National College of Ireland

A lightweight failure management pattern for microservices

MSc Research Project
Cloud Computing

Sai Dhawanjewar
Student ID: x22130063

School of Computing
National College of Ireland

Supervisor: Dr Giovani Estrada

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Sai Dhawanjewar
Student ID:	x22130063
Programme:	Cloud Computing
Year:	2023
Module:	MSc Research Project
Supervisor:	Dr Giovani Estrada
Submission Due Date:	05/04/2024
Project Title:	A lightweight failure management pattern for microservices
Word Count:	7570
Page Count:	27

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	
Date:	5th April 2024

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	☐
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☐
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	☐

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

A lightweight failure management pattern for microservices

Sai Dhawanjewar
x22130063

Abstract

The project develops a novel lightweight failure management pattern for distributed transactions in microservices. We focus on issues related to data consistency, fault tolerance, and atomicity. As the adoption of microservices architecture becomes prevalent due to its advantages over monolithic architecture, concerns arise regarding the coordination of distributed transactions and the preservation of data consistency across microservices. The specific scenario explored involves a web application with interconnected microservices, highlighting the potential for inconsistent data and performance issues in the face of microservice failures.

To address these challenges, the research proposes a novel lightweight failure management pattern for microservices. It takes the bare-minimal fault-tolerant features of Two-Phase Commit and Saga Cloud Pattern, which are the retry mechanism and temporary databases. The 2PC protocol ensures atomicity, reliability, and data consistency in distributed transactions and Saga patterns automate the failure recovery functionality. The unique aspect of this solution lies in the incorporation of a temporary database, serving as a safeguard against failures. In the event of a microservice failure, transactions are logged in the temporary database. Upon the restoration of the failed service, it retrieves the data from the temporary database, and the 2PC protocol verifies the availability of all participating services before finalizing the transaction and Saga pattern automates the regaining original state of the microservice.

To demonstrate the expected results, a web application (called *My ArtGallery*) consisting of two microservices (Admin and Main) was developed. The implementation involves React JS for webpages, Python and Django for backend processing, MySQL as the database, RabbitMQ for message publication and subscription, and Docker for deployment. Two use cases demonstrate the main features of the proposed algorithm with encouraging results. We therefore think the novel lightweight failure management pattern can be of great academic and practical interest.

1 Introduction

1.1 Software development

The traditional monolithic architecture in software development refers to building an application as an integrated whole. Despite being straightforward for small applications, this method becomes difficult to scale and maintain as applications get larger. The microservice architecture is an architectural pattern designed to overcome these development difficulties as it separates services with their own codebase, database, and

functionality. Large and complex systems can benefit from the flexibility and scalability that microservices offer by facilitating independent development, deployment, and scaling. Christian et al. (2023),Al-Debagy and Martinek (2018)

A research gap is identified in Microservice architecture, particularly regarding the possibility of inconsistent data and performance issues in the event of microservice failure. An example is presented where a web application with connected web pages uses separate microservices with their databases. The communication between microservices is highlighted as a potential source of inconsistency, especially when a microservice goes down during data processing.

Fault flow is discussed as a solution to these problems, which include retrying messages in a queue and storing raw data in the database in case of failure. Both of these approaches have benefits and cons, so a cost-effective and reliable solution is required.

We will later review methodologies like Paxos, Saga, and the Two-Phase Commit (2PC) as well as distributed system concepts, such as transactions, data consistency, and fault tolerance. After analysing these three approaches (2PC, Saga, and Paxos), we will show how a lightweight design pattern based on 2PC and Saga pattern is more effective in situations where applications require atomicity, consistency in distributed databases, and cost-effectiveness, especially for smaller applications.

It is acknowledged that the conventional Two-Phase Commit (2PC) method has certain drawbacks when adopting these approaches for microservices. To address the drawbacks of the conventional 2PC approach, suggested solution deliberately integrates aspects influenced by the 2PC methodology and Saga. This adaptation uses temporary databases to address particular issues with distributed transaction management in microservices, offering a more effective and customised solution for instances in which automaticity, consistency, and economy of scale are critical considerations.

1.2 Microservices

In the traditional, monolithic software development approach, the entire application is constructed as a single, integrated unit. All components, including the user interface, business logic, and database interfaces, are closely integrated into a single codebase. The entire application is delivered and scaled as a single unit. While monolithic designs are simple and straightforward to create, they can be challenging to maintain and expand as the application grows in complexity. Any changes or updates to the application require the entire monolith to be re-deployed.

To address the challenges posed by the monolithic approach, the Microservices architecture has been introduced. Microservices architecture is a modern development approach that structures an application as a collection of small, independent services. Each service focuses on a specific business capability and operates autonomously with its own codebase, database, and functionality. These services communicate with each other through well-defined APIs, allowing for loose coupling and modularity. Microservices facilitate independent development, deployment, and scaling of individual services Raharjo et al. (2022), Hasselbring and Steinacker (2017).

This modular approach enhances flexibility, scalability, and ease of maintenance, making it particularly suitable for large and complex systems. The microservices design allows for greater flexibility in selecting a distinct technology stack for each service. For example,one service may use a relational database, while another may use a NoSQL database, allowing individual services to manage domain data independently. Further-

more, the microservices architecture allows for on-demand dynamic scaling of data stores. Each microservice keeps its own database of important business transactions. As a result, dealing with distributed transactions and providing data consistency across many services offer significant challenges. Figure 1 Illustrates the monolithic and microservice architecture.

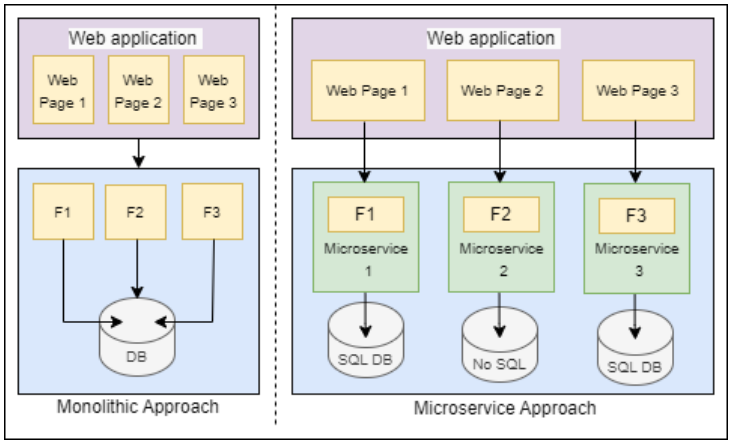

Figure 1: Illustration of Monolithic and Microservice architectures

1.3 Research Gap

When dealing with microservice architecture and distributed databases, the possibility of inconsistent data and performance issues arises in the event of a failure in any microservice. This example will clarify the concept. Let us consider a web application with two connected web pages, each linked to a separate microservice for processing, to better understand the scenario, please refer to Figure 1. Each microservice has its own database for storing and processing data, with some common fields shared between both databases. Microservices communicate through message passing via queues to maintain data consistency.

The 2^{nd} microservice goes down while the 1^{st} microservice receives data for processing that is common between both databases. In this case, the 1^{st} microservice processes and stores the data into its database, then publishes a message to the 2^{nd} microservice. However, since the 2^{nd} microservice is down, it cannot process the data, resulting in a failure to update its database, leading to inconsistency. This inconsistency is particularly critical for applications dealing with e-commerce, retail, etc.

To address such scenarios, fault tolerance becomes crucial. Microservice fault management is essential for maintaining the stability and performance of the application. In fault tolerance, we attempt to handle these conditions in different ways, some of which are effective, such as retrying fault messages in a queue (Country (2021),Perikov (2020)) and storing raw data in the database in case of failure (Struk (2021),Laigner et al. (2021)). However, these techniques also have drawbacks, which may lead to inconsistency in the database, as discussed in detail in Section 2.7. To address these issues, we need an effective solution that helps maintain data consistency, atomicity, and cost-effectiveness, especially for small-scale businesses.

1.4 Research question

The research questions aim to explore how proposed flow can effectively address challenges in managing distributed transactions, ensuring data consistency and improving fault tolerance in microservices. Research objectives include setting up a development environment, implementing a sample microservice application, developing a novel flow with a temporary database, and testing the proposed technique against existing literature.

1. What are the challenges and limitations of 2PC and Saga pattern in the handling of data integrity for microservices?

2. To what extent and how can 2PC and Saga pattern be combined to address transactional integrity in a minimalist way.

1.5 Research objectives

In order to address the research questions shown in Section 1.4, we are going to set up a microservice environment to focus on data consistency:

1. Set up a development environment.

2. Implement a sample microservice application with dependencies between components, specifically tailored for fault-tolerance research.

3. Develop and implement a novel fault-tolerant flow, inspired by the principles of 2PC and Saga pattern, incorporating a temporary database.

4. Test and evaluate the proposed technique against existing literature.

1.6 Outline

The structure of the report includes sections on relevant theory, details of the proposed approach, techniques used to address the problem, proposed test cases, presentation of novel methods, evaluation results, and conclusions with discussions of future research directions.

The remainder of the report is organised as follows. Section 2 presents the relevant theory and works closely related to the proposed one. Section 3 describes details of the proposed approach. In Section 4, we describe the techniques that are used to address the problem, as well as all proposed test cases. The two novel methods are presented in Section 5, while evaluation results appear in Section 6. Finally, we provide conclusions and discussions of future research directions in Section 7.

2 Related Work

2.1 Distributed Microservice Architecture

Microservices are an architectural style that organizes applications as a collection of services, delivering large, complex applications regularly and reliably. Key properties include high availability, scalability, loose coupling, agility, and reliability. Microservices address

issues in traditional monolithic applications, such as single shared databases causing scalability and failure concerns. They function as a distributed system, with transactions distributed across multiple services, each with its own database. Implementing distributed transactions for data consistency and rollback operations is crucial in a microservices architecture.Messina et al. (2016),Fan et al. (2020).Figure 1 illustrates the distributed microservice architecture.

Three approaches were identified for the implementation of effective synchronization and fault tolerance in distributed microservices architectures: the Paxos consensus algorithm, the Saga pattern, and the Two-Phase Commit. Following a comprehensive assessment of the application's requirements, particularly emphasizing data consistency and atomicity in small-scale scenarios, a critical analysis of these approaches is outlined below.

2.2 Exploring Algorithmic Solutions for Microservice Architecture

To address the identified research gap, we analysed several algorithms, including Raft Consensus, Quorum-Based Techniques, and Chain Replication. Upon evaluation, we determined that the Saga pattern, 2PC, and Paxos offer promising solutions for the proposed implementation within a microservice architecture. This determination was made with careful consideration of key constraints such as atomicity, performance, and scalability.

2.2.1 Paxos consensus algorithm

A key protocol for attaining consensus and fault tolerance in distributed computing is the Paxos consensus algorithm, which Leslie Lamport created in 1989. It guarantees that, even in the event of failures and inconsistent communication channels, a group of nodes can agree on a single value or a series of values.

Operating in phases with nodes labeled as "acceptors" and a distinguished "proposer," Paxos follows rounds of Prepare, Promise, Accept, and Learn. In the Prepare phase, the proposer sends a proposal to the acceptors, who respond with a promise not to accept lower-sequence proposals. If a threshold of acceptors promises, the proposer proceeds to the Accept phase, achieving consensus once the value is accepted.

Despite its complexity, Paxos ensures fault tolerance by allowing a majority of nodes to agree, even in the face of failures. Variations like Multi-Paxos and Fast Paxos address practical challenges. Paxos remains a cornerstone in fault-tolerant distributed systems, influencing consensus algorithms in modern distributed databases and cloud infrastructure. (Kończak et al. (2021)).Figure 2 illustrates the Paxos consensus algorithm flow (image source[1]).

2.2.2 Saga Pattern

The Saga cloud design pattern is a failure management pattern long-lived transactions without a two-phase commit protocol, ensuring fault tolerance and data consistency across microservices[2]. In the Saga pattern, there is a sequence of transactions, each

[1] https://www.scylladb.com/glossary/paxos-consensus-algorithm/

[2] https://docs.aws.amazon.com/prescriptive-guidance/latest/modernization-data-persistence/saga-pattern.html

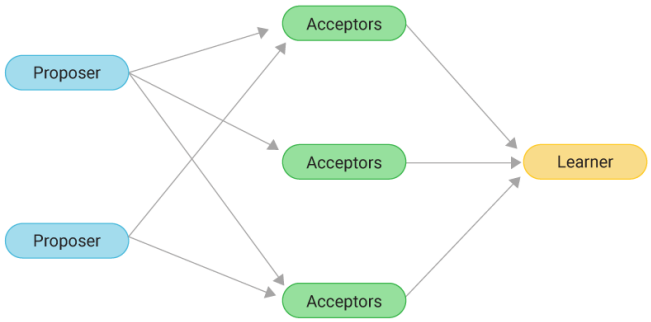

Figure 2: Illustration of paxos consensus algorithm.

one responsible for updating a microservice's state and triggering subsequent transactions. If a failure occurs, compensating transactions undo changes, restoring a consistent state without a global, blocking two-phase commit. This pattern is ideal for scenarios like e-commerce order processing, providing graceful recovery from failures. Illustrated in an online product purchase system, the workflow involves multiple microservices (Warehouse, Order, Billing, Shipping) with compensating transactions for error handling. Daraghmi et al. (2022).

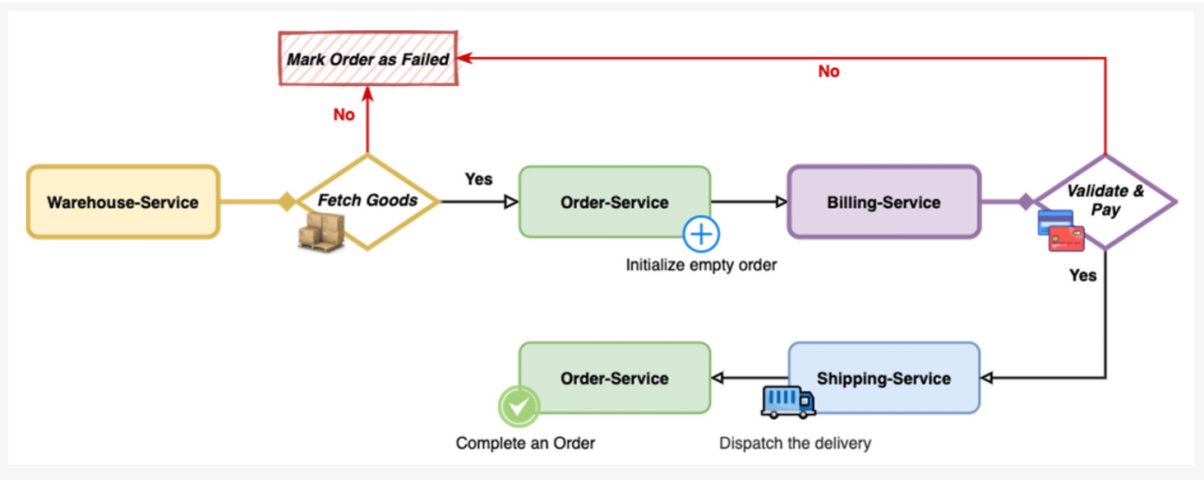

Figure 3: Saga workflow of a standard e-commerce microservices-based system.

Figure 3 illustrates the event workflow within a microservices-based e-commerce system. The system facilitates online product purchases, allowing users to choose products, payment methods, and shipping options. This transaction spans multiple microservices, namely the Warehouse-Service, Order-Service, Billing-Service, and Shipping-Service. The depicted workflow encompasses both the warehouse-before-billing and billing-before-warehouse scenarios to emulate real-world e-commerce processes Books.com.tw (2022). The sequence begins with the Warehouse-Service fetching goods, followed by the Order-Service initializing an "IN-PROGRESS" order. If goods retrieval fails, the order status is set to "FAILED". Subsequently, the Billing-Service validates the specified payment method, proceeding to collect payment upon successful validation. If validation fails, the flow terminates with the order marked as "FAILED". The Shipping-Service then dispatches the

delivery, and finally, the Order-Service completes the order, updating relevant information such as status, shipping ID, and amount.

2.2.3 2-Phase Commit (2PC) Protocol

While implementing distributed transactions in a microservice architecture, one of the most widely used patterns is the two-phase commit protocol (2PC) Uyanık and Ovatman (2020). As shown in Figure 5., this protocol uses a coordinator who is in charge of transaction control and management logic, while the microservices (participating nodes) execute their respective local transactions.

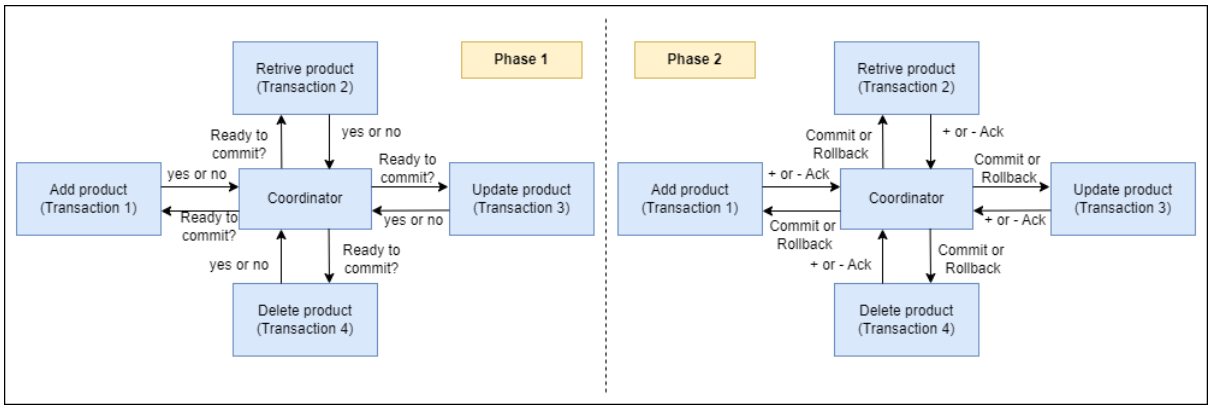

Figure 4: Illustration of the two-phases commit protocol.

A distributed transaction follows in two stages according to the 2PC protocol. The coordinator instructs participating nodes to commit the transaction during the initial phase, known as the prepare phase, eliciting a yes or no response. Following that, in the second stage, the coordinator instructs all nodes to proceed with the commit after receiving affirmative responses from all participating nodes.

The two-phase commit (2PC) protocol proves highly effective in scenarios requiring atomicity, where transaction consistency and reliability are paramount. Consider a scenario in a web application where a batch processing is initiated for data transfer. The application involves multiple microservices responsible for data processing and storing processed data in respective databases. Employing 2PC ensures atomicity by orchestrating the transaction in two phases. In the first phase, the coordinator requests participating nodes to prepare for the transaction, prompting a yes or no response. If all nodes confirm readiness, the second phase, the commit phase, ensues. Here, the coordinator instructs all nodes to execute the transaction. If any participating node encounters an issue or responds negatively at any point, the coordinator triggers a rollback, ensuring that the entire batch processing either succeeds uniformly or fails entirely. This preserves the atomic nature of the batch processing, preventing inconsistencies in data storage.

2.3 Comparison of approaches to handle distributed transactions

A number of algorithms have been proposed over the decades for the synchronization of distributed systems to address the challenges, such as distributed transactions, data consistency, and fault tolerance. Potential solutions include considering approaches such as Paxos, Saga pattern, and 2PC.

Two-Phase Commit (2PC)

The Two-Phase Commit Protocol guarantees atomicity and reliability while offering a structured method for distributed transactions. It is a classical reference algorithm for consistency and atomicity in distributed databases. Preparation and commit are two separate stages in which this protocol orchestrates transactions. Its blocking potential and increased latency, particularly in scenarios with frequent transactions, are significant drawbacks despite its effectiveness in maintaining transaction consistency. Furthermore, by coordinating individuals to agree on transaction outcomes, the Two-Phase Commit (2PC) protocol assures atomicity and consistency across several nodes in a transaction, enabling fault tolerance. The main drawback of 2PC and similar algorithms is the blocking mechanism, which can lead to deadlocks and uncommitted transactions trapped in the prepare state.

Paxos

The Paxos consensus algorithm enables fault tolerance by establishing agreement among distributed nodes, allowing the system to function in the face of failures and reach a judgement on crucial decisions. It has been recognised for its strong consensus and fault tolerance, even when communication channels are unpredictable and nodes fail. The way Paxos works is in phases. It involves nodes called "acceptors" and a node called "proposer". It accomplishes fault tolerance quite well, but in real-world applications, its complexity of implementation gives rise to variants like Multi-Paxos and Fast-Paxos presents even more complex design patterns for distributed applications. Kończak and Wojciechowski (2021)

Saga Cloud Design Pattern

The Saga Pattern is a design pattern to handle long-term transactions in distributed systems without requiring a two-phase commit protocol. It deals with distributed transactions by organising local transactions within each microservice and using compensating transactions to handle failures, allowing the system to recover and retain data consistency. Saga pattern performs good in demanding scenarios like e-commerce order processing, providing fault tolerance and consistency across microservices. Its application, however, might be restricted to particular use cases, and the complexities involved in handling compensating transactions add to the system's complexity. The key component of the Saga pattern is the temporary database to continue (retry) or compensate (rollback) a transaction[3]. Djerou and Tibermacine (2022)

2.4 Three-Phase commit (3PC) over Two-Phase Commit (2PC)

This paper Kumar et al. (2014) identifies the drawbacks of the Two-Phase Commit Protocol (2PC) and proposes an improved Three-Phase Commit Protocol (3PC) as a solution. The primary enhancement of 3PC is the addition of a second pre-commit decision phase to reduce blocking issues inherent in 2PC. The protocol enables the sender side to communicate pre-commit messages and obtain acknowledgments from the recipient side

[3] `https://docs.aws.amazon.com/prescriptive-guidance/latest/cloud-design-patterns/saga.html`

before committing or canceling the transaction, enhancing decision-making. Implementation analysis demonstrates that this approach significantly improves the performance and reliability of distributed database systems. However, concerns arise regarding the superiority of 3PC over 2PC tikv.org (2023), particularly regarding network partitions leading to blocking problems and impairing protocol execution. The longer latency and increased communication overhead introduced by the three phases of 3PC reduce efficiency, and its assumption of a fail-stop model limits compatibility with asynchronous communication and network partitions. Additionally, 3PC's higher complexity compared to 2PC restricts its adoption in real-world applications.

2.5 Saga pattern implementation with Choreography or Orchestration

Saga pattern implementation can be achieved through either Choreography or Orchestration. Orchestration, as a centralized coordination method, aligns well with the principles of the Saga pattern, facilitating coordinated execution of saga steps through a central coordinator. This centralized control simplifies managing complex sagas and ensures atomicity, fault tolerance, and performance. In contrast, Choreography lacks centralized control, making it challenging to coordinate saga steps and maintain a global view of the transaction process. Additionally, choreography may struggle to ensure atomicity, manage failures efficiently, and scale as the system expands, leading to performance overhead and reliability issues. Therefore, when coordinating complex distributed transactions, Orchestration is preferred over Choreography due to its ability to address these challenges effectively Richardson (2018).

2.6 Combining approaches Two-Phase Commit (2PC) and Saga pattern

In large-scale applications, managing and coordinating microservices poses challenges in ensuring their collaborative functioning. This paper proposes a solution by combining Two-Phase Commit (2PC) and Saga pattern techniques. While 2PC involves distinct prepare and commit phases, Saga pattern offers choreography and orchestration-based implementations. The paper underscores the complexity of controlling rollback scenarios in the event of transaction issues, particularly with each microservice typically possessing its own solutions in Saga pattern. The proposed solution suggests separating the rollback system from microservices for simplified and parallel execution to enhance system safety and speed. Emphasizing the response control service within the microservices architecture raises concerns about data inconsistency, mainly linked to unique identifiers in response management. The paper suggests managing and synchronizing responses using unique identifiers, but highlights potential gaps in matching responses if issues arise during identifier creation or transmission, compromising data consistency in partial or incorrect rollback scenarios. The robustness and dependability of the unique identifier mechanism are crucial for the approach's efficacy, with any weaknesses potentially leading to inconsistent system behavior.Gördesli et al. (2022).

2.7 A Comparative Analysis of Retrying Queue vs. Storing in Database

Fault tolerance is essential to maintaining the stability and performance of applications in the dynamic world of microservices. Two common methods of handling errors are analysed : storing raw data in the database in the case of a failure (Struk (2021); Laigner et al. (2021)) and retrying error messages in a queue Country (2021); Perikov (2020). Every strategy has advantages and drawbacks that affect the system's overall resilience and consistency. The merits and cons of these approaches are evaluated in this comparative analysis, which also clarifies the complexities of fault management in microservices architecture.

1. **Retrying in a Queue:**

   - Queue management: Retrying unsuccessfully executed messages stored in the message queue can lead to increased queue lengths and potential performance issues.
   - Timeouts and deadlines: Setting appropriate timeouts and deadlines is essential to prevent the application from getting stuck in continuous retries, which can lead to performance degradation.
   - Rate limiters: Rate limiters can control the rate at which requests are sent to a service, preventing excessive usage that could lead to service unavailability.
   - Circuit-breaking: Circuit-breaking is a pattern that helps reduce the number of unsuccessful requests by limiting the number of retries.
   - Retryable vs. non-retryable errors: It's essential to distinguish between retryable errors and non-retryable errors. Retryable errors can be retried, while non-retryable errors should not be retried, as they might lead to infinite retries and degrade performance
   - Stateful failure handling: Resolving failures might be stateful, and state handling is a key question for failure handling in microservices. This means that you need to consider the state of the service and its ability to recover from failures.

2. **Storing Raw Data in DB in Case of Failure:**

   - Data synchronization challenges: Synchronizing data between services using separate databases can become a significant challenge, requiring additional effort and resources.
   - Increased complexity: Managing multiple databases for each microservice increases the overall complexity of the system.
   - Limited code and resource reuse: Each microservice having its own data repository can lead to limited code and resource reuse, resulting in duplication and inefficiencies.
   - Dependency on database technology: The choice of database technology can impact the overall performance and scalability of the microservice architecture.

- Lack of centralized data management: In a microservice architecture, data is often stored in separate databases per service, which can lead to a lack of centralized data management and difficulty in maintaining a consistent view of the overall data landscape

3 Methodology

Detailed comparative analysis of the approaches—2-phase commit, Paxos, and Saga patterns to determine which approach is more efficient in scenarios where data consistency, atomicity, and budget are the primary considerations for a small-scale application

3.1 Critical analysis of Paxos vs. Saga vs. 2PC

1. **Purpose**
   - **2PC (Two-Phase Commit):** It is designed for coordinating transactions. In distributed systems, it ensures that all participating nodes either commit or roll back a transaction.
   - **Paxos:** This is a consensus algorithm primarily used for replication in stateful services. It facilitates agreement among distributed nodes on a specific value or state.
   - **Saga Pattern:** It is used for managing distributed transactions in a sequential manner. It employs a series of local transactions to maintain consistency across distributed systems.
2. **Resilience**
   - **Paxos:** Exhibits more resilience to manager failures and involves minimal blocking, allowing the system to continue functioning effectively.
   - **2PC (Two-Phase Commit):** Tends to block if the coordinator fails, requiring human intervention to restart and resume normal operations.
   - **Saga Pattern:** Requires developers to implement rollback logic themselves, potentially leading to less efficient fault recovery.
3. **Message Complexity**
   - **Paxos:** Requires more messages but is efficient in terms of message delay, ensuring effective communication within the distributed system.
   - **2PC (Two-Phase Commit):** Involves less message complexity, which can contribute to quicker transaction coordination.
   - **Saga Pattern:** Involves increased complexity, especially in managing compensating transactions, which are used to undo the effects of previously committed transactions.
4. **Fault Tolerance**
   - **Paxos:** Provides fault tolerance through replication, ensuring that the system remains operational even if some nodes fail.

- **2PC (Two-Phase Commit):** Assumes all nodes never fail, which can be a limitation in scenarios where failures are possible.
- **Saga Pattern:** Relying on compensating transactions, it might be potentially less reliable in handling failures compared to other approaches.

5. **Transaction Type**

   - **Paxos:** Oriented towards achieving consensus in replication scenarios.
   - **2PC (Two-Phase Commit):** Enforces single commit, adhering to ACID (Atomicity, Consistency, Isolation, Durability) transactions for strong consistency.
   - **Saga Pattern** Operates sequentially, not as a single commit, making data eventually consistent through a series of committed operations.

6. **Read Isolation**

   - **Paxos:** Not explicitly focused on read isolation, as its primary goal is achieving consensus.
   - **2PC (Two-Phase Commit):** Provides read isolation, ensuring that intermediate reads are allowed only if the transaction is successful.
   - **Saga Pattern:** Allows for intermediate reads, which may lead to potential issues like dirty reads, considered a disadvantage in many systems.

7. **Complexity**

   - **Paxos:** Involves complexity and may have a performance impact on the system.
   - **2PC (Two-Phase Commit):** Generally considered easier for application developers, reducing the complexity of troubleshooting.
   - **Saga Pattern:** Can be more complex to implement and maintain, especially when dealing with multiple microservices and event messages.

Table 1: Approaches and its drawbacks

Approch	**Drawbacks**	**Paper**
Paxos consensus algorithm	Complexity and performance impact	Ailijiang et al. (2016)
	Resource requirements	Roth and Haeberlen (2021)
	Message complexity	
Saga Pattern	Increased complexity	Documentation (2024e)
	Confusing workflow	
	Lack of isolation	microservices patterns (n.d.)
	Debugging challenges	
	Resource requirements	
2 Phase Commit Protocol	Synchronous nature	Xiang (2028)
	Resource requirements	Haroon (2016)
	Network overhead	
	Single point of failure	
	Lack of scalability	

3.2 Proposed idea inspired on 2PC and Saga Pattern: usage of temporary databases and retry mechanisms

As we previously described, the bottom line of existing fault-tolerant transactions is the usage of temporary databases and a retry mechanism. In other words, after thoroughly analyzing all three approaches Paxos, Two-Phase Commit Protocol, Saga Pattern, and techniques, such as retrying in a queue and the use of a database, as discussed in the Section 2. It is thus tempting to create a 2PC-Saga pattern inspired algorithm from their bare-minimal, fault-tolerant features.

The proposed lightweight failure management pattern was developed and tested using small scale microservice applications. Future work should however concentrate on the analysis of its behaviour on larger microservice systems. The selection of this innovative approach is guided by essential factors, including data consistency, atomicity, simplicity in design, and considerations related to the computing budget. These criteria play a critical role, directing the choice toward a solution that is not only robust but also adaptable, ensuring the seamless fulfillment of the application's specific requirements.

1. **Data Consistency and Atomicity**

   - **Two-Phase Commit (2PC)**
     - **Advantages**
       Provides a mechanism for ensuring atomicity and consistency of distributed transactions. Coordinates a commit or rollback decision among participating nodes, preventing inconsistencies.
     - **Limitations:**
       Can face challenges in scenarios where any participating microservice is temporarily unavailable, leading to blocking issues. Synchronous nature can impact system responsiveness.
   - **Saga Pattern:**
     - **Advantages:**
       Supports distributed transactions by employing a sequence of local transactions. Enables compensating transactions for handling failures, contributing to fault tolerance.
     - **Limitations:**
       Potential for increased complexity, especially in implementation and maintenance. Compensation logic must be carefully designed to ensure reliability.

2. **Justification for Combining 2PC, Saga Pattern, and Temporary Database:**

   - **Enhanced Fault Tolerance:**
     - **Two-Phase Commit (2PC):** Ensures transactional integrity and coordination among microservices.
     - **Saga Pattern:** Manages compensating transactions, handling failures gracefully.
     - **Temporary Database:** Acts as a buffer during failures, allowing the system to store data without impacting overall functionality.

- **Flexibility in Handling Failures:**
  - **Two-Phase Commit (2PC):** Addresses issues synchronously, ensuring atomicity when all services are available.
  - **Saga Pattern:** Handles failures by rolling back or compensating for transactions, providing flexibility.
  - **Temporary Database:** Safeguards against data loss during transient failures, contributing to a smoother recovery process.
- **Consistency Maintenance:**
  - **Two-Phase Commit (2PC):** Ensures consistent state across participating microservices during transactional operations.
  - **Saga Pattern:** Manages compensating transactions to maintain data consistency, even in the face of partial failures.
  - **Temporary Database:** Supports consistent data storage during temporary unavailability.
- **Efficiency in Recovery:**
  - **Two-Phase Commit (2PC):** Coordinates the recovery process when a faulty microservice returns to its original state.
  - **Saga Pattern:** Utilizes compensating transactions for efficient recovery.
  - **Temporary Database:** Facilitates seamless processing of stored data during recovery.

4 Design Specification

The integration of Two-Phase Commit (2PC), Saga Pattern, and a temporary database is used to enhance fault tolerance in the microservices architecture. Varying from alternatives like Paxos and the Saga Pattern, the chosen approach aims to enhance fault tolerance, ensure data consistency, and address challenges posed by distributed transactions in microservices-based applications. This integration provides a robust foundation for the system's reliability and consistency under varying operational scenarios.

4.1 Algorithm/Model Functionality

In the proposed lightweight failure management pattern merges the very basics of 2PC and Saga Pattern. By creating temporary databases for uncommitted transactions and implementing a retry mechanism, we achieved a robust fault tolerance mechanism. Effectively handling CRUD operations during faults is achieved by utilizing a dynamically assigned temporary database from the primary database, thereby minimizing the risk of errors in commits. This approach, integrated with the Saga Pattern, not only reduces the likelihood of mistakes but also amplifies the capability for successful transaction rollbacks in the event of microservice failures. The inclusion of the 2PC paradigm further enhances the coordination of distributed transactions, ensuring improved read-isolation and overall system performance. The workflow, illustrated in Figure 5, illustrates the thorough management of database changes, contributing to a seamless and uninterrupted state throughout the application, aligning closely with the lightweight failure management pattern proposed. Here, a web service application is implemented using a microservice architecture.

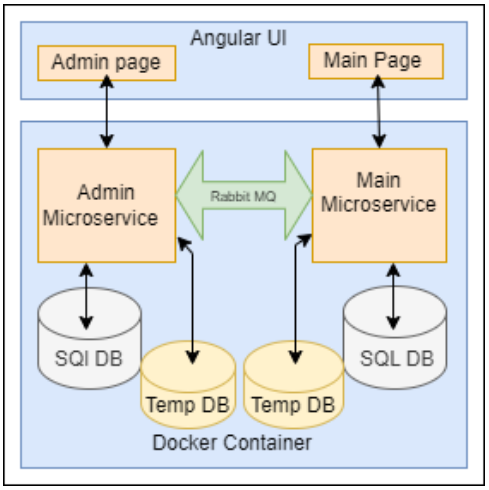

Figure 5: Illustration of the Micoservice architecture fault management with two-phases commit, saga pattern and Temporary DB

4.2 2PC Implementation in Proposed Solution with Saga Pattern Features and Temporary database

The following algorithm outlines the core steps of the proposed algorithm:

Step 1: Function handleTransaction(transaction):

- This function manages the coordination of distributed transactions between the Admin and Main microservices.

Step 2: Check the availability of all participating microservices:

- Send a request to each participant microservice (Admin and Main) to check their availability.

Step 3: If all microservices are available:

- Proceed with the transaction coordination process between the Admin and Main microservices.

Step 4: Perform Prepare phase:

- The coordinator (Admin microservice) sends a prepare message to all participants (Main microservice).
- Each participant (Main microservice) responds with a vote to commit or abort the transaction.
- One API request will be sent to each participant. Upon its availability, it will respond with a status code 200 for a commit vote, and in case of abort, it will respond with a status code 400.
  - Status code 200: Indicates a successful response to the request.

  - Status code 400: Indicates a client error, often used to signify that the request could not be understood by the server due to malformed syntax or other clientside errors.

Step 5: If all participants vote to commit:

- Proceed with the commit phase, with HTTP status code 200.

Step 6: Perform Commit phase:

- The coordinator (Admin microservice) sends a commit message to all participants (Main microservice).
- Each participant (Main microservice) executes the final commit operation.

Step 7: Else (if any participant votes to abort):

- Proceed with the abort phase, with HTTP status code 400.

Step 8: Perform Abort phase:

- The coordinator (Admin microservice) sends an abort message to all participants (Main microservice).
- Each participant rolls back any changes made during the transaction.

Step 9: Else (if any participant is not available):

- Store transaction details in a temporary database for future processing of the working microservice.
- If the Main service encounters a failure and the coordinator service (Admin microservice) receives an abort message (status code 400) from the Main service, the failure logs will be stored in the temporary database of the Admin microservice for future processing.
- Failure logs consist of the raw requests originating from the respective service's UI/API, requesting the execution of CRUD operations or other types of transactions.

Step 10: Function recoverFromFailure():

- This function handles the recovery process in case of participant unavailability.

Step 11: Send heartbeat messages to participants:

- Periodically send heartbeat messages to both the Admin and Main microservices to check their availability.
- A heartbeat message constitutes an API request sent to all participating microservices.

Step 12: If participant responds positively:

- If a participant (either the Admin or Main microservice) responds positively to the heartbeat message with a status code 200, it indicates a successful response.

Step 13: Retrieve failed transactions from the temporary database:

- Retrieve failed transactions from the temporary database for processing.

Step 14: Process transactions and commit changes to the primary database:

- Process transactions and commit changes to the primary database (Main database).
- This operation ensures that any pending transactions are completed and committed successfully.
- Additionally, send any dependent data to the associated microservice for further processing via a queue mechanism, ensuring seamless communication and coordination between microservices.

Step 15: Delete processed transactions from the temporary database:

- Once the transactions are successfully processed and committed to the primary database, delete them from the temporary database.

Step 16: Repeat steps 11-15 periodically:

- Periodically repeat the process of sending heartbeat messages, retrieving, processing, and deleting transactions for continuous fault recovery.
- The system will send heartbeat messages to the failed microservice repeatedly every 30 milliseconds.
- If the microservice responds with an HTTP status code 200 (OK), indicating availability, the system will proceed with fault recovery.
- If there is no response or the response is not 200 within 30 milliseconds, the system will continue sending heartbeat messages until the 30 milliseconds time limit is reached.
- After 30 minutes of continuous attempts without success, the retry operation will be exhausted.

This algorithm ensures the coordination of transactions between the Admin and Main microservices, even in the event of participant unavailability. By utilizing a temporary database and periodic heartbeat messages, the system maintains data consistency and integrity, thereby enhancing reliability and fault tolerance.

4.3 Temporary database set up

The temporary databases were set up and managed as follows:

Temporary Database Configuration:

- For demonstration purposes, an in-memory space within each microservice was utilized as the temporary database.
- This in-memory space was accessed using arrays, providing a lightweight and easily accessible storage solution.

Scalability Considerations:

- While in-memory storage was sufficient for demonstration, scalability concerns necessitated the consideration of using separate database instances in production environments.
- Separate database instances could provide more robust and scalable storage capabilities to handle larger volumes of data and ensure better fault tolerance.

Flexibility and Resource Optimization:

- Using a separate database instance for temporary storage offers greater flexibility and resource optimization, particularly in scenarios where multiple microservices may need to access and manage failure logs concurrently.
- This approach ensures that temporary database operations do not impact the performance or stability of the main database used for storing permanent data.

5 Implementation

5.1 Research project implementation

The practical implementation of the proposed lightweight failure management pattern for microservices was carried out through the development and testing of a microservices-based application called My ArtGallery. This application was designed to demonstrate the functionality and effectiveness of the fault tolerant mechanisms proposed in the research.

The implementation involved creating two main microservices: the Admin microservice and the Main microservice. The Admin microservice was responsible for handling administrative tasks such as adding, updating, and deleting products, while the Main microservice focused on displaying products and managing user interactions such as liking products.

Key components of the implementation include:

- **Web Application Development:** The front end of the application was developed using ReactJS Documentation (2024h), providing both an Admin page and a Main page for different user roles. These pages allowed users to perform actions like managing products and liking items.

- **Microservices Development:** The back end of the application was implemented using Python Documentation (2024f) and the Django framework Documentation (2024a). Two microservices, Admin and Main, were created to handle different functionalities of the application. APIs were developed to facilitate communication between the front end and the microservices and deployed it on Docker.

- **Database Management:** MySQL Documentation (2024d) was used as the primary database management system to store and manage structured data for both microservices. Each microservice had its own database to maintain data isolation.

- **Message Queue Implementation:** RabbitMQ Documentation (2024g) was used as the message broker to facilitate communication between the Admin and Main microservices. It enabled asynchronous communication and decoupling between the microservices, enhancing system scalability and resilience.

- **Fault Tolerance Mechanisms:** The fault-tolerant mechanisms, inspired by the TwoPhase Commit (2PC) algorithm and the Saga pattern, were integrated into the microservices architecture. A temporary database was introduced to store uncommitted transactions during fault scenarios, ensuring data consistency and integrity.

The practical implementation was carried out in various scenarios, including:

- **Normal Flow Without Failure:** This scenario validated the seamless interaction between the Admin and Main microservices, demonstrating the successful addition, update, retrieval, and deletion of products, as well as user interactions like liking products.

- **Fault Flow with 2PC, Saga, and Temporary Database:** This scenario simulated microservice unavailability and tested the fault-tolerant mechanisms. The integration of 2PC and Saga, along with the temporary database, facilitated fault recovery while maintaining data consistency and system integrity.

5.2 Candidate tool consideration and discarded tools:

Tools used to create the development environment were chosen based on their proven track record, reliability, and ability to meet the specific requirements of the project, including backend and frontend development, database management, message brokering, deployment, and version control.

- **Backend Development:** Python Documentation (2024f) and Django Documentation (2024a) were chosen for their readability, versatility, modularity, and scalability, providing a strong foundation for microservices. Java and Ruby on Rails were considered but discarded due to concerns regarding community support and scalability compared to Python and Django.

- **Frontend Development:** ReactJS Documentation (2024h) was chosen for its component-based architecture and good performance, outperforming competitors such as AngularJS due to its popularity and extensive community support.

- **Database Selection:** MySQL Documentation (2024d) was chosen for its performance, scalability, and stability, whereas competing databases such as Postgres and

MongoDB were ruled out due to MySQL's established track record and extensive acceptance.

- **Integrated Development Environment (IDE):** Visual Studio Code (VS Code) Documentation (2024i) was chosen due to its lightweight yet robust features and good support for a variety of programming languages and frameworks, above alternatives such as Eclipse in terms of versatility and ease of use.

- **Message Broker Selection:** RabbitMQ Documentation (2024g) was chosen for its dependability, scalability, and support for complex messaging patterns such as publish-subscribe, using MQTT, which is more compatible with microservices architecture.

- **Deployment Tool:** Docker Documentation (2024b) was chosen for its lightweight containerisation strategy, portability, and scalability, which ease the deployment process and ensure consistency across environments. Cloud services like AWS and Azure were rejected in favour of Docker due to its flexibility and ease of use.

- **Version Control:** Git Repository: Git Documentation (2024c) allows for version control, fostering collaboration by providing seamless access to code updates, improving reproducibility and transparency through branching for parallel development, and serving as a central hub for efficient issue reporting and improvement suggestions.

6 Evaluation

To validate the proposal's results, a microservices-based application was developed, with two interdependent microservices for data processing. Inspired by the Saga pattern and Two-Phase Commit, a temporary fault-tolerant database was introduced. The application comprises a web interface for an Art Gallery, including Main and Admin pages connected to respective microservices via APIs. The Main microservice communicates with the Admin microservice through RabbitMQ for publishing and subscribing to dependent data. The Admin page allows CRUD operations on images and titles, processed by the Admin microservice, stored in the Admin database, and published to maintain consistency. The Main page displays images and titles, facilitates liking images, and ensures consistency by updating the Main database and notifying the Admin microservice.

To demonstrate the proposed approach, the following use cases were implemented within the microservices-based application for the Art Gallery web interface.

6.1 Experiment / Case Study 1: Normal flow without failure

Use Case 1 explains the normal or expected flow when a user attempts to add, update, retrieve, or delete a product through the Admin page and tries to retrieve and like products through the Main page. Figure 6 illustrates the expected behavior of the system.

Following steps explain the verified expected flow of the system

1. When the admin Add, Update, or Delete an image or title by accessing the Admin page, the system processed and reflect the changes in the Admin database. Additionally, it published the added, updated, or deleted product (image, title)

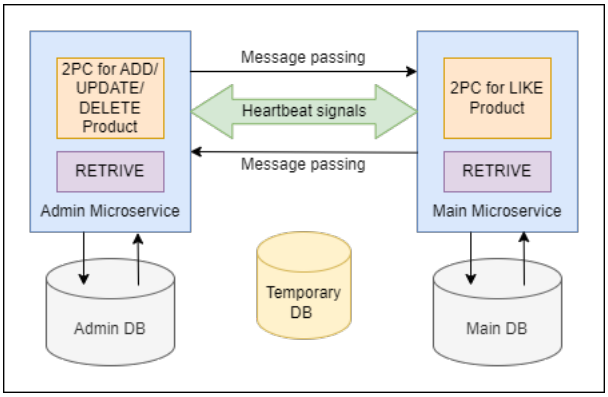

Figure 6: Illustration of Normal flow without failure.

along with a product ID, acting as a primary key, to the main microservice through RabbitMQ.

2. RabbitMQ then consumed the product published by the admin microservice and stored it in the Main database with the corresponding product ID.

3. Whenever a normal user accesses the Main page of the art gallery, the Main microservice retrieved all the products available in the Main database and sent the results to the main webpage, allowing users to see all the available products.

4. The main page also provided the functionality to LIKE a product. When the main page receives a like request, the Main microservice updated the like count in the Main database and published the count to the Admin microservice.

5. The Admin microservice then consumed the like count from the Main microservice and updated it in the Admin database to maintain consistency throughout the application.

6.2 Experiment / Case Study 2: Fault flow with 2PC, saga and temporary DB

Use Case 2 Outlines the fault-handling mechanism when any of the microservices is not operational. It illustrates how the Two-Phase Commit and Saga patterns collaborate with a temporary database during the recovery process. The Figure 7. illustrates the implementation of fault flow with Two-Phase Commit and a temporary database when any of the services is not responding.

Step 1

1. When the Admin attempts to ADD, UPDATE, or DELETE a product, or when a normal user tries to LIKE any product, the functionality influenced by the Two-Phase Commit (2PC) algorithm should verify the availability of all participating microservices.

2. If, due to some failure, the coordinator microservice (for ADD, UPDATE, or DELETE - Microservice Admin; for LIKE - Microservice Main) does not receive a 'Yes' response from the participant microservice, the coordinator microservice should

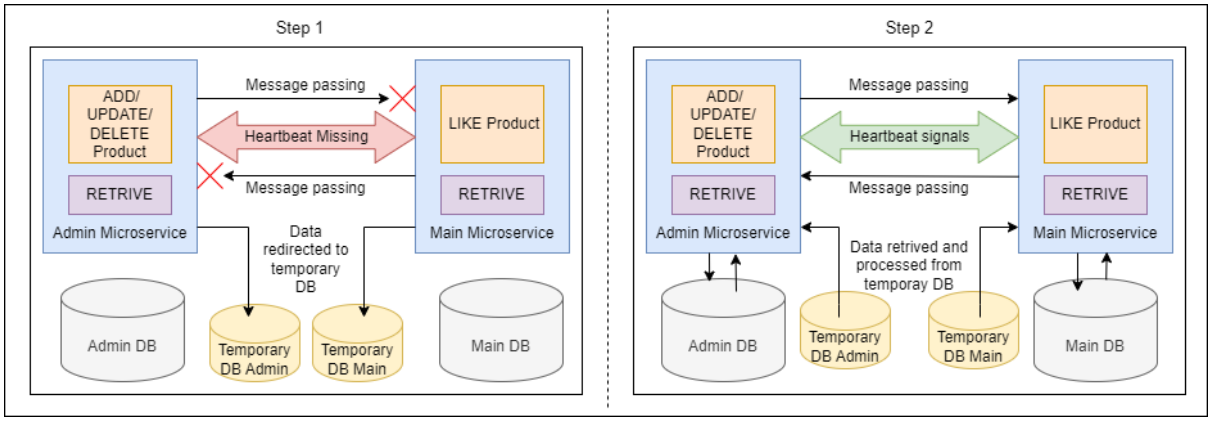

Figure 7: Illustration of fault flow.

store the failed logs in the respective temporary database without affecting other application functionalities and the primary database.

Step 2

1. The coordinating microservice should send heartbeat messages to the participating microservice at specific intervals to check its availability status. This automated recovery mechanism is influenced by the Saga pattern.

2. If the coordinator microservice receives a 'Yes' response from the participant microservice, it should first check for all available failed records in the temporary database. Retrieve all records from the temporary database, process them according to the normal flow, and store the results in the respective primary database. Additionally, a message should be sent to the dependent microservice to maintain data integrity. Once all data stored in the temporary database is processed, the corresponding failed logs should be deleted to free up memory.

3. While processing the normal flow, if any regular request arises, the system should process it according to the standard flow without involving the temporary database. In case of any subsequent failures, the system should then follow the fault recovery flow.

6.3 Experiment / Case Study 3: Fault flow without temporary database

Use Case 3 outlines the fault-handling mechanism without utilizing a temporary database. In the traditional approach, the primary database is used to store fault logs in the event of failure, or alternative techniques such as retrying in a queue may be utilized. However, these methods may introduce inconsistency in distributed database systems.

Fault flow with Storing Raw Data in primary DB

(Struk (2021),Laigner et al. (2021))

When implementing a fault-handling mechanism by storing raw data in the primary database while dealing with distributed databases, the potential issue may arise in the

following scenarios like for data inconsistency, high budget implementation, data redundancy.

1. Dealing with database failures is critical, and having a backup database with all of the data replicated from the main database reduces the risk of data loss. This redundant database can be utilized to service data demands until the primary database is ready to take on the load again. (geeksforgeeks (2022))

2. Fault tolerance mechanisms in distributed systems are used to sustain reliability and availability in the interconnected systems. These mechanisms include replication, redundancy, and high availability, and they help reduce the risks of human and economic loss due to system failures. (Sari and Akkaya (2015))

6.4 Discussion

This section presents an in-depth discussion of the findings from the experiments and case studies, offering insights and interpretations regarding the fault-tolerant flow, the tools used, and the suggested microservices-based application.

6.4.1 Validation Approach and Metrics Utilized in the Art Gallery Application

The art gallery application we created features CRUD operations using images and a like button. The important metrics within the algorithm are the HTTP return codes 200 for success and 400 for failure. These codes allowed us to check if the application is up and running. As for the retry mechanism, We followed the Circuit Breaker design pattern, with a retry every 30 milliseconds and exhaustion after reaching 30 minutes.

Generated a graph illustrating the recovery time from failure, depicting the time it takes Docker to reestablish the service. Our code is deterministic, so there is not much variability between runs. The graph illustrates the Request Volume Over Time following a system failure, with Time in milliseconds on the y-axis and Request Volume on the x-axis. Before the failure, requests were typically processed swiftly, with response times ranging from 0.141 to 0.159 milliseconds. After the failure, there was a significant disruption in system performance. During the recovery phase, marked by response times exceeding 100 milliseconds, there was a notable surge in request volume, with response times ranging between 103.396 and 171.727 milliseconds. Once the system stabilized, response times returned to their pre-failure levels, with requests being efficiently processed within milliseconds.

6.4.2 Measuring Data Integrity

This research addresses the identified gap of potential data loss in microservices architecture by conducting a comprehensive analysis of case studies on the proposed implementation. The study focuses on scenarios where users interact with microservices, simulating sudden application failures. Notably, inputted data remains intact and is committed to the main database upon application recovery, defining data integrity from a perspective of data consistency. Two case studies are presented:

- Case Study 1 evaluates normal flow without failure, validating interaction between Admin and Main microservices for product management and data consistency.

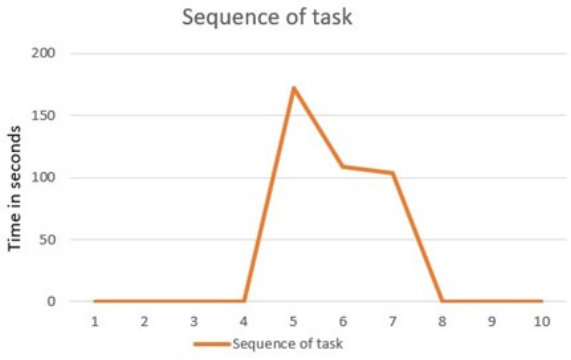

Figure 8: Illustration of Sequence of tasks. Only tasks 4-8 were affected by the failure

- Case Study 2 explores fault-handling mechanisms employing 2PC, Saga Pattern, and temporary databases, assessing the system's ability to maintain data integrity during microservice unavailability.

Results demonstrate the effectiveness of implemented fault-tolerant mechanisms, providing validation of hypotheses and highlighting the necessity for robust fault-tolerant mechanisms in distributed systems to ensure data consistency and system reliability.

6.4.3 Summary

Overall, the presented fault-tolerant mechanism for microservices provides a lightweight solution for data consistency using a temporary database for fault recovery. The retry mechanism is similar to other fault-tolerant approaches, but with far less overhead found in the full 2PC and Saga pattern architectures. It was therefore said the proposed algorithm is minimalist in that respect. These findings contribute to a deeper understanding of what the essential features for fault-tolerant microservices are. The proposed algorithm inherits the single-point of failure from 2PC, but extensions could be made to Paxos-like progressions when the coordinator node fails.

7 Conclusion and Future Work

A lightweight failure management pattern for microservices is introduced, merging elements from the Two-Phase Commit (2PC) algorithm and the Saga pattern. This approach ensures reliable handling of database transaction failures by utilizing retry mechanisms and temporary databases. Data consistency and dependability are maintained through RabbitMQ communication between Admin and Main microservices. Validation experiments confirm system functionality under various conditions, including fault scenarios and service recovery. Leveraging the 2PC algorithm preserves data consistency during service failures, safeguarding user experience with temporary database usage. The proposed pattern offers significant contributions compared to existing literature, enhancing fault tolerance with flexible recovery mechanisms and ensuring improved data consistency. Minimized overhead and complexity make it suitable for microservices-based applications, with adaptability to handle various failure scenarios and operational conditions. These contributions advance fault-tolerant distributed transactions, providing a lightweight, efficient, and adaptable solution tailored for microservices architectures, thus improving robustness and reliability in distributed systems.

The proposed research's future scope includes several important areas. At the moment, the lightweight failure management pattern inherits a single point of failure from 2PC, but obvious extensions (Paxos) were left out for time constraints. Performance optimization is another area of interest, requiring a thorough examination of bottlenecks and architectural improvements to enhance system responsiveness. Integration of serverless architecture, load balancing techniques, and additional containerization could address scalability concerns. On the security side, integrating failure management with data encryption and secure communication mechanisms remains an interesting aspect to explore in distributed transactions.

References

Ailijiang, A., Charapko, A. and Demirbas, M. (2016). Consensus in the cloud: Paxos systems demystified, *2016 25th International Conference on Computer Communication and Networks (ICCCN)*, IEEE, pp. 1–10.

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

Books.com.tw (2022). Shopping process. Accessed on 10 May 2022.
URL: *https://www.books.com.tw/web/sys$_q$alist/qa$_{12}$/0?loc* = 000$_0$02

Christian, J., Steven, Kurniawan, A. and Anggreainy, M. S. (2023). Analyzing microservices and monolithic systems: Key factors in architecture, development, and operations, *2023 6th International Conference of Computer and Informatics Engineering (IC2IE)*, pp. 64–69.

Country, S. (2021). Techniques for handling service failures in microservice architectures. Accessed: [2021].
URL: *https://softwarecountry.com/company/our-blog/fault-handling-in-dotnet/*

Daraghmi, E., Zhang, C.-P. and Yuan, S.-M. (2022). Enhancing saga pattern for distributed transactions within a microservices architecture, *Applied Sciences* **12**(12).
URL: *https://www.mdpi.com/2076-3417/12/12/6242*

Djerou, M. and Tibermacine, O. (2022). Saga distributed transactions verification using maude, *2022 International Symposium on iNnovative Informatics of Biskra (ISNIB)*, pp. 1–6.

Documentation, D. (2024a). Django guides walkthroughs. Accessed: [2024].
URL: *https://docs.djangoproject.com/en/4.2/*

Documentation, D. (2024b). Docker guides walkthroughs. Accessed: [2024].
URL: *https://docs.docker.com/guides/walkthroughs/*

Documentation, G. (2024c). Github. Accessed: [2024].
URL: *https://docs.github.com/en*

Documentation, M. (2024d). Mysql guides walkthroughs. Accessed: [2024].
URL: *https://dev.mysql.com/doc/*

Documentation, M. (2024e). Saga distributed transactions pattern. Accessed: [2024].
URL: *https://learn.microsoft.com/en-us/azure/architecture/reference-architectures/saga/saga*

Documentation, P. (2024f). Python guides walkthroughs. Accessed: [2024].
URL: *https://docs.python.org/3/*

Documentation, R. (2024g). Rabbitmq guides walkthroughs. Accessed: [2024].
URL: *https://www.rabbitmq.com/documentation.html*

Documentation, R. (2024h). React guides walkthroughs. Accessed: [2024].
URL: *https://react.dev/blog*

Documentation, V. (2024i). Vscode guides walkthroughs. Accessed: [2024].
URL: *https://code.visualstudio.com/docs*

Fan, P., Liu, J., Yin, W., Wang, H., Chen, X. and Sun, H. (2020). 2pc*: a distributed transaction concurrency control protocol of multi-microservice based on cloud computing platform, *Journal of Cloud Computing* **9**(1): 40.
URL: *https://doi.org/10.1186/s13677-020-00183-w*

geeksforgeeks (2022).
URL: *https://www.geeksforgeeks.org/handling-failure-in-distributed-system/*

Gördesli, M., Nasab, A. and Varol, A. (2022). Handling rollbacks with separated response control service for microservice architecture, *2022 3rd International Informatics and Software Engineering Conference (IISEC)*, pp. 1–4.

Haroon, R. (2016). Saga vs 2pc: An exhaustive exploration of distributed transaction protocols. Accessed: [2016].
URL: *https://blog.stackademic.com/saga-vs-2pc-an-exhaustive-exploration-of-dist*

Hasselbring, W. and Steinacker, G. (2017). Microservice architectures for scalability, agility and reliability in e-commerce, *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pp. 243–246.

Kończak, J. and Wojciechowski, P. T. (2021). Failure recovery from persistent memory in paxos-based state machine replication, *2021 40th International Symposium on Reliable Distributed Systems (SRDS)*, pp. 88–98.

Kończak, J., Wojciechowski, P. T., Santos, N., Żurkowski, T. and Schiper, A. (2021). Recovery algorithms for paxos-based state machine replication, *IEEE Transactions on Dependable and Secure Computing* **18**(2): 623–640.

Kumar, N., Sahoo, L. and Kumar, A. (2014). Design and implementation of three phase commit protocol (3pc) algorithm, *2014 International Conference on Reliability Optimization and Information Technology (ICROIT)*, pp. 116–120.

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

Messina, A., Rizzo, R., Storniolo, P., Tripiciano, M. and Urso, A. (2016). The database-is-the-service pattern for microservice architectures, *in* M. E. Renda, M. Bursa, A. Holzinger and S. Khuri (eds), *Information Technology in Bio- and Medical Informatics*, Springer International Publishing, Cham, pp. 223–233.

microservices patterns (n.d.). Chapter 4. managing transactions with sagas. Accessed: [2023].
URL: *https://livebook.manning.com/book/microservices-patterns/chapter-4/*

Perikov, I. (2020). 5 patterns to make your microservice fault-tolerant. Accessed: [2020].
URL: *https://itnext.io/5-patterns-to-make-your-microservice-fault-tolerant-f3a1*

Raharjo, A. B., Andyartha, P. K., Wijaya, W. H., Purwananto, Y., Purwitasari, D. and Juniarta, N. (2022). Reliability evaluation of microservices and monolithic architectures, *2022 International Conference on Computer Engineering, Network, and Intelligent Multimedia (CENIM)*, pp. 1–7.

Richardson, C. (2018). Microservice architecture : Pattern- saga.
URL: *https://microservices.io/patterns/data/saga.html*

Roth, E. and Haeberlen, A. (2021). Do not overpay for fault tolerance!, *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, IEEE, pp. 374–386.

Sari, A. and Akkaya, M. (2015). Fault tolerance mechanisms in distributed systems, *International Journal of Communications, Network and System Sciences* **08**: 471–482.

Struk, V. (2021). All you need to know about microservices database management. Accessed: [2021].
URL: *https://relevant.software/blog/microservices-database-management/*

tikv.org (2023). Distributed algorithms. Accessed: [2023].
URL: *https://tikv.org/deep-dive/distributed-transaction/distributed-algorithms/*

Uyanık, H. and Ovatman, T. (2020). Enhancing two phase-commit protocol for replicated state machines, *2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pp. 118–121.

Xiang, K. (2028). Patterns for distributed transactions within a microservices architecture. Accessed: [2028].
URL: *https://developers.redhat.com/blog/2018/10/01/patterns-for-distributed-transactions-within-a-microservices-architecture#*