed framework to evaluate how foreign key constraints can be used to maintain referential integrity in distributed databases within a microservices architecture. To test our approach, we designed a case study involving two central microservices: ProductCategoryService and ProductService, which simulate a real-world e-commerce scenario. These services are logically connected through a foreign key relationship, where each product belongs to a specific product category.

B. Case Study

In the designed case study, we implemented a distributed database system using the Database per Service pattern. The ProductCategoryService manages the ProductCategory entity and its database, while the ProductService manages the Product entity. A logical foreign key constraint is defined between the Productcategory-id field and the ProductCategory-id, ensuring that each product must belong to a valid category.

In this section the results of Response time of the three distributed database patterns are presented. For clarity, we presented the most-significant results obtains. In the first section, we examine how distributed database patterns compare with each other and in the last section we observed the response time while performing CRUD operation in distributed databases and we define the performance for all the patterns considered. The response time of the three pattern were compared in a pairwise such as:

- Shared database using APIs vs. Event Driven Architecture.
- Shared DB Using APIs vs. choreography saga pattern.
- Event driven Architecture vs. Saga Choreography pattern.
- Response time for all the pattern.

C. Evaluation Criteria

Microservices architecture can be evaluated using variety of criteria such as data integrity, data availability, data scalability, transaction management and response time. In this research, we will mainly focus on how response time and critical aspects such as scalability, fault isolation, and operational complexity behave when performing CRUD operations. Response time within a distributed database in microservices architecture, refer to the duration when the request is send to and when the client receive the response. This include numerous stages such as transmission time (time for request to reach the target microservices), service processing time (time which involve business logic and database CRUD operations on the distributed database) which consist of data access, data consistency and communication across the node or replica, inter services communication alongside response transmission time back to client.

D. Shared Database vs. Event Driven Architecture

The result in below Table III shows the response time for distributed database pattern. Some test cases were executed against the deployed patterns. Each test case consists of a HTTP request which consists a server side overview for data access. In this case, each test cases were executed using well defined APIs in shared database. This was carried out turn by turn. First shared database using APIs/HTTP and then database per service pattern such as event driven architecture using events were carried out and perform CRUD operation and calculate the average response time.

TABLE III. Comparison Between Event Driven Architecture vs. Shared DB

DB Operation	**Event Driven Pattern**			**Shared DB**		
	Number of Request (ms)					
	10	**100**	**1000**	**10**	**100**	**1000**
CREATE	73.17	695.47	7155.34	43.89	428.65	3987.45
READ	40.23	385.51	3993.04	35.33	359.09	3367.77
UPDATE	92.45	859.44	8967.71	58.2	575.17	5734.63
DELETE	89.88	887.04	8852.08	59.91	603.45	6072.11
Avg. Response Time	73.93	507.10	7242.04	49.33	491.59	4790.49

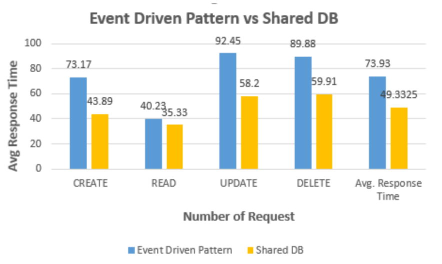


Fig. 17. Avg. Response time between shared DB VS Event driven architecture.

Taking an average of all the tests carried out indicates that shared database pattern performed better than database per services (EDA) while performing CRUD operation. Shared database pattern have relatively Fast response time as compare to event driven Architecture. Response time in Shared database is Fast due to access of same database. While event driven architecture takes time for coordination between different services, performing CRUD operations and inter-communication services and many more. Due this reason event driven architecture takes more time as show in Fig-17.

While analyzing other factors shared Database has limited scalability because all services depend on a single database, which becomes a bottleneck as traffic increases. It also has weak fault isolation. If one service or its database fails, it can affect others since everything is tightly connected. While it's easy to manage at first, complexity grows with more services due to shared schema and organization needs. In contrast, Event-Driven Architecture offers very high scalability because services and messaging systems can grow independently. It provides excellent fault isolation, as services are decoupled and communicate through events that can be retried if one fails. However, it brings very high operational complexity, requiring careful handling of events, retries, and monitoring, which adds to the system's management effort.

E. Shared Database vs. Saga Choreography Pattern

The result presented in table shows the response time of distributed database between the shared database using APIs/HTTP request and database per services. In this case, each test cases were executed using SAGA as well as database per services. This was carried out turn by turn. First shared database using APIs/HTTP were carried out and perform CRUD operation and calculate the average response time. Secondly, SAGA pattern using database per services pattern was carried out and perform CRUD operations and calculate the average response time (Table IV).

TABLE IV. Comparison Between Saga Choreography Pattern vs. Shared DB

DB Operation	Saga Pattern			Shared DB		
	Number of Request (ms)					
	10	100	1000	10	100	1000
CREATE	67.19	598.43	6495.35	43.89	428.65	3987.45
READ	37.48	332.29	3577.61	35.33	359.09	3367.77
UPDATE	83.67	789.11	7982.47	58.2	575.17	5734.63
DELETE	75.89	668.56	6830.03	59.91	603.45	6072.11
Avg. Response Time	66.05	597.09	6221.36	49.33	491.59	4790.49

The comparison between shared database and Saga choreography pattern using database per services pattern were carried out on the bases of average response time while performing CRUD operations.

Taking an average of all the tests carried out in fig. 18 that shared database pattern performed better than saga choreography pattern while performing CRUD operation. Shared database pattern have relatively Fast response time as compare to SAGA. Response time in Shared database is Fast due to access of same database. While saga choreography pattern involved stages such as transmission time, service processing time which involve business logic and database CRUD operations and global coordination time alongside response transmission time back to client. Due to this reason Saga choreography takes more time as compare to shared DB.

In terms of scalability, the Shared Database is harder to scale because all services trust on a single database, which can become a bottleneck as the system grows. In contrast, Saga

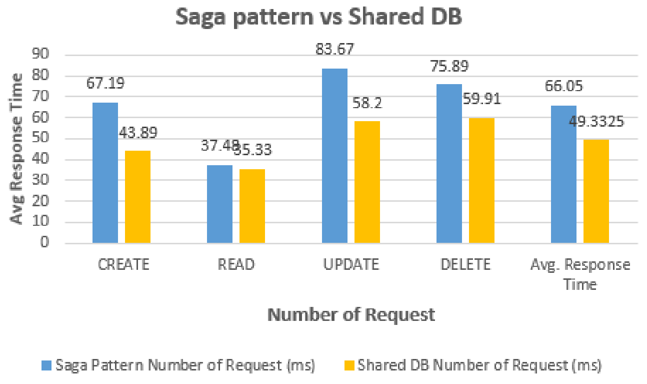


Fig. 18. Avg. Response time between shared DB vs. Saga choreography pattern.

Choreography scales more easily since each service has its own database and can grow independently. For fault isolation, the Shared Database is weaker. Issues in one service or changes to the shared schema can affect others. Saga Choreography, however, offers stronger fault isolation, as services are decoupled and operate independently. When it comes to operational complexity, the Shared Database is easier to manage at first due to its simplicity, but complexity grows as more services are added. Saga Choreography is more complex from the beginning, as it requires managing inter-service communication and ensuring data consistency through event handling.

F. Database Per Services (Choreography Saga Pattern) vs. Database Per Services (EDA)

The result presented in the table shows the response time between distributed databases. Both distributed databases used database per services pattern. The results of the table indicate an average response time of choreography saga pattern and event driven architecture.

TABLE V. Comparison between Saga Choreography pattern vs. Event Driven Architecture

DB Operation	Saga Pattern			Event Driven Pattern		
	Number of Request (ms)					
	10	100	1000	10	100	1000
CREATE	67.19	598.43	6495.35	73.17	695.47	7155.34
READ	37.48	332.29	3577.61	40.23	385.51	3993.04
UPDATE	83.67	789.11	7982.47	92.45	859.44	8967.71
DELETE	75.89	668.56	6830.03	89.88	887.04	8852.08
Avg. Response Time	66.05	597.09	6221.36	73.93	507.10	7242.04

The data in Table V: shows that Event Driven Architecture performed better for each batch update, deletion, insertion, then Saga choreography pattern.

Saga choreography design pattern choose asynchronous communication between the services and many other factors such as network latency may low their response time. In the

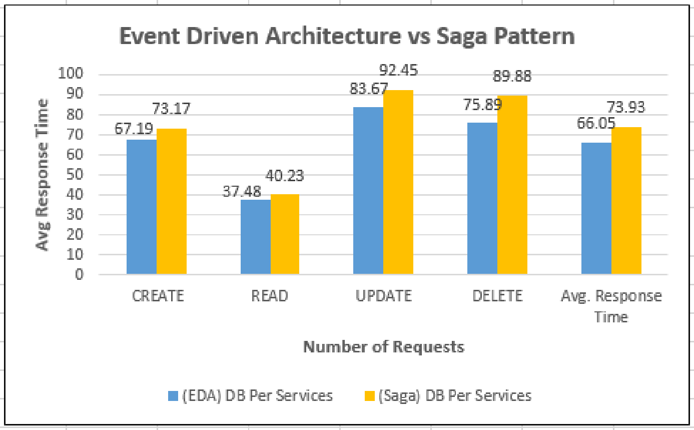


Fig. 19. Avg. Response time between event driven Architecture vs. Saga choreography pattern.

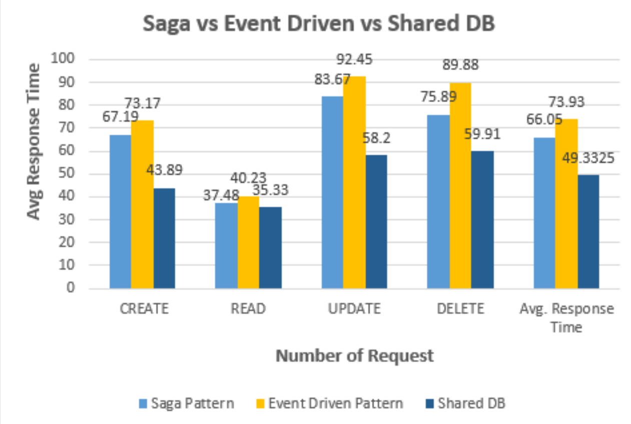


Fig. 20. Avg. Response time of different methods.

TABLE VI. Comparison Between Saga Choreography Pattern vs. Event Driven VS Shared DB

DB Op	Saga Pattern			Event Driven Pattern			Shared DB		
	Number of Request (ms)								
	10	100	1000	10	100	1000	10	100	1000
C	67	598	6495	73	695	7155	43	428	3987
R	37	332	3577	40	385	3993	35	359	3367
U	83	789	7982	92	859	8967	58	575	5734
D	75	668	6830	89	887	8852	59	603	6072
Avg RT	66	597	6221	73	507	7242	49	491	4790

above graph clear show that in each test case event driven architecture perform better then saga choreography pattern. The comparison between Saga Choreography Pattern and Event-Driven Architecture (EDA)—both using the database-per-service approach—shows that while both patterns offer good scalability and fault isolation, EDA takes these aspects further. Saga Choreography provides high scalability, as services operate with their own databases and coordinate through events, and offers strong fault isolation through loose coupling. However, it requires managing compensating transactions and choreography logic, making it operationally complex. On the other hand, EDA achieves very high scalability and fault isolation, thanks to full service decoupling and asynchronous communication. But this comes at the cost of very high operational complexity, involving advanced event design, ensuring idempotency, managing retries, and implementing distributed tracing to monitor the system.

While comparison between the three patterns of distributed database. We observe each test case with each three pattern such as shared database, saga choreography and event driven architecture as shown in the Table VI.

After observing each test with 10,100,1000 number of request respectively, we conclude that shared database performed faster as compare to other two patterns. While Event driven architecture performs better then saga choreography pattern while performing CRUD operation. For more clarity in the results we represent the result in graphical representation in Fig. 19 and Fig. 20.

V. Conclusion and Recommendation

In this research distributed databases in microservices has been broadly analyze while performing CRUD operations and how we achieve loose coupling between the services while maintaining data integrity and very efficient data access performance. Furthermore, an evaluation of foreign key constraints to maintain referential integrity in distributed databases (shared database, database per services) as shown in this research bring out one of the core challenge of microservices. This research further proven that shared database pattern give better performance then database per services while performing simple data related operation such as simple CRUD operation.

In this research after examined the test case, it could be deduced that foreign key constraints in a database per services pattern does not carried out at database level. While future could find a way to mitigate this challenge. When designing data driven microservices, it remains a crucial factor to consider.

References

[1] S. Li, H. Zhang, Z. Jia, C. Zhong, C. Zhang, Z. Shan, J. Shen, and M. A. Babar, "Understanding and addressing quality attributes of microservices architecture: A systematic literature review," *Information and software technology*, vol. 131, p. 106449, 2021.

[2] L. Jiang and F. Naumann, "Holistic primary key and foreign key detection," *Journal of Intelligent Information Systems*, vol. 54, pp. 439–461, 2020.

[3] W. Lu, "Replacing a monolithic web application with a new backend framework," 2018.

[4] C. K. Rudrabhatla, "Impacts of decomposition techniques on performance and latency of microservices," *International Journal of Advanced Computer Science and Applications*, vol. 11, no. 8, 2020.

[5] X. Liu, S. Jiang, X. Zhao, and Y. Jin, "A shortest-response-time assured microservices selection framework," in *2017 IEEE International Symposium on Parallel and Distributed Processing with Applications and 2017 IEEE International Conference on Ubiquitous Computing and Communications (ISPA/IUCC)*. IEEE, 2017, pp. 1266–1268.

[6] B. Barua, M. Whaiduzzaman, M. Mesbahuddin Sarker, M. Shamim Kaiser, and A. Barros, "Designing and implementing a distributed database for microservices cloud-based online travel portal," in *Sentiment Analysis and Deep Learning: Proceedings of ICSADL 2022*. Springer, 2023, pp. 295–314.

[7] R. Laigner, Y. Zhou, M. A. V. Salles, Y. Liu, and M. Kalinowski, "Data management in microservices: State of the practice, challenges, and research directions," *arXiv preprint arXiv:2103.00170*, 2021.

[8] X. Wu, N. Wang, and H. Liu, "Discovering foreign keys on web tables with the crowd," *Computing and Informatics*, vol. 38, no. 3, pp. 621–646, 2019.

[9] M.-D. Pham, L. Passing, O. Erling, and P. Boncz, "Deriving an emergent relational schema from rdf data," in *Proceedings of the 24th International Conference on World Wide Web*, 2015, pp. 864–874.

[10] L. M. D. S. dos Santos *et al.*, "Data distribution and access in a microservices architecture," 2024.

[11] 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)*. IEEE, 2018, pp. 000 149–000 154.

[12] J. Motl and P. Kordík, "Foreign key constraint identification in relational databases." in *ITAT*, 2017, pp. 106–111.

[13] G. Blinowski, A. Ojdowska, and A. Przybyłek, "Monolithic vs. microservice architecture: A performance and scalability evaluation," *IEEE access*, vol. 10, pp. 20 357–20 374, 2022.

[14] V. Talaver and T. A. Vakaliuk, "Reliable distributed systems: review of modern approaches," *Journal of edge computing*, vol. 2, no. 1, pp. 84–101, 2023.

[15] Y. Romani, O. Tibermacine, and C. Tibermacine, "Towards migrating legacy software systems to microservice-based architectures: a data-centric process for microservice identification," in *2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C)*. IEEE, 2022, pp. 15–19.

[16] E. Volynsky, M. Mehmed, and S. Krusche, "Architect: A framework for the migration to microservices," in *2022 International Conference on Computing, Electronics & Communications Engineering (iCCECE)*. IEEE, 2022, pp. 71–76.

[17] S. Lungu and M. Nyirenda, "Current trends in the management of distributed transactions in micro-services architectures: A systematic literature review," *Open Journal of Applied Sciences*, vol. 14, no. 9, pp. 2519–2543, 2024.

[18] E. Daraghmi, C.-P. Zhang, and S.-M. Yuan, "Enhancing saga pattern for distributed transactions within a microservices architecture," *Applied Sciences*, vol. 12, no. 12, p. 6242, 2022.

[19] R. Laigner, Y. Zhou, and M. A. V. Salles, "A distributed database system for event-based microservices," in *Proceedings of the 15th ACM International Conference on Distributed and Event-based Systems*, 2021, pp. 25–30.

[20] T. de Oliveira Rosa, J. F. L. Daniel, E. M. Guerra, and A. Goldman, "A method for architectural trade-off analysis based on patterns: Evaluating microservices structural attributes," in *Proceedings of the European Conference on Pattern Languages of Programs 2020*, 2020, pp. 1–8.

[21] R. Laigner, M. Kalinowski, P. Diniz, L. Barros, C. Cassino, M. Lemos, D. Arruda, S. Lifschitz, and Y. Zhou, "From a monolithic big data system to a microservices event-driven architecture," in *2020 46th Euromicro conference on software engineering and advanced applications (SEAA)*. IEEE, 2020, pp. 213–220.

[22] M. Manouvrier, C. Pautasso, and M. Rukoz, "Microservice disaster crash recovery: a weak global referential integrity management," in *International Conference on Computational Science*. Springer, 2020, pp. 482–495.

[23] A. Katsifodimos and M. Fragkoulis, "Operational stream processing: Towards scalable and consistent event-driven applications." in *EDBT*, 2019, pp. 682–685.

[24] R. Heinrich, A. Van Hoorn, H. Knoche, F. Li, L. E. Lwakatare, C. Pahl, S. Schulte, and J. Wettinger, "Performance engineering for microservices: research challenges and directions," in *Proceedings of the 8th ACM/SPEC on international conference on performance engineering companion*, 2017, pp. 223–226.

[25] S. R. Vangala, R. K. Mallidi, and V. P. Appili, "Micro-services transactions resilience across bounded domains: An architecture perspective," *International Journal of Computer Applications*, vol. 975, p. 8887.

[26] M. André, "Automated database schema evolution in microservices," in *Conference on Very Large Data Bases (VLDB 2023)*, 2023.

[27] A. Mparmpoutis and G. Kakarontzas, "Using database schemas of legacy applications for microservices identification: A mapping study," in *Proceedings of the 6th International Conference on Algorithms, Computing and Systems*, 2022, pp. 1–7.

[28] A. Singjai, U. Zdun, O. Zimmermann, M. Stocker, and C. Pautasso, "Patterns on designing api endpoint operations," 2021.

[29] I. K. Aksakalli, T. Çelik, A. B. Can, and B. Tekinerdoğan, "Deployment and communication patterns in microservice architectures: A systematic literature review," *Journal of Systems and Software*, vol. 180, p. 111014, 2021.

[30] K. Munonye and P. Martinek, "Evaluation of data storage patterns in microservices archicture," in *2020 IEEE 15th International Conference of System of Systems Engineering (SoSE)*. IEEE, 2020, pp. 373–380.

[31] M. El Kholy and A. El Fatatry, "Framework for interaction between databases and microservice architecture," *IT Professional*, vol. 21, no. 5, pp. 57–63, 2019.

[32] N. Viennot, M. Lécuyer, J. Bell, R. Geambasu, and J. Nieh, "Synapse: a microservices architecture for heterogeneous-database web applications," in *Proceedings of the tenth european conference on computer systems*, 2015, pp. 1–16.