A
Project Report
on

Web Interface for Distributed Transaction Systems

SUBMITTED TOWARDS THE
FULFILLMENT OF THE REQUIREMENTS OF

Bachelor Of Engineering (Computer Engineering)

BY

Hardik Pandya (72021513H)
Shubham Babasaheb Bhosale (72028592F)
T Rohith Kumar (72021692D)
Godage Suraj Dinkar (72034314D)

Under The Guidance of

Asst. Prof. Rushali S Patil

Department Of Computer Engineering
Army Institute Of Technology, Pune - 411015.

SAVITRIBAI PHULE PUNE UNIVERSITY
2022-23

ARMY INSTITUTE OF TECHNOLOGY, DEPARTMENT OF COMPUTER ENGINEERING

CERTIFICATE

This is to certify that the Project entitled

Web Interface for Distributed Transaction Systems

Submitted by

Hardik Pandya (72021513H)
Shubham Babasaheb Bhosale (72028592F)
T Rohith Kumar (72021692D)
Godage Suraj Dinkar (72034314D)

is a bonafide work carried out by students under the supervision of Asst. Prof. Rushali S Patil, and it is submitted towards the fulfillment of the requirement of Bachelor Of Engineering (Computer Engineering) Project.

Asst. Prof. Rushali S Patil
Project Guide

Prof. Dr. S.R Dhore
H.O.D

External Examiner

Dr. B P Patil
Principal

Place : AIT, Pune

Date :

PROJECT APPROVAL SHEET

A

Final Project Report

on

Web Interface for Distributed Transaction Systems

is successfully completed by

Hardik Pandya (72021513H)
Shubham Babasaheb Bhosale (72028592F)
T Rohith Kumar (72021692D)
Godage Suraj Dinkar (72034314D)

at

Department Of Computer Engineering
Army Institute of Technology, Pune-411015.

SAVITRIBAI PHULE PUNE UNIVERSITY
2022-23

Asst. Prof. Rushali S Patil
Project Guide

Prof. Dr. S.R Dhore
HOD

ACKNOWLEDGEMENT

It is great pleasure for us to undertake this project. We would like to express our gratitude toward staff of Computer Engineering department for providing us this great opportunity to work on B.E project **Web Interface for Distributed Transaction Systems.**

I am overwhelmed with deference and appreciation to acknowledge my gratitude towards all those who have assisted me in taking these concepts beyond the level of simplicity and developing them into something substantial. The success of this project report necessitated a great deal of direction and cooperation from many people, and I am grateful and privileged to have received it from all of the assisting entities.

I owe a debt of gratitude towards **Asst. Prof. Rushali S Patil**, my project guide for paying close attention to and taking an interest in my project, and for steering me in the proper way throughout my project duration by providing all of the required information for establishing a good system. Despite her busy schedule, I am grateful to her for providing me with her fascinating assistance and direction.

I also would like to extend my appreciation towards the Head of the Computer Department Dr. Sunil Dhore for his assistance and his constant counsel and leading me in the right direction in the research I performed.

This project would not have completed without their enormous help and worthy experience. Whenever we were in need, they were there to help.

Hardik Pandya
Shubham Babasahed Bhosale
T Rohith Kumar
Godage Suraj Dinkar
(B.E. Computer Engg.)

ABSTRACT

The concept of the saga pattern, which is a technique used to manage local sequential transactions across distributed microservices. While the saga pattern is effective in ensuring data consistency, it lacks isolation, which can result in incorrect commits on databases if transactions are left unfinished. To address this issue and improve existing solutions like transaction management protocols such as the two-phase commit, this research proposes enhancements such as the quota cache and commit-sync service.

The quota cache and commit-sync service facilitate operations between database layers, preventing invalid or incomplete commitments from occurring on main databases. An experiment was conducted to evaluate the effectiveness of these enhancements in a microservices-based e-commerce system. The results of the experiment showed that the proposed approach could handle both regular scenarios and exceptions, effectively addressing isolation issues.

In case of service failures, compensation transactions are performed to undo adjustments made exclusively within the caching layer. Once all processes are completed correctly, the alterations are committed back into the database. The results of the experiment demonstrate promising features of the approach, but further investigation is needed to optimize it before it can be widely implemented as an industry-standard approach.

Contents

List of Figures

Chapter 1

INTRODUCTION

1.1 Details of Project Work

Creation of Microservices

- Microservices will be created using Springboot in backend.
- Communication between microservices will be done using REST APIs.
- Orchestration approach will be used to manage transation across Microservices.

Queue Middleware for Events handling

- Several message queue middleware were also utilised to manage the numerous events, such as the buy events, completion events, and failure events. The system as a whole may operate at a greater throughput thanks to these message queues.
- Mainly APACHE KAFKA will be utilized as a message Queue Milldleware.

Isolation Handling in SAGA

We will use quota cache database using REDIS database server.The commits sync pattern will be used to solve the issue of isolation.A distributed system called a microservice-based application uses several, smaller services that cooperate to offer the functionality of the application. The monolithic application has been divided into numerous distinct services thanks to the microservice design, which offers lossened and low coupling, increased maintainability, increased availability, and scalability for the creation of applications. The technological stack for each service may be chosen using the microservices architecture. For instance, the services

may maintain the domain data independently if the relational database was used to implement one service while the NoSQL database was used to implement the other. Additionally, the microservice design enables on-demand scalability of data repositories.

Since each microservice is assosciated with its database that holds specific business activities, it is challenging to manage dispersed transactions and maintain data integrity when transactions span several services.The Software Automation, Generation, and Administration (Saga) Pattern was suggested as a way to manage and handle distributed transactions in and accross the microservice architecture. Through distributed microservices transactions, the Saga design pattern handles transactions and upholds consistency of data. The microservices are updated by Saga, a group of local sequential transactions, and messages are published to start the subsequent transactions. If a transaction fails, compensatory transactions will be conducted to offset the failed transaction. The saga structure lacks read isolation, though. In contrast to ACID (atomicity, consistency, isolation, durability), ACD stands for "atomicity, consistency, durability."

Data reading and writing from an unfinished transaction are permitted as a result of the absent isolation, which causes a number of isolation anomalies [4,5]. Therefore, to solve this issue, this research suggests an improved saga structure that makes use of the quota cache and the commit-sync service to provide eventual consistency. By dedicating a portion of the primary database to the in-memory data caching server, it is intended to combine the traditional saga design with an in-memory data caching layer. As a result, the CRUD (create, read, update, and delete) operations will be performed through the quota cache rather than the main database, preventing any errors from being committed to the main database.

When a microservice fails to complete, the remaining microservices carry out compensatory transactions to undo modifications that only impact the cache layer and not the core database layer. In addition to overcoming the read-isolation issue with the saga design, this will increase performance.

Once all transactions have successfully completed, the database commit will be postponed and handled through the message queue middleware in order to achieve eventual consistency. Since all of the domain use cases were established, the clean architecture method was applied throughout implementation.

To compare the basic baseline standard version of the saga pattern with the suggested upgraded version, a e-commerce system that is lightweight microservices-based was created for demonstration purposes. Several tests were run for validation and assessment. Results show that the suggested strategy is able to address the saga pattern's lack of isolation. Results also show that the suggested method outperforms the baseline standard version not only in typical situations but also when handling exceptions because using cache operations rather than database operations improves performance and lowers latency time.

1.2 Objectives

Build a microservice architecture

Microservice architecture is a design strategy that divides an application into a number of tiny, independent services, each with a particular business capability. It entails segmenting a huge application into several services that can be created, set up, and scaled separately. The well-defined APIs used by these services to communicate with one another. Service independence, where each microservice may be built and scaled independently, boosting flexibility and agility, is one of the key features of microservice based architecture. In order to ensure modularity and maintainability, each microservice concentrates on a particular business feature. With microservices having their own exclusive databases, data management is decentralised, enabling data separation and minimising dependencies. Services communicate with one another through APIs, which promotes loose coupling and technological variety. By preventing problems in one service from affecting others, fault isolation enhances fault tolerance.

Microservice design has advantages including increased fault tolerance, scalability, and agility. It encourages modularity, makes it possible for autonomous service development and deployment, and makes it possible for more efficient resource use. In conclusion, a microservice architecture divides an application into separate services, each of which has a particular business competency. Through the use of APIs for communication, these services offer scalability, flexibility, and adaptability.

Distributed transaction handling across microservices architecture

To maintain consistency, dependability, and atomicity in distributed systems, distributed transaction management in microservices architecture entails managing transactions that span many microservices. Coordination of transactions is a challenging problem in a microservices architecture since each microservice functions as an independent and autonomous component. Microservices divide data across numerous services, each of which has its own database or data store, in contrast to monolithic programmes, which allow a single database transaction to execute multiple actions. Maintaining transactional integrity across services becomes difficult as a result.

For distributed transactions management in a microservices based architecture, there are many methods available:

Saga Pattern: Using the Saga pattern, a lengthy transaction is broken down into number of smaller, local and less important operations that are carried out by each participating microservice. To preserve consistency, each microservice retains its own transactional logic and mitigating actions. If an error occurs, corrective measures are used to reverse the modifications made in the earlier processes.

Two-Phase Commit: The 2PC protocol is an established method for managing dispersed transactions involving several parties. To decide whether to commit or cancel the transaction, a coordinator must communicate with each microservice. Although 2PC offers excellent consistency, it has drawbacks such blocking behaviour and failure susceptibility.

Eventual Consistency: This strategy relies on eventual reconciliation or consistency techniques to let microservices eventually update their local data asynchronously and eventually converge to a consistent state. With this strategy, scalability and availability take precedence above instant consistency.

These methods make it easier to handle distributed transactions in microservices architecture and guarantee that, despite the difficulties of a distributed environment, transactions across several microservices preserve consistency, dependability, and atomicity.

Implement enhanced SAGA pattern by using redis database cache online server and kafka message queue of events middleware

Implementing standard saga pattern with kafka message queue middleware and redis online cache server database will provide security to prevent uncommited read and write. This will be an improved saga design that uses the quota cache database server of redis and the commit and sync type of service to address the lack of isolation problem. There will be certain transactions moved from the database layer to the memory core layer.

As a result, there won't be any errors in main database commits. The other microservices will conduct compensation transactions to reverse any modifications that solely affect the cache layer rather than the database layer if a microservice fails to finish. The database will be committed once all transactions have been properly completed. For comparison, a simple microservices-based e-commerce system will be put into use. The purpose of the experiments is to validate and assess the results.

1.3 Motivation

The necessity to solve the shortcomings of monolithic transaction systems and propose a distributed transaction system to meet these issues served as the primary impetus for the creation of this project. The ACID (Atomicity, Consistency, Isolation, Durability) qualities are frequently neglected by monolithic transaction systems, which can result in a number of problems and inconsistencies. The 2-phase commit and 3-phase commit are examples of patterns seen in the world of distributed transaction systems. However, effective concurrency management techniques spanning numerous transactions are frequently absent from these patterns. The SAGA pattern became a popular choice as a result of the necessity to investigate other alternatives.

Enhancing the SAGA pattern's performance and efficacy was this project's second goal. Absence of isolation, which can cause problems and impede the overall system's consistency, is major problem of SAGA pattern faces. The project attempts to enhance the performance and reliability of the SAGA pattern by addressing this problem and implementing isolation measures inside the pattern.The overall goal overcomes the drawbacks of monolithic systems and, by guaranteeing isolation, improves the efficacy of the SAGA pattern.

Chapter 2

LITERATURE SURVEY

2.1 Methods and process of service migration from monolithic architecture to microservices

The issue of service migration from monolithic architecture to microservices is examined in this article. Microservice architecture is gaining popularity as a result of traditional monolithic design's failure to satisfy the demands of scalability and quick development cycles. Both types of architecture are explored, along with their key differences, benefits, and drawbacks. A monolithic application migration is described. The internal consistency of the code, which makes it simpler for the developer to understand precisely how the programme functions [49], is one of its important characteristics. The developer can quickly identify several hundred lines in many modules if the software is effectively divided into its individual component modules, saving him from having to read a large amount of code.

The development of a single software as a collection of smaller programmes, each having its own process and interacting with others through HTTP, is known as microservices architecture [50]. These apps are provided separately using fully automated deployment techniques and are constructed around business operations. These applications have a minimal amount of centralised management and may be created in a variety of programming languages and storage systems [51]. The monolith must have all logic related to price and customer visualisation removed before moving on to the final phase. To establish a unified flow for external users or services, the API gateway may be viewed as the main entry point for all services.

There are a few important things to note: The monolithic architecture organises the programme as a single unit, while the microservice architecture breaks the system down into a number of independently deployed services, each with its own database. The monolithic architecture is good for simple applications, but the microservice architecture is typically a better option for large, complex applications.

2.2 Poster: a declarative approach for updating distributed microservices

Microservices offer a significant advantage by enabling easier modification of programs through their division into independently deployable components. This, in conjunction with Continuous Delivery (CD) and Platform as a Service (PaaS), has made the microservice paradigm vital for implementing agile procedures [47].

To facilitate the upgrading of distributed microservice applications, a proposed system has been introduced for DevOps teams. The system adopts a declarative approach, where teams simply declare the desired target architecture and strategy for the application. The program is then transformed into intermediate architectures through a series of transitions until the target architecture is achieved [48]. Notably, the framework includes a preview mode that allows visualization of the intermediate structures without actually applying them to the system.

Using this framework, DevOps teams can upgrade a microservice application by stating the desired architecture and strategy. They have the flexibility to process updates in stages, modifying the strategy or adjusting the target architecture at each stage to ensure a smooth transition.

2.3 Away from migrant legacy from software systems to architectures based on microservices: process for the identification of microservices (data centric)

"Microservice-based architecture" refers to an architectural approach used to create software systems that focus on independent maintainability, deployability, and scalability[43]. Because of these critical characteristics in current software development and operation contexts, many businesses have migrated their old (legacy) monolithic software systems

to microservice-based architectures. The migration procedure is a difficult effort. It necessitates dividing the system into uniform components that constitute the collection of microservices. Existing studies in this division mostly focus on functional concerns [44].

The first step in achieving that goal is this study. We suggest a method for identifying microservices that analyses database models, preprocesses their symbols, enriches them with terms that are semantically related using lexical databases (such as WordNet or WordWeb) [45], and then groups the symbols into a set of clusters that we consider labels for potential microservice candidates. Following that, each identified microservice is given access to a set of connected tables or documents called a database sub-model [46].

2.4 Enhancing performance and scalability in microservices-based systems with eventdriven architecture

Microservices may be thought of as a group of tiny, autonomous services or processes that often interact to create sophisticated applications. The ability of the microservice architectural style to adapt to technological developments and aid in better organising the development team makes it a viable replacement for monolithic architecture [60].

Loosely connected microservice systems that communicate with one another using publish and listen events to share information or data. EDA enables information to be ingested into an ecosystem that is event based driven and then it will be broadcast to the service that will listen to the (message) event or receive it.

EDA operates asynchronously and uses message events for communication compared to the API based Driven Architecture, which operates synchronously, communicates using calls made by API, and is popular in a variety of fields including network instruction detection, sensor networks, fast trading, real-time system control, healthcare monitoring, mobile computing, and wearable computing. The primary reason is because EDA offers distributed system development technologies that enable high concurrency and flexibility.

The usage of containers, which provide a straightforward method to extend activities by creating more copies of the service, has been attempted as one of several strategies in earlier research to solve the scal-

ability and performance challenges in microservice designs. Containers can also aid with elasticity and scalability problems. An example of a system that makes use of containerization techniques is Docker. It is lightweight, can help with and run a number of microservices, and contributes to better resource utilisation and microservice performance.

2.5 Research on distributed transaction and implementing processing middleware

The in-depth design and implementation details of the advanced distributed transaction processing system POST are covered in detail in this paper [1]. POST's main objective is to offer a simple and effective method for handling transactions in a distributed setting. POSTBOX, a distributed database system based on Berkeley DB's foundation with extra data partitioning features, and POSTMAN, a middleware created expressly for distributed transaction processing, are two key elements that are incorporated into POST to accomplish this. By incorporating data splitting techniques, POSTBOX expands Berkeley DB's existing extremely accessible capabilities. In doing so, the system is able to distribute enormous volumes of data among a number of nodes and manage them effectively.

POSTMAN serves as a customised middleware that continually checks POSTBOX to guarantee the accuracy and dependability of transaction processing. Through the application of the Partition Replication Body State Array (PRBSA), it enables the appropriate operation of transaction migration and preserves transactional integrity, even in the case of a node loss. It's crucial to remember that a distributed system has inherent difficulties providing availability, consistency, and partition tolerance at the same time. The well-known CAP theorem is frequently based on these three principles. Consequently, one of these three must be sacrificed in order to improve the other two. According to the CAP principle, the POST system chooses eventual consistency.

By giving high availability and partition tolerance priority while allowing for some eventual consistency, it achieves a balance. However, it is important to note that when the master node does uploads, the consistency method used for POST does not ensure the consistency of replica nodes. This feature draws attention to a weakness in the consistency mechanisms of the system. POST, which combines the strength of POSTMAN and POSTBOX, offers a complete solution for distributed transaction processing. It offers high availability, partition tolerance,

and eventual consistency in handling transactions across a distributed environment.

2.6 Distributed database and its transaction processing

The complex demands of large-scale data storage have proven to be very difficult for conventional information storage systems to handle [5]. Ensuring transaction integrity in distant and varied situations has been harder as the number of data and applications has grown. The capacity to use MapReduce parallel processing in each processing link is the key distinction between basic and conventional data processing. When processing massive volumes of unstructured data, which is typical in the field of big data processing, this skill becomes important. New data gathering methods have emerged as a result of the development of big data technology. These technologies go beyond the standard ones, such RFID RF and sensors, and can now also collect system logins.

This study collects a thorough review of the many methods employed to assess brand and e-commerce websites in this context. It highlights how crucial it is for companies to take into account the particulars of their operations while performing evaluations. Unfortunately, a sizable proportion of e-commerce websites are still evaluated using antiquated technology standards that no longer accurately represent the changing environment of online commerce.

2.7 2PC* : concurrency controling protocol of multiple distributed microservice transaction

When it comes to establishing transactions(distributed), storage systems like databases(relational) and databases(distributed) heavily rely on the two-phase commit (2PC) protocol. Serialised transaction execution is made possible by this protocol, which is renowned for its great consistency and centralised design. A group of academics built on this foundation by creating middleware that makes use of the 2PC protocol to enable distributed microservices inside transactions. They used Spring's AOP (aspect-oriented programming) to build the annotation "@TxTransactional" and inject distributed transactional capabilities into certain microservices in order to achieve decoupling from the original business module. With this strategy, the microservices are given the freedom to func-

tion independently while yet taking part in the larger transactional operation.

A transaction actor, a transaction coordinator, and a transaction initiator make up the prototype of their solution. The Netty framework, a high-performance networking toolkit made for distributed systems, is used to connect these parts to one another. Microservice configurations are common in distributed systems, particularly those set up on cloud computing platforms, and they can be vulnerable to unanticipated service disruptions including service loss and network delays. When XML-RPC or SOAP is used as a remote call protocol for microservices across a gateway, such the Simple Object Access Protocol (SOAP), certain events are more likely to occur. The researchers improved the transaction compensation approach to guarantee the ultimate consistency of distributed transactions across microservices. This improvement makes it possible for the system to recover and lessens the effects of disturbances.

2.8 Distributed transactions in systems which are reliable

Due to three critical qualities they provide, transactions are essential in many distributed applications. First, concurrent readers and writers of data are prevented from obstructing one another by synchronisation qualities like serializability. This promotes scalability and performance by enabling effective and concurrent access to data. Second, by preventing changes from being left in a partially completed state and ensuring data consistency, failure of atomicity ensures the maintenance of all the invariants found on data. Finally, the idea of permanence gives programmers the confidence that only extremely rare failures may result in harm to or loss of previously performed alterations, guaranteeing the longevity of data.

We choose to use operation invocation that are location-transaperant, lock data (servers), use logging (write-ahead) with a shared log for data consistency, and store objects which are permanent in virtual memory. By taking into account these design options, we want to develop useful and effective distributed systems that can accommodate a variety of applications. We now have a strong basis for implementing distributed transactions in a dependable and scalable way thanks to the confluence of study, development, and use of these approaches.

2.9 Concurrency control and it's distributed transaction

Database systems are essential in today's organisational landscape since they are the foundation for storing and protecting operational data. Organisations might choose between a distributed or centralised strategy when creating database systems. A database system that has been constructed using a distributed technique is referred to as a "distributed database system." The need for increased performance, speed, and availability led to the transition from centralised to distributed systems. Distributed concurrency management techniques are used to make sure that distributed transactions are carried out correctly without compromising the ACID (Atomicity, Consistency, Isolation, Durability) features.

The overall performance and efficiency of the system may be enhanced by serialising these transactions and making sure they are carried out consecutively. Adopting distributed database systems gives businesses more scalability, fault tolerance, and performance. The management of dispersed data, coordinating transactions, and preserving data consistency throughout the network become more difficult [52]. Organisations may efficiently employ distributed database systems to fulfil the rising needs of modern applications by utilising distributed concurrency management technologies and carefully considering data dependencies [53].

2.10 XPC: new 2-commit and 3-phase commit transmission protocols with better synchronous transmission

The authors of this study present X Process Commit (XPC), a synchronous transmission-based communication-based architecture for an atomic commit protocol. They also provide Hybrid, a method that combines the benefits of Chaos and Glossy Synchronous Transmission primitives in order to have reduced latency and more dependability than any primitive used alone. They show through rigorous testing that the suggested protocols perform better in terms of performance and reliability than Glossy or Chaos utilised separately as dissemination primitives. The authors demonstrate how to build protocols for the conventional 2-phase and 3-phase commit abstractions using XPC and Hybrid. These protocols are especially useful for controlling numerous applications found in the infrastructure of smart cities, including lifts, sewer pipes, and pollution monitoring systems.

The authors are creating this work as part of a substantial sensor deployment over the whole smart city in order to increase the effectiveness and dependability of these vital systems. The introduction of X Phase Commit (XPC) as a framework for creating atomic commit protocols that make use of synchronous transmissions and the Hybrid method is the main topic of this paper. Reference implementations for both two-phase commit and three-phase commit are included, and the concept of XPC is fully discussed. These implementations make use of Glossy and Chaos, two well-liked synchronous transmission primitives.

2.11 Distributed transactions on functions which are stateful and serverless

One of the most well-known serverless services is function-as-a-service (FaaS), in which users create functions and the cloud handles deployment, upkeep, and scalability. But when it comes to supporting stateful designs like microservices and scalable, low-latency cloud applications, FaaS falls short. Effectively executing transactions across stateful tasks is the difficulty. The execution of transactions across stateful workloads in a serverless architecture is of utmost importance in addressing this restriction. The paper uses a benchmarking approach to look at this issue. By gradually raising the input throughput generated by benchmark clients, the system's output throughput in Kafka is monitored. As a result, the maximum throughput that can be achieved for each workload and system configuration is determined, giving information about the potential and constraints of the system.

The consistency of programmers and the integrity of transactions are both at stake when non-deterministic activities are a part of transactions. Dealing with the difficulty of providing transactional activities across cloud apps inside a serverless architecture becomes crucial. The article discusses two possible options. The Saga process, which provides eventual atomicity and consistency, is the preferred choice. It offers a method to manage lengthy transactions among distributed functions while maintaining eventual consistency.

2.12 Incorporation of 3-phase commit protocol into the existing intrusion detection systems in the mobile ad-hoc network

A collection of mobile nodes that connect to create a network on their own, independent of any centralised infrastructure, is what makes up a mobile ad hoc network (MANET) [18]. Because of its special qualities, MANETs are increasingly being used for security and business objectives. Security is a serious problem since these networks are vulnerable to numerous external attacks from uninvited parties. For efficiently detecting, identifying, and reacting to threats and incursions in MANETs, the present models of intrusion detection systems (IDS) require a thorough logic [19]. Improved IDS algorithms are therefore increasingly needed to increase the security of these networks. To meet this demand, researchers and developers are working hard to create IDS algorithms that are both more effective and efficient [20].

The three-phase commit approach has been used in the context of distributed database management systems to prevent data consistency problems. By ensuring that distributed transactions are correctly coordinated, committed, and propagated over the network, this method preserves the accuracy of the data kept in the system. Some weaknesses have been found in the current models of intrusion detection systems, including CONFIDANT, CORE, and OCEAN. A fresh enhanced intrusion detection system algorithm was created in response to these faults with the goal of resolving the restrictions and deficiencies of the existing models. The improved method is made to offer stronger and more precise detection and response mechanisms, hence boosting the general security of MANETs.

2.13 Transaction processing with the microservice architecture and its problems

Creating an online purchasing system based on the microservices architectural paradigm is the research's goal. All business processes and the objects they are connected with are assigned to a distinct service called the "Order Microservice" as part of the system's migration to a microservice architecture [21]. Similar items and business operations per-

taining to reservations and inventory use are assigned to the "Inventory Microservice." In order to solve the issues of transaction processing in microservice-based systems, this architectural change requires an analysis of transaction handling features. The possibility for inconsistent transaction results in such systems is one problem[22].

2.14 Method for distributed database transition

The two-phase commit technique is a popular distributed transaction processing technology used in distributed database systems. In the two-phase commit method, there is a coordinator and a lot of people [23][24]. The heartbeat mechanism is introduced and its benefits are explored in order to assure the effective execution of distributed transaction processing utilising the two-phase commit protocol. The two-phase commit procedure moves through two phases and enables efficient participant failure resolution. Immediate evaluation of each database server's performance is possible by constructing a distributed transaction processing system based on the two-phase commit protocol and making use of the heartbeat mechanism [25].

This method is very useful for distributed databases since distributed transaction processing is so important in these systems.Businesses rely significantly on database systems for their operations across a variety of sectors. It is crucial to comprehend the distributed transaction processing paradigm and how to use the two-phase commit technique when managing distributed transactions[26]. This paper offers a thorough investigation of these ideas. The drawbacks of the conventional two-phase commit mechanism in handling distributed transactions are also examined in this paper. The distributed database of a news system is given as an example to show these restrictions [27].

2.15 A Reference Architecture for Saga Pattern Microservices for Elastic SLO Violation Analysis

The microservice saga pattern is implemented in this paper's proposal for a self-adaptive microservice reference architecture. In order to explain self-adaptation and the spread of service-level objective (SLO) breaches throughout an architecture with complicated patterns, the reference architecture seeks to serve as a benchmark [57]. The Eventuate Tram saga

orchestration framework is used, and the architecture is developed in Java Spring Boot. For proper system execution and modifications, it includes monitoring, asynchronous, lightweight communication, and system-wide tracing. In terms of out-of-date technology, a lack of lightweight asynchronous communication, a lack of self-adaptation, and a lack of comprehensive coverage of microservice patterns, the study emphasises the shortcomings of current reference designs. Modern technologies, the saga pattern, self-adaptation, performance analytics [58], and load profiles are all included in the suggested T2-Project reference design to solve these restrictions.

The architecture has eight services, including UI, Inventory, Cart, Order, Orchestrator, Payment, and CreditInstitute, and it uses a database-per-service design pattern. Both synchronous REST APIs and asynchronous messaging using Apache Kafka are used by the services to exchange information. The Custom Auto Scaler (CAUS) is used by the Kubernetes-deployable T2-Project for self-adaptation. Additionally, it connects with Zipkin for distributed tracing and Prometheus for monitoring. The TeaStore and SockShop reference architectures are discussed in the article, along with the reasons they weren't modified for the suggested design [59]. The T2-Project's commercial operations, which include choosing items, placing orders, managing payments, and directing the narrative, are also described. The architecture and implementation of the T2-Project show how different patterns and frameworks may be used to assist self-adaptation and take SLOs into account.

2.16 A review of saga frameworks for distributed transactions that are implemented using event-driven microservices

The implementation of distributed transactions using the Saga pattern for event-driven microservices is the main topic of this article. With the use of events for inter-service communication, event-driven microservices provide scalability and autonomous development [54]. The microservice design recommends employing sagas, which are a collection of local transactions constituting a distributed transaction, as opposed to conventional 2-phase commits, which are unworkable in this asynchronous communication paradigm. Although they need careful design and error management throughout each microservice, sagas eventually give consistency. The asynchronous nature of interactions makes it particularly

difficult to implement sagas in event-driven microservices [55]. The study suggests leveraging tried-and-true frameworks rather than creating sagas from scratch in order to deal with these complications.

The document lists the saga frameworks that are currently supported by Java, Python, and NodeJS. It emphasises the necessity of a vendor-neutral abstraction to divide the business layer from the transaction layer and make saga implementation easier. The transition from monolithic systems to microservices to fulfil scalability requirements is highlighted in the introductory section. Microservices are tiny services that are related to particular domains and help businesses efficiently meet Time to Market criteria. Transactions frequently span several microservices [56], and synchronous REST interface implementation results in distributed monoliths. Distributed transactions in microservices were addressed by the Saga design, which permits momentary local locks and non-blocking event-based communication.

2.17 Sagamas: a framework for transactions in the distributed microservice architecture

SagaMAS, a multi-agent development framework especially created to manage dispersed transactions inside the microservices architecture, is described in this article as it is introduced. A distributed system is divided into several highly linked and autonomous services using the microservices design, providing for flexibility and scalability [28]. As a result, the system becomes heterogeneous since various implementations and data persistence techniques may be used by various microservices. In such a varied and scattered architecture, managing distributed transactions that span several microservices can be difficult. The SagaMAS framework offers efficient distributed transaction management as a response to this problem [29]. The framework is still in its early stages, thus it has to be followed and built upon in the following stages, which include the execution of the architectural design and the full system design.

Although SagaMAS is intended to be functionally deployed, careful testing and problem-solving are required to guarantee its dependability and effectiveness. By utilising Multi-Agent Systems (MAS), the framework enables the creation of advanced artificial intelligence-based negotiation and agent reasoning systems. This creates possibilities for improving several process-related elements in the distributed system. Developers may enhance transactional workflows, negotiation procedures,

and overall system performance by utilising SagaMAS and its intelligent agent-based methodology [30]. The framework shows potential for properly managing complicated distributed transactions in the architecture of microservices. But in order to realise its full potential and guarantee its effective integration inside various distributed systems, more development, testing, and optimisation are required.

2.18 State of practices, difficulties, and future direction for research in data management in microservices

Due to their effectiveness in effectively dividing an application into small, autonomous services, enabling scalability, strong isolation, and specialisation of database systems based on the workloads and data formats of each service, microservices have grown in popularity as an architectural style for data-driven applications [31]. Nevertheless, despite the industry's growing acceptance of microservices, little study has been done on the status of the practise at the moment and the ongoing difficulties faced by practitioners in the area of data management inside microservices. We carried out a detailed analysis to fill this gap, which entailed analysing well-known open-source microservice applications, doing a complete literature review, and conducting an online survey to verify our findings [32].

More than 120 seasoned practitioners and academics responded to the poll, and their perspectives and experiences contributed to strengthen and confirm our findings. We were able to discover numerous basic issues that require system-level support rather than just software engineering changes by categorising the existing state of data management in microservices [33]. These issues are intended to lighten the load on practitioners and offer useful information for enhancing data management in microservices [34].

2.19 A modified three-phase commit protocol for discrete systems concurrency control

An technique that explicitly addressed global abortion caused by a few unimportant sites was presented in the paper "An Extended Three Phase Commit Protocol for Concurrency Control in Distributed Systems" [35].

Four examples of the method's success were used to document it. This work, however, presents a more useful and all-encompassing strategy to treat worldwide abortion induced by both important (main) and minor (secondary) locations.

A brand-new table called the TIT (Transaction Identifier Table) has been included to the study to help with this. A single transaction used to be split up into several smaller ones and dispersed among several sites. However, even with the FLAG variable present, which provided the CONSISTENT or INCONSISTENT choices, it was unable to adequately solve the problem of abortion caused by big (main) sites.

The paper offers the Extension to Modified Three Phase Commit Protocol to address this issue[36]. Only single database object access transactions are permitted to use this protocol. Transaction abortion is considerably decreased by assuring the commitment of transactions that would have otherwise failed in the Modified Three Phase Commit Protocol. This not only raises the overall performance of distributed systems but also increases their dependability. The goal of the work is to offer a more useful and effective method for concurrency control in distributed systems. It solves the shortcomings of prior strategies and provides a complete solution to prevent transaction abortion caused by both major and minor sites by introducing the Extension to Modified Three Phase Commit Protocol and using the TIT table[37].

2.20 Rapid: a real time commit protocol

If write transactions try to access data that has already been accessed by read transactions [38], they run the danger of starvation in distributed systems. When priority inversion happens concurrently, the issue is made worse. It becomes much more difficult if the proposed write transaction has a higher priority level than the lock-holding read transaction with the highest priority from the group that is presently reading the conflicting data. The Reads-Write Avoid Starvation and Priority Inversion Period Decreased (RAPID) commit protocol provides a workable solution to this problem.

By cancelling certain reader transactions that have either been requested or are currently holding the conflicting data item in read mode, the RAPID protocol resolves the issue [39]. As a result, priority inversion is avoided and write transactions are not starved. The Priority Inheritance Commit (PIC) protocol faces difficulties with the Reads-Write dilemma because it takes time for people to understand the idea of prior-

ity inheritance. In the MR-SWO (Multiple Readers-Single Writer Optimisation) situation, a conflict between read and write operations causes an execution-committing conflict.

The problems of unbounded priority inversion and hunger brought on by MR-SWO conflicts are successfully handled by the RAPID protocol. The system performance is enhanced by lowering the size of the linked list of records related to the data items.Distributed systems can reduce the issues with write transaction starvation, priority inversion, and conflicts between read and write transactions by adopting the RAPID protocol. This protocol offers a workable approach to guarantee equitable access to data, improve system efficiency, and shield distributed systems from the harmful effects of concurrent read and write operations.

2.21 Design and implementation of three phase commit protocol (3PC) algorithm

A distributed database system is a sophisticated system that combines many databases to meet organisational and technological needs. Replication, transparency, and fragmentation issues, which are all closely tied to the communication network, are made more difficult by this integration [40]. To enable effective data collection, retrieval, and storage activities inside distributed database systems, these issues must be resolved. Diverse strategies are used in the context of homogenous distributed database systems, when the participating databases have comparable traits. These methods try to deal with problems including load balancing, query optimisation, and data consistency [41]. These systems may provide smooth integration and interoperability across the remote databases by employing a homogenous methodology.

The coordination of transactions among the participating databases is a key component of distributed database systems. The Three Phase Commit (3PC) Protocol and the Two-Phase Commit (2PC) Protocol are used to accomplish this. The 3PC Protocol, which adds a second cycle of message transmission, beats the 2PC Protocol in terms of system performance, according to our research, which compared the two protocols. A vote commit message is always transmitted to the receiving side of the 3PC Protocol. By adding this stage, the transaction coordination process becomes more reliable and resilient, lowering the possibility of errors or inconsistencies. The 3PC Protocol overcomes some of the shortcomings of the 2PC Protocol by including an additional phase, eventually

resulting in enhanced system performance.

2.22 Non-blocking one-phase commit made possible for distributed transactions over replicated data

Through incremental processing and materialised view maintenance, distributed transactions can handle both large-scale commercial transactions and quick in- formation extraction from massive amounts of data in highly scalable datastores. Thanks to incremental processing and materialised view maintenance, distributed transactions can manage large-scale business transactions and quickly extract information from vast volumes of data in highly scalable datastores.

A fundamental component of improving the commitment process is Pronto, a framework designed to provide universal distributed transaction commitment in massively scalable datastores that shard and duplicate data. It is critical to contrast Pronto with other protocols like 2PC (Two-Phase Commit) and RCommit in order to evaluate the effectiveness of commitment methods. The quantity of participants and data items in a transaction has a significant impact on how well commitment methods operate. The objective of Pronto is to demonstrate its superiority in terms of commitment performance by analysing the distinctive capabilities of each protocol.

The architecture of Pronto, which uses a non-blocking commitment approach for distributed transactions over replicated data, is the specific subject of this paper. This strategy guarantees effective transaction processing without sacrificing the underlying datastores' capacity to scale. An examination utilising an expanded YCSB (Yahoo! Cloud Serving Benchmark) benchmark is done to confirm Pronto's efficacy. The outcomes reveal that Pronto works better than cutting-edge commitment mechanisms, highlighting its resilience and scalability. When it comes to commit performance, Pronto continuously outperforms other commit tools even as the number of transaction participants rises. It differs from conventional protocols in that it can manage sharded and replicated data, which makes it an invaluable tool for massive commercial transactions and data extraction operations.

2.23 Improvising two phase-commit protocol

The Two-Phase Commit (2PC) approach is frequently used in distributed systems to guarantee the consistency of an atomic transaction across several peers. In this paper, we suggest a 2PC technique improvement designed particularly for setups of repeating state machines [42]. In our method, if a higher priority state machine wants to commit a value that has to be recorded, state executions in that state machine are prematurely terminated. When compared to the conventional 2PC technique, our method dramatically reduces the number of state executions needed by using this process, especially in situations involving more than three replicates.

We add an additional step, known as the M step, which comes after the READ, CHECK, and WRITE operations in order to clarify the spacing between the check and commit phases. Before committing the ultimate value, this interim stage offers the chance to assess and resolve problems. The inclusion of an early write conflict control mechanism into the 2PC protocol, as a result of extensive testing, has been found to effectively reduce the cost associated with unnecessary state executions that may result from concurrent modifications made to the same object across different computers.

2.24 Study development of e-commerce website

Businesses who use business-to-customer (B2C) e-commerce properly can benefit greatly. Businesses may improve customer happiness, boost sales, save administrative costs, and speed up product delivery by using effective B2C e-commerce solutions. The creation of designs, interfaces, and content that adhere to a system's minimal requirements requires the modification of HTML, CSS3, JavaScript, and Bootstrap modules.

E-commerce is the practise of making purchases, sales, and money or information transfers through a network, most frequently the internet. However, e-commerce success has its own unique set of difficulties. Online identity verification, which is essential for building confidence and security in online transactions, is one of the main problems. A dependable payment system must also be included for efficient financial transactions.

Chapter 3

REQUIREMENT ANALYSIS

3.1 Functional Requirements

- **Registration :** The user will be able to register on the site.
- **Login :** Now the user will login through the details provided during registration.
- **Wallet :** User will be pre provided a wallet where the user can add money.
- **Add to Cart :** User can select desired items and add it to cart.
- **ACID property across Transaction :** ACID property would be maintained across transaction using SAGA.

3.2 Hardware Requirements

- Processor : Core-i5 1123HE
- Speed : 2.3 GHz
- GPU support.
- Ram : 8 GB
- Hard Disk : 128 GB
- Keyboard : Standard Windows Keyboard
- Mouse : Two or Three button mouse
- Monitor : LCD/LED

3.3 Software Requirements

- Technologies : SAGA distributed transaction pattern.
- NodeJs
- JavaScript
- ReactJS
- DBMS
- PostgreSQL
- REDIS database server
- APACHE KAFKA

3.4 Use Case Model

User Case 1 : Registration
Primary Actor : User
Pre-Conditions : Internet should be available
Main Scenario :

- Terms and conditions are digitally signed by the user.
- Valid personal information is filled into the registration form.
- The registration form is submitted for verification, and once it gets verified, the user is entered in the Marketplace.
- User account is Created.

Alternate Scenario : Network failure. Now, the user needs to try again when a secure internet connection is re-established.

User Case 2 : Login
Primary Actor : User
Pre-Conditions : The user should have an existing account.
Main Scenario :

- Open the website.
- Fill in your username and password.
- Authentication is done

- User is directed to the main screen or Marketplace.

User Case 3 : Check Product Details
Primary Actor : User
Pre-Conditions : Internet should be available. Database should have Product that user wants to see
Main Scenario :

- User makes use of Search to search the Products he wants to buy.
- Then if Product is available, it shows the details of the Product with Buy Product option.

Alternate Scenario : If Product is not there in the Item Database it will fetch details from other sites and show it to the user so that he can purchase it.

User Case 4 : Create Order
Primary Actor : User
Pre-Conditions : User should be logged in to the website and should select at least one item.
Main Scenario :

- User must select his desired items.
- User must add all the selected items to cart.
- They will be shown option to create order.

Alternate Scenario : The internet stops working while the user starts the transaction, the user is then prompted to re-establish an internet connection and then repeat use case 4.

3.5 Non-Functional Requirements

- **Security :** Data provided by user during registration should be kept secured.
- **Performance :** System should perform well even on load.
- **Compatibility :** System should be compatible across different platoforms.
- **Easy to Use :** User should be able to easily understand the UI and use the system.

Chapter 4

PROPOSED SYSTEM AND ITS MODULES

4.1 Distributed Transactions in Microservices

The monolithic programme was reorganised into numerous distinct services using the microservice design. The usage of a single shared original core database, which further complicates scalability and one point failure difficulties, is the most frequent problem with classic monolithic applications [6]. A distributed system with several services that are invoked sequentially or concurrently to complete the whole process might be considered as a microservice architecture [2].

Transactions must traverse several databases since the database per service pattern may be used thanks to the microservice design. The handling and implementation of distributed transactions that ensure data consistency as well as rollback operations are crucial issues that must be taken into account with a microservice architecture. The implementation patterns for transactions in a microservice architecture that are distributed and are outlined in the ensuing subsections.

4.2 Two-Phase Commit

The two-phase commit protocol (2PC) is most often used techniques to perform transactions in a microservice architecture which are distributed [7]. The coordinator is the element that manages transactions in this protocol (Figure 1), while the microservices (participant nodes) carry out their local operations. A distributed transaction is carried out in two steps using the 2PC protocol. The coordinator requests that the partici-

pating nodes commit the transaction during the first phase, also referred to as the preparation phase. As a result, just a yes or no response will be given. When each participating node responds “yes” to the coordinator’s question during the second phase (commit phase), the coordinator then asks each node to commit.

Note that after getting at least one negative answer, the coordinator requests that everyone roll back their local transactions. The coordinator in orchestrator may become a single failure point even if the 2PC is seen to be a suitable method for implementing distributed transactions [8]. Additionally, because all other rest of services must wait and keep patient until the slowest service completes its work and they recieve their confirmation, the total performance of the transactions depends on it. As a result, it doesn’t function well in systems that are vast and heavily loaded[9].

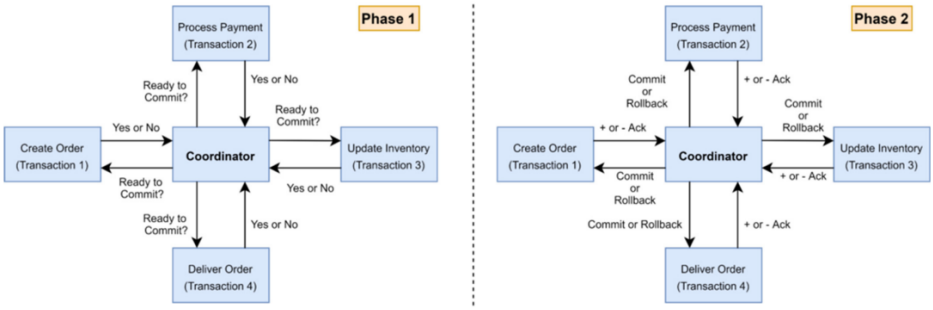


Figure 4.1: Two-Phases commit Protocol

4.3 Pattern of SAGA

The saga pattern was created to organise the communication involved in a microservices architecture and overcome the problems with the 2PC protocol [3,10]. The SAGA [11], project investigated both the formal and the practical, was first introduced by Campbell and Richards in 1981. For the purpose of enabling the construction of a feasible software development environment, features of computer assisted assistance for the lifecycle of software are needed. Garcia-Molina and Salem [10] It was first suggested to structure long-lasting transactions as a series of local transactions, where each database update broadcasts an event or message that triggers the following local transaction. To update data across different services in a microservices architecture without utilising distributed transactional databases, the saga has been introduced, in other words. In the series of compensating transactions will be carried

out by Saga to redo the made changes by the previous transactions done at local level when one local transaction fails due to noncompliance with the business rules.

The saga was characterised by the authors in [10] as a series of actions that each carry out a specified task and are often performed in succession. Saga operations may be reversed by a compensatory action. The Saga pattern guarantees either the valid operation of all the execution of the appropriate compensating actions for all conducted operations to undo any previously completed work. According to the saga pattern, large transactions will be divided into a number of brief transactions that commit one after the other in order to avoid long transactions securing locks[10].

The two most frequent methods for synchronising the tale pattern are generally choreography and orchestration. With choreography, sagas are organised without a single point of control where players may trade occurrences. Local transactions broadcast domain events that cause other services to do local transactions. The orchestration, on the other hand, is a further method of coordinating sagas in which sagas are coordinated with a centralised controller who instructs saga participants as to what local transactions to carry out depending on the events. Saga requests are carried out by the orchestrator, which also stores and interprets task statuses. The orchestrator manages failure recovery for compensatory transactions.

The authors of [14] presented a formal model-based method for microservices and service integration utilising the UML and UML profiles. A microservice-based system can implement distributed transactions by using the saga architectural pattern, according to earlier research. However, rather of ACID (atomicity, consistency, isolation, durability), the saga pattern is ACD [3]. Because read-isolation is absent in the saga design, data from uncompleted transactions can be read and written [4], and In Saga, each microservice updates its own database. Read-isolation is lacking, which affects durability. Counter measures for reducing anomalies must be part of the saga implementation [13].

4.4 E-Commerce Workflow

The event process for an e-commerce microservices-based system is shown in Figure. The system enables online product purchases and gives users the choice of items, payment options, and shipping methods. The various services are some of the microservices that make up this long-lasting

transaction. To emulate real-world e-commerce applications, the system incorporates both the warehouse-before-billing and the billing before-warehouse processes, as shown in Figure [15]. The Warehouse-Service, which gets the products, initiates the flow. The Order-Service then creates a blank order that is designated as "IN-PROGRESS." When the items cannot be retrieved, the order will be listed as "FAILED." The selected payment is then verified by the Billing Service. If the validation is successful, the billing service will take the money.

If not, it stops the flow and marks the order as "FAILED." The delivery is dispatched by the shipping service. In the end, the order is completed by the order service, which also updates the status, shipping ID, and total.

4.5 Approach and it's components

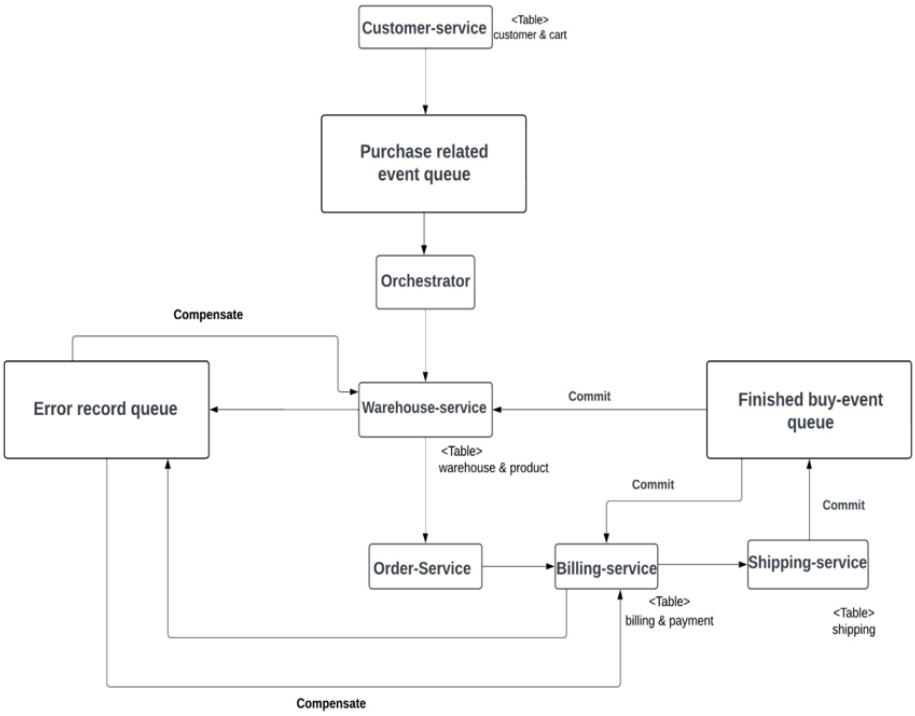


Figure 4.2: Architecture Diagram

The suggested augmented saga pattern's backend elements are illustrated in this section. The e-commerce system's architecture and tech stacks are depicted in Figure, and its workflow is explained in Section. The implementation method used the clean architecture methodology

[16]. The use cases for every domain have all been developed beforehand. For instance, the Billing-Service contains the following use cases: add-payment, create-billing, validate-payment, payment-pay, and revert-payment-pay.

4.6 Microservices in intergrated system

The WarehouseService, OrderService, BillingService, ShippingService, and CustomerService are the five microservices that make up the system. There is a separate database for each microservice. The spring boot technology was used to create the microservices, which inherited and implemented the pertinent use cases [17]. Internally, the microservices were exposed via the REST API [18,19], enabling communication via the straightforward HTTP protocol.

4.7 Quota Cache

In computing, a cache is typically a piece of hardware or software that stores data. A high-speed data storage layer is what it is called. A cache improves data retrieval performance by minimising main memory accesses. The quota of a particular resource from the primary database is stored in the quota cache (see Figure). A feature called quota calculates the number of bytes—or the amount of storage space—that are accessible.

In caching of data in memory, is used to address the isolation problem for read operation, property in the saga design. The memory online cache database server will be given a certain amount of space from the primary database. The quota cache will be used to handle the CRUD actions. The main database will never get an incorrect commit as a result. Additionally, the in-memory operations will help the microservices that use the quota cache ensure lower latency and high output, resulting in better performance compared to the baseline standard saga.

The quota cache is used by microservices that need validation, as shown in Figure, since validation errors may lead to exceptions. When a microservice fails to complete, the other microservices will undertake compensation transactions to reverse the alterations. In the event that the Warehouse-Service makes a mistake when getting the goods, the first good fetching will update the main database, which would allow a client to see the newly formed order. The order will be cancelled in the next few seconds when the compensation transaction is executed to undo the alterations. The CRUD operations are transferred from the main

database level to the cache level in the improved version, in contrast.

4.8 Commit Sync Service

An ultimate commit sync service is necessary to accomplish this synchronisation. Before committing the changes to the database, this service makes sure that any events or operations that have been carried out while utilising the cache quota have been properly finished. The "done message" is not sent to the message queue middleware until all events have occurred and the system has verified their success.The microservices that have used up their cache quota then fetch the event record and carry out the database commit after receiving the "done message". In essence, this indicates that the database will not be committed until each successful event has been finished.

4.9 Orchestrator Module

In order to manage and coordinate the microservices inside a system, the orchestrator module is essential. In this instance, the RxJava package, a well-liked toolkit for creating asynchronous and event-driven systems based on the observer design pattern, has been used to organise the microservices in the orchestrator module[23]. The microservices may be organised as observers thanks to the observer design pattern, which enables them to respond to system events and changes. The orchestrator module may effectively manage and control the microservices by using this approach.

4.10 Main Database

The primary database was created using PostgreSQL [24,25]. Java application development can use the open-source object-relational database PostgreSQL.

Chapter 5

DETAIL DESIGN

5.1 Architecture Diagram

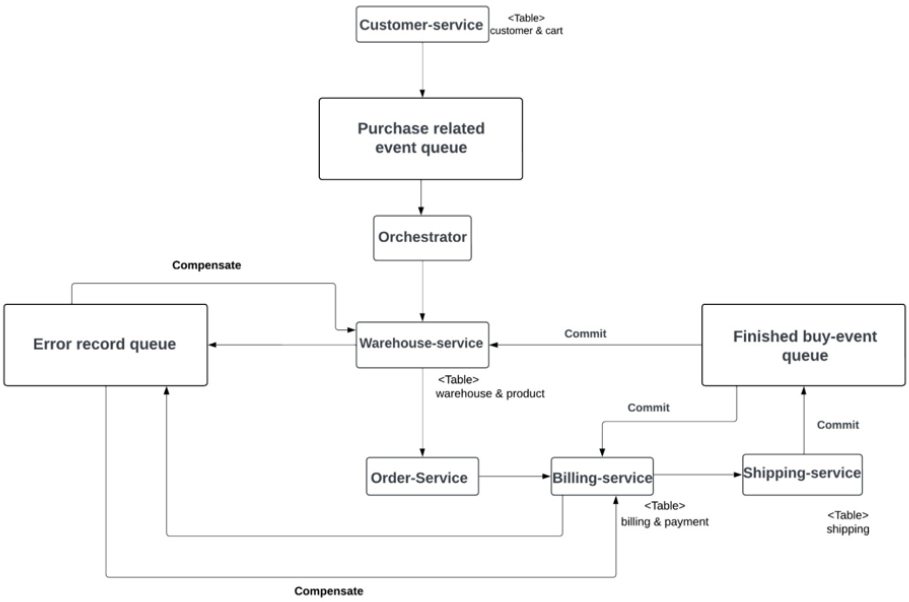


Figure 5.1: Architecture Daigram

5.2 Activity Diagram

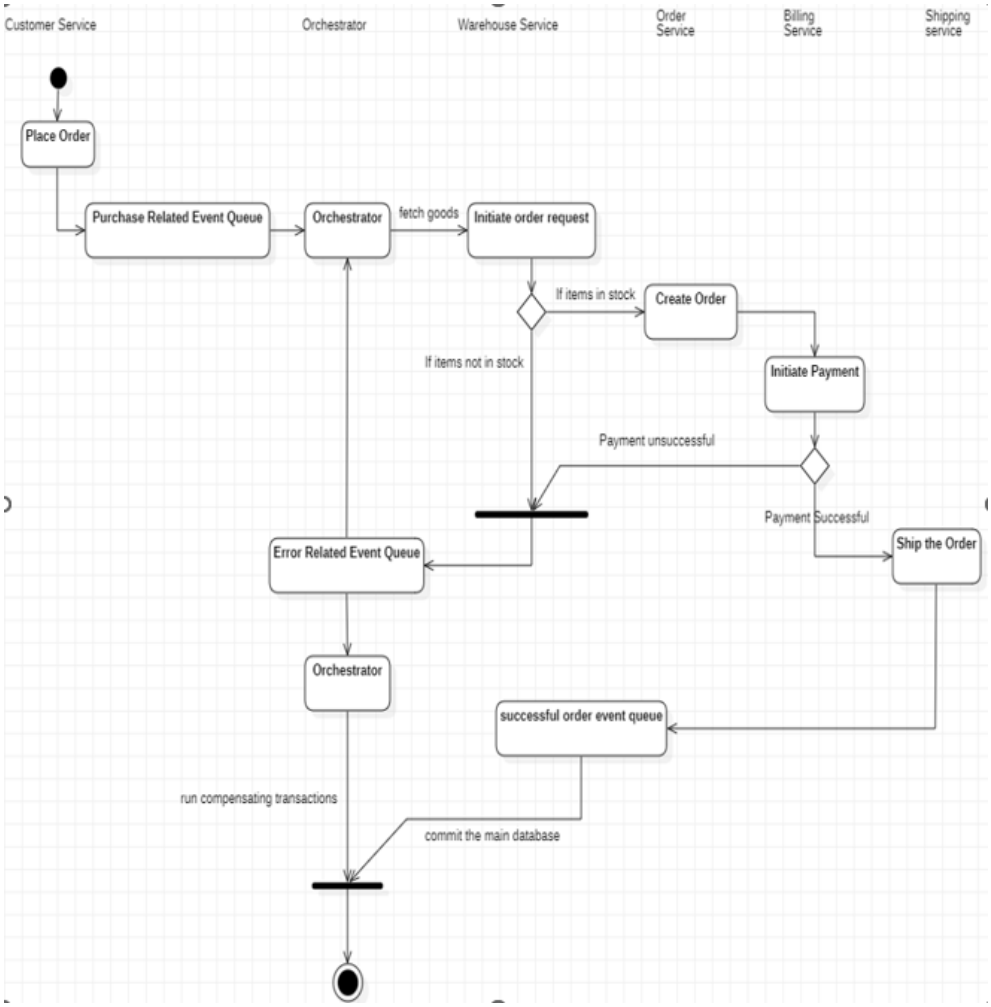


Figure 5.2: Activity Diagram

5.3 State Diagram

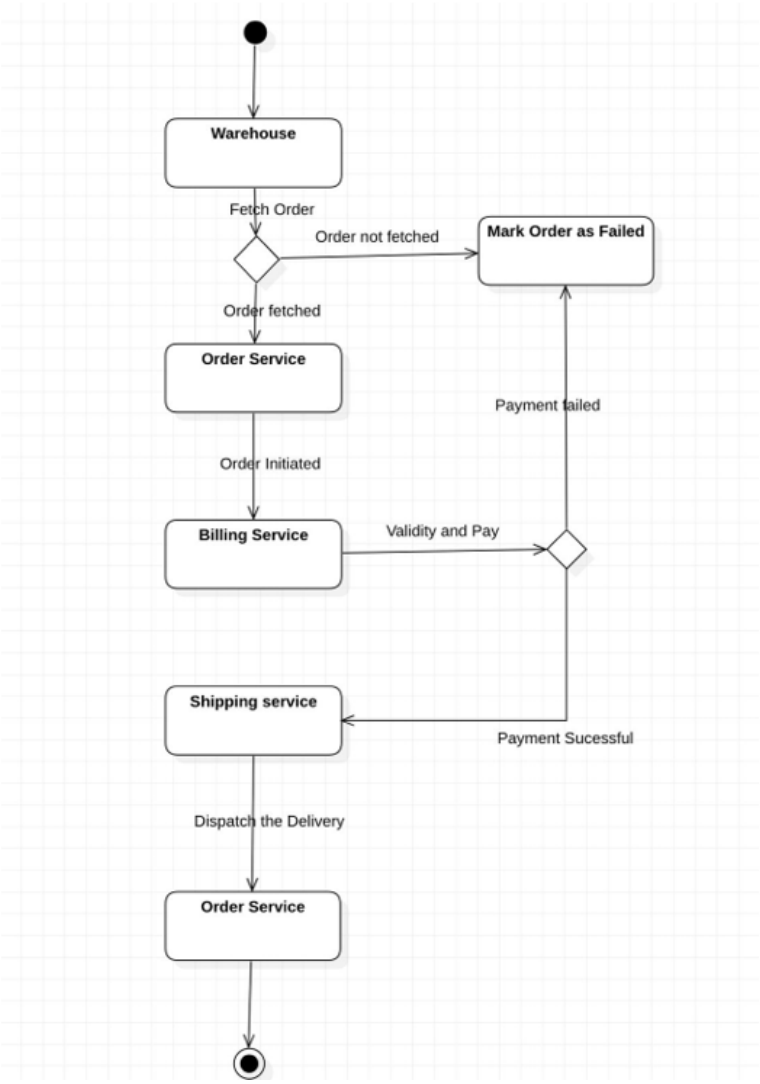


Figure 5.3: State Diagram

5.4 Class Diagram

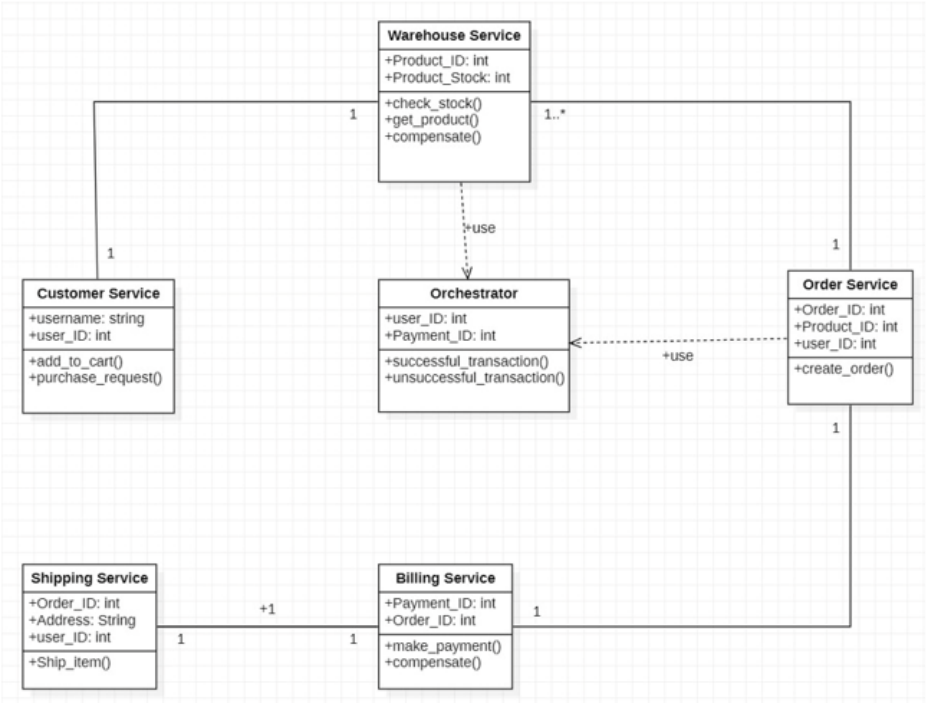


Figure 5.4: Class Diagram

5.5 Use Case Diagram

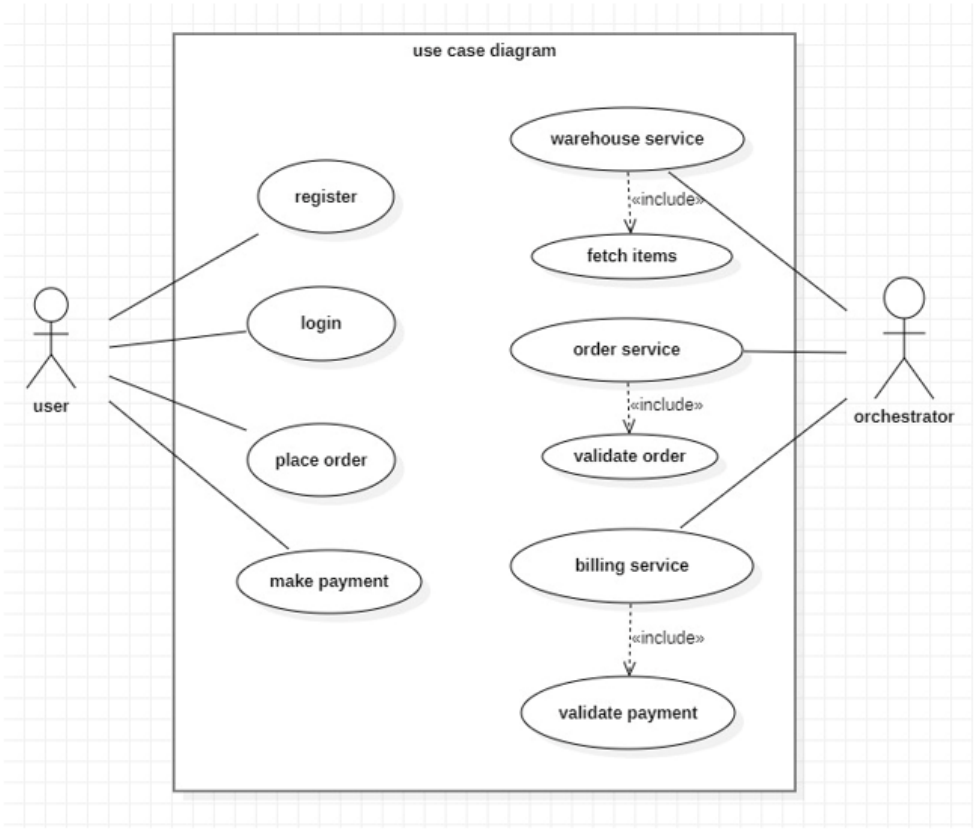


Figure 5.5: Use Case Diagram

Chapter 6

CODING

6.1 Algorithm

SAGA

Data consistency across microservices in dispersed transaction settings may be managed using the Saga design pattern. A saga is a series of transactions that update every service and publish a message or event to start the following transaction step. The saga conducts compensatory transactions that reverse the previous transactions if a step fails.

A transaction is a discrete chunk of logic or labour that occasionally consists of several operations. An event happens when the status of an entity changes during a transaction, and a command contains all the information required to carry out an action or start a subsequent event. Atomic, consistent, isolated, and durable transactions (ACID) are required. Although transactions within a single service are ACID, a cross-service transaction management strategy is necessary to ensure data consistency.

Using a series of local transactions, the Saga pattern offers transaction management. The atomic work effort put out by a saga participant is known as a local transaction. Every local transaction in the saga modifies the database and broadcasts a message or event to start the subsequent local transaction. A sequence of compensatory transactions are carried out by the system in the event that a local transaction fails, undoing the modifications that the previous local transactions made.
In SAGA patterns :

1. Compensable transactions in Saga patterns are those that could be undone by performing an other transaction that has the opposite result.

2. The go/no-go decision point in a tale is a pivotal transaction. The saga continues until it is finished if the pivot transaction commits.

3. Retryable transactions are transactions that come after the pivot transaction and are guaranteed to succeed. A pivot transaction can be a transaction that is neither compensable nor retryable or it can be the final compensable transaction or the first retryable transaction in the saga.

Enhanced SAGA pattern

Enhancing saga with Redis cache server so that the complete sequence of transactions takes place on cache server and the final commit is performed once any transaction is successfully completed or if it fails then the transaction is rolled back on Redis database and hence np changes or rollbacks take place on main database and hence it prevents uncommitted read and write.

This part provides examples of how the previous system is used in practise (for more information, check the Supplementary Materials section). Two versions of the system will be put into use for comparing purposes later. The upgraded saga pattern through quota cache and eventual commit sync service is used in the second version, which employs the baseline standard saga pattern in the first. The spring boot framework serves as the foundation for the implementation. The microservices interact with one another via the REST API, which enables the other services to adhere to the REST standards. The numerous events, including the buy events, completion events, and failure events, were also handled by a number of message queue middleware. These message queues provide increased system throughput.

There are two techniques to coordinating sagas in the microservices architecture: orchestration and choreography. For implementation, we used both orchestration and event choreography methodologies in our proposal.

The manager controller oversees all microservice communications in the orchestration-based saga. the orchestrator module in our approach is in charge of informing the associated microservice about which transactions must be conducted.

As a result, the process order may be readily managed without modifying any microservices. When the orchestration module detects a "buy-event,"It gives the Warehouse-Service the order to start recovering goods.

Additionally, the orchestration manages asynchronously occuring failure and completion events via the message queue middleware. The orchestration delivers a "complete-event" to the message queue middleware when all transactions are successfully finished, enabling the WarehouseService and the BillingService to consume it and commit the database.

The choreography service composes services in a decentralised manner. In our proposal, after a microservice completes its local transaction, it will publish domain events to which the other microservices will subscribe in order to trigger their local transactions.

This version makes use of all of the back-end components, including the memory cache server and the message queue middleware. The memory cache server and the message queue middleware, as shown in the figure, only apply to the WarehouseService and the Billing-Service because only these two microservices can perform rollbacks when an error occurs in the system (i.e., the fetched goods and the collected payment should be reverted in the case of exception).

The BillingService verifies the customer's payment and collects it in the memory layer, the ShippingService sends the delivery for a customer, and the Order-Service completes the order. The Warehouse-Service utilises Redis to retrieve items in the memory layer. The system will simultaneously post the finished event to the related message middleware. The subscription services will use it and then commit the desired change to their primary database. In the case of an error, this system also provides compensation transactions that are carried out.

Compensation transactions only affect the allocated quota cache and not the primary database (i.e., the memory layer as opposed to the disc). Additionally, to maintain eventual consistency, the message queue middleware handles the database commit at the conclusion of the process after being handled by the microservices that use the quota cache. This means that the database commit will only be carried out if all of the events involved have been properly completed by using the quota cache and the eventual commit sync service.

6.2 Software used

- **Kafka message queue middleware :** Middleware that supports sending and receiving messages between distributed systems is known as message-oriented middleware (MOM). The complexity of creating applications that run across various operating systems

and network protocols is reduced by MOM, which enables the distribution of application modules across heterogeneous platforms. The application developer is shielded from the specifics of the multiple operating systems and network interfaces by a dispersed communications layer that the middleware develops.

- **Redis cache server** An open-source, in-memory data structure store called Redis Cache is utilised as a database, cache, and message broker. Data may be stored in and retrieved from memory using this key-value store. Redis Cache is used to speed up web applications by temporarily storing frequently requested data.

6.3 Hardware specification

- **Processor:** Core-i5 1123HE
- **Speed:** 2.3 GHz
- GPU support.
- **Ram:** 8 GB
- **Hard Disk:** 128 GB
- **Keyboard:** Standard Windows Keyboard
- **Mouse:** Two or Three button mouse
- **Monitor:** LCD/LED

6.4 Programming language

Nodejs for backend

- The open source, cross-platform runtime environment known as Node.js (Node) allows JavaScript code to be run. Because Node is frequently used for server-side development, developers may utilise JavaScript for both client-side and server-side programming without needing to learn another language. Node is sometimes referred to as a framework for software development or a programming language, however neither of these definitions are appropriate; it is merely a JavaScript runtime.

- Node.js is event-driven and runs asynchronously. Code created for the Node environment does not follow the typical receive, process, send, wait, and receive model used in other systems. Instead, Node processes incoming requests as they pile in the event queue, handling minor requests progressively without waiting for responses.

React Js for frontend

- A JavaScript framework called ReactJS is used to build modular user interface elements. The definition that follows is taken straight from the React documentation.

- User interfaces are built using the React framework, which is modular. It encourages the creation of reusable user interface elements that show dynamic data. React makes use of the JavaScript object known as Virtual DOM. App speed will improve since JavaScript's virtual DOM is faster than the traditional DOM. It may be used on both the client and server sides and is compatible with a variety of frameworks. By enhancing readability, component and data pattern implementation features help maintain larger programmes.

6.5 Components

Microservices

The system consists of five microservices: the order system, the billing service, the shipping service, and the customer service. There is a separate database for each microservice. The essential use cases were passed down to and implemented by the spring boot technology, which was used to create the microservices. Internally, the microservices were accessible via the REST API; as a result, communication is possible using the straightforward HTTP protocol.

Message queue middleware

As the middle of the message queue, Apache Kafka was used. An open-source distributed event streaming system called Apache Kafka may be used to publish and subscribe to message streams. Kafka's ability to process thousands of messages per second allows it to be used to develop applications with high throughput. Kafka always stores messages on the disc for persistence since it possesses the durability property.

Cache Server Redis

In computer, a cache is often a piece of hardware or software that stores data. A high-speed data storage layer is what it is called. A cache improves data retrieval performance by minimising main memory accesses. The quota of a particular resource from the primary database is stored in the quota cache. The quota functionality calculates the number of bytes that can be used to store material.

Microservices that need validation use the quota cache because errors might happen if the validation is unsuccessful. The other microservices will conduct compensation transactions to undo the modifications when a microservice fails to finish. In the event that the Warehouse-Service makes a mistake when getting the goods, the first good fetching will update the main database, which would allow a client to see the newly formed order. The order will be cancelled in the next few seconds when the compensation transaction is executed to undo the alterations. The CRUD operations are transferred from the main database level to the cache level in the improved version, in contrast. This won't result in an incorrect database change or commit. In the event of an error, the compensation request will be delivered to the relevant message queue middleware.

Orchestrator module

Sagas can be coordinated via orchestration, in which a centralised controller instructs the participants in the saga what local transactions to carry out. The saga orchestrator manages all transactions and directs participants in accordance with occurrences as to which operation to carry out. The orchestrator manages failure recovery with compensatory transactions, performs saga requests, and maintains and interprets each task's state.All microservices can be managed easily and the workflow can be changed as needed thanks to the orchestrator module.

Chapter 7

TESTING

The SAGA pattern efficiently manages transaction across distributed system containing multiple microservices, but it faced a problem of uncommitted read and write which was solved using enhanced SAGA pattern which has extra support from the cache database server that helped the SAGA pattern to have the complete series of transaction required to complete a whole transaction to take place on the cache database server and not on the main database and hence the changes in the main database are committed only and only if whole transaction is completed successfully, if the transaction at any stage meets a failure or records an error than the previous steps are rolled back on the cache database only and hence the problem of uncommitted read and write was solved.

The system testing was done based on its working and as the objective claims the testing was done based on two scenarios:

- **Scenario 1 :** when any transaction among the complete sequence of transaction does not fail or incur any error and the whole sequence of transaction is completed successfully.
- **Scenario 2 :** when any transaction among the complete sequence of transactions fails or incurs an error such as insufficient balance in case of billing service.

And the whole sequence of transaction is not completed and there is a need to roll back the previous transaction steps.

Chapter 8

RESULT

Experiment 1: Testing the Baseline Standard System (version 1: Utilizing the standard saga pattern) with Scenario 1

The given scenario follows the SAGA pattern, which involves the orchestration of events by different microservices when the orchestrator module captures a "buy-event." Each microservice plays a specific role in the workflow to ensure a successful order process.

The first stage is executed by the Warehouse-Service, which is responsible for retrieving the required amount of goods. This operation is processed in the main database, and the "logger name" field indicates its execution.

After the Warehouse-Service completes its task, the Order-Service takes over. Its responsibility is to initiate the order based on the received information. The Order-Service coordinates with other microservices to ensure the successful execution of the order.

Next, the Billing-Service comes into play. Its main task is to validate the payment associated with the order. If the payment validation fails, indicating an issue with the payment, the workflow is terminated, and the order is marked as "FAILED." This step ensures that inconsistent or erroneous orders are not processed further.

In the scenario described, the payment validation is successful, allowing the process to proceed to the next phase. The Billing-Service proceeds to collect the appropriate payment for the order, ensuring that the financial transaction is properly handled.

Once the payment is successfully collected, the Shipping-Service takes charge of dispatching the delivery based on the customer's specified re-

quirements. This step involves coordinating with the relevant shipping providers and ensuring that the order is shipped to the correct destination.

Finally, the Order-Service completes the order by updating crucial information in the system. This includes updating the order status, shipment ID, quantity, and any other relevant details. By maintaining accurate and up-to-date information, the system ensures transparency and facilitates efficient order management.

Throughout this workflow, each microservice operates independently, focusing on its specific responsibilities. The SAGA pattern ensures that the events are coordinated and handled in a sequential manner. By breaking down the order process into smaller stages and leveraging the event-driven nature of the pattern, the system achieves consistency, reliability, and scalability.

This scenario represents a successful case without any errors. However, in real-world scenarios, the SAGA pattern also provides mechanisms to handle exceptional cases and compensating actions when errors or failures occur during the transaction process. These features contribute to the robustness and resilience of the overall system.

Experiment 1: Testing the improved System (version 2: Utilizing the enhanced saga pattern) with Scenario 1


```
start {
  product: [
    {
      quantity: 1,
      pid: '9',
      id: '644ff37ad32769f612fc9051',
      title: 'WD 2TB Elements Portable External Hard Drive - USB 3.0 ',
      price: 64,
      image: 'https://fakestoreapi.com/img/61IBBVJvSDL._AC_SY879_.jpg',
      rating: 3.3,
      description: 'USB 3.0 and USB 2.0 Compatibility Fast data transfers Improve PC Performance High Capaci
ty; Compatibility Formatted NTFS for Windows 10, Windows 8.1, Windows 7; Reformatting may be required for ot
her operating systems; Compatibility may vary depending on user’s hardware configuration and operating syste
m'
    }
  ],
  customer: {
    name: 'Test Eleven',
    username: 'test11',
    email: 'test11@test.com',
    customerId: '644837456fbc51fb166dc579'
  }
}
```


Figure 8.1: order request

In enhanced SAGA pattern also the Step 1 and Step 2 are nearly identical to original SAGA since only the Warehouse-Service and the Billing-

```
Saga started
{"level":"INFO","timestamp":"2023-05-25T14:22:11.863Z","logger":"kafkajs","message":"[Consumer] Starting","g
roupId":"saga-consumer"}
===> message received { index: 0, phase: 'STEP_FORWARD' } payload {
  product: [
    {
      quantity: 1,
      pid: '9',
      id: '644ff37ad32769f612fc9051',
      title: 'WD 2TB Elements Portable External Hard Drive - USB 3.0 ',
      price: 64,
      image: 'https://fakestoreapi.com/img/61IBBVJvSDL._AC_SY879_.jpg',
      rating: 3.3,
      description: 'USB 3.0 and USB 2.0 Compatibility Fast data transfers Improve PC Performance High Capaci
ty; Compatibility Formatted NTFS for Windows 10, Windows 8.1, Windows 7; Reformatting may be required for ot
her operating systems; Compatibility may vary depending on user's hardware configuration and operating syste
m'
    }
  ],
  customer: {
    name: 'Test Eleven',
    username: 'test11',
    email: 'test11@test.com',
    customerId: '644837456fbc51fb166dc579'
  }
}
```

Figure 8.2: SAGA initiated

Service implement the proposed approach. As demonstrated, data access was handled by Redis, the memory cache server, rather than the main database. In other words, no transactions on the primary database are required throughout these processes. As shown in figure 1 demonstrate the request of items the customer wants to buy.

Initially message is received in kafka message queue middleware with all necessary details such as product-id, number of items, customer-id, price, rating etc. as shown in Figure 2.

```
STEP1 FORWARD
{"level":"INFO","timestamp":"2023-05-25T14:22:11.940Z","logger":"kafkajs","message":"[ConsumerGroup] Consume
r has joined the group","groupId":"saga-consumer","memberId":"kafkajs-4d8f4266-bc12-4919-8928-c691af738aa8",
"leaderId":"kafkajs-4d8f4266-bc12-4919-8928-c691af738aa8","isLeader":true,"memberAssignment":{"order-complet
e":[0],"order-failed":[0]},"groupProtocol":"RoundRobinAssigner","duration":69}
res {
  product: [
    {
      quantity: 1,
      pid: '9',
      id: '644ff37ad32769f612fc9051',
      title: 'WD 2TB Elements Portable External Hard Drive - USB 3.0 ',
      price: 64,
      image: 'https://fakestoreapi.com/img/61IBBVJvSDL._AC_SY879_.jpg',
      rating: 3.3,
      description: 'USB 3.0 and USB 2.0 Compatibility Fast data transfers Improve PC Performance High Capaci
ty; Compatibility Formatted NTFS for Windows 10, Windows 8.1, Windows 7; Reformatting may be required for ot
her operating systems; Compatibility may vary depending on user's hardware configuration and operating syste
m'
    }
  ],
  customer: {
    name: 'Test Eleven',
    username: 'test11',
    email: 'test11@test.com',
    customerId: '644837456fbc51fb166dc579'
  },
  warehouseCheck: true
}
```

Figure 8.3: Warehouse Microservice - I

```
===> message received { index: 1, phase: 'STEP_FORWARD' } payload {
  product: [
    {
      quantity: 1,
      pid: '9',
      id: '644ff37ad32769f612fc9051',
      title: 'WD 2TB Elements Portable External Hard Drive - USB 3.0 ',
      price: 64,
      image: 'https://fakestoreapi.com/img/61IBBVJvSDL._AC_SY879_.jpg',
      rating: 3.3,
      description: 'USB 3.0 and USB 2.0 Compatibility Fast data transfers Improve PC Performance High Capaci
ty; Compatibility Formatted NTFS for Windows 10, Windows 8.1, Windows 7; Reformatting may be required for ot
her operating systems; Compatibility may vary depending on user's hardware configuration and operating syste
m'
    }
  ],
  customer: {
    name: 'Test Eleven',
    username: 'test11',
    email: 'test11@test.com',
    customerId: '644837456fbc51fb166dc579'
  },
  warehouseCheck: true
}
```

Figure 8.4: Warehouse Microservice - II

Step 1: as shown in Figure 3 the warehouse-service checks whether the desire items that the customer want to buy are in stock or not. And if the items that the customer wants to buy are in stock then it updates the warehouse-check status to true else it updates it as false. And after updating the warehouse-check status this status is received by kafka message queue middleware as a message as shown in Figure 4.

```
STEP2 FORWARD
res {
  oid: '30741c5f9d5e3ab3',
  product: [
    {
      quantity: 1,
      pid: '9',
      id: '644ff37ad32769f612fc9051',
      title: 'WD 2TB Elements Portable External Hard Drive - USB 3.0 ',
      price: 64,
      image: 'https://fakestoreapi.com/img/61IBBVJvSDL._AC_SY879_.jpg',
      rating: 3.3,
      description: 'USB 3.0 and USB 2.0 Compatibility Fast data transfers Improve PC Performance High Capaci
ty; Compatibility Formatted NTFS for Windows 10, Windows 8.1, Windows 7; Reformatting may be required for ot
her operating systems; Compatibility may vary depending on user's hardware configuration and operating syste
m'
    }
  ],
  customer: {
    name: 'Test Eleven',
    username: 'test11',
    email: 'test11@test.com',
    customerId: '644837456fbc51fb166dc579'
  },
  warehouseCheck: true
}
```

Figure 8.5: Order Microservice - I

Step 2: Now, the message that was received as shown in Figure 4 is sent to order-service where the requested order is provided with an unique order-id as shown in Figure 5. And then again the message is published to kafka message queue middleware containing a unique order-

```
===> message received { index: 2, phase: 'STEP_FORWARD' } payload {
  oid: '30741c5f9d5e3ab3',
  product: [
    {
      quantity: 1,
      pid: '9',
      id: '644ff37ad32769f612fc9051',
      title: 'WD 2TB Elements Portable External Hard Drive - USB 3.0 ',
      price: 64,
      image: 'https://fakestoreapi.com/img/61IBBVJvSDL._AC_SY879_.jpg',
      rating: 3.3,
      description: 'USB 3.0 and USB 2.0 Compatibility Fast data transfers Improve PC Performance High Capaci
ty; Compatibility Formatted NTFS for Windows 10, Windows 8.1, Windows 7; Reformatting may be required for ot
her operating systems; Compatibility may vary depending on user's hardware configuration and operating syste
m'
    }
  ],
  customer: {
    name: 'Test Eleven',
    username: 'test11',
    email: 'test11@test.com',
    customerId: '644837456fbc51fb166dc579'
  },
  warehouseCheck: true
}
```

Figure 8.6: Order Microservice - II

id as shown in Figure 6.

Step 3: After the second step where the requested order was provided with unique order-id now the billing-service is initiated and here validating the payment is transferred from the main database to the memory cache server and the payment status is updated as true if payment is done successfully and false otherwise as shown in Figure 7. And then again a message is published in message queue middleware that carries the payment status to shipping microservice as shown in Figure 8.

```
STEP3 FORWARD
res {
  oid: '30741c5f9d5e3ab3',
  product: [
    {
      quantity: 1,
      pid: '9',
      id: '644ff37ad32769f612fc9051',
      title: 'WD 2TB Elements Portable External Hard Drive - USB 3.0 ',
      price: 64,
      image: 'https://fakestoreapi.com/img/61IBBVJvSDL._AC_SY879_.jpg',
      rating: 3.3,
      description: 'USB 3.0 and USB 2.0 Compatibility Fast data transfers Improve PC Performance High Capaci
ty; Compatibility Formatted NTFS for Windows 10, Windows 8.1, Windows 7; Reformatting may be required for ot
her operating systems; Compatibility may vary depending on user's hardware configuration and operating syste
m'
    }
  ],
  customer: {
    name: 'Test Eleven',
    username: 'test11',
    email: 'test11@test.com',
    customerId: '644837456fbc51fb166dc579'
  },
  warehouseCheck: true,
  status: 'PAYMENT_SUCCESS'
}
```

Figure 8.7: Billing Microservice - I

Step 4: Now the message that was generated in step 3 is passed to

```
===> message received { index: 3, phase: 'STEP_FORWARD' } payload {
  oid: '30741c5f9d5e3ab3',
  product: [
    {
      quantity: 1,
      pid: '9',
      id: '644ff37ad32769f612fc9051',
      title: 'WD 2TB Elements Portable External Hard Drive - USB 3.0 ',
      price: 64,
      image: 'https://fakestoreapi.com/img/61IBBVJvSDL._AC_SY879_.jpg',
      rating: 3.3,
      description: 'USB 3.0 and USB 2.0 Compatibility Fast data transfers Improve PC Performance High Capaci
ty; Compatibility Formatted NTFS for Windows 10, Windows 8.1, Windows 7; Reformatting may be required for ot
her operating systems; Compatibility may vary depending on user's hardware configuration and operating syste
m'
    }
  ],
  customer: {
    name: 'Test Eleven',
    username: 'test11',
    email: 'test11@test.com',
    customerId: '644837456fbc51fb166dc579'
  },
  warehouseCheck: true,
  status: 'PAYMENT_SUCCESS'
}
```

Figure 8.8: Billing Microservice - II

shipping-service which checks whether the payment status and warehouse status are true or not and if both of them are true then and then only it would confirm the order and ship it and mark the order-status as success and also update the shipping status as dispatched as shown in Figure 9.

```
STEP4 FORWARD
res {
  oid: '30741c5f9d5e3ab3',
  product: [
    {
      quantity: 1,
      pid: '9',
      id: '644ff37ad32769f612fc9051',
      title: 'WD 2TB Elements Portable External Hard Drive - USB 3.0 ',
      price: 64,
      image: 'https://fakestoreapi.com/img/61IBBVJvSDL._AC_SY879_.jpg',
      rating: 3.3,
      description: 'USB 3.0 and USB 2.0 Compatibility Fast data transfers Improve PC Performance High Capacity; Compatibility Formatte
d NTFS for Windows 10, Windows 8.1, Windows 7; Reformatting may be required for other operating systems; Compatibility may vary depend
ing on user's hardware configuration and operating system'
    }
  ],
  customer: {
    name: 'Test Eleven',
    username: 'test11',
    email: 'test11@test.com',
    customerId: '644837456fbc51fb166dc579'
  },
  warehouseCheck: true,
  status: 'PAYMENT_SUCCESS',
  shipping_id: '099192eb24bafe01',
  shipping_status: 'DISPATCHED'
}
====> Saga finished and transaction successful
```

Figure 8.9: Shipping Microservice - I

Then the orchestrator module sends the completion event to the appropriate message queue middleware. When the Warehouse-Service and Billing-Service receive the "completion-event" message, they will execute the specified transaction against the main database. Both the Warehouse-Service and the Billing-Service have already transferred the update procedure to the memory cache server. As long as they receive the completion event message, it will complete the required transactions to the main database.

As shown in below mentioned figure 10, on successful completion of a complete transaction an JavaScript pop up is shown stating that the transaction has been completed successfully.

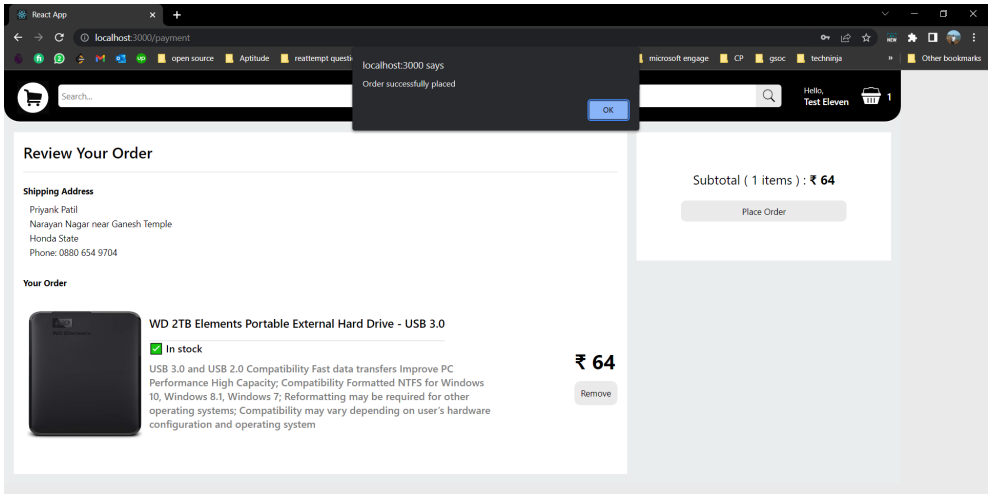


Figure 8.10: Order Placed Successfully

Experiment 2: Testing the Baseline Standard System (version 1: Deploying standard saga pattern) with Scenario 2

In this experiment, an e-commerce microservices-based system is used to demonstrate the event process, specifically focusing on a payment exception scenario. The goal is to compare the behavior of two system versions: the standard version using the standard saga pattern. The steps involved in the workflow are carried out to simulate the scenario and observe the response of each version.

The first two phases of the workflow involve retrieving the products and setting up the order. These steps are performed without any issues and progress smoothly. However, in the subsequent step, an exceptional situation arises when a high sum is provided for payment, exceeding the affordability of the consumer. This leads to a payment validation failure, indicating that the payment cannot be processed.

When such an exception occurs, the system needs to handle it appropriately to maintain data integrity and consistency. In the case of the

standard saga pattern, rollbacks are initiated at step 4. This means that any changes made in the previous steps, such as retrieving the products, need to be reverted to ensure that the system returns to its previous state.

The Order-Service, responsible for managing the order process, completes the necessary actions for rolling back the fetched products. It ensures that the order is marked as unsuccessful, reflecting the failure in the payment validation step. This allows for proper tracking and recording of unsuccessful transactions.

By observing and analyzing this experiment, we can gain insights into how the system handles exceptions and maintains the integrity of the overall process. The standard saga pattern, with its built-in mechanisms for compensation and rollbacks, ensures that any errors or failures are properly handled and do not lead to inconsistent or incorrect results.

This experiment highlights the importance of robust error handling and proper compensation mechanisms in microservices-based systems. By effectively managing exceptions, the system can recover from failures, maintain data consistency, and provide a seamless experience for both the service providers and consumers.

It is worth noting that this experiment focuses specifically on the payment exception scenario. In real-world scenarios, various other exceptional cases and compensating actions may need to be considered, depending on the specific requirements of the e-commerce system.

Experiment 2: Testing the improved System (version 2: Deploying the enhanced saga pattern) with Scenario 2

In the initial phase all the steps are same and hence the order request is received in message form in kafka message queue middleware and passed on to warehouse-service where the warehouse-service checks in it database for getting the quantity of items available and updates the warehouse-status and publishes a message In message queue middleware that is forwarded to order service. And as shown in previous experiment the order-service provides an unique order-id.

Step 3: After the second step where the requested order was provided with unique order-id now the billing-service is initiated and here validating the payment is transferred from the main database to the memory cache server and the payment status is updated as false as per the test scenario defined earlier, and hence now the compensating transaction

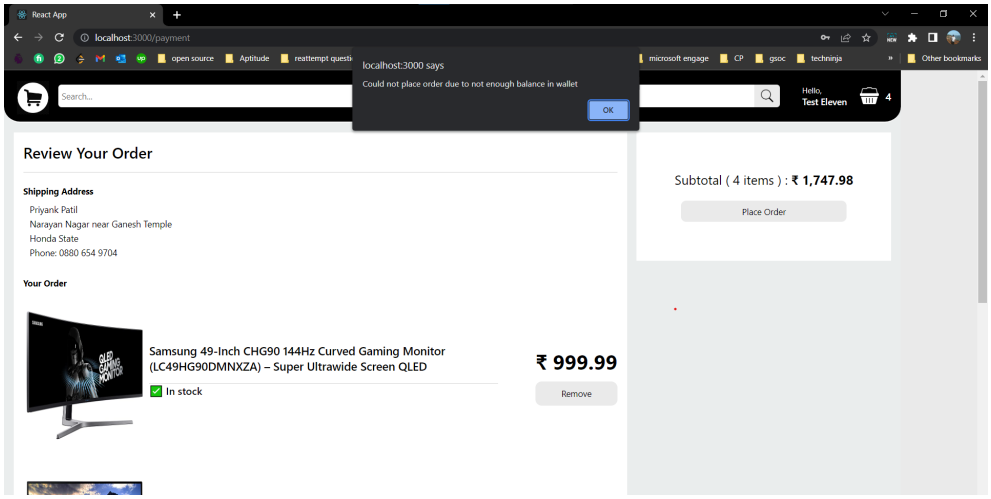


Figure 8.11: Billing Service Exception

needs to be carried out. As shown in figure 11 the payment status would be updated as false and the message would be published to kafka message queue middleware.

Step 4: The figure 12 shows how the items were fetched using GET request from warehouse service on demand of consumer, and the fetched data was stored on redis cache server and further series of transaction was done using data store in redis cache database server and not using the data of main database. Figure 12 shows caching of data and storing it in redis cache database server and also it shows the POST request to compensate the transaction and roll back previous steps.

In the last two steps, the timestamps of both logs are nearly identical, indicating that the system transmitted the failure event and finished the order at the same time. Figure 13 depicts how the Warehouse-Service compensates for fetched products via the memory cache server while making no changes to the original main database.

```
STEP3 FORWARD
res { status: 'failed', msg: 'not enough balance in wallet' }
in catch block Billing-service
===> message received { index: 1, phase: 'STEP_BACKWARD' } payload { status: 'failed', msg: 'not enough balance in wallet' }
STEP2 COMPENSATION
===> message received { index: 0, phase: 'STEP_BACKWARD' } payload { status: 'failed', msg: 'not enough balance in wallet' }
STEP1 COMPENSATION
===> Saga finished and transaction rolled back
final rollback payload { status: 'failed', msg: 'not enough balance in wallet' }
```

Figure 8.12: SAGA Compensation

```
cacheing warehouse order data
POST /warehouse 200 11.311 ms - 1096
stored data in redis
[
  {
    quantity: 1,
    pid: '14',
    id: '644ff37ad32769f612fc905d',
    title: 'Samsung 49-Inch CHG90 144Hz Curved Gaming Monitor (LC49HG90DMNXZA) – Super Ultrawide Screen QLED ',
    price: 999.99,
    image: 'https://fakestoreapi.com/img/81Zt42ioCgL._AC_SX679_.jpg',
    rating: 2.2
  },
  {
    quantity: 1,
    pid: '13',
    id: '644ff37ad32769f612fc9059',
    title: 'Acer SB220Q bi 21.5 inches Full HD (1920 x 1080) IPS Ultra-Thin',
    price: 599,
    image: 'https://fakestoreapi.com/img/81QpkIctqPL._AC_SX679_.jpg',
    rating: 2.9
  },
  {
    quantity: 1,
    pid: '17',
    id: '644ff37ad32769f612fc906d',
    title: 'Rain Jacket Women Windbreaker Striped Climbing Raincoats',
    price: 39.99,
    image: 'https://fakestoreapi.com/img/71HblAHs5xL._AC_UY879_-2.jpg',
    rating: 3.8
  },
  {
    quantity: 1,
    pid: '11',
    id: '644ff37ad32769f612fc9055',
    title: 'Silicon Power 256GB SSD 3D NAND A55 SLC Cache Performance Boost SATA III 2.5',
    price: 109,
    image: 'https://fakestoreapi.com/img/71kWymZ+c+L._AC_SX679_.jpg',
    rating: 4.8
  }
]
POST /warehouse/compensate 200 3.539 ms - 27
```

Figure 8.13: Warehouse Service Transaction on Cache Server

According to the logs, the enhanced recommended solution can manage exceptions and never do rollbacks to the main database when mistakes occur.

Exception Scenario : A Microservice declined

Lets suppose any error occurs in the software and a microservice goes down i.e. stops working, in this case the shipping microservice stopped working and hence an exception was thrown using javaScript as shown in figure 14 and also compensating transactions were carried out by billing service on redis cache database server, compensating transaction would be also carried out by warehouse service as shown in figure 13, and figure 15 shows the compensating transaction performed by billing service on redis cache server.

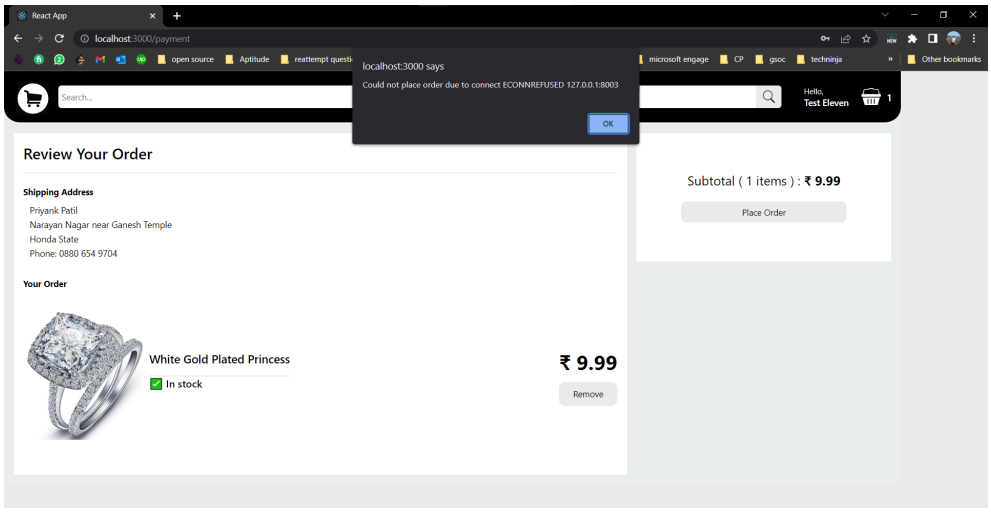


Figure 8.14: Shipping Service Exception

```
current wallet amount before purchase 644.8799999999992
paymentStatus true
POST / 200 4.344 ms - 400
wallet amount after purchase 634.8899999999992
{ status: 'failed', msg: 'connect ECONNREFUSED 127.0.0.1:8003' }
POST /compensate 200 7.628 ms - 24
wallet amount after compensating restore 644.8799999999992
```

Figure 8.15: Billing Service Compensation on Cache Server

Chapter 9

CONCLUSION

This article suggests a better method for fixing the saga pattern's lacking read-isolation characteristic. The idea incorporates the quota into the established system. Both the temporary commit sync service and cache. The ability of the suggested technique to handle the saga pattern's lack of read isolation by the transfer of few transactions in the DB layer towards the memory layer is what gives this research its significance. The proposal states that the memory cache server will be used to perform CRUD operations rather than the primary database. Never will this result in an incorrect commit to the main database. recommended feature to add in our website.

Some other microservices would conduct transactions for compensating to undo any modifications that solely affect the cache level if a microservice fails to complete. In order to achieve eventual consistency, DB commit will be delayed and processed through the middleware for message queue at the conclusion of the work path.

Chapter 10

FUTURE SCOPE

Companies nowadays must have effective search capabilities within their databases since they are dealing with enormous quantities of data. By providing a system that provides quick and accurate transaction processing capability within a business's own database, our project seeks to meet this demand. We can successfully address the issues brought on by dispersed transactions by implementing and improving the SAGA pattern. Our project's capacity to enhance the SAGA pattern's isolation feature is one of its main benefits. This improvement will help to address persistent problems with distributed transactions by improving the consistency, dependability, and atomicity of transactional processes.

Additionally, our project has room for the inclusion of other microservices, which might provide the system new features and functionalities. We can improve the user experience by increasing the number of microservices and providing a more complete and effective solution. There is a sizable opportunity for our project to succeed in the industry given the rising need for effective database search and the possible enhancements that we may offer. This project has the potential to develop into a successful commercial endeavour with appropriate execution and development, meeting the demands of organisations looking for quick and accurate search capabilities within their databases.

Chapter 11

REFERENCES

1. Bucchiarone, A.; Dragoni, N.; Dustdar, S.; Lago, P.; Mazzara, M.; Rivera, V.; Sadovykh, A. Microservices: Science and Engineering; Springer International Publishing: Cham, Switzerland, 2019; ISBN 978-3-030-31646-4.

2. Hasselbring, W.; Steinacker, G. Microservice Architectures for Scalability, Agility and Reliability in E-Commerce. In Proceedings of the 2017 IEEE International Conference on Software Architecture Workshops (ICSAW), Paris, France, 21–25 May 2017; IEEE: Gothenburg, Sweden, 2017; pp. 243–246

3. Stefanko, M.; Chaloupka, O.; Rossi, B. The Saga Pattern in a Reactive Microservices Environment. In Proceedings of the 14th International Conference on Software Technologies, Prague, Czech Republic, 26–28 July 2019; SCITEPRESSScience and Technology Publications: Prague, Czech Republic, 2019; pp. 483–490.

4. Ahluwalia, K.S.; Jain, A. High availability design patterns. In Proceedings of the 2006 conference on Pattern Languages of Programs—PLoP '06, Portland, OR, USA, 21–23 October 2006; ACM Press: Portland, OR, USA, 2006; p. 1.

5. Richardson, C. Microservices Patterns: With Examples in Java; Manning Publications: Shelter Island, NY, USA, 2019; ISBN 978-1- 61729-454-9

6. Messina, A.; Rizzo, R.; Storniolo, P.; Tripiciano, M.; Urso, A. The Database-is the-Service Pattern for Microservice Architectures. In Information Technology in Bio- and Medical Informatics; Renda, M.E., Bursa, M., Holzinger, A., Khuri, S., Eds.; Lecture Notes in Computer Science; Springer International Publishing: Cham, Switzerland, 2016; Volume 9832, pp. 223–233, ISBN 978-3-319-43948-8.

7. Uyanik, H.; Ovatman, T. Enhancing Two Phase-Commit Protocol for Replicated State Machines. In Proceedings of the 2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), V¨asteras, Sweden, 11–13 March 2020; IEEE: V¨asteras, Sweden, 2020; pp. 118–121

8. Mohan, C.; Lindsay, B.; Obermarck, R. Transaction management in the R* distributed database management system. ACM Trans. Database Syst. 1986, 11, 378–396. [CrossRef]

9. Thomson, A.; Diamond, T.; Weng, S.-C.; Ren, K.; Shao, P.; Abadi, D.J. Calvin: Fast distributed transactions for partitioned database systems. In Proceedings of the 2012 International Conference on Management of Data—SIGMOD '12, Scottsdale AL, USA, 20–24 May 2012; ACM Press: Scottsdale, AL, USA, 2012; p. 1.

10. Garcia-Molina, H.; Salem, K. Sagas. In Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data SIGMOD '87, San Francisco, CA, USA, 27–29 May 1987; ACM Press: San Francisco, CA, USA, 1987; pp. 249–259

11. Campbell, R.H.; Richards, P.G. SAGA: A system to automate the management of software production. In Proceedings of the May 4–7, 1981, National Computer Conference on—AFIPS '81, Chicago, IL, USA, 4–7 May 1981; ACM Press: Chicago, IL, USA, 1981; p. 231.

12. Luckow, A.; Lacinski, L.; Jha, S. SAGA BigJob: An Extensible and Interoperable Pilot-Job Abstraction for Distributed Applications and Systems. In Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, Melbourne, Australia, 17–20 May 2010; IEEE: Washington, DC, USA, 2010; pp. 135–144.

13. Lim´on, X.; Guerra-Hern´andez, A.; S´anchez-Garc´ıa, A.J.; Arriaga, J.C.P. Saga- ´ MAS: A Software Framework for Distributed Transactions in the Microservice Architecture. In Proceedings of the 2018 6th International Conference in Software Engineering Research and Innovation (CONISOFT), San Luis Potos´ı, Mexico, 24–26 October 2018; pp. 50–58.

14. Petrasch, R. Model-based engineering for microservice architectures using Enterprise Integration Patterns for inter-service communication. In Proceedings of the 2017 14th International Joint

Conference on Computer Science and Software Engineering (JCSSE), Nakhon Si Thammarat, Thailand, 12–14 July 2017; IEEE: Nakhon Si Thammarat, Thailand, 2017; pp. 1–4.

15. Shopping Process. Available online: https://www.books.com.tw/web/sysqalist(accessed on 10 May 2022).
16. Martin, R.C.; Martin, R.C. Clean Architecture: A Craftsman's Guide to Software Structure and Design; Robert C. Martin series; Prentice Hall: London, UK, 2018; ISBN 978-0-13-449416-6.
17. Walls, C. Spring Boot in Action; Manning Publications: Shelter Island, NY, USA, 2016; ISBN 978-1-61729-254-5.
18. Mass, M. REST API Design Rulebook; O'Reilly: Sebastopol, CA, USA, 2012; ISBN 978-1-4493-1790-4.
19. Rodr´ıguez, C.; B´aez, M.; Daniel, F.; Casati, F.; Trabucco, J.C.; Canali, L.; Percannella, G. REST APIs: A Large-Scale Analysis of Compliance with Principles and Best Practices. In Proceedings of the ICWE, Lugano, Switzerland, 6–9 June 2016.
20. Kreps, J. Kafka: A Distributed Messaging System for Log Processing; ACM Pres: Athens, Greece, 2011.
21. Narkhede, N.; Shapira, G.; Palino, T. Kafka: The Definitive Guide: Real-Time Data and Stream Processing at Scale, 1st ed.; O'Reilly Media: Sebastopol, CA, USA, 2017; ISBN 978-1-4919-3616-0
22. Redis. Available online: https://redis.io/ (accessed on 11 April 2022).
23. Nurkiewicz, T.; Christensen, B. Reactive Programming with RxJava: Creating Asynchronous, Event-Based Applications, 1st ed.; O'Reilly Media, Inc.: Sebastopol, CA, USA, 2016; ISBN 978-1-4919-3165-3.
24. Douglas, K.; Douglas, S. PostgreSQL: A Comprehensive Guide to Building, Programming, and Administering PostgresSQL Databases, 1st ed.; Developer's Library; Sams: Indianapolis, IN, USA, 2003; ISBN 978-0-7357-1257-7.
25. PostgreSQL on Linux. In DBAs Guide to Databases Under Linux; Elsevier: Frisco, CO, USA, 2000; pp. 359–418, ISBN 978-1-928994-04-6.
26. Bailis, P.; Ghodsi, A. Eventual consistency today: Limitations, extensions, and beyond. Commun. ACM 2013, 56, 55–63. [CrossRef]

27. Zhang Dongxia, Miao Xin, Liu Liping et al. Research on the development of large data technology for smart grid [J]. proceedings of the China Electrical Engineering, 2015, 35 (1):2-12.

28. Yao Hongyu. Big data and cloud computing [J]. information technology and standardization of big data and cloud computing, 2013 (5):21-22.

29. Zhao Gang. Guide to large data technology and application [M]. Beijing: Publishing House of electronics industry, 2013.

30. Kong Ying. Research on data stream technology and its application in electric power information processing [D]. Hebei: North China Electric Power University, 2009.

31. Laurence T. Yang, Liwei Kuang, Jinjun Chen, et al. A Holistic Approach to Distributed Dimensionality Reduction of Big Data[J]. IEEE Transactions on Cloud Computing, 2015, 39:1-1.

32. Gao Cheng. PHP dynamic web construction [M]. dynamic homepage construction.PHP Beijing: National Defence Industry Press, 2002.

33. Feng Zuhong.MySQL distributed database access method [J]. computer application, 2002, 22 (8): 4-6.

34. Sandhya Harikumar, M. Shyju; M. R. Kaimal. SQL-MapReduce Hybrid Approach towards Distributed Projected Clustering[C]. International Conference on Data Science Engineering. 2014:18-23.

35. Larrucea X, Santamaria I, Colomo-Palacios R, Ebert C (2018) Microservices. IEEE Softw 35(3):96–100. https://doi.org/10.1109/MS.2018.2141030

36. Corbett JC, Dean J, Epstein M, Fikes A, Frost C, Furman J, Ghemawat S, Gubarev A, Heiser C, Hochs-child P et al (2013) Spanner: googles globally distributed data-base. ACM Transact Comput Syst (TOCS) 31(3):8. https://doi. org/10.1145/2491245

37. Zhang G, Ren K, Ahn JS et al (2019) GRIT: consistent distributed transactions across polyglot microservices with multiple databases[C]. In: 2019 IEEE 35th international conference on data engineering (ICDE). IEEE. https://doi.org/10. 1109/ICDE.2019.00230

38. Mohan C, Lindsay B, Obermarck R (1986) Transaction management in the R* distributed database management system. ACM Trans Database Syst (TODS) 11(4):378–396

39. Thomson A, Diamond T, Weng SC et al (2012) Calvin: fast distributed transactions for partitioned database systems[C]. In: Acm Sigmod international conference on management of data. ACM. https://doi.org/10. 1145/2213836.2213838

40. Hwang E, Kim S, Yoo TK et al (2015) Resource allocation policies for loosely coupled applications in heterogeneous computing systems[J]. IEEE Trans Parallel Distributed Syst:1–1. https://doi.org /10.1109/TPDS.2015.2461154

41. Du J, Sciascia D, Elnikety S, Zwaenepoel W, Pedone F (2014) Clock-RSM: low-latency inter-datacenter state machine replication using loosely synchronized physical clocks. 2014 44th annual IEEE/IFIP international conference on dependable systems and networks, Atlanta, pp 343–354. https://doi.org/10.1109/DSN.2014.42

42. YugaByte. https://www.yugabyte.com/. Accessed 10 May 2019.

43. Zhang S, Zhu S (2013) Server structure based on netty framework forinternet-based laboratory [C]. In: Control and automation (ICCA), 2013 10th IEEE international conference on. IEEE. https://doi.org/10.1109/ICCA.2013. 6564990

44. Yamina Romani, Okba Tibermacine, Chouki Tibermacine, Towards Migrating Legacy Software Systems to Microservice-based Architectures: a Data-Centric Process for Microservice Identification2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C) — 978-1-6654-9493-9/22/31.00 ©2022 IEEE — DOI: 10.1109/ICSA-C54293.2022.0001

45. Anup K Kalia, Jin Xiao, Rahul Krishna, Saurabh Sinha, Maja Vukovic, and Debasish Banerjee. Mono2micro: a practical and effective tool for decomposing monolithic java applications to microservices. In Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pages 1214 1224, 2021.

46. Alessandra Levcovitz, Ricardo Terra, and Marco Tulio Valente. Towards a technique for extracting microservices from monolithic enterprise systems. CoRR, abs/1605.03175, 2016.

    Li, He Zhang, Zijia Jia, Zheng Li, Cheng Zhang, Jiaqi Li, Qiuya Gao, Jidong Ge, and Zhihao Shan. A dataflow-driven approach to identifying microservices from monolithic applications. Journal of Systems and Software, 157:110380, 2019

47. Fabienne Boyer, Nol de Palma, Fabienne Boyer, Nol de Palma, Poster: A Declarative Approach for Updating Distributed Microservices 2018 ACM/IEEE 40th International Conference on Software Engineering: Companion Proceedings

48. J. Kramer and J. Magee. 1990. The Evolving Philosophers Problem: Dynamic Change Management. IEEE TSE 16, 11 (1990), 1293–1306.

49. Marian Kyryk, Oleksandr Tymchenko, Nazar Pleskanka, Mariana Pleskanka, Methods and process of service migration from monolithic architecture to microservices, 2022 IEEE 16th International Conference on Advanced Trends in Radioelectronics, Telecommunications and Computer Engineering (TCSET) — 978-1-6654-6861-9/22/31.00 ©2022 IEEE — DOI: 10.1109/TCSET55632.2022.976705

50. P. Di Francesco, P. Lago, and I. Malavolta, "Migrating towards microservice architectures: An industrial survey," In 2018 IEEE International Conference on Software Architecture, ICSA 2018, Seattle, USA, April 30 - May 4, 2018, pages 29-38, May, 2018.

51. Tapia, Freddy, Miguel Á. Mora, Walter Fuertes, Hernán Aules, Edwin Flores, and Theofilos Toulkeridis. 2020. "From Monolithic Systems to Microservices: A Comparative Study of Performance" Applied Sciences 10, no. 17: 5797. https://doi.org/10.3390/app10175797

52. F. Li, J. Fröhlich, D. Schall, M. Lachenmayr, C. Stückjürgen, S. Meixner, F. Buschmann "Microservice Patterns for the Life Cycle of Industrial Edge Software", in Proc. EuroPLoP'18 Conf., 2018, art. 4, pp. 111

53. G. Samaras, K. Britton, A. Citron, C. Mohan, "Two-phase commit optimizations and tradeoffs in the commercial evironment", in Proc. IEEE Conf., April 1993, pp. 17-22

54. Krishna Mohan Koyya, Dr. B Muthukumar, A Survey of Saga Frameworks for Distributed Transactions in Event-driven Microservices, 2022 Third International Conference on Smart Technologies in Computing, Electrical and Electronics (ICSTCEE) — 978-1-6654-5664-7/22/31.00 ©2022 IEEE — DOI: 10.1109/ICSTCEE56972.2022.10099533

55. amírez,Francisco, et al. "An Empirical Study on Microservice software development." 2021 IEEE/ACM Joint 9th International Workshop on Software Engineering for Systems-of-Systems and 15th Workshop on Distributed Software Development, Software Ecosystems and Systems-of- Systems (SESoS/WDES). IEEE, 2021.

56. Larrucea, Xabier, et al. "Microservices." IEEE Software35.3 (2018): 96-100.

57. Sandro Speth, Sarah Stieß, Steffen Becker, A Saga Pattern Microservice Reference Architecture for an Elastic SLO Violation Analysis, 2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C) — 978-1-6654-9493-9/22/31.00 ©2022 IEEE — DOI: 10.1109/ICSA-C54293.2022.00029

58. C. M. Aderaldo, N. C. Mendonc¸a, C. Pahl, and P. Jamshidi, "Benchmark requirements for microservices architecture research," in 2017 IEEE/ACM 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering (ECASE). IEEE, 2017, pp. 8–13.

59. S. Speth, "Semi-automated cross-component issue management and impact analysis," in Proceedings of 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, Nov. 2021, pp. 1090–1094.

60. Ridhima Mishra, Ridhima Mishra, Rishi Prakash, Paras Nath Barwal, Transition from Monolithic to Microservices Architecture : Need and proposed pipeline, 2022 International Conference on Futuristic Technologies (INCOFT) — 978-1-6654-5046-1/22/31.00 ©2022 IEEE — DOI: 10.1109/INCOFT55651.2022.10094556

Web Interface for Distributed Transaction System

Suraj Godage
dept. of computer engineering
Army Institute of Technology, Pune
Pune, India
surajgodage_21954@aitpune.edu.in

Hardik Pandya
dept. of computer engineering
Army Institute of Technology, Pune
Pune, India
hardikpandya_19122@aitpune.edu.in

Shubham Bhosale
dept. of computer engineering
Army Institute of Technology, Pune
Pune, India
shubhambhosale_21953@aitpune.edu.in

T Rohith Kumar
dept. of computer engineering
Army Institute of Technology, Pune
Pune, India
trohith_19013@aitpune.edu.in

Rushali Patil
dept. of computer engineering
Army Institute of Technology, Pune
Pune, India
rpatil@aitpune.edu.in

***Abstract*—This research delves into the concept of saga pattern, an effective means for managing local sequential transactions across distributed micro services. Despite its effectiveness in ensuring data consistency, it has a lack of isolation which could lead to incorrect commits being made on databases due to unfinished transactions. To mitigate this issue and further enhance current possibilities provided by other solutions like transaction management protocols (e.g., two-phase commit), this study introduces enhancements such as quota cache and commit-sync service allowing for certain operations between database layers - ultimately preventing invalid or incomplete commitments from occurring on main databases. An experiment was conducted to evaluate the success of a microservices-based e-commerce system. Results indicated that this novel idea could handle both regular scenarios and exceptions, thus tackling insularity issues. In case of any service failures, compensation transactions would be run in order to undo adjustments made solely within the caching layer; after all processes had been completed correctly, committing alterations back into the database followed suit. Consequently, it can be concluded that while promising features have beenshowcased by these results - further investigation is still needed for optimization purposes before its widespread implementation as an industry standard approach.**

Keywords – microservices, two-phase commit, databases, SAGA, distributed transaction, Three-Phase commit.

I. Introduction

A. Overview

To create an effective E-commerce website, the use of Microservices created with Spring boot in the backend is highly recommended. REST APIs will be used to facilitate communication between microservices, while Orchestration approaches are required to manage transactions across various services. In order to correctly manage events including buying and completion processes as well as failure cases, several message queue middleware needs to be utilized; these queues enhance system throughput significantly by providing isolation handling through SAGA quotacache database connected via REDIS server technology - utilizing a commit sync pattern for optimal performance results.

B. Objectives

- To Build website using microservice architecture so the implementation and working of distributed transaction patterns across it becomes easier and smoother.
- Distributed transaction handling across microservices architecture using SAGA pattern for distributed transaction.
- Implement SAGA pattern and enhance it using queue middleware and quota cache database server to enhance isolation.

C. Scope and Motivation

Every company has huge datasets to handle nowadays, and everyone needs search in their database to be fast and precise. Therefore, our project can be used by those companies for fast and precise search inside their own database. Improvising SAGA pattern by enhancing isolation property will solve many on-going problems related to distributed transaction problems. The main motivation behind idea of this project was toimplement distributed transaction system to abort the problems created due to monolithic transactionsystem. There are problems with monolithic transaction systems such as they do not maintain ACID properties across transactions and hence there is need of distributed systems. There are present distributed transaction system patterns like 2 phase commits, 3 phase commit which do lack in concurrency control across multiple transactions. The third most widely used pattern isSAGA, so the second motive was to enhance the performance of SAGA pattern by solving the issueof absence of isolation in saga.

II. Literature Survey

Microservice Architecture

A. Comparison of Monolithic and Microservice Architecture

In this paper we not only compare performance and scalability between monolithic and microservices. We had covered the difference between then along with their different implementation style like working with java or with c#, NET, microservices based architecture has the advantage of improved availability, fault tolerance, and having a great software development community and because of such benefits, the most small-scale organization wants to change the software architecture [1]. The benefits like as they are loosely coupled with eachother follow the abstraction method to do so and itis

presented with well-defined interfaces and communicates with users. In analyzing the most effective architectural design for various systems, it becomes evident that microservices do not always reign supreme. In fact, monolithic architectures can often prove to be more successful in small-scale applications where user requests are relatively simple. The capacity to break down and organize systems into logically coherent, loosely linked modules that conceal their implementation from one another and expose services through well defined interfaces is referred to as "separation of concerns."[2][3] Additionally, due tothe extra overhead associated with directing requests through multiple components within a microservice architecture, these solutions commonly perform poorly on an individual system basis compared to their predecessors.

B. A scalable and protected Microservice Architecture

This research presents a highly promising architecture for both academic research and practical implementation by industry experts. In order to fully understand the potential of this approach, future experiments should compare microservice-based systems to traditional monolithic ones in terms of features and logged data. Performing load testing can provide valuableinsights into which approach offers superior performance capabilities [21]. A microservices-based application requires tight coordination of the supporting services (such as security monitoring, authentication/authorization, etc.) through a specialized infrastructure, such as the Service Mesh [4]. This comparison is crucial as it can provide a clear understanding of how changes made in a monolithic structure differfrom those made in a microservice-based system, and what implications they may have on system maintenance.

C. A high-speed integrated control platform based on microservice architecture

This study examines microservices as a potential remedy for the integrated highway management platform. The authors want to present a complete study of the possible advantages of employing microservices for this particular application through a careful investigation of microservices theoretical thinking, architectural patterns, and platform design implementation [22]. The authors also emphasized that the growth of highway platforms would inevitably lead to traffic informatization, which is a future trend. This project intends to offer a new architectural model of microservices for highway platform framework, which should be able to replace the current model, in order to fully realize the promise of this technology. The proposed model is believed to be more suitable for the specific needs of the highway platform and is expected to bring significant improvement in terms of stability and service protection. According to the authors, by segmenting the business into different dimensions and decomposing the formerly monolithic app into its component functions, each functional module can be in charge of a distinct area of the business and be deployed and maintained independently, resulting in a system that is expected to be more stable and easy to maintain.

One Phase Commit

A. one-phase commit – Non blocking in nature

This paper presents the Pronto Non-Blocking Commitment Protocol as a highly promising solution for transactions are distributed and also involving replicated data. Through extensive evaluation ofan extended YCSB benchmark, the authors have demonstrated that pronto exhibits significantly higher reliability than its cutting-edge counterparts, even with increasing numbers of transaction participants [23]. The authors also point out that this protocol manages efficient operation despite lock acquisition failures, ensuring timely termination tests, in contrast to Variants of 2PC, which suffer from serious performance degradation in case of server failure, and current non-blocking atomic commitment methods, which are too expensive for daily use due to their requirement for log writings and data item values storage. The availability of large-scale systems is limited by blocking protocols like 2PC and its variants. Two non-blocking atomic commitment mechanisms already in existence, Paxos Commit [7] and three-phase commit [8], are too costly for broad application. According to the authors, this protocol can be a good substitute for distributed systems that demand great performance and dependability.

Two Phase Commit

A. 2pc agent method: achieving serializability in presence of failures in heterogeneous multi-database.

This method of involving 2PC Agent provides a better solution to the challenge of integrating common systems of databases with a one phased interface that is transactional into systems containing multi-databases. This method makes it possible for coordinators to use conventional two-phase commit protocols and integrated concurrency control and recovery, which are necessary for the consistency and integrity of data in multi-database systems. The method is particularly notable for its ability to maintain global serializability even in the event of unilateral aborts or site failures, which are common occurrences in distributed systems[24]. A global sub transaction is a series of all the operations of a global transaction, performed at the 2PC interface of a particular site. Thus, similarly to transaction models used elsewhere [9][10], for every global transaction there is at most one global sub transaction per site [11]. A local sub transaction is a series of operations pertaining to a global sub transaction, performed at the local DBMS interface[12]. This feature ensures that the system can recover from failures and maintain its consistency despite the occurrence of such events. The authorssuggest that this method is a valuable addition to the field of distributed systems and database management, and that it has the potential to be widely adopted in practical applications. To ensure successful coordination between collaborating databases, a highly structured two-phase locking strategy or similar concurrency control policy must be implemented. Additionally,annotations such as AP operations are needed to accurately analyze the functioning of the entire system and record states in

2PCA logs for verification purposes. These activities should take place within an execution tree at each respective node to enable proper global sub-transaction processing.

B. 2PC- Enhnaced 2PC for concurrency in distributed transaction system*

This paper presents an innovative concurrency control technique called 2PC* for distributed transactions in multi-microservice modules. This mechanism utilizes a secondary asynchronous optimistic lock to reduce the probability of concurrent conflicts between distinct operations, while also allowing for higher extraction levels of concurrency than traditional two-phase commit (2PC). The authors have proposed this mechanismas an efficient solution for distributed systems that involve multiple microservices and require high levels of concurrency[25]. To showcase the effectiveness of this technique, the authors have developed and deployed Ctrip MSECP on a cloud platform using their middleware solution that offers transactional support with 2PC*. The two-phase commit (2PC) protocol is typically used to implement a distributed transaction [13]. Sadly, this does not perform well in large-scale, high-throughput systems, particularly for applications with a lot of transaction conflicts [14]. Locks are maintained throughout the 2PC process, which is the cause. For weakly connected distributed transactions, other strategies include persistent message queue patterns [15][16], which call for additional application logic to make up for missing transaction stages. The results of this deployment demonstrate that the proposed technique is highly effective in reducing conflicts and increasing concurrency, making it a valuable addition to the field of distributed systems. The authors suggest that this technique has the potential to be widely adopted in practical applications and can be used to improve the performance and scalability of distributed systems.

C. Concurrency control and it's distributedtransaction

Nowadays, most organizational applications rely on database systems since they store and protect an organization's operational data. A distributed or centralized approach can be used to build database systems. A database system that is implemented in a distributed way is referred to as a" distributed database system." The switch from centralized to distributed frameworks was motivated by the need for higher performance, speed, and availability. Distributed concurrency control offers a method for synchronizing distributed transactions so that their interleaved execution does not compromise ACID properties [9]. The distributed database system where these transactions are executed has the necessary data on a network of linked data servers [26]. As a result, in addition to local dependencies, dependencies involving other data servers must also be considered. In a distributed database system, scattered transactions are processed, and related data is stored on a network of connected computers. Different data servers handle the execution of each sub-transaction that makes up a distributed transaction. These transactions can be executed more quickly by being serialized.

Three Phase Commit

A. 3PC – design and implementation

This study presents a novel algorithm for distributed computing that addresses the issue of blocking caused by 2PC through the use of a 3PC protocol [27]. This advanced protocol includes a pre-commit phase during Phase two, which is sent from the sender side after completion and storedin K-sites. This new protocol aims to provide a more efficient and reliable solution for distributed systems by reducing the likelihood of blocking and increasing the chances of successful transaction processing between multiple parties involved. when the acknowledgment was executed by two parties and the transaction was completed. It establishes a three-step process with phases 1 and 2 being referred to as preparation, pre-commit, and phase 3 as acknowledgement commit/abort. This system is employed to prevent the blocking issue that arises in 2PC when a transaction is executed [5][6]. The authors have conducted extensive testing to evaluate the effectiveness of this protocol and have determined that further testing should be conducted to establish its performance under different scenarios and to identify any potential issues. Additionally, the authors have emphasized the importance of proper RPC connection establishment to construct a directory structure with three specific phases referred to as preparation; prep-commit; and acknowledgement commit/abort, for successful transaction processing. The authors suggest that this advanced algorithm has the potential to be a valuable addition to the field of distributed computing and can be used to improve the performance and reliability of distributed systems. This technique allows for the circumvention of any blocking issues that may occur during execution in a 2PC transaction.

SAGA

A. Performance analysis of Rollbacks and Aborts of Long-lived Transaction Processing Systems.

This paper delves into the intricacies of transaction processes that are long-lived that incorporate a small group of transactions with aborts and rollbacks. The authors analyze the complexities ofthese DBMS systems, which contains fork or join structures, where processes requires only one data item, ignoring all precedence constraints between tasks [28]. The system is composed of multiple components, including terminals, lock managers, waiting queues for fault diagnosis centers, as well as distributed computing elements such as file managers, disk storage units, and communication networks. These elements are involved in sagas, which yield partial locks during execution, allowing some short-term transactions to acquire these locks ahead of time. The authors suggest that this research can be used to improve the design and implementation of long-lived transaction processing systems, and canbe valuable to practitioners in the field of distributed systems. By implementing the saga system, it is possible to drastically reduce response times when processing short transactionsat high loads and with large LLT sizes. The result of this increased concurrency allows a remarkable improvement in performance

capabilities.

B. Analysis of 2 types of saga pattern in microservice architecture

This paper aims to analyze and differentiate the two methods of saga one being orchestration and the other being choreography for evaluating performance. Time, memory usage, and energy consumption are used as the basis for this analysis [12]. Through comprehensive studies on time utilization & memorization it can be conclusively proven that event choreography is more efficient in terms of speed compared with orchestrators [29]. Nevertheless, whilst attempting to coordinate multiple trigger actions without a coordinator there may arise significant difficulties;since every micro-service's activities become difficult to discern by one developer or team not being cognizant of each other's tasks. Programmers have likely reached the point of monolithic authority and complete independence due to these two tactics[17][18]. They now offer reusability, which decreases the cost of developing or updating programs, ensures resource efficiency, and enables applications to grow upon demand [19][20]. Choreography-based composition is an effective technique for distributed business processes and event triggers when the number of microservices involved are small. Unlike orchestration, which calls one provider at a time in a centralized manner until it receives a response before dialing another, choreography relies on decentralized events. SAGA further refines this approach by focusing specifically on certain services only; thus enabling agile delivery without having to resort to complex distributed transactions that may be required for ensuring data consistency otherwise. Each system command is given a unique saga, representing an intricate web of local transactions to update data across multiple services. To ensure reliability and consistency, ACID transaction frameworks are employed for each local operationwithin the saga with feedback provided once it's complete.

C. Procesing of transaction in microservice architecture and problems with it

This research aims to create a sophisticated online shopping system utilizing the microservices architectural pattern. To accomplish this, an 'OrderMicroservice' is used to divide all business processes and their associated elements into distinct parts. Likewise, objects linked with reservations or inventory are delegated to an 'Inventory Microservice'. The study further examines how transactions may be handled in these types of systems as well as provides recommendations for improvement when confronting issues related transaction processing using microservices architecture- which might lead clients to observe that orders were created only for them to be removed due unforeseen circumstances instantly afterwards. As microservice architecture is a widely adopted approach, its importance increases due to the changes brought by scalability and fault-tolerance. To ensure consistent results linked to numerous data sources when performing critical business processes, problem solving techniques such as introducing Transaction Coordinator or Saga Orchestrator must be implemented; though this may increase development costs and complexityin comparison with monolithic architectures. Consequently, embracing proper approaches for handling distributed transactions becomes ever more vital year after year.

III. Quota-Cache Method to Enhance SAGA.

A. Microservices

The system utilises five microservices: the WarehouseService, OrderService, BillingService, ShippingService, and CustomerService. There is a separate database for each microservice. The spring boot technology was used to create the microservices, which inherited and implemented the pertinent use cases [30]. Internally, the microservices were accessible through the REST API [31][32], enabling communication with the straightforward HTTP protocol.

B. E-Commerce Workflow

The Figure demonstrates the event process for a complex e-commerce microservices system. This system allows users to conveniently purchase products online, selecting their preferred items and payment options while choosing shipping methods that suit them best. Composed of various components such as Warehouse-Service, Order-Service, Billing-Service and Shipping Service among others; it guarantees long transactions with both warehouse before billing or billing before warehouse approaches present –as demonstrated in the figure. The flow is initiated by Warehouse Services when they procure goods which are then stored into an "IN_PROGRESS" order in Order services. This paper describes a process for placing and fulfilling orders that begins with the retrieval of items. If the retrieval of items fails, the order is listed as "FAILED." The next step in the process is the validation of payments by a Billing Service. If the payment is successful, the money is taken for the transaction, otherwise the flow stops at this stage. After the successful validation of payments, the order is dispatched from the Shipping Services. The delivery completion process belongs to the Order Service, who also updates the overall status including shipping ID and total cost. This process is a crucial part of e-commerce operations and is designed to ensure that orders are fulfilled in a timely and efficient manner. The authors suggest that this process can be used as a model for other e-commerce operations and can be adapted to fit the specific needs of different businesses.

C. Message queue middleware

The message queue middleware manages both failure and completion events using a distributed event-store and stream-processing architecture. It facilitates message sending and receiving between the microservices. To ensure that the final commitment is accurate, it maintains the order of requests from the microservices. Numerous forms of message queue middleware were used. APache Kafka was used as the middle of the message queue. You may publish and listen to message streams using Apache Kafka, an open source distributed event streaming platform. Kafka's ability to process thousands of

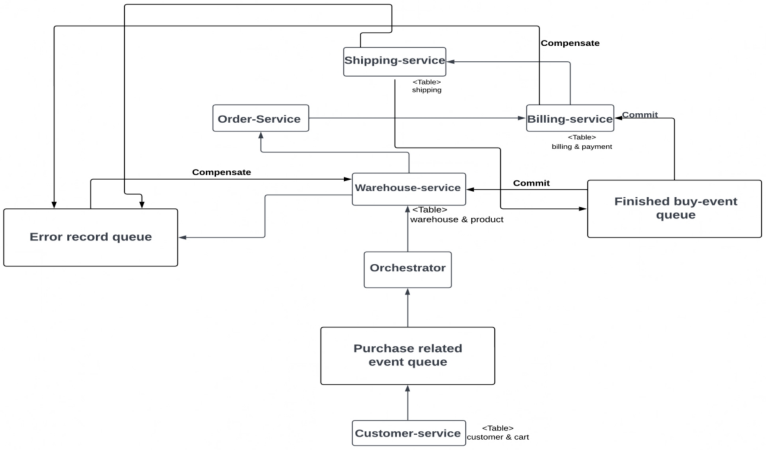
messages per second allows it to be used to develop applications with high throughput. Because Kafka includes a durability feature, it always stores messages on disc for persistence[33].

D. Quota-Cache

In computational contexts, caches are sophisticated data storage elements that can drastically reduce access time to primary memory systems. Furthermore, quota caching is a form of in-memory databasing which ensures an accurate read/write temperature through CRUD (Create Read Update Delete) actions and frees the main database from any incorrect commit disasters. In sum, utilizing a cache system helps maximize performance by decreasing latency across multiple databases or services. This paper presents an implementation of an in-memory quota cache that enables microservices to achieve improved performance as compared to the baseline with saga. The implementation of this type of caching can help to mitigate potential exceptions that may arise from failures during the process, especially when validation is required for a service's operation. The in-memory quota cache is a valuable addition to the microservices architecture as it allows for faster access to frequently used data, reducing the need for frequent database access. However, in the event of something going awry when it comes time for Warehouse-Service actions, which would update the main database and would be perceived by clients placing orders, compensation transactions are deployed quickly to undo these changes and cancel the order. The authors suggest that this implementation of an in-memory quota cache can be used to improve the performance and reliability of microservices-based systems, and can be adapted to fit the specific needs of different applications. In the improved version, these CRUD operations have been relocated from the primary database layer to an intermediate cache tier instead.

E. Commit Sync Service

This paper describes a synchronous service that remains blocked until the successful completion of a commit or until an irrecoverable error occurs, which would prompt it to be thrown back to the caller. The implementation of a memory cache server allows for CRUD activities to be executed while the primary database still requires committing. As a result, the need for ultimate sync-commit services arises. Every successful event



ultimately "delivers" a 'done message' via middleware into microservices that employ quota caches, followed by the execution of necessary commits on databases for updated records. This approach ensures that commitment only occurs after foreseeable outcomes are reached, providing an extra level of reliability and consistency for the system. This approach can be taken into consideration to improve the performance and reliability of systems that rely on synchronous services, and can be adapted to fit the specific needs of different applications.

IV. Conclusion

This article presents a novel solution to an enduring problem with the saga pattern: its lack of read isolation. The authors propose a solution that leverages both temporary commit sync services and caches, transferring transactions from database layers into memory layers in order to reliably perform CRUD (create-read-update-delete) operations without the risk of incorrect commits being made on the primary databases. This approach aims to ensure safe and consistent operation within the saga pattern, by reducing the chances of conflicts and errors.

The authors have conducted extensive research and testing to demonstrate the effectiveness of this solution and have highlighted its potential benefits for the field of distributed systems. This research is considered to be critical for ensuring the safe and reliable operation of systems that employ the saga pattern and can be used as a guide for practitioners in the field. To ensure eventual consistency between all of the microservices, any modifications that solely affect cache levels will be compensated for via a compensation transaction if one should fail to complete. At the end of its respective workflow, this compensatory action is facilitated by delayed database commit handled through message queue middleware.

V. REFERENCES

[1] Grzegorz Blinowski (member, IEEE), Anna Ojdowska, Adam Przybyłek, "Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation", 2022, DOI: 10.1109/ACCESS.2022.3152803.

[2] A. Przybyłek, ‘‘Where the truth lies: AOP and its impact on software modularity,’’ in Fundamental Approaches to Software Engineering, D. Giannakopoulou and F. Orejas, Eds. Berlin, Germany: Springer, 2011,pp. 447–461.

[3] A. Przybyłek, ‘‘An empirical study on the impact of AspectJ on software evolvability,’’ Empirical Softw. Eng., vol. 23, no. 4, pp. 2018–2050, 2018.

[4] M. Fazio, A. Celesti, R. Ranjan, C. Liu, L. Chen, and M. Villari, “Open issues in scheduling microservices in the cloud,” IEEE Cloud Computing, vol. 3, no. 5, pp. 81–88, 2016.

[5] Anshu Veda, Kaushal Mittal, Project Report One and Two Phase Commit Protocols, KReSIT, IIT Bombay, 20/1 0/2004.

[6] Peter Lannhult, Bjorn Lindblom, Review of Non-Blocking TwoPhase Commit Protocols, Master Program in Computer System Engineering, Emausgatan 29C/ Generatorgatan I.

[7] J. Gray and L. Lamport, “Consensus on transaction commit,” ACM Trans. Database Syst., vol. 31, no. 1, pp. 133–160, Mar. 2006.

[8] D. Skeen, “Nonblocking commit protocols,” in Proceedings of the 1981 ACM SIGMOD international conference on Management of data. ACM, 1981, pp. 133–142.

[9] V. Gligor and R. Popescu-Zeletin, "Transaction Management in Distributed Heterogeneous Database Management Systems", Information Systems, vol. 11, no. 4 (1986), pp. 287-297.

[10] Y. Breitbart and A. Silberschatz, "Multidatabase Systems with a Decentralized Concurrency Control Scheme", IEEE Distributed Processing Tech. Comm. Newsletter, vol. 10, no. 2 (November 1988), pp. 35-41.

[11] W. Du, A. K. Elmagarmid, Y Leu and S.D. Osterman, "Effects of Local Autonomy on Global Concurrency Control in Heterogeneous Distributed Database Systems", Proc. Second Int. Conf. on Data and Knowledge Systems for Manufacturing and Engineering, October 1989.

[12] A. K. Elmagarmid and A. A. Helal, "Supporting Updates in Heterogeneous Database Systems", Proc. IEEE Fourth Conf. Data Engineering (Los Angeles, 1 - 5 February, 1988), pp. 564 - 569.

[13] Mohan C, Lindsay B, Obermarck R (1986) Transaction management in the R* distributed database management system. ACM Trans Database Syst (TODS) 11(4):378–396

[14] Thomson A, Diamond T, Weng SC et al (2012) Calvin: fast distributed transactions for partitioned database systems[C]. In: Acm Sigmod international conference on management of data. ACM. https://doi.org/10. 1145/2213836.2213838

[15] Hwang E, Kim S, Yoo TK et al (2015) Resource allocation policies for loosely coupled applications in heterogeneous computing systems[J]. IEEE Trans Parallel Distributed Syst:1–1. https://doi.org/10.1109/TPDS.2015.2461154

[16] Du J, Sciascia D, Elnikety S, Zwaenepoel W, Pedone F (2014) Clock-RSM: low-latency inter-datacenter state machine replication using loosely synchronized physical clocks. 2014 44th annual IEEE/IFIP international conference on dependable systems and networks, Atlanta, pp 343–354. https://doi.org/10.1109/DSN.2014.42

[17] S. Neha, U. Sakthivel, and P. Raj. “Selection Mechanism of Micro-Services Orchestration vs. Choreography.” International journal of Web & Semantic Technology 10.1 (2019): 01–13.

[18] S. Andreas et al. “Orchestration vs. Choreography Functional Association for Future Automation Systems.” IFAC-PapersOnLine 53.2 (2020): 8268–8275.

[19] B. Fadhlallah, N. Le Sommer, and Y. Maheo. “Choreography-Based vs Orchestration-Based Service Composition in Opportunistic Networks.” International Conference on Wireless and Mobile Computing, Networking and Communications. Vol. 2017- October. IEEE Computer Society, 2017.

[20] S. Martin, O. Chaloupka, and B. Rossi. “The Saga Pattern in a Reactive Microservices Environment.” ICSOFT 2019 - Proceedings of the 14th International Conference on Software Technologies. SciTePress, 2019. 483–490.

[21] Rahime Yilmaz, Feza Buzluca, "A Fuzzy Quality Model to Measure the Maintainability of Microservice Architectures" in 2021 2nd International Informatics and Software Engineering Conference (IISEC), INSPEC Accession Number: 21573074, DOI: 10.1109/IISEC54230.2021.9672417.

[22] Roberto S. de O. Júnior, Ruan C. A. da Silva, Marcelo Souza Santos, Danyllo W. Albuquerque, Hyggo O. Almeida, Danilo F. S. Santos, in 2022 IEEE International Conference on Consumer Electronics (ICCE), INSPEC Accession Number: 21667735, DOI: 10.1109/ICCE53296.2022.9730757.

[23] Yuqing Zhu, "Non-blocking one-phase commit made possible for distributed transactions over replicated data" in 2015 IEEE International Conference on Big Data (Big Data), INSPEC Accession Number: 15679723, DOI: 10.1109/BigData.2015.7364107.

[24] A. Wolski, J. Veijalainen, "2PC Agent method: achieving serializability in presence of failures in a heterogeneous multidatabase" in Proceedings. PARBASE-90: International Conference on Databases, Parallel Architectures, and Their Applications, 2002, INSPEC Accession Number: 3707187, DOI: 10.1109/PARBSE.1990.77157.

[25] Pan Fan, Jing Liu, Wei Yin, Hui Wang, Xiaohong Chen & Haiying Sun , "2PC*: a distributed transaction concurrency control protocol of multi-microservice based on cloud computing platform", 2020

[26] B.P. Gbranwi, Prince, O Asagba, "Distributed Transactions and Distributed Concurrency Control", 2021, DOI:10.47760/ijcsmc.2021.v10i01.006.

[27] Nitesh Kumar, Laxman Sahoo, Ashish Kumar, "Design and implementation of Three Phase Commit Protocol (3PC) algorithm" in 2014 International Conference on Reliability Optimization and Information Technology (ICROIT), INSPEC Accession Number: 14254535, DOI: 10.1109/ICROIT.2014.6798309.

[28] D. Liang, S.K. Tripathi, "Performance analysis of long-lived transaction processing systems with rollbacks and aborts" in IEEE Transactions on Knowledge and Data Engineering (Volume: 8, Issue: 5, October 1996), INSPEC Accession Number: 5428810, DOI: 10.1109/69.542031.

[29] Sahin Aydin, Cem Berke Çebi, "Comparison of Choreography vs Orchestration Based Saga Patterns in Microservices" in 2022 International Conference on Electrical, Computer and Energy Technologies (ICECET), INSPEC Accession Number: 22028506, DOI: 10.1109/ICECET55527.2022.9872665.

[30] Walls, C. Spring Boot in Action; Manning Publications: Shelter Island, NY, USA, 2016; ISBN 978-1-61729-254-5.

[31] Mass, M. REST API Design Rulebook; O'Reilly: Sebastopol, CA, USA, 2012; ISBN 978-1-4493-1790-4.

[32] Rodríguez, C.; Báez, M.; Daniel, F.; Casati, F.; Trabucco, J.C.; Canali, L.; Percannella, G. REST APIs: A Large-Scale Analysis of Compliance with Principles and Best Practices. In Proceedings of the ICWE, Lugano, Switzerland, 6–9 June 2016.

[33] Narkhede, N.; Shapira, G.; Palino, T. Kafka: The Definitive Guide: Real-Time Data and Stream Processing at Scale, 1st ed.; O'Reilly Media: Sebastopol, CA, USA, 2017; ISBN 978-1-4919-3616-0.

plague

ORIGINALITY REPORT

13%	6%	10%	3%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	mdpi-res.com Internet Source	5%
2	Eman Daraghmi, Cheng-Pu Zhang, Shyan-Ming Yuan. "Enhancing Saga Pattern for Distributed Transactions within a Microservices Architecture", Applied Sciences, 2022 Publication	1%
3	Catalina Munoz Morales, Bruno Honorio, Alexandro Baldassin, Guido Araujo. "Improving Phased Transactional Memory via Commit Throughput and Capacity Estimation", 2021 IEEE 33rd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), 2021 Publication	1%
4	Submitted to University of Greenwich Student Paper	1%
5	Alam Rahmatulloh, Fuji Nugraha, Rohmat Gunawan, Irfan Darmawan. "Event-Driven	1%