TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

MUSTAFA TIKIR 223606IASM

ANALYSIS OF DISTRIBUTED TRANSACTION PATTERNS WITHIN MICROSERVICE ARCHITECTURE

Master's thesis

Supervisor: Tarmo Robal
PhD

Tallinn 2024

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

MUSTAFA TIKIR 223606IASM

HAJUSATE TEHINGUMUSTRITE ANALÜÜS MIKROTEENUSTE ARHITEKTUURIL

Lõputöö liik: magistritöö

Juhendaja: Tarmo Robal
Teadur

Tallinn 2024

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature, and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Mustafa Tikir

06.05.2024

Abstract

The main aim of the thesis is to study and analyze distributed transaction patterns that solve the problem of data consistency in transactions operating over multiple microservices. The three most popular patterns are chosen to be analyzed which are Two-Phase Commit, Saga Orchestration, and Saga Choreography. During analysis, three different versions of an e-commerce system were implemented, each implementation having different transaction patterns. Ordering a product is the main flow analyzed in three implementations. Ordering a product operation includes multiple microservices and operations such as updating inventory, deleting charts, and saving payment.

The analysis is made based on three research questions which mainly investigate the design, performance, and updates of the systems. The design question investigates static code analysis metrics like cyclomatic complexity, cognitive complexity, lines of code, and communication overhead. The performance question investigates response times, transaction throughput, central processing unit usage, and memory usage. The update question investigates system updates causing failures like changing data-transfer objects, introducing new endpoints or new message queues, and adding a new microservice to the order flow.

The results of experiments show that each distributed transaction pattern has trade-offs where one or the other performs better. In the design analysis, code metrics do not highlight one pattern, but Saga Choreography has the best communication overhead score. Performance analysis shows Saga Choreography has the best response time and transaction throughput, but the Two-Phase commit has the best resource usage. Finally, update analysis shows that update failures can be mitigated in all the patterns but updating systems in Two-Phase commit and Saga Orchestration is easier because of the central orchestrator.

This thesis is written in English and is 68 pages long, including 6 chapters, 30 figures, and one table.

Annotatsioon
Hajusate tehingumustrite analüüs mikroteenuste arhitektuuril

Magistritöö peamine eesmärk on analüüsida hajutatud tehingumustreid, mis lahendavad andmekonsistentsi probleemi tehingutes, mis toimivad mitme mikroteenuse kaudu. Analüüsiks valitakse kolm populaarseimat arhitektuurimustrit: kahefaasiline kohustus, Saga orkestreerimine ja Saga koreograafia, mida rakendatakse e-kaubanduse süsteemil. Toote tellimine on peamine voog, mida analüüsitakse kolmes rakenduses, ja mis hõlmab mitmeid mikroteenuseid ja operatsioone, näiteks laoseisu uuendamine, ostukorvi kustutamine ja makse salvestamine.

Analüüs põhineb kolmel uurimisküsimusel, mis uurivad süsteemide arhitektuuri, jõudlust ja uuendamise võimalusi. Arhitektuuri kavandamise juures uuritakse staatilise koodianalüüsi mõõdikuid nagu tsükliline keerukus, kognitiivne keerukus, koodiridade arv ja suhtluskoormus. Jõudlusküsimuste juures uuritakse vastuseaegu, tehingu läbilaskevõimet, keskprotsessori kasutust ja mälukasutust. Uuendamiste võimalused keskenduvad arhitektuuri valikust sõltuvatel süsteemi uuendustel, mis võivad põhjustada tõrkeid, nagu andmeedastusobjektide muutmine, uute otspunktide või sõnumijärjekordade lisamine ning uue mikroteenuse lisamine tellimuse voogu.

Töö tulemused näitavad, et iga hajutatud tehingumustri puhul esineb kompromissikohti, kus üks või teine muster toimib paremini. Analüüs ei tõsta koodimõõdikute alusel ühte mustrit esile, kuid Saga koreograafial on parim suhtluskoormuse hinnang. Jõudlusanalüüs näitab, et Saga koreograafial on parim reageerimisaeg ja tehingu läbilaskevõime, kuid kahefaasiline kohustus arhitektuurilise mustrina kasutab ressursse kõige paremini. Uuendamiste analüüs kinnitab. et tõrkeid saab kõigis mustrites leevendada, kuid süsteemide uuendamine kahefaasilises kohustuses ja Saga orkestreerimises on lihtsam tänu kesksele orkestreerijale.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 68 leheküljel, 6 peatükki, 30 joonist, ühte tabelit.

List of abbreviations and terms

2PC	Two-Phase Commit
ACID	Atomicity, Consistency, Isolation, Durability
API	Application Programming Interface
BFF	Back-end for Front-end Pattern
CPU	Central Processing Unit
CQRS	Command Query Responsibility Segregation
DTO	Data Transfer Object
ERD	Entity Relationship Diagram
LLT	Long-Lived Transactions
G1	Garbage-First
MAS	Multi-Agent Systems
ms	Millisecond
MSA	Microservice Architecture
OS	Operating System
RAM	Random Access Memory
REST	Representational State Transfer
RQ	Research Question
TPS	Transactions Per Second
UI	User Interface
UML	Unified Modeling Language
UX	User Experience

Table of contents

List of Figures

List of Tables

1 Introduction

Software systems are complex and designing complex systems is a challenging task. Imagine an e-commerce system having one million visitors per day and the system is being developed by 18 development teams globally and simultaneously. The system needs to meet the functional requirements of the e-commerce system such as listing products, taking products into shopping lists, purchasing, etc. Also, the system needs to meet non-functional requirements like scalability, performance, and fault tolerance [1]. In such a complex system, management of teams, operation of applications, and number of deployments become an issue that needs to be handled.

System designers, analysts, architects, and developers put substantial effort into software systems so that systems can be managed easily, perform well, scale according to the load, new feature time to market is fast, monitored well, and bug-free. To overcome these challenges software architectures are being refined in terms of usefulness, effectiveness, and maturity [2].

Architecture design is considered one of the most important processes in producing successful software [3]. However, there is no such architecture that solves all the problems. Designing architecture is tightly related to system requirements. There are numerous software architectures used in industry such as Layered architecture [4], Peer-to-peer architecture [5], Monolith architecture [6], and Microservice architecture (MSA) [7]. Architects may choose to benefit from multiple architectures to solve design problems.

For decades, software systems in enterprises have been developed as monolith systems. Monolith architecture is a type of architecture in which all the modules of the system are combined into one larger component. Monolith architecture has benefits like managing a single code repository easily and having one single database, therefore, using the benefit of the database management system's transaction mechanisms. However, monoliths bring the problem of having complex [8], less flexible [9], and poorly scalable [10] systems. Imagine a scenario in a big monolith system in which only one line of code must be

updated due to an urgent production bug. To solve the bug, the code gets changed. However, certain processes are required to be run such as the big monolith application is required to be rebuilt, automated tests are rerun, crucial functions are retested, and the application is redeployed [11]. The process is inefficient due to processes like building and testing happen on one big chunk of a monolith application.

Over the years, monolith systems have gotten bigger and more complex. MSA is presented as an alternative architecture to the monolith. The idea is to create small services having their database per each business function so that the overall system will provide better scalability, faster time to market, and business agility [11]. However, MSA has challenges such as design complexity, management overhead, and keeping data consistent on distributed transactions [12].

In this thesis, the design patterns solving the problem of data consistency of distributed transactions in MSA are analyzed. In business transactions operating over multiple microservices, transaction consistency is a challenge. When a microservice in the transaction chain fails for any reason, proper rollback mechanisms must be applied to keep the system in a consistent state. In monolith systems, keeping transaction consistency is easier, because most of the database management systems provide transaction managers that ensure the transaction consistency. However, in MSA, benefiting only from transaction managers is not enough, because each microservice has its database. Therefore, to overcome the problem of data consistency in MSA, distributed transaction patterns are used. In this thesis, distributed transaction patterns in MSA are analyzed in aspects of design, performance, and updates to answer the following research questions (RQ):

RQ1: How much does the choice of transaction consistency pattern affect overall MSA design?

> Hypothesis: The choice of distributed transaction pattern in MSA influences overall system design regarding code complexity and communication overhead. Each pattern has a trade-off where one or the other performs better.

RQ2: What is the impact of various transaction consistency patterns on the performance of microservices?

Hypothesis: Various transaction consistency patterns have a quantitative impact on the performance of MSA. It is expected that certain consistency patterns may lead to improved response times, increased transaction throughput, better Central Processing Unit (CPU) usage, and memory usage, while others may exhibit trade-offs in one or more of these performance metrics.

RQ3: What are the failure scenarios and methods to mitigate failures when updating microservices using different transaction consistency patterns?

Hypothesis: Different transaction consistency patterns are affected by the challenges of updates causing application downtimes. There are methods to mitigate failures during development and deployment time such as endpoint versioning and feature flags. Depending on the pattern and update case some of the mentioned techniques can be used.

In this thesis, three popular distributed transaction patterns which are the Two-Phase Commit Pattern (2PC) [13], Saga Orchestration Pattern [14], and Saga Choreography Pattern [15] are analyzed with the research questions (RQ1-RQ3). For this, the functionality of ordering a product is implemented as a microservice for an e-commerce system using 2PC, Saga Orchestration, and Saga Choreography patterns. Ordering a product was chosen because ordering a product contains transactions that are distributed to multiple microservices. In the e-commerce system, the order microservice receives the order request, sends an update request to the inventory microservice, sends a delete request to the chart microservice, and sends a save request to the payment microservice. The difference in systems is the distributed transaction pattern implemented. Distributed transaction patterns are responsible for handling errors when ordering a product to keep the system in a consistent state. Because in monolith systems transaction consistency is ensured by transaction managers, these systems are left out of the thesis scope.

The rest of the thesis is structured as follows. Chapter 2 addresses related work by providing a literature review of MSA distributed transaction pattern analyses. Chapter 3 provides an overview of different distributed transaction patterns in general and in-depth for the chosen three patterns. Chapter 4 provides details about the three e-commerce systems' design and implementation. Chapter 5 presents the experiments, test scenarios, test conditions, and gathered results. In conclusion, Chapter 6 summarizes the thesis.

2 Related Work

Migration from monolith architecture to MSA gained speed in the last decade. A wide number of studies [8], [9], [10], [11], [16] indicate the motivations, benefits, and challenges of migrating to MSA exist, where the main motivations are maintainability, scalability [17], performance issues, fault tolerance, agility, and cost efficiency [18]. The benefits of migrating to MSA are improved scalability, availability, performance [16], development velocity/team productivity, deployment velocity, maintainability, compliance, and cost efficiency [9].

The challenges of migrating to MSA are categorized mainly under two categories. The first category is operational challenges, which are role assignments, communication/coordination between team members, and organization of teams [10]. The second category is technical challenges, which are the decomposition of the monolithic system, monitoring, integration, management of microservices, programming language issues, complexity of the testing process, and managing data consistency [19]. In this thesis, distributed transaction patterns that solve the data consistency challenge in MSA are analyzed.

Michael Müller [20] compared three patterns: Shared Database, Saga Orchestration, and Saga Choreography based on consistency and autonomy attributes for detecting benefits, problems, and use cases of patterns. The Shared Database pattern leads to strict consistency in systems, also it is an easy pattern to understand, implement, test, and debug. However, it interferes the autonomy. Saga Orchestration is also considered easy to implement, test, and debug. However, the orchestrator in the system is a bottleneck. Saga Choreography is the most autonomous pattern in which microservices scale well and provide loose coupling. In performance wise, it is expected to perform well since services communicate with each other. The main problem with this solution is that it only guarantees eventual consistency which is the weakest consistency model. Therefore, the most autonomous model is identified as Saga Choreography where it has the weakest consistency. Whereas the Shared Database can provide the strongest consistency, yet it provides the least autonomous solution. As future work to the research, it is mentioned

that selected solutions could be implemented for further testing of performance, scalability, and fault tolerance.

Malyuga and colleagues [21] analyzed 2PC, Saga Orchestration, and Saga Choreography patterns and suggested a fault-tolerant Saga Orchestration in Representational State Transfer (REST) Architecture. Their idea is to store data at the orchestrator level and execute the REST actions when necessary so that the system remains in a consistent state. The research suggests two optimization approaches: analysis of response payload from service to replicate and preserve only meaning data and optionality of data preservation. These optimizations can be used together to different degrees. This research provides time and memory metrics only for Saga Orchestration. The time difference in the preservation of only failed Sagas data and no preservation at all is defined as 0.12%. In systems with low Saga failure rates, saving only failed Sagas makes the biggest memory consumption reduction 59.02% comparatively. Even though the research considers the same three patterns analyzed in this thesis, it does not compare patterns, it provides an improved Saga Orchestration pattern.

Koyya and colleagues [14] conducted a survey about Saga frameworks for distributed transactions in event-driven MSA. The survey analyzed available Saga frameworks for mainly three programming languages which are Java, Python, and NodeJS. In the survey, it is pointed out that the only distributed transaction framework for Python is saga-framework, and for NodeJS is node-sagas. However, for Java, there are multiple frameworks such as Axon, Eventuate Tram, Apache Camel, and Netflix Conductor. Axon is a suitable option for implementing both orchestrated and choreographed Sagas in applications that are architected along the lines of the Command Query Responsibility Segregation (CQRS) pattern. Eventuate Tram is a developer-friendly framework because of its builder model. The downside of the Eventuate Tram is that developers are expected to write some number of classes to represent each of the entities, commands, and events and it often leads to duplicate code or forced to use a shared library. Apache Camel is a proven framework for enterprise-level application integrations; however, it supports only REST-based orchestration. Netflix Conductor is a natural choice for Spring Boot applications which provides a user interface (UI) for state of execution. Koyya and colleagues' [14] survey compares frameworks for different programming languages however the main difference with this thesis is that the research does not compare distributed transaction patterns.

Aydin and colleagues [15] compared Saga Choreography and Saga Orchestration patterns in MSA. Their study suggests which Saga methodology to employ in which scenario by analyzing performance and complexity using real-world data obtained by modeling a range of use cases with a unique project built on Spring Boot-driven microservices. The analysis is based on time, memory, and power usage. The result of time and memory consumption analysis shows Saga Choreography is substantially faster than Orchestration. However, when several events are triggered by microservices, Saga Choreography becomes difficult to manage. Also, having an orchestrator in the system helps the development team's services to communicate. So, Saga Orchestration is sluggish, but it comes in handy when dealing with complex transaction scenarios. Also, about the dynamic updates, the research mentions that in case there is a plan to upgrade, delete, or add additional services often, Saga Choreography is suggested to be used. In the Saga Choreography pattern, with little work and no damage to existing systems, the whole software can be updated. The research summarizes the use cases such as the Orchestration is more effective for transactional services or services that rely on others to perform a task. On the other hand, Choreography is well suited to work that is asynchronous, self-contained, and decoupled. The research compares two Saga patterns and provides the pros and cons of each pattern and in what kind of business cases they are useful. However, the research does not include the 2PC pattern in the comparison.

Rudrabhatla [22] performed a quantitative analysis of performance, complexity, and load-based test of both event Saga Choreography and Orchestration patterns in MSA. According to the performance analysis, the Saga Choreography pattern performs almost five times better than the Saga Orchestration pattern. In terms of complexity, a sample test scenario is performed for four microservices and 6 microservices in both Saga Choreography and Saga Orchestration for five test runs. It was observed that the time taken for the Saga Orchestration pattern is approximately 40 times more than the Saga Choreography. However, it was noted that as the number of events increased, it became more and more complex to handle the code in individual microservices. Whereas the orchestrator proved to be more elegant in handling multiple events with less confusion as the event handlers are the orchestrators at a single location. In terms of load-based tests, a similar test scenario is repeated one more time with a scenario where the frequency of events that are triggered is increased by five-fold and 10-fold. This is obtained by writing a test client that fires parallel requests. It is calculated that the ratio of response times with

the frequency of one vs five vs 10. It is observed that the event modal began to respond slowly as the frequency increased, whereas the orchestrator was able to handle the load better. The response times varied as 1:3.6:8.2 for the Saga Choreography, whereas the ratios for Saga Orchestration came out as 1:3.9:6.4. As Aydin and colleagues' [15] research, Rudrabhatla [22] also compared the Saga Choreography and Orchestration but results include more numeric comparison values. However, both research does not include the 2PC pattern in the comparison.

Megargel and colleagues [23] provided a decision framework for solution architects to choose between Saga Choreography and Saga Orchestration patterns in MSA. The research identifies the Saga Choreography pattern as loosely coupled and low chattiness, meaning less communication is required, whereas as downsides the Saga Choreography pattern has poor process visibility, complex design, poor reusability, and intermediate response time. The Saga Orchestration pattern has clear process visibility, simple design, high reusability, and predictable response time. However, as a downside, the Saga Orchestration pattern has tightly coupled services and high chattiness. The research also provides a hybrid solution combining the asynchronicity and flexibility of Saga Choreography with the process visibility of Saga Orchestration. Instead of having atomic microservices subscribing to state changes in each other, a composite microservice (the "controller") becomes responsible for ensuring the execution of steps by publishing and subscribing to events. The decision framework considers the pros and cons of the patterns mentioned. The framework employs 6 collaboration factors: a) distribution of microservices across the wide area network, impacting "chattiness", b) predictability of response time from the application process, c) loose coupling of microservices to the process, impacting deployment, d) reusability (i.e., atomicity) of microservices, e) complexity (i.e., number of microservices in the application process), and f) runtime visibility of the application process flow. The solution architect answers the questions, how capable is each collaboration pattern (Saga Choreography and Saga Orchestration) in satisfying each of the six collaboration factors? In terms of "Ability", each collaboration factor is to be assessed using a five-point Likert scale; Incabable, Slightly Incapable, Neutral, Capable, Very Capable. As Aydin and colleagues' [15] research and Rudrabhatla's [22] research, also Megargel and colleagues' [23] research compares the Saga Choreography and Saga Orchestration patterns, and in addition to the comparison,

the research offers a decision framework and a hybrid solution. However, the research does not include the 2PC pattern in the comparison.

Nylund [24] compared 2PC and Saga Choreography patterns in his master's thesis. The 2PC commit and the Saga Choreography patterns are evaluated based on their performance, scalability, and complexity. Performance and scalability metrics show that the Saga Choreography pattern performs 10 times better than 2PC in max response times. The complexity of patterns is calculated and compared with Halstead metrics [25], and results show that Saga Choreography is more complex than 2PC. The evaluation shows that the 2PC commit could be better in edge cases that require strong consistency such as crucial operations in banking systems. However, the Saga Choreography is the superior choice in most use cases, with its ability to operate well under heavy loads and high availability in the case of a network partition. Nylund research compares 2PC and Saga Choreography but does not include Saga Orchestration in the comparison.

Laigner and colleagues [26] conducted research on data management in MSA. In the research, database systems, deployment patterns, online queries, stream processing, even-driven computing, and interservice-communication mechanisms in MSA are investigated. Especially the interservice-communication mechanism concept is related to this thesis work. The research shows that the most popular interservice-communication mechanisms in MSA are the Orchestration, Choreography, 2PC, and Back-end for Front-end (BFF). However, BFF does not provide a distributed transaction pattern. When it is asked experts who developed those systems about how they dealt with distributed transactions, they mentioned custom-made solutions implemented [26].

Vural and colleagues [27] discussed in their systematic literature review on Microservices, that there are not enough empirical studies to clarify many issues under discussion related to MSA and not enough research targeting fragile points of MSA such as distributed transactions. Laigner and colleagues [26] mentioned in their data management in MSA research, that the lack of a holistic data management solution for microservices leads practitioners to resort to ad-hoc designs. One of the data management problems mentioned in the research [26] is the data consistency in transactions spanning multiple microservices which require design solutions such as distributed transaction patterns. Also, Laigner and colleagues' [26] research indicates that 2PC, Saga

Orchestration, and Saga Choreography are the most popular distributed transaction patterns.

The literature review conducted in this section clarifies that there is no research comparing the 2PC, Saga Orchestration, and Saga Choreography distributed transaction patterns. The research gaps identified in the literature review are the lack of enough empirical studies to research the fragile parts of MSA and the lack of research on holistic data management solutions such as distributed transactions. Therefore, 2PC, Saga Orchestration, and Saga Choreography distributed transaction patterns are analyzed with the design, performance, and update attributes.

3 Distributed Transactions

Distributed transaction patterns in MSA are solutions for the data consistency problem of transactions spanning multiple microservices [28]. In this section, 2PC [13], 2PC* [29], 2PC+ [30], GRIT [12], Saga Orchestration [15], Saga Choreography [14], SagaMas [31], and Buffered Serialization [32] distributed transaction patterns are explained. 2PC, Saga Orchestration, and Saga Choreography workflows are visualized, and advantages/disadvantages are stated.

2PC is the traditional approach for solving data consistency problems in MSA [12]. There are prepare and commit phases in 2PC. In the context of 2PC, there exists a controller node and other microservices. When a transaction starts, the controller sends Prepare requests to other microservices. The microservices reply with their first acknowledgment (ready) to the controller indicating they can commit or reject. When all microservices send a successful acknowledgment message, the controller starts the second phase and sends a commit message. Microservices processes the commit operation and returns the acknowledgment message. Figure 1 visualizes the 2PC flow. If any of the services returns fail, then the whole transaction fails. So that the system remains in a consistent state. 2PC satisfies ACID (Atomicity, Consistency, Isolation, Durability) principles therefore 2PC provides a solution for data consistency. Here are the advantages and disadvantages of the 2PC [29], [32], [33]:

Advantages:

- Strong consistency,
- The controller provides ease of management.

Disadvantages:

- System locked during the two phases,
- Does not scale well to large and high-throughput systems,
- The controller is the single point of failure.

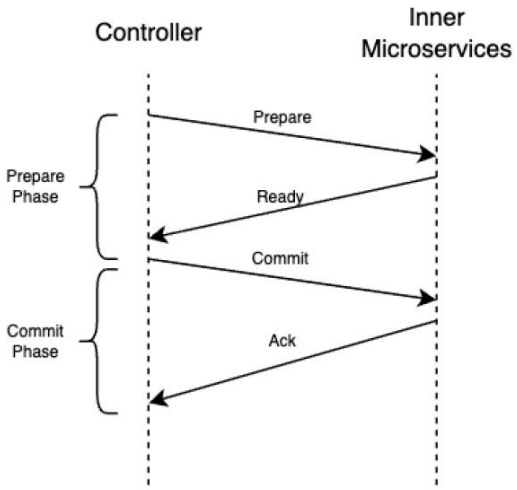

Figure 1. 2PC transaction management [33].

2PC* [29] is an extension of 2PC. 2PC* aims to improve concurrency, reduce overhead, and improve fault tolerance. 2PC* achieves better performance than 2PC because locks are held throughout the transaction process. In the load test scenario, 2PC* achieves at most a 3.3x improvement in throughput and a 67% reduction in latency compared to 2PC. Fault tolerance is improved with transaction compensations. 2PC* also provides a middleware solution for transaction management.

2PC+ [30] is another extension of 2PC. 2PC+ extends 2PC with transaction thread synchronization blocking optimization algorithm SAOLA. Then, the SAOLA algorithm gets verified by temporal logic of action TLA+ language. 2PC+ suggests the real-time performance is increased 2.87 times to 3.77 times compared to 2PC and transaction per second (TPS) performance is 323.7% to 514.4% higher than 2PC.

GRIT [12] is a framework that resolves the distributed transactions in the MSA challenge by leveraging deterministic database technologies and optimistic concurrency control protocol. It is another pattern built on top of 2PC. In GRIT, transactions are optimistically executed with its read-set and write-set captured during the execution phase. Then at the commit phase, conflicts are checked, and a global commit or reject decision is made. A logically committed transaction is persisted into logs first, and then asynchronously applied to the physical databases deterministically.

The Saga framework was first published in 1987 [34]. The main target of the framework is to handle Long-Lived Transactions (LLT) alternatively named Sagas. LLT is defined

as a transaction that can be split into a collection of sub-transactions. Every sub-transaction corresponds to a microservice operation in MSA. On an LLT request, each microservice performs its action and returns either success or error. In case of any error, rollback steps are performed to undo the changes so that the end of the LLT system remains in a consistent state. This has formed a basis for managing transactions in distributed architectures including MSA.

Saga patterns are mainly applicable for event-driven microservices that use events for interservice communication. Typically, a message bus is responsible for microservices to communicate via events. Each event corresponds to a local transaction in a microservice. However, Saga patterns do not provide traditional(strong) consistency, Sagas provide eventual consistency [14]. Eventual consistency means that the system allows temporary inconsistencies between services but ensures that all services will eventually be consistent. There are two basic types of Sagas which are Orchestration and Choreography. In Saga Orchestration, a controller (orchestrator) manages the microservices, whereas in Saga Choreography, no controller is present.

In the Saga Orchestration pattern, a controller service controls and manages the transaction flow. A message bus is employed for communication between the *Orchestrator* and microservices. In this pattern, every transaction starts via the *Orchestrator*. The *Orchestrator* puts a message onto the queue and triggers a corresponding microservice, the corresponding microservice executes its action and puts the result back into the queue. The *Orchestrator* receives the result and decides what to do next. In case it is a successful flow, the flow continues until it reaches the end. In case there is an error received from the queue, proper compensation(rollback) events are published onto the queue [14], [15], [32], [33]. Figure 2 illustrates a successful orchestration execution of the Saga pattern.

Advantages:

- Communication through *Orchestrator*,
- Ease of debugging and tracing,
- Eventual consistency.

Disadvantages:

- *Orchestrator* is the single point of failure,
- Complexity increases by adding additional components.

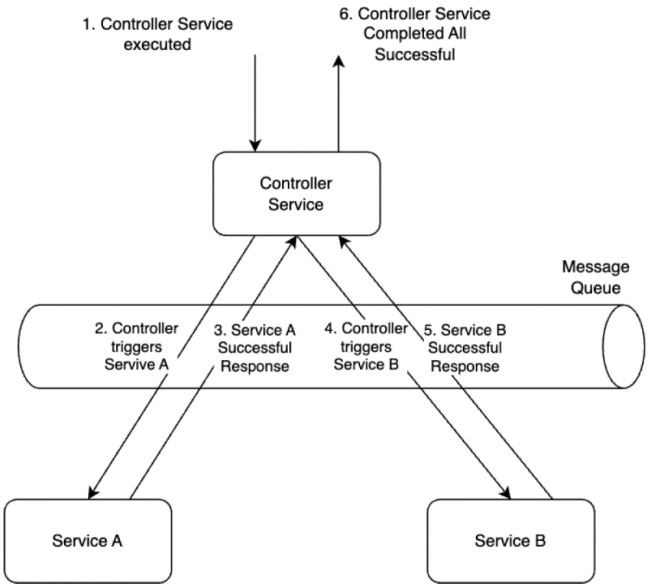

Figure 2. Successful Saga Orchestration [33].

In the Saga Choreography pattern, there is no orchestrator to handle transaction management. Every microservice publishes its response message to the queue and the next microservice receives the message from the queue. Every microservice triggers its process when the previous microservice publishes a successful message. When an error happens, the microservice publishes an unsuccessful message to the queue, and microservices that already published the successful messages receive the unsuccessful message and perform compensation actions [14], [15], [32], [33]. Figure 3 shows the successful execution of the Saga Choreography pattern.

Advantages

- No central orchestrator so no single point of failure,
- Faster since there is no controller,
- Eventual consistency.

Disadvantages

- Complex due to high coupling between services,
- Communication overhead in case of failure.

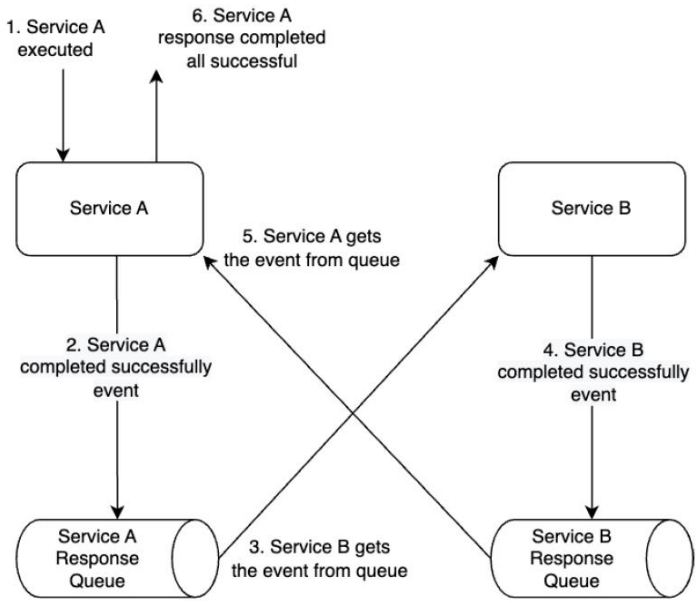

Figure 3. Successful Saga Choreography [33].

SagaMAS [31] is a Saga pattern built on top of Saga patterns. MAS stands for Multi-Agent Systems. In SagaMAS, transaction management is delegated to the agent by having one agent assigned to each microservice which deals with the microservices communication. Therefore, microservice developers are revealed from dealing with microservice coordination tasks. This also promotes loose coupling in the system and removes the single point of failure of coordinating microservices via an orchestrator. Also, SagaMAS provides a high level of workflow management and error-handling strategy.

Buffered Serialization [32] is the last pattern explained in this chapter. The pattern combines the benefits of both the Saga Choreography and Saga Orchestration patterns but introduces an additional component, the buffered serializer. The buffered serializer ensures that transaction metadata across services is serialized and deserialized using a common structured format. In addition, the buffered serializer makes the failed transaction data available for retries. Making retries possible is a feature of this pattern

that does not exist in others. For storing the transaction metadata, server-side caches are used.

In terms of the popularity of distributed transactions in MSA, Laigner and colleagues' [26] research shows mechanisms used for inter-microservice coordination in the software systems developed in MSA. There is no one-to-one mapping of inter-service communication and distributed transaction patterns, but they are closely related. The research shows that even though the (BFF) is one of the most common mechanisms for inter-microservice coordination it does not provide distributed transactions. Therefore, as shown in Table 1, the Saga versions of Orchestration and Choreography and 2PC are the most popular distributed transaction mechanisms used in MSA.

Table 1. Mechanisms for inter-microservice coordination [26].

Coordination mechanism	Usage percentage
Orchestration	22.8
Sagas (centralized approach)	14.8
BFF	14.8
Choreography	13.6
Sagas (decentralized approach)	8.6
Distributed transactions (e.g., via 2PC)	8.6
2-Phase Commit	6.8
Others	9.9

In this section, 2PC [13], 2PC* [29], 2PC+ [30], GRIT [12], Saga Orchestration [15], Saga Choreography [14], SagaMas [31], and Buffered Serialization [32] patterns were explained for how 2PC and Saga patterns and their modified versions(2PC*, SagaMAS, etc.) are implemented and what possible improvements the patterns contain. Also, 2PC, Saga Orchestration, and Saga Choreography pattern workflows are visualized and pros/cons are listed for creating a knowledge base for implementation and experiments which are explained in Chapter 4 and Chapter 5.

4 E-Commerce Applications

In order to investigate how distributed transaction patterns in MSA affect overall system design, performance, and updates, and to answer the research questions RQ1-RQ3, three e-commerce application backend systems were developed. Those three applications have similar functionality, ordering a product, but the distributed transaction consistency patterns are different in each system. The distributed transaction consistency patterns chosen to be implemented are 2PC, Saga Orchestration, and Saga Choreography.

The functionality chosen to be implemented is ordering a product. Ordering a product is a function that exists in many e-commerce systems. It is a good candidate for this thesis work because the functionality interacts with multiple microservices such as inventory microservice, chart microservice, and payment microservice. In the scenario implemented, which is illustrated in Figure 4, when a user orders a product in the e-commerce system, the order microservice receives the order and executes a series of calls to the subsequent microservices. First, the order microservice executes an update request to the inventory service to update(reduce) the available inventory items. Secondly, the order service executes a delete request to the chart microservice to delete the chart related to the user because the products are purchased. Finally, the order microservice executes a save request to the payment service. The requests are chosen to be operations that change data like update, delete, and save. So that, when one of the operations fails, the others must be rolled back to keep the system in a consistent state. For example, when a user orders a product and updating inventory and deleting chart is successfully finished, but payment is failed, updating inventory, and deleting chart get rolled back. The Figure 4 shows a simple ordering of a product flow, more detailed versions are found in this chapter when each distributed transaction system is explained.

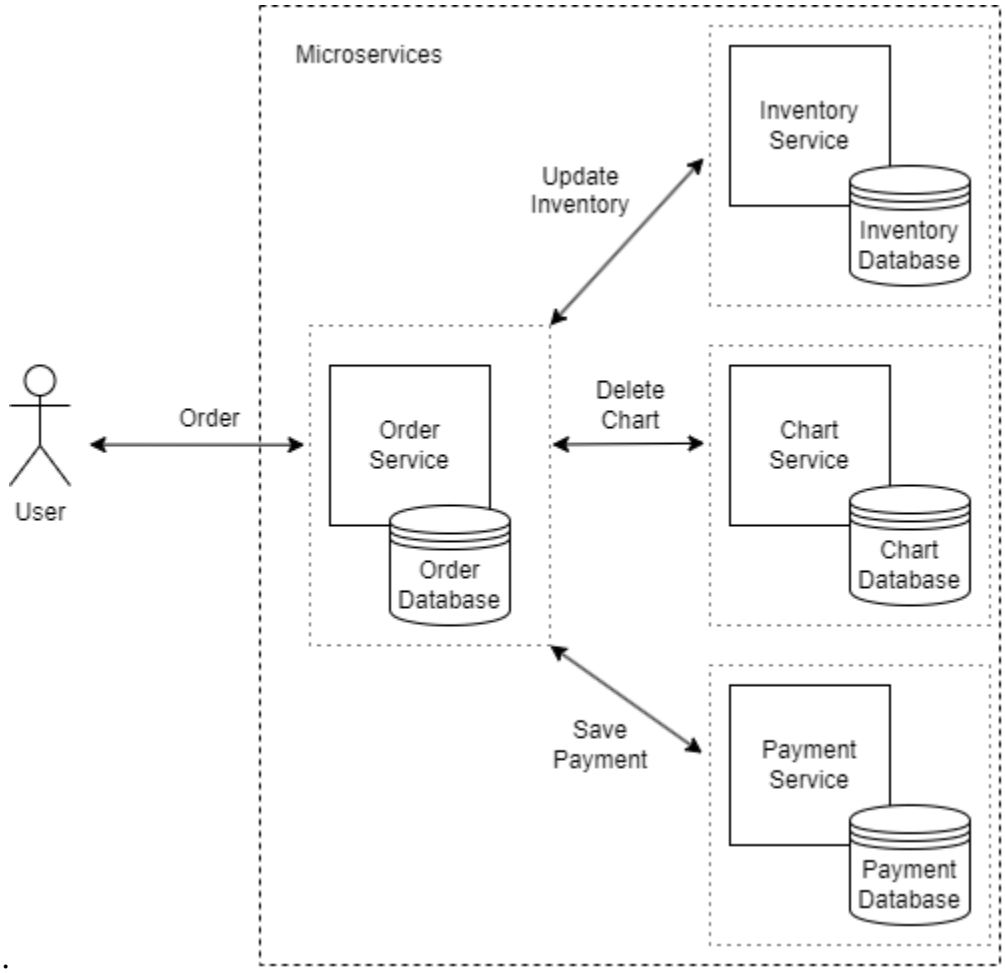

Figure 4. Simple flow of ordering a product in the e-commerce system.

E-commerce application backend systems are developed in the Spring Boot[1] application framework. Spring Boot is one of the most popular Spring framework modules that provides rapid application development with ease of automatic configuration management of software components. Mythily's research shows that the Spring Boot framework is used 24% directly, and 41% indirectly in information technology systems [35]. In this research, using Spring as an application framework plays a crucial role because it provides integration with Spring Cloud, Spring Data, Spring Web, Spring Actuator, Spring Kafka, etc. which are particularly used for different purposes in the e-commerce applications. Using the libraries improves the application development time because those libraries provide many functionalities out of the box. For example, Spring Data makes database connection and database querying easy for developers. Spring Data provides most of the boilerplate code that developers would need to implement like

[1] https://spring.io/projects/spring-boot

creating transactions, committing, rollbacking, etc. Providing those functionalities out of the box helps developers to focus on the business logic, therefore it helps to increase the development time. Spring Boot version[1] 3.2.1 is used in the implemented applications as it was the latest version of Spring Boot at the time of development in January-April 2024.

Java[2] is the main programming language used to implement e-commerce applications in the Spring Boot framework. Java is a general-purpose, concurrent, strongly typed, class-based object-oriented language. It is compiled to the bytecode instruction set and binary format defined in the Java virtual machine specification. Even though it is possible to develop applications in other programming languages in the Spring framework, Java is the most popular choice. Java 17[3] is the version used to develop e-commerce applications because it is the minimum version for applications developed in Spring Boot. Java 17 was modern and new enough but at the same time stable enough to find sources online at the time of development.

Maven[4] is used as a build tool for e-commerce applications. Because microservices are multiple separate applications, Maven is a useful tool to build the executable artifacts of the microservices. With a single command in Maven, all the deployable microservice artifacts can be produced, i.e. *mvn package*. In the e-commerce applications, Google Cloud Jib Maven Plugin[5] is used to put produced artifacts into the Docker hub. Docker is a containerization platform that provides the ability to package and run applications in a loosely isolated environment. Docker is used to run the implemented e-commerce microservices and to run related services that microservices need like databases for every microservice, an authentication server, a message bus, a distributed tracing tool, monitoring tools, and a code quality tool.

In MSA, each microservice has its database (Figure 4). For example, payment mircorservice has its database, and payment microservice accesses only to payment database. Also, no other microservice accesses the payment database. Separating

[1] https://github.com/spring-projects/spring-boot/wiki#release-notes

[2] https://www.oracle.com/java/technologies/java-se-glance.html

[3] https://www.oracle.com/java/technologies/downloads/#java17

[4] https://maven.apache.org/

[5] https://github.com/GoogleContainerTools/jib/tree/master/jib-maven-plugin

databases per microservice provides loose coupling but it is the source of distribution transaction problems in MSA. In e-commerce systems, PostgreSQL[1] is chosen as a database provider because PostgreSQL is an open-source, object-relational database that has a strong reputation for reliability, feature robustness, and performance. However, because Spring Data provides data access level abstraction, the database connection is configured only via configuration files. No database provider-specific code is implemented, so when it is required to change the database provider to Oracle or MySQL or a NoSQL database like MongoDB, only configuration changes are sufficient.

Spring Cloud is a natural choice for microservices developed in the Spring framework. Spring Cloud provides plenty of functionalities out of the box that every MSA needs in a cloud environment. In e-commerce microservices, Spring Cloud Gateway is used for providing a secure endpoint. The e-commerce backend environment is accessible only via Gateway Endpoint. Gateway checks authentication credentials in every request before forwarding it to the corresponding service. For an authentication service, KeyCloak[2] is used. Keycloak provides user federation, strong authentication, user management, and fine-grained authorization as a service. For example, when a user executes an order request to purchase a product, the e-commerce gateway receives the request, and checks if the authentication token is valid and the user is authenticated, if all checks are okay, then the request is forwarded to the order service. Keycloak supports OAuth 2.0 which is a protocol used for authorization by a third party to get permission to access sources from a service [36].

Service discovery provided by Spring Cloud Netflix[3] is another library used from the Spring Cloud framework. Finding and using the available services is an important component for microservice applications, and is one of the major challenges in cloud computing [37]. Because microservices can be run with multiple instances, and each microservice can run different numbers of instances, it is crucial to identify which instances are available. Spring Cloud Netflix's service discovery library is used in the e-commerce systems implemented. A discovery server is implemented and all the other

[1] https://www.postgresql.org/

[2] https://www.keycloak.org/

[3] https://spring.io/projects/spring-cloud-netflix

microservices are identified as clients. The service discovery mechanism automatically registers or unregisters the microservices depending on their availability to the discovery server's registry.

In the e-commerce system, synchronous and asynchronous communication happens between microservices. Synchronous communication involves two parties: a client and a server. The client executes a request to the server and the server immediately returns a result. However, in asynchronous communication, a message bus is responsible for delivering messages between microservices [38]. Asynchronous communication is based on the fire-and-forget principle. The message sender sends a message to the message bus and continues with other processes, it does not wait for a response. It is the message bus' responsibility to deliver the message to the receiver. Asynchronous communication provides non-blocking communication between the message sender and receiver parties. Each communication type has pros/cons and has applicable use cases. In the e-commerce system using the 2PC transaction consistency pattern, synchronous communication is used via REST protocol. In Saga Orchestration and Saga Choreography systems, Apache Kafka[1] is used as a message bus for asynchronous communication.

Before starting the implementation of e-commerce applications, existing ready-to-use e-commerce microservice implementations available online are considered as alternatives. Microsoft's eShopOnContainers[2] is the most popular open-source e-commerce application by looking number of stars on their GitHub[3] project. Also, design, documentation, and tutorials made the project a good starting point, however, the project tech stack is in C# and .NET framework, which the author of this thesis did not have experience with. Vert.x[4], an open-source, reactive, and polyglot software development toolkit, has an e-commerce microservice application[5] to illustrate how to implement applications within Vert.x toolkit, however, the project has limited documentation.

---

[1] https://kafka.apache.org/

[2] https://learn.microsoft.com/en-us/dotnet/architecture/cloud-native/introduce-eshoponcontainers-reference-app

[3] https://github.com/dotnet/eShop

[4] https://vertx.io/

[5] https://github.com/sczyh30/vertx-blueprint-microservice

Some other open-source microservice-based e-commerce applications are considered as a starting point. A REST Microservices architecture for e-commerce[1] has modern tools like Spring Boot, Spring Cloud, and Spring Cloud Netflix, but does not have message bus implementation in it and is poorly documented. Another considered open-source project was e-Commerce-boot μServices[2]. It is well documented, well designed, and has modern tools but the project was not complete. Finally, Spring Boot Microservices[3] has been chosen to be the starting point for this thesis project because it is well designed, and has modern tools for microservice development like Spring Boot, Spring Data, Spring Cloud libraries, Zipkin for distributed tracing, Prometheus and Grafana for monitoring, docker for containerization, Rest for synchronous, Apache Kafka for asynchronous communication and has application-wide hands-on tutorials. Spring Boot Microservices[4] project was used just as a starting point in the thesis projects but microservices, logic, and related applications are customized.

4.1 Two-Phase Commit

The first transaction consistency pattern implemented in the e-commerce system is 2PC. As illustrated in Figure 5, order operation is requested by a user via a front-end application. Front-end applications here can be a variety of options like a web application, a mobile application, or a tablet application. Front-end applications are not implemented in this thesis because it is out of the scope of distributed transaction patterns, but front-end applications are shown in Figure 5 to give a broad overview of how users would interact with the system in actual usage. For testing purposes, Postman, which is a popular API (Application Programming Interface) platform for building and using APIs, is used to create the requests just like a front-end Wapplication would do. When users initiate an order via a front-end application, the front-end application sends the request to API Gateway. API Gateway validates the user's authentication by checking the authentication token against the authentication server using OAuth 2.0. In case the user is authenticated, API Gateway forwards the request to the corresponding service, in this case, it is the order

[1] https://github.com/RainbowForest/e-commerce-microservices

[2] https://github.com/SelimHorri/ecommerce-microservice-backend-app

[3] https://github.com/SaiUpadhyayula/spring-boot-microservices

[4] https://github.com/SaiUpadhyayula/spring-boot-microservices

service. After API Gateway, the request is in the microservices environment. 2PC e-commerce system implementation can be found in the GitHub repository[1].

Figure 5 shows that there are five microservices in the system. User microservice is responsible for providing operations for user data stored in the user tables such as *user_id*, *name*, *email_address*, and *user_addresses*. The "user" keyword is a special keyword in the PostgreSQL database, so the "t" prefix was added to the table name. The *t_user* table and *user address* relation is a one-to-many relation meaning, one user can have multiple addresses. The user database diagram is shown in Figure 6. User service has only read-only functionality in the ordering of a product context. When the order service receives the order request, the *user_id* in the request is checked with the user service if the user exists or not. User service is only used to check user data in the ordering of a product flow, before the actual flow starts. User service is not included in the compensation transactions. Numbers on arrows on the Figure 5 show the order of requests in a successful execution of ordering a product in the e-commerce systems.

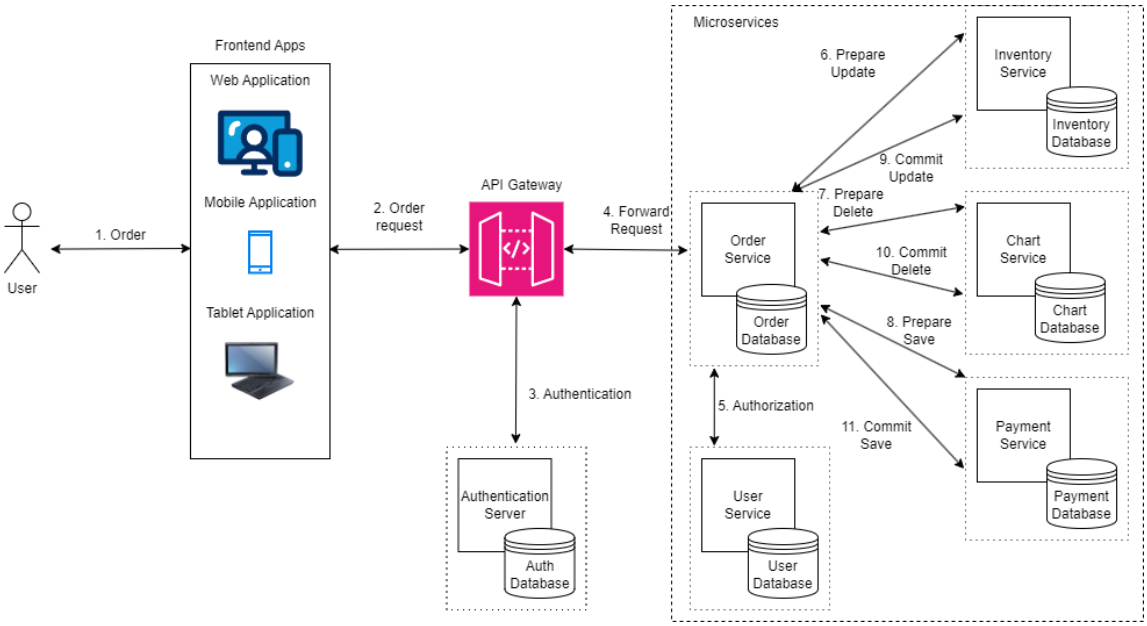

Figure 5. 2PC in the e-commerce system.

[1] https://github.com/mstftikir/e-commerce-microservices-2pc

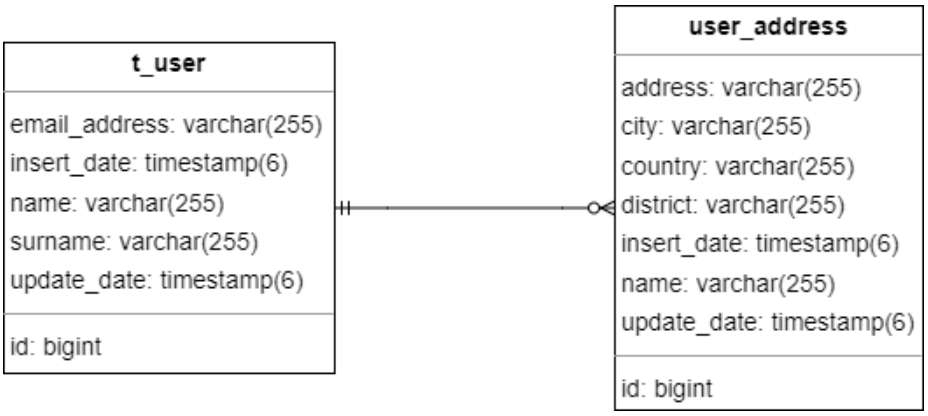

Figure 6. User database diagram (ERD using Crow's foot notation).

Entity Relationship Diagram (ERD) is used in this work to express the table relations as can be seen on Figure 6. ERD is a high-level conceptual model that describes information as entities, attributes, relationships, and constraints [39]. For example, as seen in the Figure 6, t_*user* and *user_addresses* tables have one mandatory to many optional relationship as illustrated on the arrow edges.

The *insert_date* and *update_date* columns exist in almost all the tables created in the e-commerce system. The insert date column stores the first date the entity was inserted, updated date stores the latest update date to the entity. The same logic applies to all other tables. Also, the *id* column exists in all tables which provides uniqueness for entities. The *id* column's data is an auto-generated value.

Order microservice is responsible for providing operations for order data stored in the order database but also initiating and controlling the requests to other microservices. As shown in Figure 7, the *t_order* table stores the *event_id*, *user_id*, and *total_price*. Like the user table, there is "t" prefix for the order table because "order" is a special keyword in the PostgreSQL database. There are two related tables with the *t_order* table. The first one is the *order_event_status*, which stores event statuses in subsequent microservices and has a one-to-one relation with the t_order table. The second table is the *order_item* table, which stores related order item data like inventory code, quantity, and price, and has a one-to-many relation with the *t_order* table. When the order service receives the order request, the first thing it does is validate if the user is active. The second thing the order service does is store the data as it is which indicates the start of an ordering a product flow. The third thing the order service does is, execute requests to subsequent services and control the data flow.

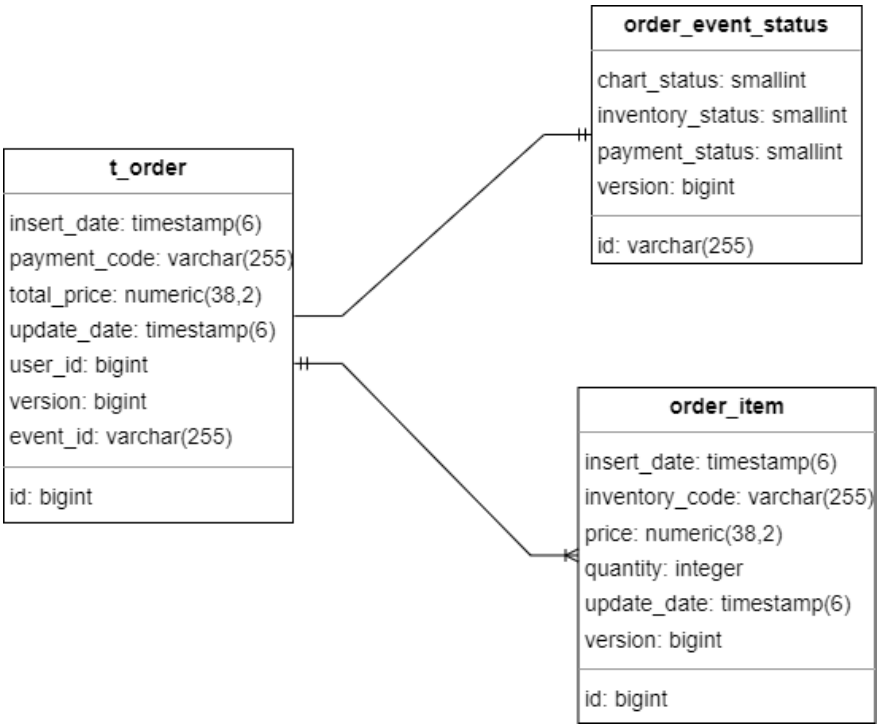

Figure 7. Order database diagram (ERD using Crow's foot notation).

In the 2PC e-commerce system, the order service is responsible for controlling the data flow. As explained in Chapter 3, the first phase executed is the *Prepare* phase. Order service executes prepare requests synchronously with REST protocol to the inventory, the chart, and the payment microservices. If any of the services return fail, execution is stopped, and an error is returned. If all the services return a success message, the second phase takes place, the *Commit* phase, in which commit messages are sent to the inventory, the chart, and the payment microservices. If any of the services return fail messages, other services are triggered to be rolled back and an error message is returned. If all calls to subsequent services finish successfully. Successful execution of the 2PC in the e-commerce system is shown in Figure 8 as a Unified Modeling Language (UML) sequence diagram.

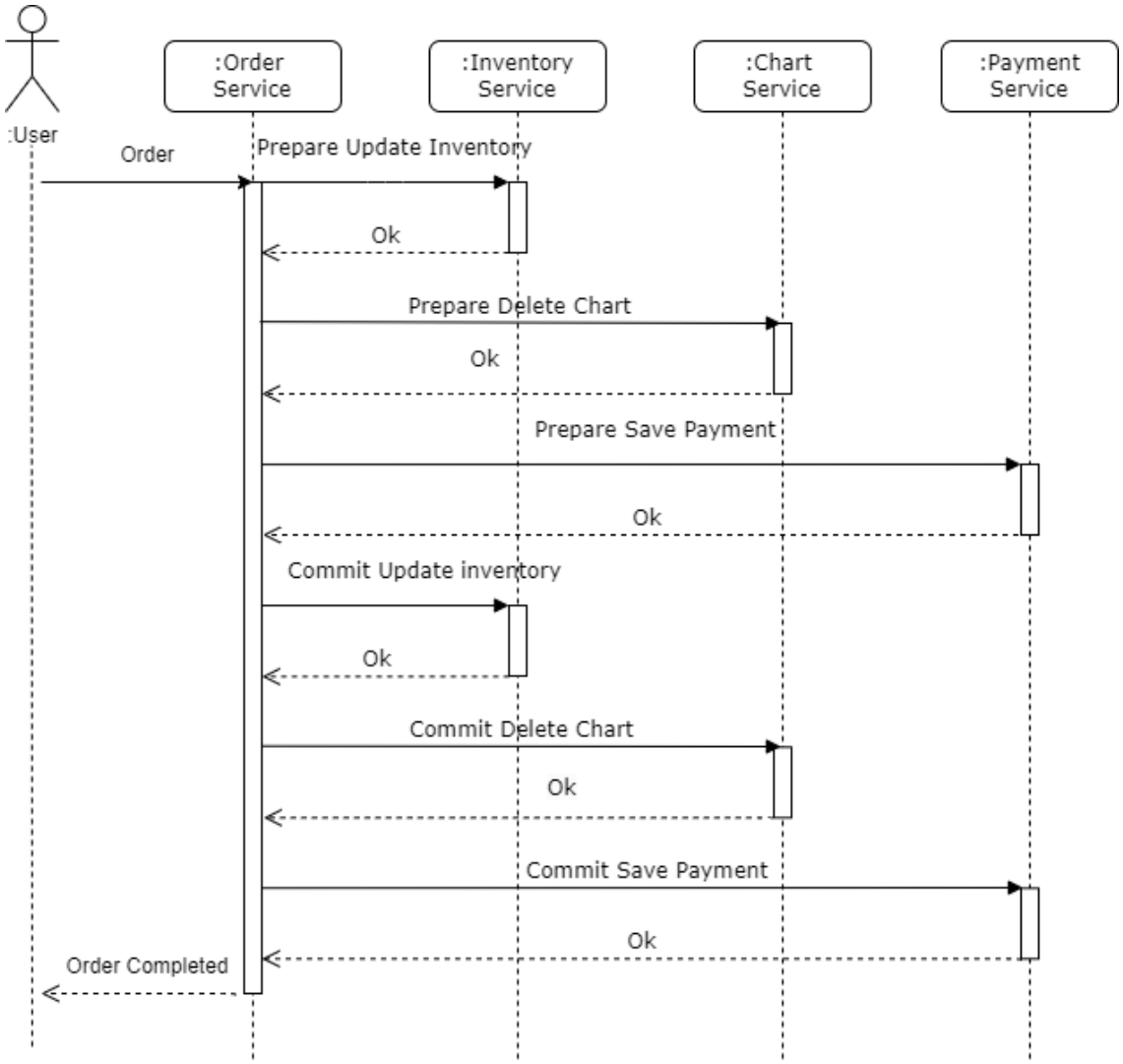

Figure 8. 2PC UML sequence diagram for placing an order.

The inventory microservice is responsible for providing inventory operations like updating inventory, Inventory data is stored in the inventory database. Figure 9 shows the inventory database diagram, which only contains an *inventory* table with *code*, *description*, *name*, *price*, and *quantity* data. Quantity is the updated (reduced) field in the order request. For example, product quantity is equal to 100. When an order request contains two items, At the end of the successful order execution, the update on the inventory item results in 98. If the order request fails for any reason, the quantity is rolled back to 100.

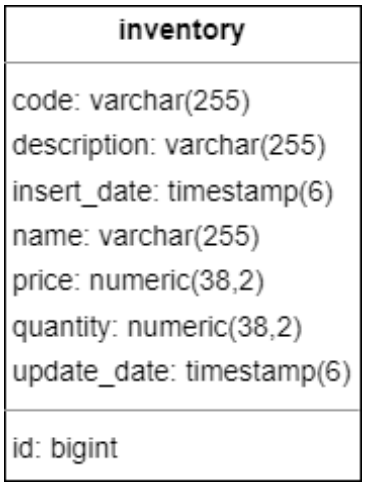

Figure 9. Inventory database diagram (ERD using Crow's foot notation).

Chart microservice is responsible for serving chart data stored in the chart database. The chart database diagram can be seen in Figure 10. The chart table contains the *user_id*, *total_price*, and *active* column. one user can have only one active chart item at a time. To place an order, the user must already have an active chart item. When a successful placing order action occurs in the system, the corresponding entity's active column is set to false, this action is called soft delete. The actual delete operation is not performed here because deleting the entity is a non-revertible action. The chart table has a one-to-many relation with the *chart_item* table, meaning one chart can have multiple chart items.

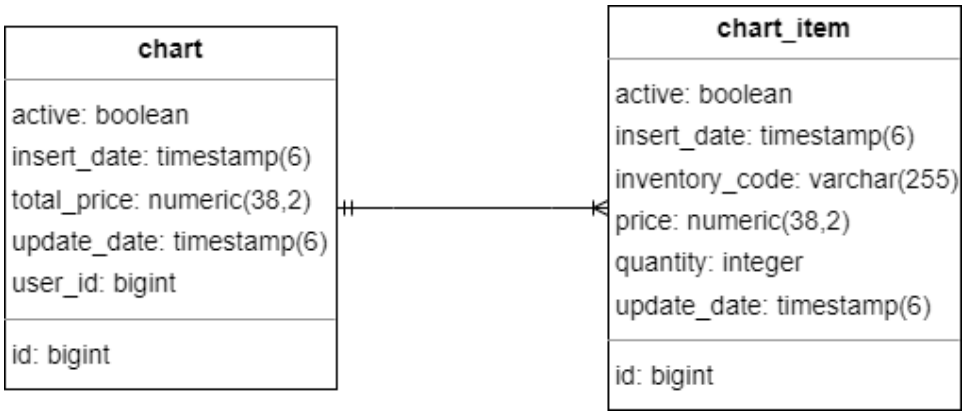

Figure 10. Chart database diagram (ERD using Crow's foot notation).

Payment microservice is responsible for providing operations for payment data which is stored in the payment database. The *payment* table, visible in the Figure 11, stores the *code*, *user_id*, *total_price*, and *active* columns. The payment table has a one-to-many relation with the payment item table. Each order corresponds to a new entity in the *payment* table. Also, each *order_item* corresponds to a new entity in the *payment_item*

table. When a payment is saved in the database but later needs to be rolled back because some other microservice failed, payment and related payment items are soft deleted meaning their active column is updated to false.

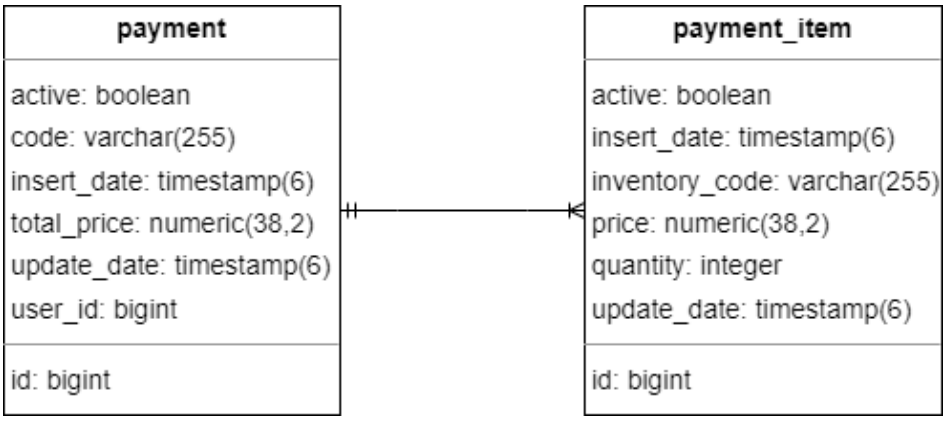

Figure 11. Payment database diagram (ERD using Crow's foot notation).

4.2 Saga Orchestration

Saga Orchestration implementation of the e-commerce system is similar to 2PC, but it involves a message bus to the system. The message bus is responsible for sending and receiving messages. Apache Kafka is used as message bus technology. All the database diagrams in microservices, which are depicted for 2PC in Figure 6, Figure 7, Figure 9, Figure 10, and Figure 11, are the same for Saga Orchestration and Saga Choreography, therefore they are not explained again for the Saga patterns. Order service acts as an orchestrator in this system. None of the microservices communicate with each other directly, but communication happens over the message bus, and the order service orchestrates the messaging. Also, none of the subsequent microservices, which are the inventory, the chart, and the payment microservices, communicate with each other. They only communicate with order service via message bus. Order service controls the system and decides what the next step is to execute depending on the state of the product ordering. As shown in Figure 12, in the Saga Orchestration there is no prepare phase like in the 2PC, nevertheless, actions are atomic actions such as updating inventory, deleting chart, and saving payment. However, this brings the consistency issue. As explained in the Distributed Transactions chapter, Saga Orchestration provides eventual consistency, not strong consistency. When an order is finished successfully or fails, the user gets notified

asynchronously. Notifying user functionality is not implemented in this thesis because it is out of the scope of distributed transactions.

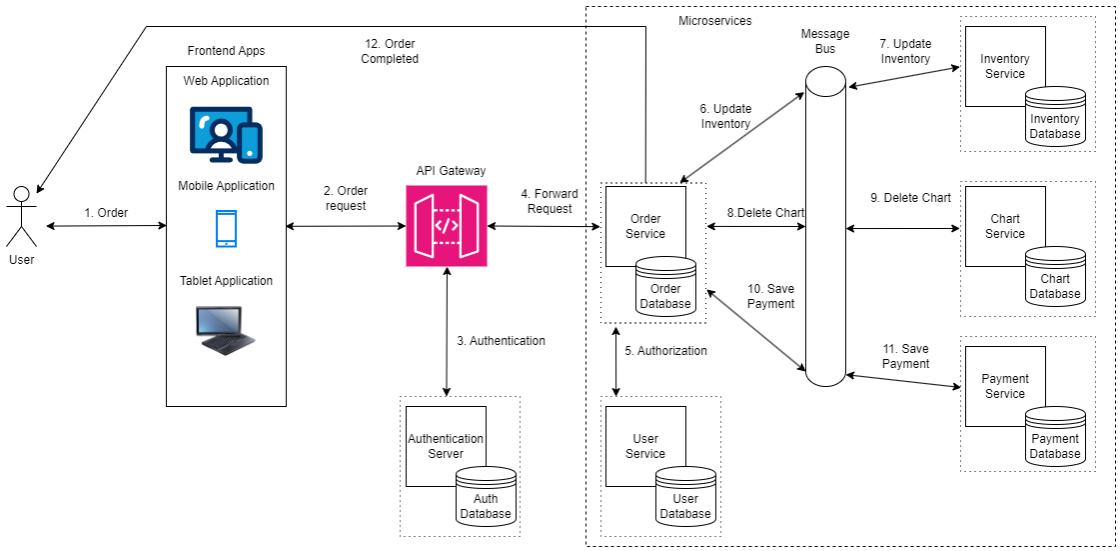

Figure 12. Saga Orchestration in the e-commerce system.

A successful execution flow of ordering a product in Saga Orchestration is seen in Figure 13. Order service acts as the orchestrator in the system and controls the flow via message bus. When the order service receives an order request it saves the order to its database as started and then, puts a message to the message bus for updating the inventory, The inventory service receives the message updates the inventory, and puts the inventory updated message to the bus. The order service receives the inventory updated message and puts a new message, a delete chart message, on the bus. The chart service receives the message, soft deletes the chart, and puts the chart deleted message on the bus. The order service receives the chart deleted message and puts the save payment message on the bus. The payment service receives the save payment message, saves the payment, and puts the payment saved message on the bus. The order service receives the payment saved message and sends the order completed message to the user. When a microservice fails to do action what it needs to do, the orchestrator initiates the proper rollback operations so that the system remains in a consistent state. and the user is notified that the order has failed. Because the system is asynchronous, the event ID in order provides uniqueness for

each order and the event ID is passed in all the messages. The Saga Orchestration e-commerce system implementation can be found in the GitHub repository[1].

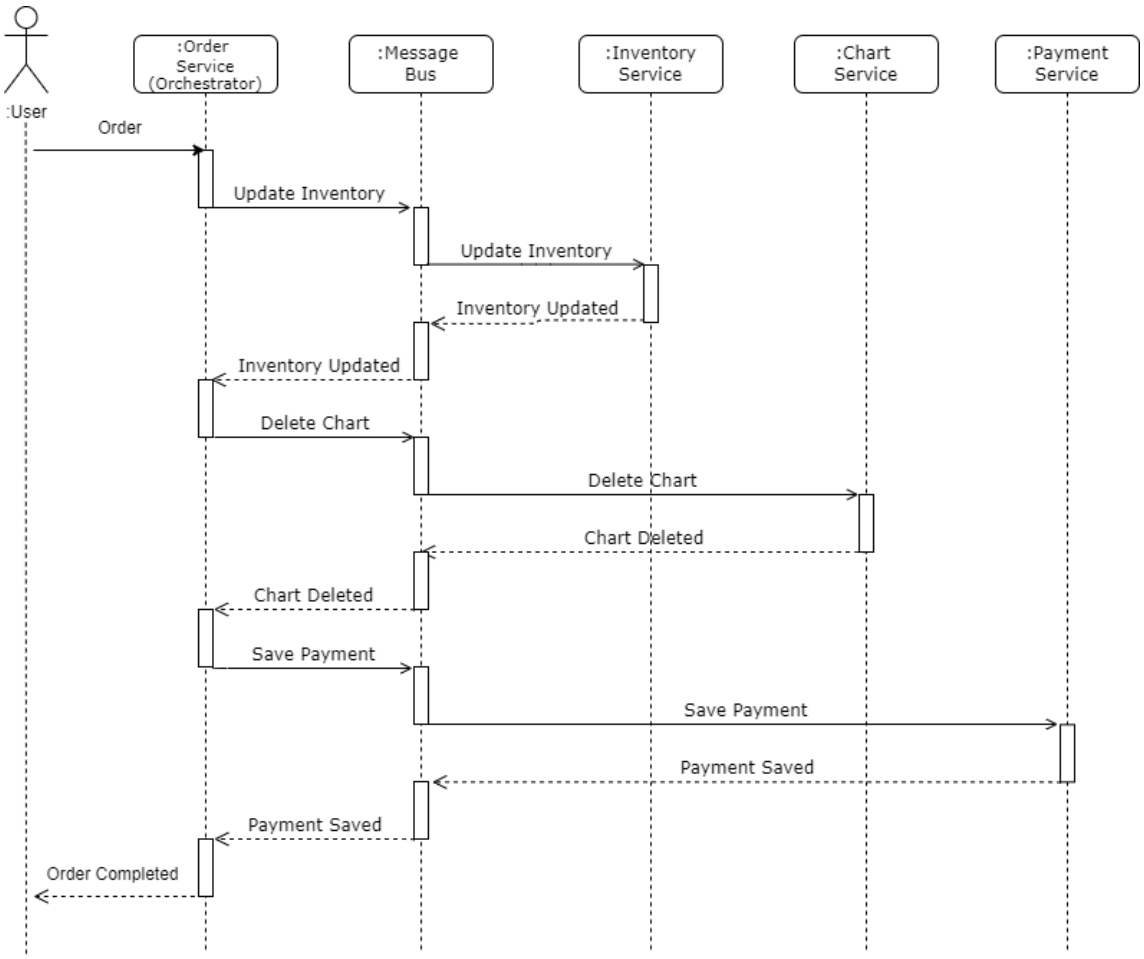

Figure 13. Saga Orchestration UML sequence diagram for placing an order.

4.3 Saga Choreography

Saga Choreography implementation of the e-commerce system is more similar to the Saga Orchestration than the 2PC. Saga Choreography also contains a message bus like the Saga Orchestration. However, the main difference between Saga Choreography and Saga Orchestration is the orchestrator in the system. As shown in the Figure 14, there is no orchestrator in Saga Choreography. So, order service does not act as an orchestrator. When the order service receives an order request, it saves the request as initialized and puts a message to the bus for the next steps. In Saga Choreography, the subsequent

[1] https://github.com/mstftikir/e-commerce-microservices-saga-orchestration

microservices, which are the inventory, the chart, and payment microservices, are independent of sending and receiving messages between each other but messaging still happens over the message bus. When subsequent actions are finished or failed in the microservices, the order service receives the final message and sends a message to the user with the status of the order. Like in the Saga Orchestration, the event ID provides the uniqueness of orders.

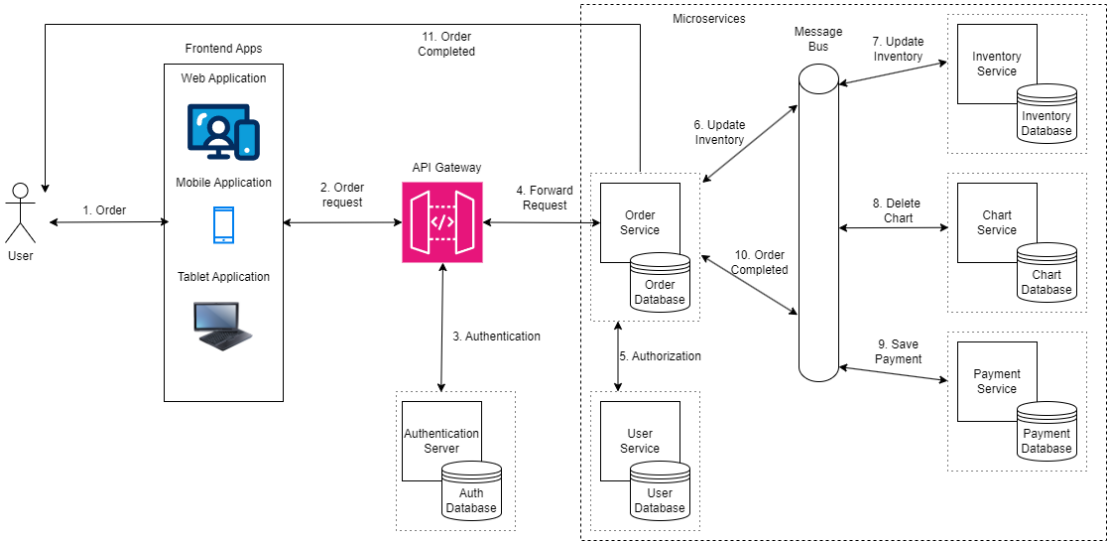

Figure 14. Saga Choreography in the e-commerce system.

A successful execution flow of ordering a product in Saga Choreography is seen in the Figure 15. When the order service receives an order request it saves the order to its database as started and then, puts an update inventory message on the bus. The inventory service receives the update inventory message, updates the inventory, and puts a delete chart message on the bus. The chart service receives the delete chart message, soft deletes the chart, and puts the save payment message on the bus. The payment service receives the save payment message, saves the payment, and puts an order completed message on the bus. The order service receives the message and notifies the user about the order status. So, order service involves the flow at the beginning and the end. The Saga Choreography e-commerce system implementation can be found in the GitHub repository[1].

[1] https://github.com/mstftikir/e-commerce-microservices-saga-choreography

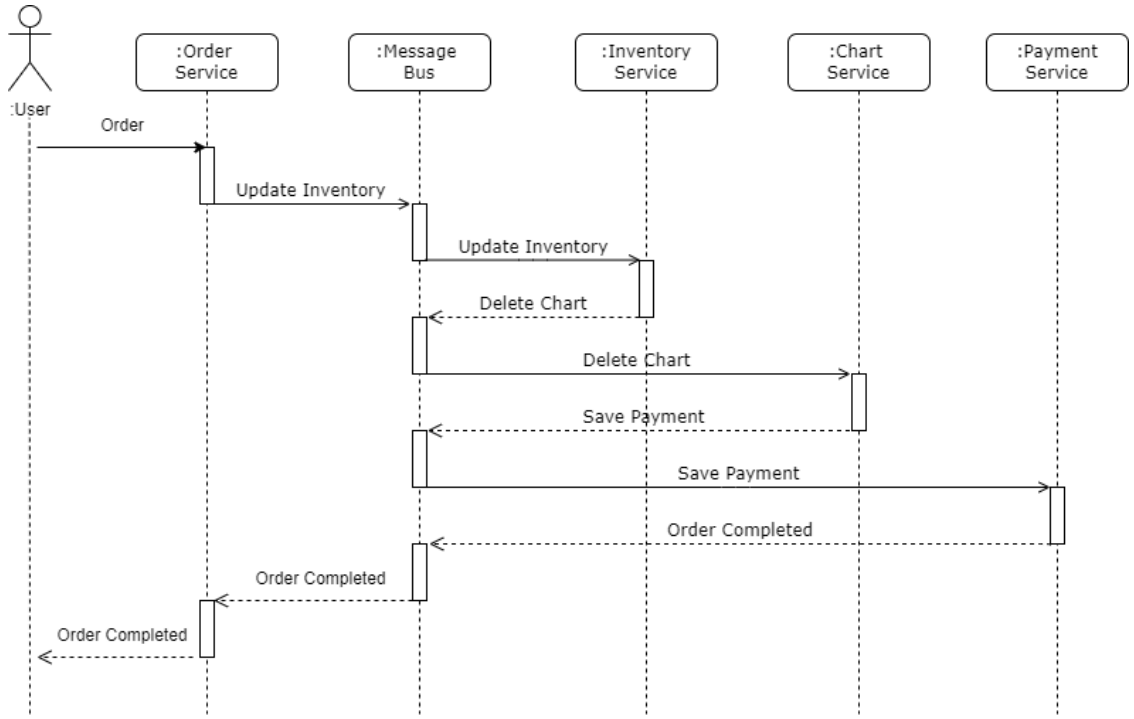

Figure 15. Saga Choreography UML sequence diagram for placing an order.

5 Experiments and Results

To answer RQ1-RQ3, three e-commerce systems using different MSA patterns: 2PC, Saga Orchestration, and Saga Choreography were implemented. First, the system design is compared (RQ1) for the three implementations. Next, the performance of the three implementations is compared (RQ2). Finally, system update scenarios are tested (RQ3).

5.1 Systems Design Comparision

RQ1, how much does the choice of transaction consistency pattern affect overall MSA design, was analyzed under two main categories: code metrics and communication overhead. Code metrics were evaluated via a static code analysis tool named SonarQube[1]. It is an open-source platform for detecting bugs, code smells, and code metrics. SonarQube was integrated into e-commerce systems by using the official image of SonarQube provided in the Docker Hub[2]. Static code analysis reports are produced by the Maven command line tool and uploaded to a locally running SonarQube server. SonarQube provides a UI to observe the analysis results.

Three main code metrics, which are cyclomatic complexity, cognitive complexity, and line of codes are considered in this thesis work. SonarQube measures cyclomatic complexity by starting the values of one and incrementing by one whenever it detects a split in the control flow of a function. Essentially, a higher cyclomatic complexity implies more branching in the code [40]. Whereas cognitive complexity is measured by how hard the control flow of a method is to understand and maintain. Cognitive complexity differs from the practice of using mathematical models to assess software maintainability. Cognitive complexity starts from the precedents set by cyclomatic complexity but uses human judgment to assess how structures should be counted and to decide what should be added to the model. The cognitive complexity is given by a positive number which is

[1] https://www.sonarsource.com/products/sonarqube/

[2] https://hub.docker.com/_/sonarqube

increased every time a control flow sentence appears. Control flows' nested levels also contribute to the cognitive complexity of a method [41]. In summary, the difference between cyclomatic complexity and cognitive complexity is understandability. The lines of code metric in SonarQube is straightforward, it counts the lines of code in the project.

Cyclomatic complexity analysis results are shown Figure 16 per distributed transaction pattern. In each pattern's implementation, the cyclomatic complexity results can be seen as all and per microservice main service class. Service classes are the place where the logic of the microservice is placed. It can be deduced from the Figure 16 that Saga Orchestration has the most cyclomatic complexity, and 2PC has the least cyclomatic complexity. The service class complexity shows the order service of Saga Orchestration is the most complex, and the order service in Saga Choreography is the least complex. This is an expected situation because the order service handles the orchestration in Saga Orchestration. So, the results show that 2PC is the best option out of the three patterns implemented because it provides the least cyclomatic complexity in general and particularly in service levels. Less complexity in systems results in better maintainability. However, when other restrictions like scalability force the system designers to use the Saga patterns, Saga Choreography is 45% less complex than Saga Orchestration. Therefore, Saga Choreography is preferable over Saga Orchestration in terms of cyclomatic complexity.

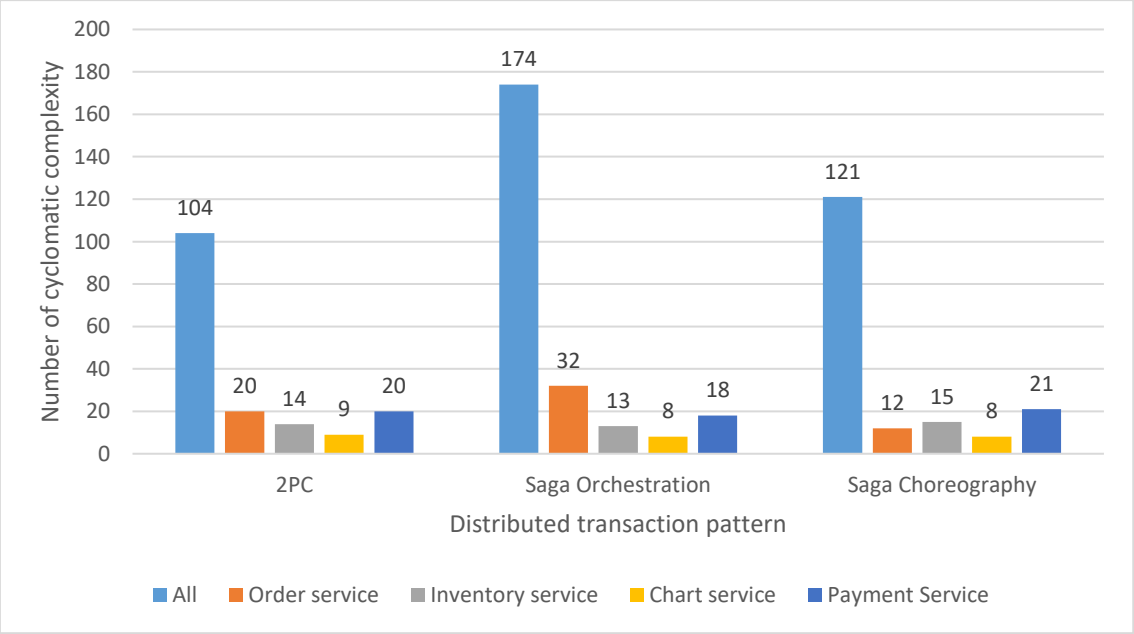

Figure 16. Comparison of cyclomatic complexity for the three MSA implementations.

Cognitive Complexity analysis results are shown in Figure 17 per distributed transaction pattern. It can be deduced from the figure that all the patterns have similar complexity. The most complex pattern is 2PC which has 38 cognitive complexity, and Saga Choreography has 35. However, the order service in 2PC, has the most complexity which is 13, whereas it is three for Saga Orchestration and two for Saga Choreography. The summary for the cognitive complexity is that even though all the patterns have similar complexity, Saga patterns have slightly less complexity. For the order service, Saga patterns have less complexity, and the complexity is distributed over other services. Because the Saga patterns have less complexity in general and in order service, they are preferable over 2PC. In the comparison between Saga Choreography and Saga Orchestration, Saga Choreography has 5% less cognitive complexity and it is the best pattern regarding cognitive complexity.

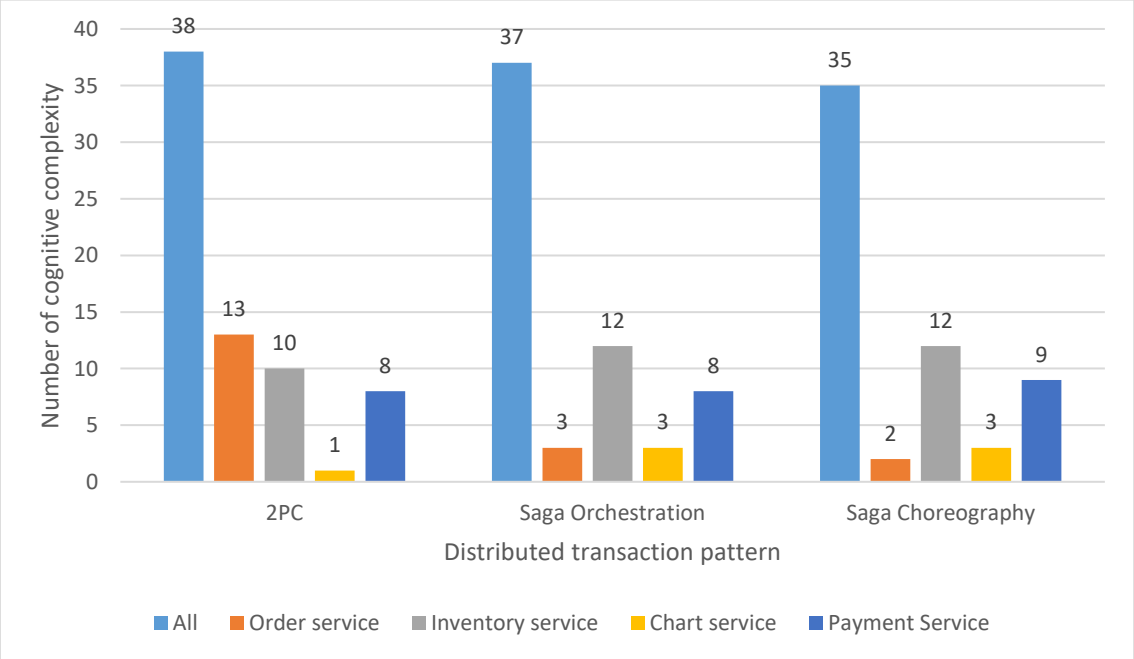

Figure 17. Comparison of cognitive complexity for the three MSA implementations.

Lines of code analysis results are shown in Figure 18 per distributed transaction pattern. The 2PC has the least lines of code because of the Two-Phase logic in which the Prepare and Commit phases use the same methods, but the action differs by a parameter. Another reason to consider is that 2PC uses synchronous communication whereas Saga patterns use asynchronous communication. Lines of code in the classes to establish the communication differ depending on the communication type. The last important metric is the lines of code in the order service class in 2PC and Saga Orchestration is noticeably

higher than the Saga Choreography. In Saga Choreography, the lines of codes in the services are more equally distributed between services. Therefore, 2PC has the least lines of code and the best out of the three patterns. 2PC has 9% fewer lines of code than Saga Choreography. However, because, Saga Choreography distributes lines of codes to services, Saga Choreography becomes the more maintainable pattern.

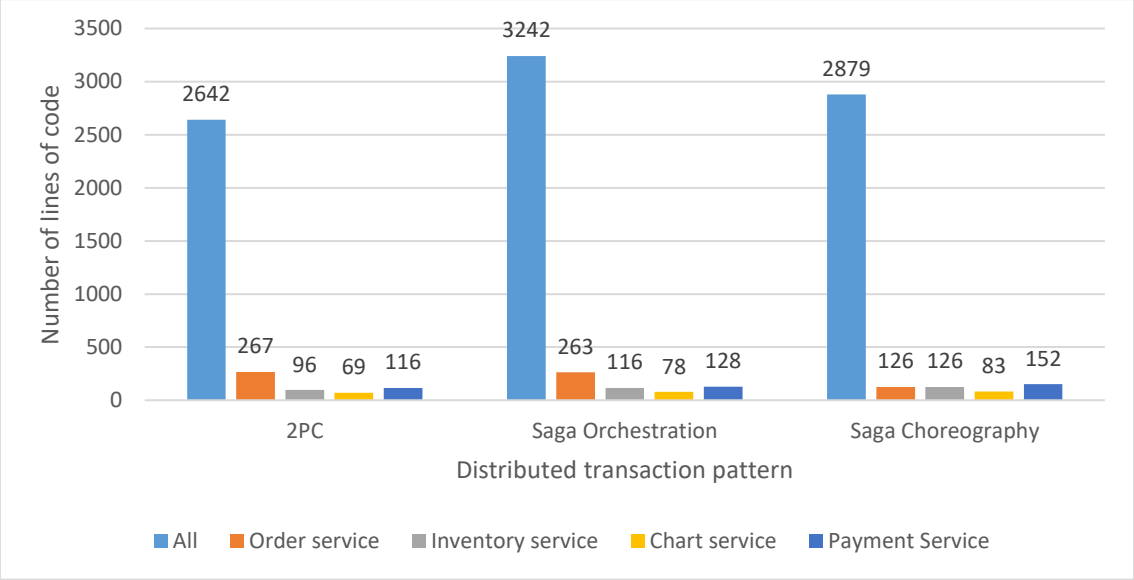

Figure 18. Comparison of lines of code for the three MSA implementations.

The other topic researched within the RQ1 is communication overhead. Communication overhead refers to how much extra communication happens to handle failure scenarios in the system. In monolith architectures, because there is only one big application, placing an order would happen inside one transaction scope. So, users of the system would have one request and one response for placing an order, a total of two interactions, with the system. However, in MSA, each application creates its transaction scope when requested, and the number of transactions in the system grows.

Figure 8 shows a successful execution of placing an order in the 2PC pattern. In 2PC, one interaction to start the order, 6 requests to subsequent services, 6 replies from subsequent services, and finally one more interaction to let the user know the state of the order takes place. In total 14 sequences of interactions happen. Those interactions are Order, Prepare Inventory Update, Ok, Prepare Delete Chart, Ok, Prepare Save Payment, Ok, Commit Inventory Update, Ok, Commit Chart Delete, Ok, Commit Save Payment, Ok, and Order Completed. Figure 13 shows 14 interactions for the successful execution of placing an

order in the Saga Orchestration. Figure 15 shows 10 interactions for the successful execution of placing an order in the Saga Choreography.

In distributed transaction systems, cases in which a request fails in one of the microservices are also important to analyze because systems are designed to handle these fail scenarios with compensation transactions. In all the three e-commerce systems implemented in this thesis, the payment service is the latest in the chain of execution. The total number of interactions between services in case the latest service fails is an important indicator of systems design. Figure 20 shows when the payment service fails to save payment in 2PC, 18 interactions happen. Four compensation interactions take place Rollback Update Inventory, Ok, Rollback Delete Chart, and Ok to keep the system in a consistent state. Figure 21 shows a similar case for Saga Orchestrator, where there is a total of 22 interactions happening, and 8 compensation interactions are happening. Figure 22 shows a similar case for Saga Choreography, the total interaction count is 14 and the compensation interaction count is four.

Figure 19 shows that the Saga Orchestration is the chattiest pattern, which has the most interactions with the number 22, in case of payment service failure. Even though Saga Orchestration has the same number of interactions with 2PC in the success case, in the failure case 8 compensation interactions take place. On the other hand, Saga Choreography has the least number of interactions in the success case which is 19 and the payment fails case which is 14. A low number of interactions makes Saga Choreography preferable in terms of communication overhead.

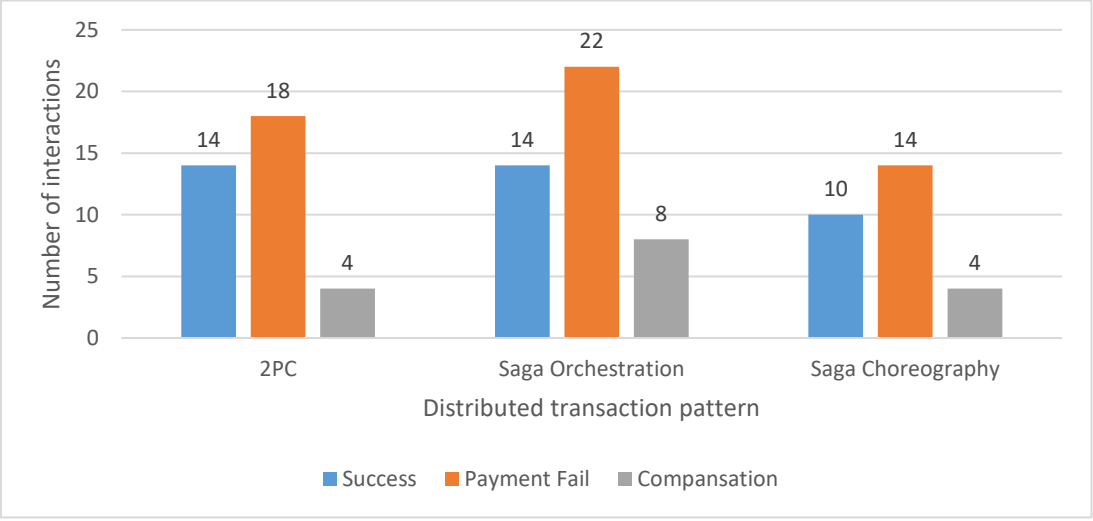

Figure 19. Comparison of the number of interactions for the three MSA implementations.

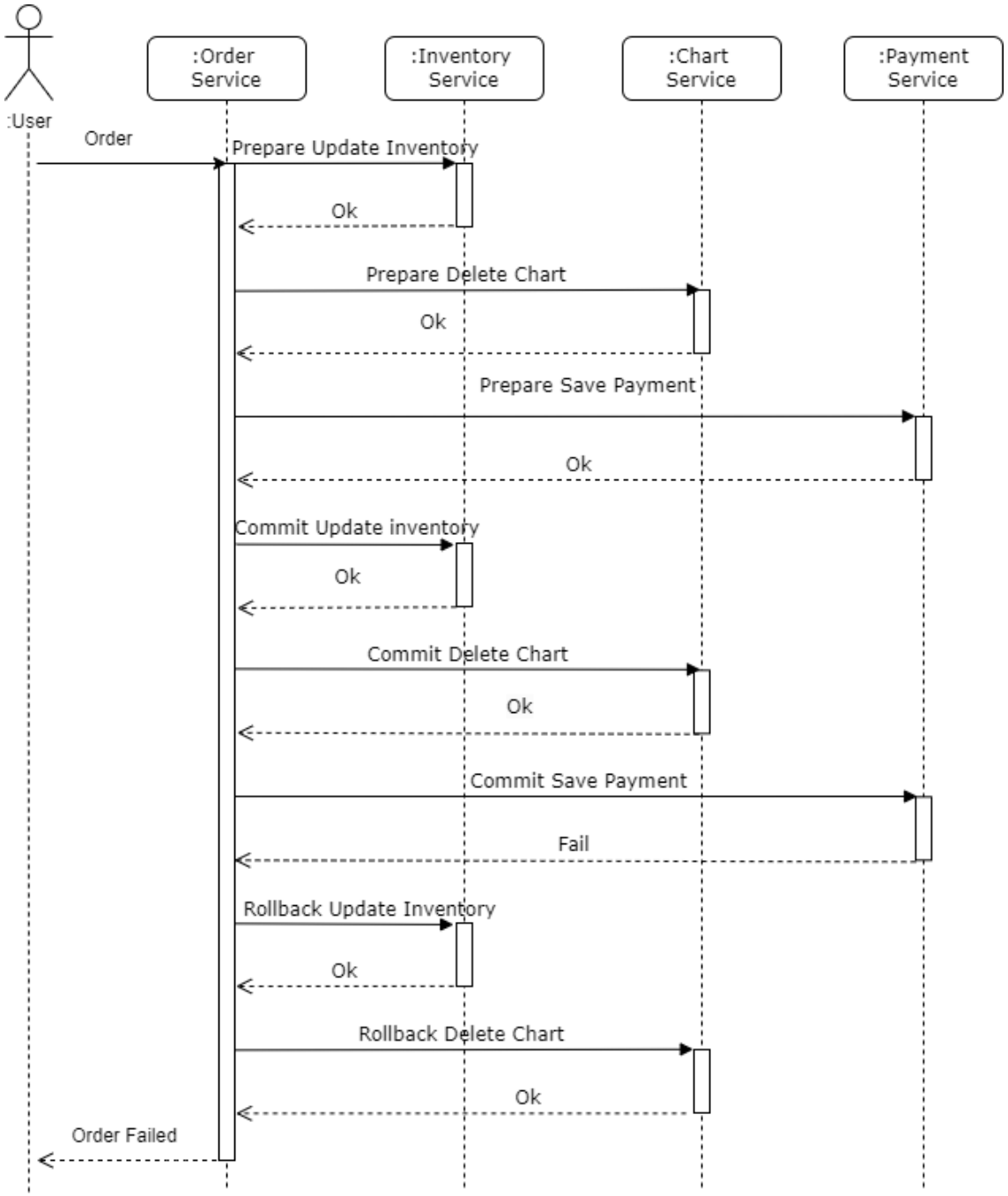

Figure 20. 2PC payment fails case UML sequence diagram.

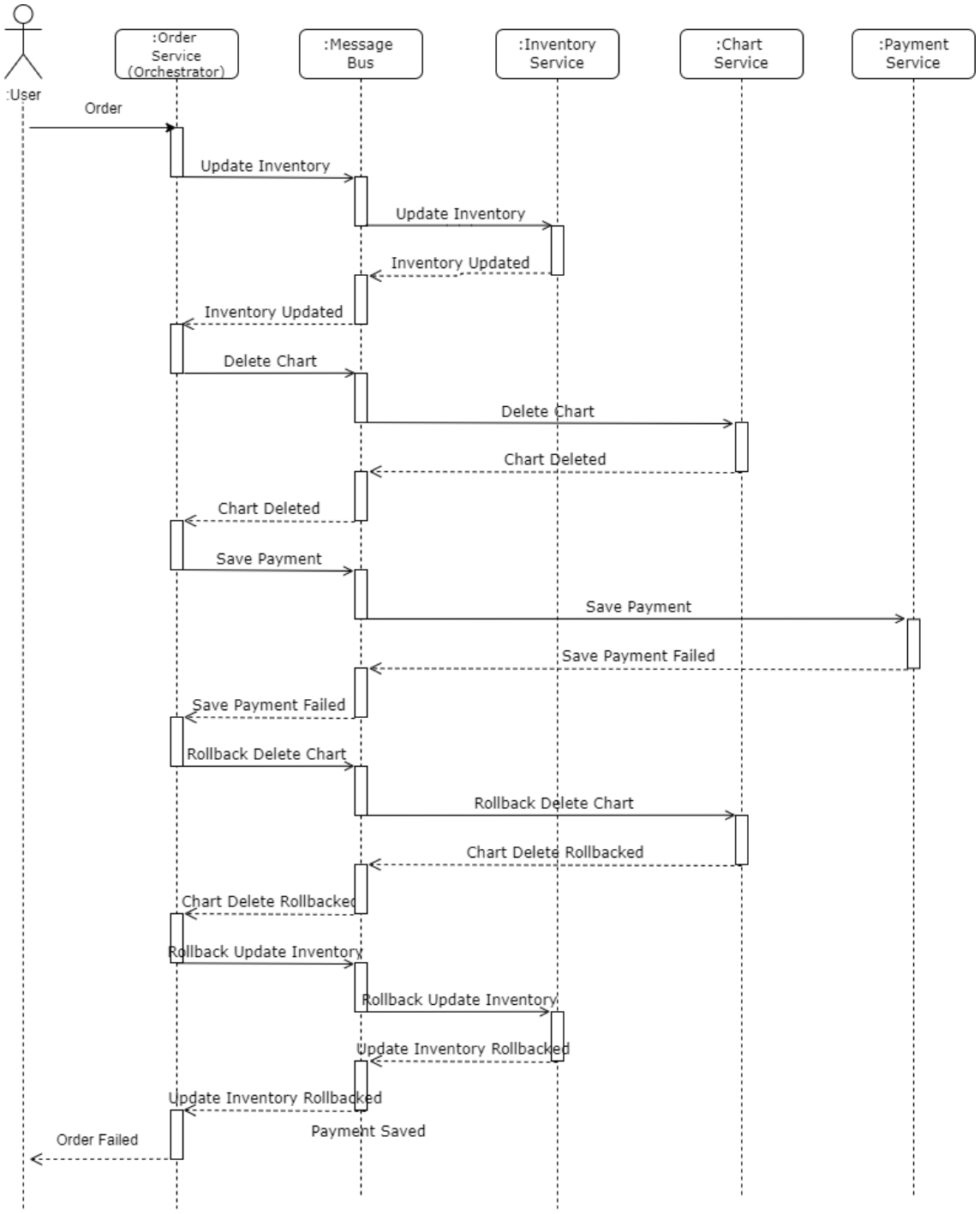

Figure 21. Saga Orchestration payment fails case UML sequence diagram.

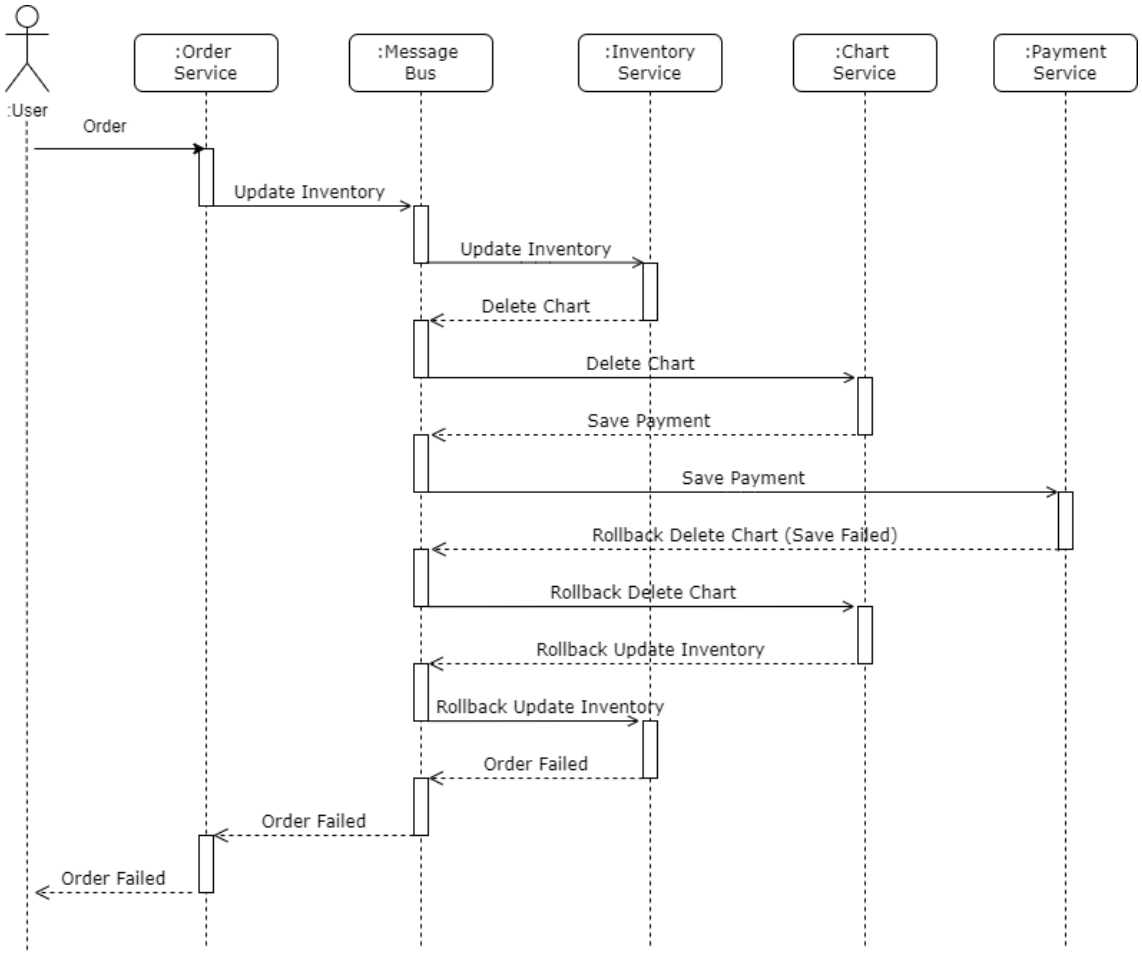

Figure 22. Saga Choreography payment fails case UML sequence diagram.

Experiments related to RQ1 show that the chosen distributed transaction pattern has design effects over systems implemented in MSA. In terms of cyclomatic complexity, 2PC has the least complexity among the three patterns however, Saga Choreography has 45% less cyclomatic complexity than Saga Orchestration and this makes Saga Choreography a better choice than Saga Orchestration. In terms of cognitive complexity, Saga Choreography has the least complexity and it has 5% less complexity than the Saga Orchestration. 2PC has the most cognitive complexity. In terms of lines of code, 2PC has the least lines of code with a 9% difference with Saga Choreography, however, because Choreography has distributed the lines of code to its services equally, it makes Choreography preferable in terms of maintainability. Finally, the last design metric, communication overhead, shows that Saga Choreography has the least overhead among the three patterns in both best-case and worst-case scenarios. So, the design experiments and results show that none of the patterns have the best results for all the aspects of system complexity, each has trade-offs where one or the other performs better. However, Saga Orchestration does not perform best in any of the experiments, so it makes Saga

Orchestration the weakest option in terms of design choice. 2PC has the best scores in cyclomatic complexity and lines of code. Saga Choreography has the best scores in cognitive complexity and communication overhead. Therefore, in terms of design, 2PC or Saga Choreography may be chosen as a distributed transaction pattern of a system depending on system requirements and constraints.

5.2 Systems Performance Comparision

RQ2, what is the impact of various transaction consistency patterns on the performance of microservices, was analyzed with response time, transaction throughput, CPU usage, and memory usage metrics. Experiments were run on the computer of the thesis author. The computer has an 11th Generation Intel Core i7-11800H @2.30GHz processor and 64,0 GB Random Access Memory (RAM). All the applications implemented in the e-commerce system ran in the Docker containers. Even though the computer ran Windows 10 operating system (OS), Docker containers could be run on different OS like Linux, MacOS, etc. The interactions with the e-commerce systems were done via Postman. Load to systems was created by Postman collection runner which can simulate up to 100 users requesting simultaneously.

To observe response time, transaction throughput, CPU usage, and memory usage metrics systems were loaded with 100 users requesting to the systems simultaneously for one minute. These tests were repeated 10 times for each MSA implementation in order to get better coverage of test results for response time, transaction throughput, CPU usage, and memory usage metrics. In the successful execution of ordering a product scenario with 100 users for one minute, the 2PC was capable of processing around 2900 requests, the Saga Orchestration around 3100, and the Saga Choreography around 4500. In each pattern, every execution had fluctuation around 100 requests, because of the computer background processes. When some other processes were consuming computer resources, the computer had fewer resources available for tested processes such as ordering a product in the e-commerce system. Therefore, background processes affected the experiment results, however, the results still provide answers to RQ2 with response time with load, transaction throughput, CPU usage, and memory usage metrics.

Figure 23 illustrates the average response times of each e-commerce system developed with different distributed transaction patterns in cases when the ordering of product

finishes successfully, fails in the inventory service, fails in the chart service, and fails in the payment service. Response time refers to the time taken from the start of the order request to finish. Figure 23 shows the average response times without load in which 100 interactions with one millisecond (ms) delay. Figure 24 shows the average response times under load in which 100 users simultaneously execute requests to the order product service for one minute. Note that failures in 2PC happen in the first phase which is Prepare phase. Average response times (A) are calculated by Equation 1 in which "a" is the last update time of an order, "b" is the insert time of an order, and "n" is the number of orders in the series.

$$A = \frac{1}{n} \sum_{i=1}^{n} a_i - b_i \quad (1)$$

Both Figure 23 and Figure 24 show that Saga Choreography performs the best in all conditions except the payment service fails without load case, yet the difference with the 2PC is only three ms. Results show that Saga Choreography is preferable in terms of response times. The other noticeable output is the Saga Orchestration performs the worst when payment service fails. It makes the Saga Orchestration the weakest choice for the worst-case scenarios. When each transaction pattern without load and under load is compared, the Saga Chorography has the least growth in response times. Success case in Saga Choreography does not change without load and under load, however, 2PC's average response time grows 13 ms and Saga Orchestration's average response time grows 16 ms.

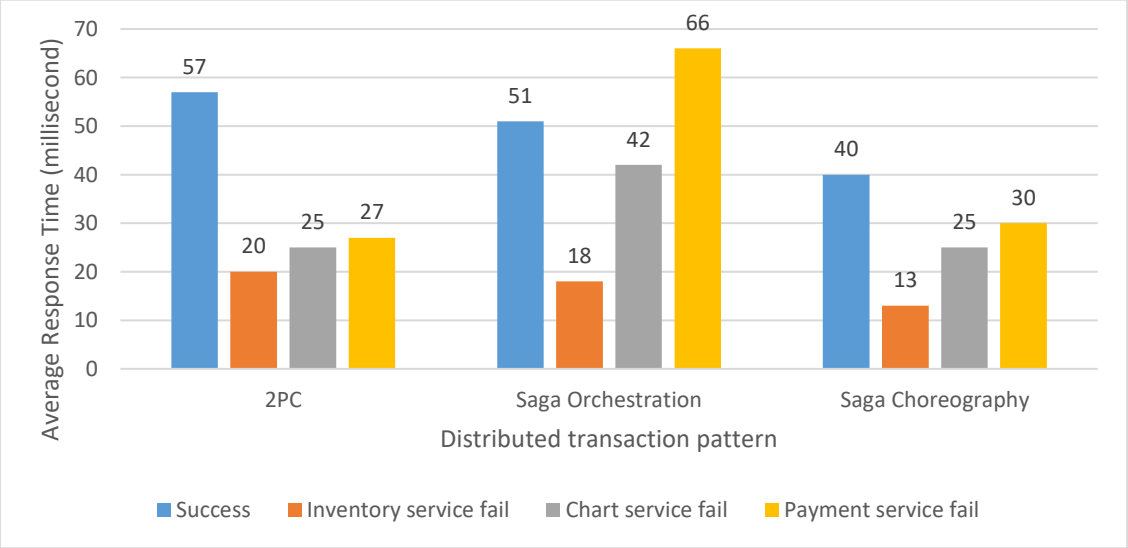

Figure 23. Response times without load for the three MSA implementations.

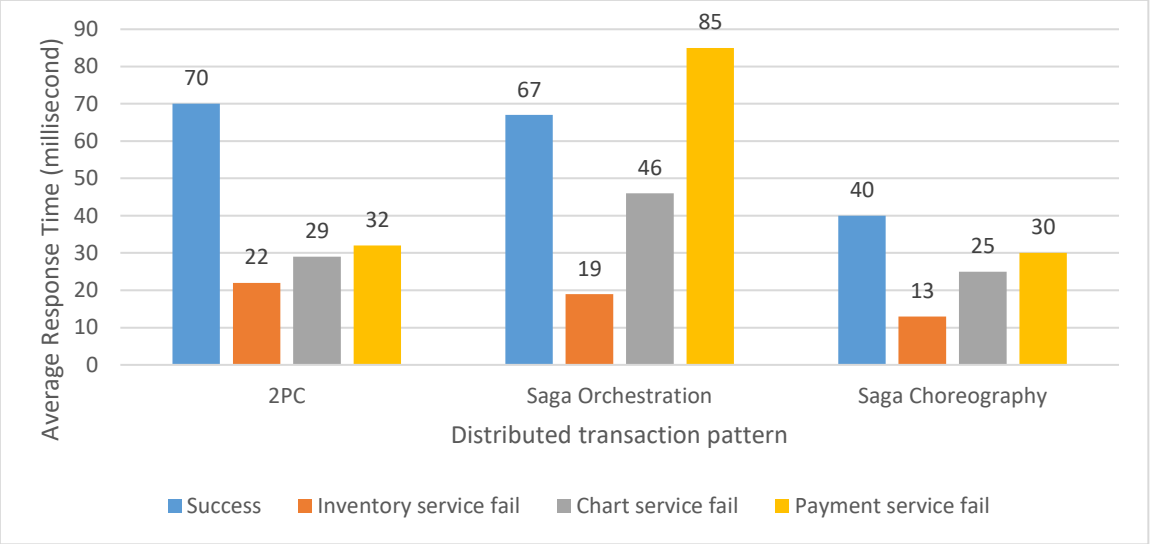

Figure 24. Response times under load for the three MSA implementations.

The second metric showing the system performance is transaction throughput. Transaction throughput is the capability of a system to handle transactions in a given time frame. A common unit to represent transaction throughput is the Transaction Per Second (TPS) which depicts the capability of the system to process a number of transactions in a second. Transaction refers to from the start of ordering a product to finish. For calculating the TPS, the systems were loaded with 100 users requesting the systems simultaneously for one minute. Tests were run 10 times for each MSA implementation and the average of the total results are considered. At the end of one minute, on average, 2938 transactions were completed for 2PC, 3216 for Saga Orchestration, and 4480 for Saga Choreography. Transaction Per Second (TPS) is calculated by Equation 2 where T is the number of transactions, and S is time in seconds, which is 60 in this case.

$$TPS = \frac{T}{S} \qquad (2)$$

Figure 25 shows that Saga Choreography has the best score with 74.6 TPS. The 2PC and Saga Orchestration have quite close TPS, the difference is only 4,7 TPS. It needs to be considered that the Saga frameworks are eventually consistent. For systems such as banking in which strong consistency is needed, 2PC is a good candidate. However, as Figure 25 depicts Saga Choreography is capable of processing transactions in one second 34% more than 2PC and 28% more than Saga Orchestration. Therefore, Saga Choreography is the natural choice for systems that are required to have better transaction throughput.

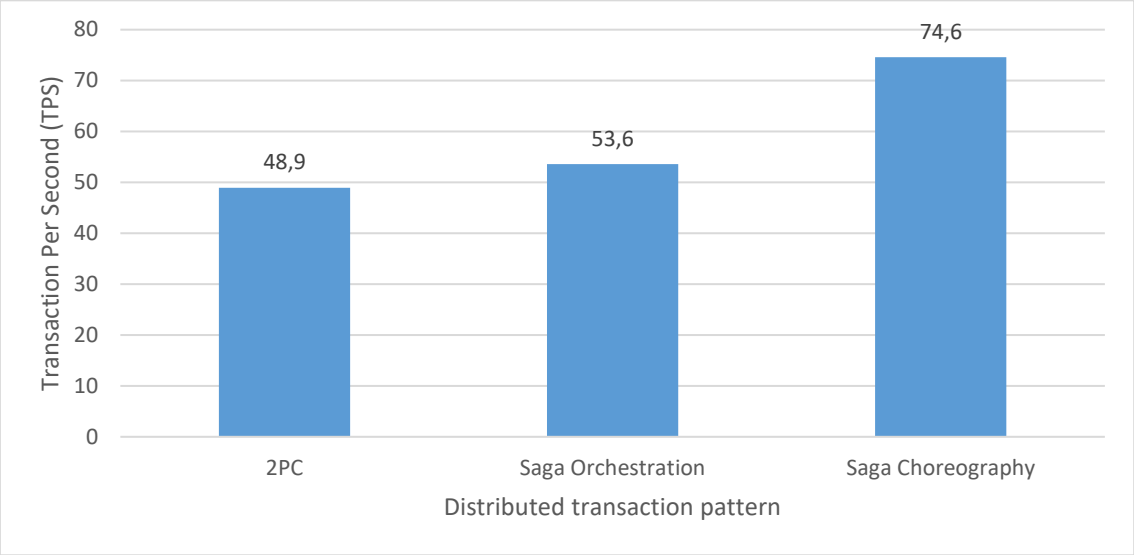

Figure 25. Transaction throughput for the three MSA implementations.

The third and fourth metrics showing the system performance are CPU usage and memory usage. For measuring CPU and memory usage, the Spring Actuator[1], Prometheus[2], and Grafana[3] were used. Spring Actuator exposes operational information about running applications such as health, metrics, info, etc. Prometheus scrapes the metrics via the Spring Actuator endpoint and stores them in a time-series database. Grafana uses Prometheus as a data source and visualizes the metrics in a more human-readable format like charts, graphs, etc.

For both the CPU usage and memory usage experiments systems are loaded with 100 users for one minute. The results show average CPU usage and memory usage. 30 seconds before and 30 seconds after one minute of execution time are included in the average. So, the total of two minutes is considered in the results. Figure 26 shows the third metric, which is CPU usage. CPU usage is illustrated per service in a distributed transaction pattern and as a total (sum of services). Each docker image is assigned to one virtual CPU and the values in the Figure 26 show the usage percentages of one CPU. Full usage of a CPU is considered as 100%. For example, total CPU usage in 2PC is 10.5%. It can be

[1] https://docs.spring.io/spring-boot/docs/current/reference/html/actuator.html

[2] https://prometheus.io/

[3] https://grafana.com/

seen in Figure 26 that Saga Orchestration uses the CPU most. Saga Choreography comes second and 2PC uses the CPU least. 2PC is favorable in terms of CPU usage out of the three distributed transaction patterns. Saga Choreography is preferable compared to Saga Orchestration in terms of CPU usage.

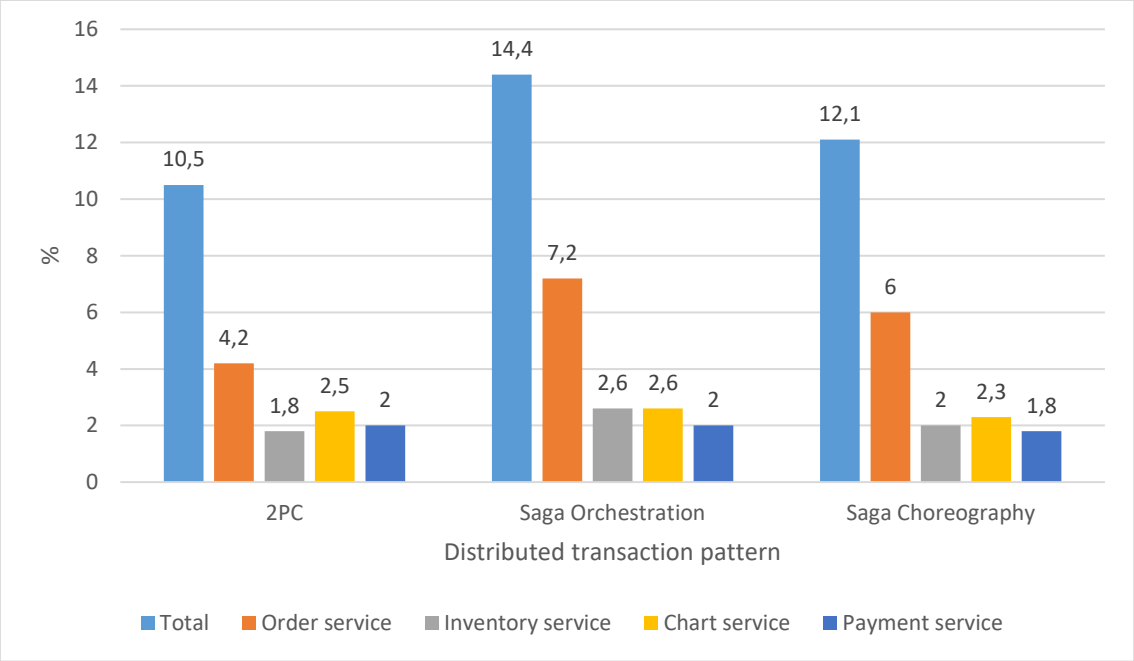

Figure 26. Comparison of average CPU usage for the MSA implementations.

Figure 27 shows the last performance metric which is memory usage. For the memory usage, Garbage-First (G1) Eden Space is considered. G1 is a garbage collector algorithm in Java. There are three main types of memory regions in G1 which are Eden, Survivor, and Old Generation. Eden is where newly created objects are allocated, Survivor is where objects that survived from a couple of garbage collections are allocated, and Old Generation is where long-lived objects are allocated. For memory usage in this thesis work, Eden Space is considered because it is the memory region where most of the objects' memory allocation and release happens. Figure 27 shows that Saga Orchestration consumes the most memory, the second is Saga Choreography and 2PC is the least memory-consuming pattern. The result of memory usage is similar to CPU usage and 2PC is the favorable pattern in terms of memory usage.

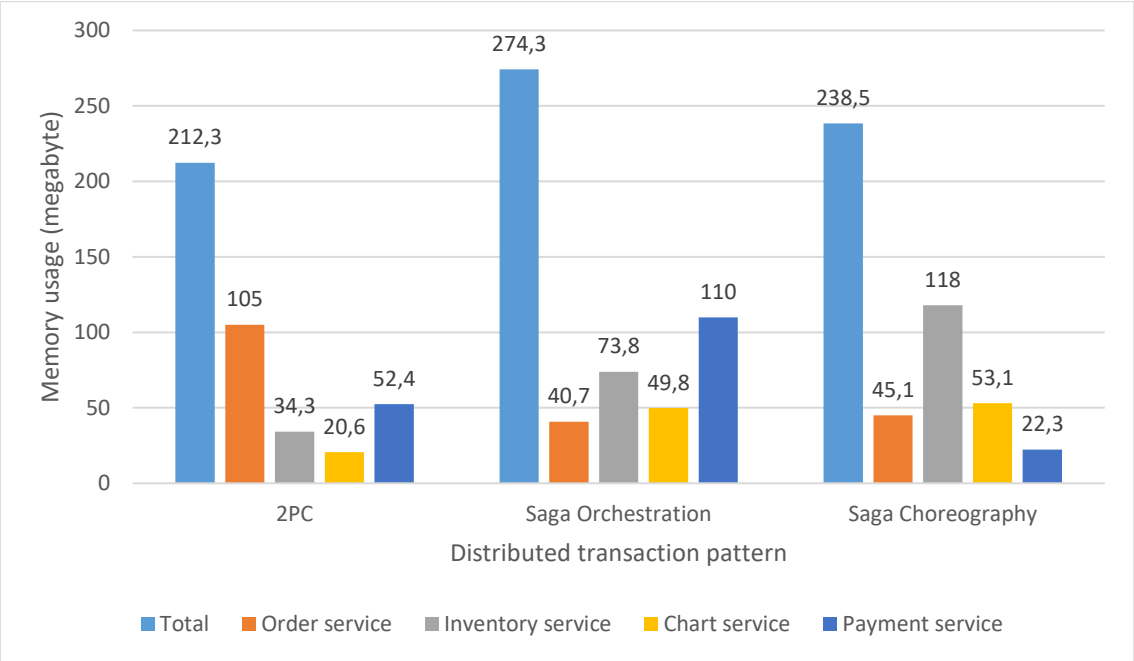

Figure 27. Comparison of memory usage for the three MSA implementations.

Experiments related to RQ2 show that the chosen distributed transaction pattern affects system performance. None of the patterns perform best in all cases, they have trade-offs. In terms of average response time under load and without load, Saga Choreography performs the best in both scenarios, and 2PC performs the worst, however, in the payment service fail case the Saga Orchestration performs noticeably badly. Therefore, for those systems in which failure is expected in the latest steps of execution, Saga Orchestration is not the best choice. The second performance metric investigated was transaction throughput and Saga Choreography has the best transaction throughput. However, because the Saga patterns are eventually consistent, not strongly consistent, for those systems that require strong consistency such as banking, 2PC is preferable. The third and fourth performance experiments were related to CPU usage and memory usage. In both of the experiments, 2PC performs the best with having less CPU and memory usage, and, Saga Choreography comes the second. Therefore, in terms of response times and transaction throughput Saga Choreography is the best option, whereas in terms of CPU and memory usage, 2PC is the best option. Thus, in terms of performance, 2PC or Saga Choreography may be chosen as a distributed transaction pattern of a system depending on system requirements and constraints. Considering the main performance criteria for developers, architects, or system designers is based on speed or efficiency, if their concern is speed, Saga Choreography is a better option, however, if the concern is resource utilization 2PC consumes fewer resources.

5.3 Systems Update Scenarios

RQ3, what are the failure scenarios and methods to mitigate failures when updating microservices using different transaction consistency patterns, was analyzed with the three scenarios: changing microservices Data Transfer Objects (DTO), using a new version of an endpoint or a new message queue, and adding a new microservice to the systems. For the experiments to answer RQ3, these changes were introduced to each MSA implementation, to observe potential problems these modifications may bring and investigate mitigation plans for the problems detected.

Critical systems are expected to be available all the time without any downtime. Downtimes for systems may cause financial and repudiation loss for companies and bad user experience (UX). One of the reasons for downtimes for systems is system updates. During the update, all the systems may be required to be down, to avoid data corruption or catastrophic failures. RQ3 aims to highlight possible failure scenarios while updating the services and methods to mitigate the failures.

Microservices are independent applications and each microservice has its own release cycle. One of the major benefits of MSA is having faster life cycle times per application. This brings faster new features to market time too. However, when a new feature is required to be delivered, such cases may occur in which one microservice is ready to deliver the feature but the other is not ready yet. Also, even though microservices are deployed at the same time because there are generally multiple instances of microservices running, data inconsistencies may occur during the deployment time.

For experimenting with the DTO updates, mock changes were made in the system. For example, a new requirement that is discount feature needs to be introduced to the e-commerce system. For the discount feature, a field named “discountId” needs to be added to the *OrderDto* class in the order microservice and the *PaymentDto* class in the payment microservice (Figure 4). Two possible update scenarios below are tested. No downtime happens in either of the scenarios. 2PC communicates via Spring WebFLux[1] and it uses

[1] https://docs.spring.io/spring-framework/reference/web/webflux.html

Jackson[1] serializers, Saga frameworks use Apache Kafka[2] for communication, and it uses JsonSerializers[3]. Both serializers are capable of ignoring the missing fields without any error. However, in case the *OrderDto* class has the discountId but the *PaymentDto* class does not have it. It may cause data inconsistency in the system. To avoid inconsistency, deployment with downtime would be preferable. Adding new fields to the DTO classes tested with the two cases below:

- The new field is only added to the *OrderDto* class,
- The new field is only added to the *PaymentDto* class.

The other possible change on DTO is when a field gets updated. The quantity field's type in the *OrderDto* class and *InventoryDto* class is an integer. An integer is a whole number and does not contain a fraction. For example, a new requirement for the e-commerce system is that the quantities should be able to contain half and quarter amounts. Because of the requirement, the type of quantity field is required to be changed from *Integer* to *BigDecimal*. Two scenarios below are tested. No downtime happens because Jackson and JsonSerializers are capable of casting *Integer* to *BigDecimal* and vice versa. However, when casting from *BigDecimal* to *Integer*, fraction values get lost, and this leads to data inconsistencies. Therefore, deployment with downtime would be preferable. Updating new fields in the DTO classes tested with the two cases below:

- The field is updated only in the *OrderDto* class,
- The field is updated only in the *InventoryDto* class.

The last change on DTO experimented on in this thesis work is when a field gets deleted from DTO. For example, due to a new requirement total price should not be stored in the order database, the total price should be deleted from *OrderDto* and *PaymentDto* classes, and the total price should only be calculated and stored in the payment service. Two scenarios below are tested and again serializers are smart enough to ignore missing fields without failure. Since the fields are going to be deleted from the order database as well,

[1] https://docs.spring.io/spring-framework/reference/web/webflux/controller/ann-methods/jackson.html

[2] https://kafka.apache.org/

[3] https://docs.spring.io/spring-kafka/reference/kafka/serdes.html

missing data does not cause any inconsistency in the systems. Deleting fields from the DTO classes tested with the two cases below:

- The field is deleted only from the *OrderDto* class,
- The field is deleted only from the *PaymentDto* class.

Another problematic update possibility in the microservices environment is major changes such as data structure changes in multiple DTO fields in endpoints or queues. In such cases introducing a new endpoint in synchronous systems or a new message queue in asynchronous systems is a better approach than changing the existing endpoint or queue dramatically. When a new endpoint or queue is introduced, existing endpoint/queue clients/consumers of the endpoints/queues need to be migrated to the new one. For example, in the 2PC pattern, the inventory service has a REST endpoint like "api/v1/inventory/commit" which the order service requests. The inventory service introduces a new REST endpoint with the endpoint like "api/v2/inventory/commit". In Saga patterns, the inventory service publishes events to a message queue named "inventoryUpdatedTopicV1" from which the order service listens. The inventory service introduces a new message queue named "inventoryUpdatedTopicV2". In the migration of order service to a new endpoint or message queue errors may happen, or messages may get lost. Also, the order service developers need to wait for the new endpoint or message queue to be ready so that they can start implementing changes, this brings tightly coupled project management flows. One of the best practices to solve this issue is the concept called feature flags. A feature flag is a variable in source code that enables or disables a specific feature of the application [42]. The order service developer may choose to use the new feature with a feature flag and can implement the new feature without waiting inventory service team to deliver their feature. Also, when all the services are deployed with the new feature, the feature flag corresponding to the new feature can be enabled after all the checks in the system. Therefore, migrating to the new endpoint/message queue can be done by mitigating errors. Togglz[1] is an open-source library that provides feature flag implementation for Java applications. Togglz provides storing the feature

[1] https://www.togglz.org/

toggles in database tables, file system files, or configuration files. Togglz console provides a UI library to update feature flags on run-time.

The last update scenario experimented on in this thesis is adding a new microservice to the placing an order flow. For example, a discount microservice is required to be added to the flow before the payment service so that users can purchase the product if they have a discount coupon. So in the flow, discount service is placed in between chart service and payment service as can be seen in Figure 28, Figure 29, and Figure 30 per distributed transaction pattern. Like a new endpoint or a message queue in an existing service, integrating a new microservice into the system brings problem and failure scenarios. Introducing a new microservice in 2PC brings the service discovery problem. When a new service is introduced in the MSA, the service discovery mechanism needs to be detected and it takes around 30 seconds, depending on the configuration. So, for establishing secure synchronous communication between services the new service needs to be discovered by the service discovery. Once again, the feature flag concept is useful to use in this context to mitigate service discovery failures. Also, using the feature flags helps development teams of microservices to work simultaneously therefore it results in faster project delivery. Also while experimenting with the new microservice addition to the flow, it was noticed that in 2PC (Figure 28) and Saga Orchestration (Figure 29), only the order service needs to be changed because the flow is controlled by the order service. Meaning that discount service is only coupled with order service. However, in the Saga Choreography (Figure 30), the chart and payment services need to be changed as well. Because Saga Choreography requires more changes to adapt to new requirements, it is the least preferable pattern. Therefore, it can be deduced that for systems that expect frequent microservice addition/removal, Saga Choreography is not a good candidate compared to 2PC and Saga Orchestration.

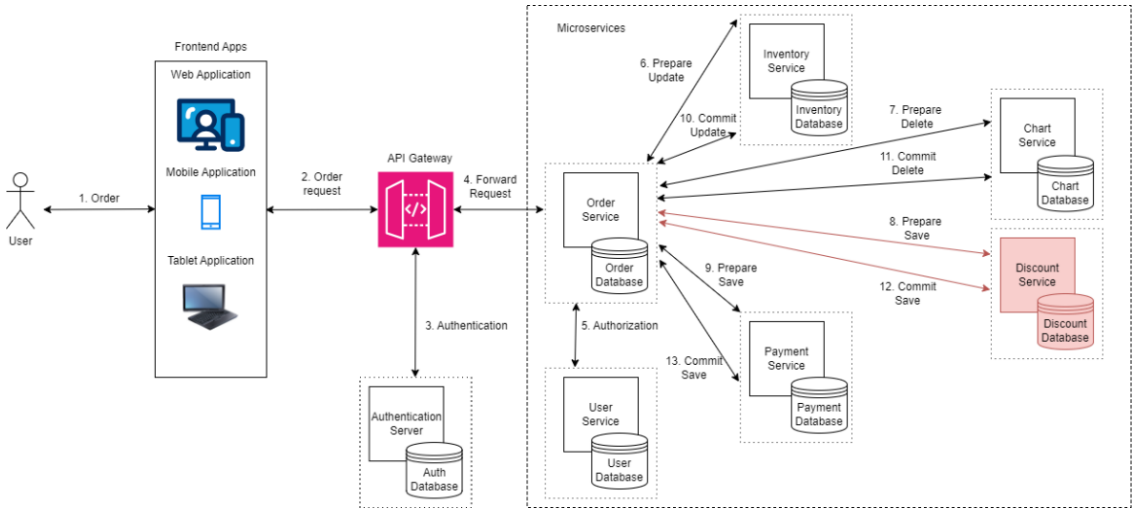

Figure 28. 2PC Discount Service.

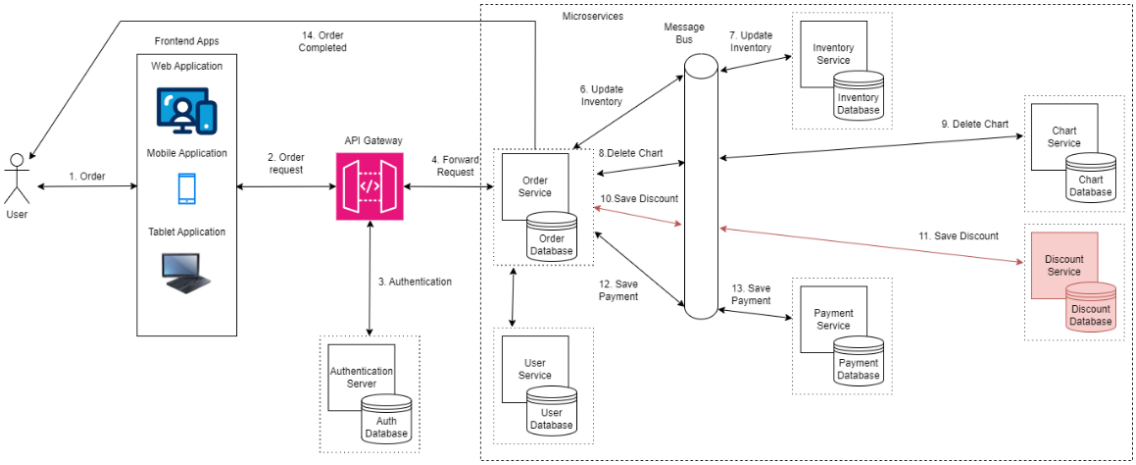

Figure 29. Saga Orchestration Discount Service.

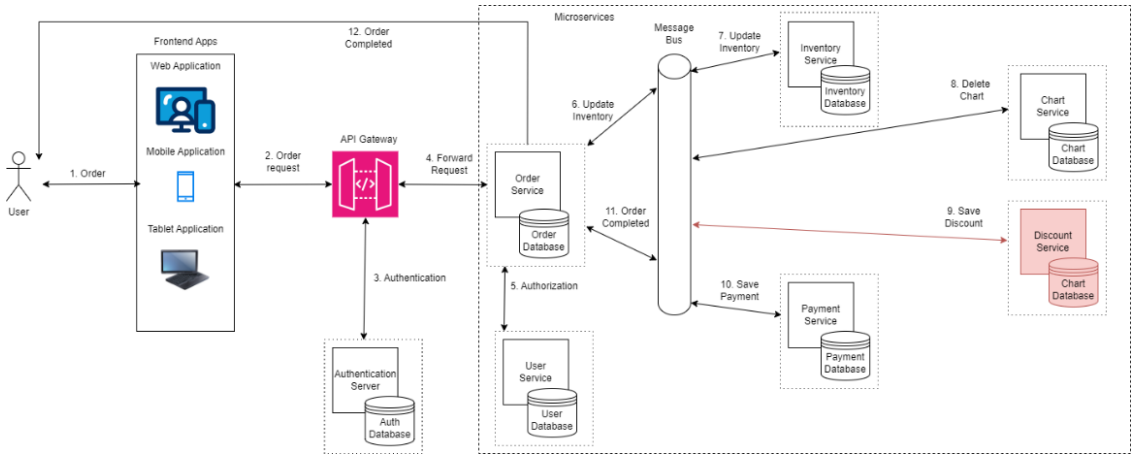

Figure 30. Saga Choreography Discount Service.

Experiments related to RQ3 show that there are risks of failures on updates, such as DTO changes, new endpoints/queues, and new microservices, however, there are methods like versioning, and feature flags to mitigate them. The first update experiment was to update DTO fields, a class field in DTO was added/updated/deleted. Experiments showed that none of the changes caused failure during downtime, however, adding and updating fields could cause data inconsistencies, and deployments with downtime would be preferable in such cases. The second update experiment was to add new a endpoint/queue to the system. When the client/consumer of the new endpoint/queue gets deployed before the new endpoint, it causes failures. To avoid such a change, the versioning of the endpoint/queue is a useful concept. Also, the feature flag concept is useful to mitigate risks and helps for better project management, because development teams can work simultaneously. The last update experiment that may cause failures was introducing new a microservice to systems. In 2PC, when a new instance of microservice is introduced in MSA, the service discovery needs to discover the new service. However, it takes around 30 seconds, so clients/consumers of the new microservices need the new microservices to be discovered by the service discovery. The feature flag is also useful in this case for loosely coupling the services. Also, the amount of microservices needed to be changed in Saga Choreography is more than the other two patterns making Saga Choreography a weak candidate in systems that expect frequent addition/removal of microservices.

6 Conclusion

In this thesis, distributed transaction patterns, 2PC, Saga Orchestration, and Saga Choreography were analyzed in terms of design, performance, and update aspects. Which pattern performs better in what conditions is explained for researchers, application developers, system architects, and system analysts.

An e-commerce backend system was implemented with three distributed transaction consistency patterns. The tech stack of the systems is Spring Boot, Java, Spring Data, Spring Cloud, Spring WebFlux, Spring Kafka, and Docker. 2PC is implemented using only synchronous communication type with REST protocol whereas Saga patterns use asynchronous communication style with Apache Kafka message bus.

One of the main flows of the e-commerce systems which is ordering a product was implemented in the three systems. Ordering a product is a suitable candidate because when ordering a product, multiple microservices and operations are involved like updating inventory via inventory service, deleting charts via chart service, and saving payment with payment service.

The design analysis shows that 2PC has the best scores in cyclomatic complexity and lines of code, whereas Saga Choreography has the best scores in cognitive complexity and communication overhead. Therefore, in terms of design, 2PC or Saga Choreography may be chosen as a distributed transaction pattern in MSA.

The performance analysis shows that in terms of response times and transaction throughput Saga Choreography is the best option, whereas in terms of CPU usage and memory usage, 2PC is the best option. Therefore, 2PC or Saga Choreography may be chosen as a distributed transaction pattern in MSA considering performance metrics. Saga Choreography provides a faster system, however, 2PC consumes fewer resources.

The update analysis shows that there are risks of failures on updates, such as DTO changes, new endpoints/queues, and new microservices, however, there are methods like versioning, and feature flags to mitigate them. Even though changing fields in DTO does

not cause failures, it may cause data inconsistency in deployment time, therefore downtime would be preferable. Versioning helps for a better transition to new endpoints/queues. Feature flag helps for a smoother transition to use new endpoints/queues and new microservices. Also, having a controller in the system provides fewer changes in the system when adding a new microservice to the system in the ordering of a product flow. Therefore, 2PC and Saga Orchestration are better options than Saga Choreography for systems that require frequent addition/removal of microservices.

Future works related to the implementation are front-end applications for the back-end system and in Saga patterns communication with the user about the order status can be implemented, communication can be any asynchronous communication channel like sending email. Future work related to experiments is running multiple instances of implemented systems simultaneously to test the scalability aspect. Future work related to continuing the research is to develop a distributed transaction decision framework based on the results of the current research to help systems designers to choose a suitable distributed transaction pattern for a specific problem or task based on key characteristics of their system.

References

[1] W. Hasselbring and G. Steinacker, “Microservice Architectures for Scalability, Agility and Reliability in E-Commerce,” in *IEEE International Conference on Software Architecture Workshops (ICSAW)*, Gothenburg, Sweden, 2017, doi: 10.1109/ICSAW.2017.11.

[2] ISO/IEC/IEEE, “Software, systems and enterprise -- Architecture evaluation framework,” *ISO/IEC/IEEE 42030:2019(E),* pp. 1-88, 2019, doi: 10.1109/IEEESTD.2019.8767001.

[3] A. Aleti, B. Buhnova, L. Grunske, A. Koziolek and I. Meedeniya, “Software Architecture Optimization Methods: A Systematic Literature Review,,” *IEEE Transactions on Software Engineering,* vol. 39, no. 5, pp. 658-683, 2013, doi: 10.1109/TSE.2012.64.

[4] P. C. Lockemann and J. Nimis, Dependable Multi-agent Systems: Layered Reference Architecture and Representative Mechanisms., Berlin: Springer, 2009, doi: 10.1007/978-3-642-04879-1_3.

[5] R. Schollmeier, “A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications,” in *Proceedings First International Conference on Peer-to-Peer Computing*, Linköping, Sweden, 2001, doi: 10.1109/P2P.2001.990434.

[6] F. Montesi, M. Peressotti and V. Picotti, “Sliceable Monolith: Monolith First, Microservices Later,” in *IEEE International Conference on Services Computing (SCC)*, Chicago, IL, USA, 2021, doi: 10.1109/SCC53864.2021.00050.

[7] Y. Shulin and H. Jieping, “Design and Implementation of Smart Teaching System Based on Microservice Architecture,” in *IEEE 2nd International Conference on Power, Electronics and Computer Applications (ICPECA)*, Shenyang, China, 2022, doi: 10.1109/ICPECA53709.2022.9718846.

[8] S. Salii, J. Ajdari and X. Zenuni, “Migrating to a microservice architecture: benefits,” in *46th MIPRO ICT and Electronics Convention (MIPRO)*, Opatija, Croatia, 2023, doi: 10.23919/MIPRO57284.2023.10159894.

[9] F. Ponce, G. Márquez and H. Astudillo, “Migrating from monolithic architecture to microservices: A Rapid Review,” 8th International Conference of the Chilean Computer Science Society (SCCC), Concepcion, Chile, 2019, doi: 10.1109/SCCC49216.2019.8966423.

[10] P. D. Francesco, P. Lago and I. Malavolta, “Migrating Towards Microservice Architectures: An Industrial Survey,” in *IEEE International Conference on Software Architecture (ICSA)*, Seattle, WA, USA, 2018, doi: 10.1109/ICSA.2018.00012.

[11] Y. Abgaz, A. McCarren, P. Elger, D. Solan, N. Lapuz, M. Bivol, G. Jackson, M. Yilmaz, J. Buckley and P. Clarke, “Decomposition of Monolith Applications Into Microservices Architectures: A Systematic Review,” *IEEE Transactions on*

Software Engineering, vol. 49, no. 8, pp. 4213-4242, 2023, doi: 10.1109/TSE.2023.3287297.

[12] G. Zhang, K. Ren, J.-S. Ahn and S. Ben-Romdhane, "GRIT: Consistent Distributed Transactions Across Polyglot Microservices with Multiple Databases," *IEEE 35th International Conference on Data Engineering (ICDE)*, pp. 2024-2027, 2019, doi: 10.1109/ICDE.2019.00230.

[13] M. Liu, D. Agrawal and A. E. Abbadi, "The performance of two-phase commit protocols in the presence of site failures," in *Proceedings of IEEE 24th International Symposium on Fault- Tolerant Computing*, TX, USA, 1994, doi: 10.1109/FTCS.1994.315637.

[14] K. M. Koyya and B. Muthukumar, "A Survey of Saga Frameworks for Distributed Transactions in Event-driven Microservices," *Third International Conference on Smart Technologies in Computing, Electrical and Electronics (ICSTCEE)*, pp. 1-6, doi: 10.1109/ICSTCEE56972.2022.10099533, 2022.

[15] S. Aydin and C. B. Çebi, "Comparison of Choreography vs Orchestration Based Saga Patterns in Microservices," in *International Conference on Electrical, Computer and Energy Technologies (ICECET)*, Prague, Czech Republic, 2022, doi: 10.1109/ICECET55527.2022.9872665.

[16] G. Blinowski, A. Ojdowska and A. Przybyłek, "Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation," *IEEE Access*, vol. 10, pp. 20357-20374, 2022, doi: 10.1109/ACCESS.2022.3152803.

[17] A. Carrasco, B. v. Bladel and S. Demeyer, "Migrating towards microservices: migration and architecture smells," in *IWoR 2018: Proceedings of the 2nd International Workshop on Refactoring*, 2018, doi: 10.1145/3242163.3242164.

[18] D. Taibi, V. Lenarduzzi and C. Pahl, "Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation," *IEEE Cloud Computing*, vol. 4, no. 5, pp. 22-32, 2017, doi: 10.1109/MCC.2017.4250931.

[19] M. Kalske, N. Mäkitalo and T. Mikkonen, "Challenges When Moving from Monolith to Microservice Architecture," in *Current Trends in Web Engineering*, 2018, doi: 10.1007/978-3-319-74433-9_3.

[20] M. Müller, "Consistency and Autonomy in the Microservice Architecture," *(IJACSA) International Journal of Advanced Computer Science*, vol. 9, no. 8, 2020.

[21] K. Malyuga, O. Perl, A. Slapoguzov and I. Perl, "Fault Tolerant Central Saga Orchestrator in RESTful Architecture," in *26th Conference of Open Innovations Association (FRUCT)*, Yaroslavl, Russia, 2020, doi: 10.23919/FRUCT48808.2020.9087389.

[22] C. K. Rudrabhatla, "Comparison of Event Choreography and Orchestration Techniques in Microservice Architecture," *International Journal of Advanced Computer Science and Applications(IJACSA)*, vol. 9, no. 8, 2018, doi: 10.14569/IJACSA.2018.090804.

[23] A. Megargel, C. M. Poskitt and V. Shankararaman, "Microservices Orchestration vs. Choreography: A Decision Framework," in *IEEE 25th International Enterprise Distributed Object Computing Conference (EDOC)*, Gold Coast, Australia, 2021, doi: 10.1109/EDOC52215.2021.00024.

[24] W. Nylund, “Comparing Transaction Management Methods in Microservice Architecture,” 2023.

[25] M. H. Halstead, Elements of Software Science, New York, United States: Elsevier Science Inc., 1977.

[26] R. Laigner, Y. Zhou, M. A. V. Salles, Y. Liu and M. Kalinowski, “Data management in microservices: state of the practice, challenges, and research directions,” *Proceedings of the VLDB Endowment,* vol. 14, no. 13, p. 3348–3361, 2021, doi: 10.14778/3484224.3484232.

[27] H. Vural, M. Koyuncu and S. Guney, “A Systematic Literature Review on Microservices,” in *Computational Science and Its Applications – ICCSA 2017*, 2017, doi: 10.1007/978-3-319-62407-5_14.

[28] E. Ntentos, U. Zdun, K. Plakidas, D. Schall, F. Li and S. Meixner, “Supporting Architectural Decision Making on Data Management in Microservice Architectures,” in *Software Architecture*, 2019, doi: 10.1007/978-3-030-29983-5_2.

[29] P. Fan, J. Liu, W. Yin, H. Wang, X. Chen and H. Sun, “2PC*: a distributed transaction concurrency control protocol of multi-microservice based on cloud computing platform,” *Journal of Cloud Computing: Advances, Systems and Applications,* vol. 9, no. 1, 2020, doi: 10.1186/s13677-020-00183-w.

[30] P. Fan, J. Liu, W. Yin, H. Wang, X. Chen and H. Sun, “2PC+: A High Performance Protocol for Distributed Transactions of Micro-service Architecture,” in *Intelligent Mobile Service Computing. EAI/Springer Innovations in Communication and Computing*, Springer, 2021, doi: 10.1007/978-3-030-50184-6_6.

[31] X. Limón, A. Guerra-Hernández, A. J. Sánchez-García and J. C. P. Arriaga, “SagaMAS: A Software Framework for Distributed Transactions in the Microservice Architecture,” in *6th International Conference in Software Engineering Research and Innovation (CONISOFT)*, San Luis Potosi, Mexico, 2018, doi: 10.1109/CONISOFT.2018.8645853.

[32] K. Munonye and P. Martinek, “Enhancing Performance of Distributed Transactions in Microservices via Buffered Serialization,” *Journal of Web Engineering,* vol. 19, no. 5-6, pp. 647-684, 2020, doi: 10.13052/jwe1540-9589.19564.

[33] M. Gördesli, A. Nasab and A. Varol, “Handling Rollbacks with Separated Response Control Service for Microservice Architecture,” in *3rd International Informatics and Software Engineering Conference (IISEC)*, Ankara, Turkey, 2022, doi: 10.1109/IISEC56263.2022.9998226.

[34] H. Garcia-Molina and K. Salem, “Sagas,” *ACM SIGMOD Record,* vol. 16, no. 3, p. 249–259, 1987, doi: 10.1145/38714.38742.

[35] M. Mythily, A. S. A. Raj and I. T. Joseph, “An Analysis of the Significance of Spring Boot in The Market,” in *International Conference on Inventive Computation Technologies (ICICT)*, Nepal, 2022, doi: 10.1109/ICICT54344.2022.9850910.

[36] Z. Triartono, R. M. Negara and Sussi, “Implementation of Role-Based Access Control on OAuth 2.0 as Authentication and Authorization System,” in *International Conference on Electrical Engineering, Computer Science and*

Informatics (EECSI), Bandung, Indonesia, 2019, doi: 10.23919/EECSI48112.2019.8977061.

[37] D. Bansal and R. Bathla, "Service Discovery Mechanism for Micro Services in Cloud Computing: Comparative Study," in *Second International Conference On Smart Technologies For Smart Nation (SmartTechCon)*, Singapore, Singapore, 2023, doi: 10.1109/SmartTechCon57526.2023.10391383.

[38] M. Gördesli and A. Varol, "Comparing Interservice Communications of Microservices for E-Commerce Industry," in *10th International Symposium on Digital Forensics and Security*, Istanbul, Turkey, 2022, doi: 10.1109/ISDFS55398.2022.9800784.

[39] P. G. T. H. Kashmira and S. Sumathipala, "Generating Entity Relationship Diagram from Requirement Specification based on NLP," in *3rd International Conference on Information Technology Research (ICITR)*, Moratuwa, Sri Lanka, 2018, doi: 10.1109/ICITR.2018.8736146.

[40] N. Nguyen and S. Nadi, "An Empirical Evaluation of GitHub Copilot's Code Suggestions," in *IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*, Pittsburgh, PA, USA, 2022, doi: 10.1145/3524842.3528470.

[41] R. Saborido, J. Ferrer, F. Chicano and E. Alba, "Automatizing Software Cognitive Complexity Reduction," *IEEE Access*, vol. 10, pp. 11642-11656, 2022, doi: 10.1109/ACCESS.2022.3144743.

[42] M. K. Ramanathan, L. Clapp, R. Barik and M. Sridharan, "Piranha: Reducing Feature Flag Debt at Uber," in *IEEE/ACM 42nd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, Seoul, Korea (South), 2020.

Appendix 1 – Non-exclusive licence for reproduction and publication of a graduation thesis[1]

I Mustafa Tikir

1. Grant Tallinn University of Technology free licence (non-exclusive licence) for my thesis “Analysis on Distributed Transaction Patterns within Microservice Architecture” supervised by Tarmo Robal.
   1.1. to be reproduced for the purposes of preservation and electronic publication of the graduation thesis, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright.
   1.2. to be published via the web of Tallinn University of Technology, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright.
2. I am aware that the author also retains the rights specified in clause one of the non-exclusive licences.
3. I confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights, the rights arising from the Personal Data Protection Act or rights arising from other legislation.

06.05.2024

1 The non-exclusive licence is not valid during the validity of access restriction indicated in the student's application for restriction on access to the graduation thesis that has been signed by the school's dean, except in case of the university's right to reproduce the thesis for preservation purposes only. If a graduation thesis is based on the joint creative activity of two or more persons and the co-author(s) has/have not granted, by the set deadline, the student defending his/her graduation thesis consent to reproduce and publish the graduation thesis in compliance with clauses 1.1 and 1.2 of the non-exclusive licence, the non-exclusive license shall not be valid for the period.