MDPI

Article

Enhancing Saga Pattern for Distributed Transactions within a Microservices Architecture

Eman Daraghmi [1], Cheng-Pu Zhang [2] and Shyan-Ming Yuan [2,]*

[1] Applied Computing Department, Palestine Technical University Kadoorie, Tulkarm p3050950, Palestine; e.daraghmi@ptuk.edu.ps

[2] Computer Science Department, National Yang Ming Chiao Tung University, Hsinchu 300, Taiwan; a2323269@gmail.com

* Correspondence: smyuan@nycu.edu.tw

Abstract: The saga pattern manages transactions and maintains data consistency across distributed microservices via utilizing local sequential transactions that update each service and publish messages to trigger the next ones. Failure by one transaction causes the execution of compensating transactions that counteract the preceding one. However, saga lacks isolation, meaning that reading and writing data from an incomplete transaction is allowed. Therefore, this research proposes an enhanced saga pattern that resolves the lack of isolation issue via the use of the quota cache and the commit-sync service. Some transactions will be transferred from the database layer to the memory layer. Thus, no wrong commit to the main database will occur. If a microservice fails to be completed, the other microservices will run compensation transactions to rollback the changes that only affect the cache layer instead of the database layer. Database commit will be performed when all transactions are completed successfully. A lightweight microservices-based e-commerce system was implemented for comparison. Experiments were conducted for validation and evaluation. Results demonstrate that the proposal has the capability of resolving the lack of isolation. Results indicate that the proposal achieves better performance not only in typical cases but also in the scenario that needs to handle exceptions.

Keywords: microservice; saga pattern; distributed transaction; read isolation

Citation: Daraghmi, E.; Zhang, C.-P.; Yuan, S.-M. Enhancing Saga Pattern for Distributed Transactions within a Microservices Architecture. *Appl. Sci.* **2022**, *12*, 6242. https://doi.org/10.3390/app12126242

Academic Editor: Federico Divina

Received: 16 May 2022
Accepted: 16 June 2022
Published: 19 June 2022

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Copyright: © 2022 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

A microservice based-application [1,2] is a distributed system where the functionalities of the application are provided by multiple smaller services that are working together. The microservice architecture has restructured the monolithic application into several individual services in order to provide loose coupling, high maintainability, high availability, and scalability for the application development. The microservices architecture enables selecting the technology stack per service. For instance, the relational database could be employed to provide one service while the NoSQL database could be utilized to implement the other one; thus, allowing the services to manage the domain data independently. Moreover, with the microservice architecture, scaling data stores on-demand is enabled. Each microservice has its own database that contains some business transactions; therefore, managing distributed transactions and maintaining data consistency when transactions span across multiple services are challenging.

To manage distributed transactions in the microservice architecture, the Saga Pattern (Software Automation, Generation, and Administration) was proposed. Saga design pattern manages transactions and maintains data consistency across distributed microservices transactions. Saga is a set of local sequential transactions that is responsible for updating the microservices and publishing messages to trigger the next transactions. In the case of failure by one transaction, compensating transactions will be run to counteract the preceding one. However, the saga pattern does not have read isolation. It is ACD (atomicity, consistency,

durability), not ACID (atomicity, consistency, isolation, durability) [3]. The missing isolation means that data reading and writing from an incomplete transaction are allowed which, in turn, introduces various isolation anomalies [4,5]. Therefore, to address this problem, this research proposes an enhanced saga pattern that aims to achieve eventual consistency via the use of the quota cache and the commit-sync service. The idea is to integrate the standard saga pattern with an in-memory data caching layer by allocating the quota of the main database to the in-memory data caching server. Therefore, the CRUD (create, read, update, and delete) tasks will be handled via the quota cache instead of the main database which ensures that no wrong commit to the main database will occur. When a microservice fails to be completed, the other microservices will run compensation transactions to rollbacks the changes that only affect the cache layer instead of the main database layer. This will resolve the lack of read-isolation of the saga pattern and, additionally, it will enhance the performance. The database commit will be postponed and handled via the message queue middleware at the end of the workflow when all transactions are completed successfully to achieve eventual consistency. The clean architecture approach was utilized for the implementation where all the domain use cases have been defined in advance.

For demonstration, a lightweight microservices-based e-commerce system was implemented to compare the standard baseline version of the saga pattern with the proposed enhanced version. Several experiments were conducted for validation and evaluation. Results demonstrate that the proposed approach has the capability to resolve the lack of isolation in the saga pattern. Results also indicate that the proposed approach achieves better performance than the standard baseline version not only in typical cases but also in the scenario that needs to handle exceptions as employing the cache operations instead of the database operation enhances the performance and reduces the latency time.

2. Literature Review

2.1. Distributed Transactions in Microservices

The microservice architecture also known as microservices is an architectural style that enables organizing an application or a system as a collection of services. It allows the frequent, rapid, and reliable delivery of large, complex applications. As presented at the otto.de, one of the biggest European e-commerce platforms [2], the properties of the microservices include: high availability, scalability, loosely coupling, agility, and reliability. The microservice architecture restructured the monolithic application into several individual services. The most common challenge with the traditional monolithic application is the use of a single shared database which raises additional issues related to the scalability and the single point of failure [6]. A microservice architecture could be viewed as a distributed system where each transaction is distributed across multiple services that are invoked in sequence or in parallel to complete the entire workflow. As the microservice architecture enables applying the database per service pattern, transactions have to span across different databases. With a microservice architecture, handling and implementing distributed transactions that guarantee data consistency in addition to the rollbacks operation are key issues that need to be considered. The following subsections summarize the patterns to be used in implementing distributed transactions in a microservice architecture.

2.1.1. Two Phases Commit Protocol

One of the most popular patterns to implement distributed transactions in a microservice architecture is the two-phase commit protocol (2PC) [7]. In this protocol (see Figure 1), a coordinator is the component that controls transactions and contains the logic for managing them, while the microservices (participating nodes) execute their local transactions. With the 2PC protocol, a distributed transaction is executed in two phases. In the first phase or what is known as the prepare phase, the coordinator asks the participating nodes to commit the transaction. Thus, a yes or no response will be returned. In the second phase (i.e., the commit phase), when a yes response is received by the coordinator from all participating nodes, the coordinator, in turn, asks all nodes to commit. Note that the

coordinator asks all participants to rollback their local transactions upon receiving at least one negative response.

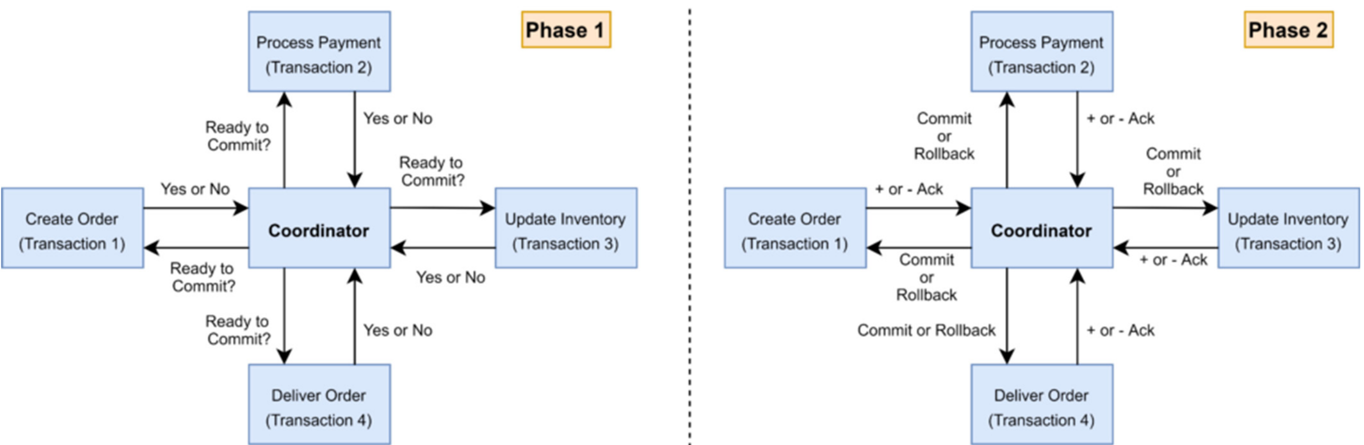


Figure 1. Illustration of the two-phases commit protocol.

Although the 2PC is viewed as a useful way to implement distributed transactions [8], the coordinator can become a single point of failure. Moreover, the overall performance of the transactions depends on the slowest service as all other services need to wait until the slowest service finishes its confirmation. Thus, it does not perform very well in large-scale and highly loaded systems [9].

2.1.2. Saga Pattern

To solve the problems with the 2PC protocol, the saga pattern was introduced [3,10] to organize the communication in a microservice architecture. In 1981, Campbell and Richards introduced the SAGA project [11] which explored both the practical and the formal characteristics of computer-aided support for the software lifecycle in order to enable the design of a practical software development environment. In 1987, Garcia-Molina and Salem [10] introduced the idea of utilizing the saga pattern for long lived-transactions by organizing them as a sequence of local transactions where each update of the database publishes an event or message to trigger the next local transaction. In other words, the saga has been introduced for updating data in multiple services in a microservices architecture without using distributed transactions. When one local transaction fails because of not complying with the business rules, a series of compensating transactions will be executed by saga to undo the changes that were made by the preceding local transactions.

The authors in [10] described the saga as a sequence of operations that perform a specific unit of work and are generally interleaved with each other. Saga operations are allowed to be rolled back by a compensating action. Saga pattern ensures either the successful completion of all operations or running the corresponding compensation actions for all executed operations to rollback any work previously done. The idea of the saga pattern is rather than having long transactions which hold into locks, long transactions will be broken into a series of short transactions that commit in sequence [10].

Generally, the choreography and the orchestration are the most two common approaches to coordinating the saga pattern. With choreography, sagas are coordinated with no centralized point of control where events are exchanged by participants. Domain events are published by local transactions to trigger local transactions in other services. On the other hand, the orchestration is another means to coordinate sagas where sagas will be coordinated with a centralized controller who tells saga participants what local transactions to execute based on the events. With the orchestrator, saga requests are executed, stored, and the task states are interpreted. With compensating transactions, the orchestrator handles failure recovery.

The saga pattern was utilized by previous research. In general, implementing the saga pattern requires a creative way of thinking in order to coordinate transactions and maintain data consistency for a business process that spans multiple microservices. The design of a SAGA-based Pilot-Job was proposed in [12]. The design proposed by the research supports several types of applications to be usable over a broad range of infrastructures. In [13], a multi-agent-based framework in the microservices architecture, namely SagaMAS is proposed where the distributed transactions are coordinated by the framework simplifying the interactions among microservices and relieving developers from coordination tasks. In [14], the authors proposed a model-based approach for microservices and service integration through formal models using the UML and UML profiles. Previous research concluded that the saga architecture pattern is a useful means to implement distributed transactions in a microservice-based system. However, the saga pattern is ACD (atomicity, consistency, durability), not ACID (atomicity, consistency, isolation, durability) [3]. Read-isolation is missing in the saga pattern which in turn allows reading and writing data from an incomplete transaction [4] and in saga, the microservices commit changes to their local databases. The lack of read-isolation imposes a durability challenge. The saga implementation must include countermeasures to reduce anomalies.

3. Proposed Approach: Design and Components

This section demonstrates the proposed approach that enhances the saga pattern via the use of the quota cache and the eventual commit sync service. In a microservices architecture, the system will be structured as a collection of services, namely microservices each of which has its own database. The proposed approach employs in-memory data caching to resolve the read-isolation issue in the saga pattern. A quota from the main database will be allocated to the memory cache server initially. The operations of the databases will be transferred to the memory cache server. Database commit will only be performed when transactions are completed successfully.

The workflow of a standard e-commerce microservices-based system will be illustrated in the next section. Moreover, the components of the proposed approach will be demonstrated in detail in Section 3.2.

3.1. Workflow of a Standard E-Commerce Application

Figure 2 presents the event workflow of an e-commerce microservices-based system. The system allows buying products over the Internet, and offers the possibility to select the products, the payment method, and the shipping means. This is a long-lived transaction that consists of several microservices: the Warehouse-Service, the Order-Service, the Billing-Service, and the Shipping-Service. As shown in Figure 2, the system includes both the warehouse-before-billing and the billing-before-warehouse to simulate real-world e-commerce applications [15]. The flow starts with the Warehouse-Service that fetches goods. Then, the Order-Service initializes an empty order marked as "IN-PROGRESS". The order will be marked as "FAILED" when the goods cannot be fetched. The Billing-Service in turn validates the specified payment. The Billing-Service collects the payment if the validation is completed successfully; otherwise, it terminates the flow with the order marked as "FAILED". The Shipping-Service dispatches the delivery. The Order-Service, finally, completes the order and updates the order information including: the status, the shipping-id, and the amount.

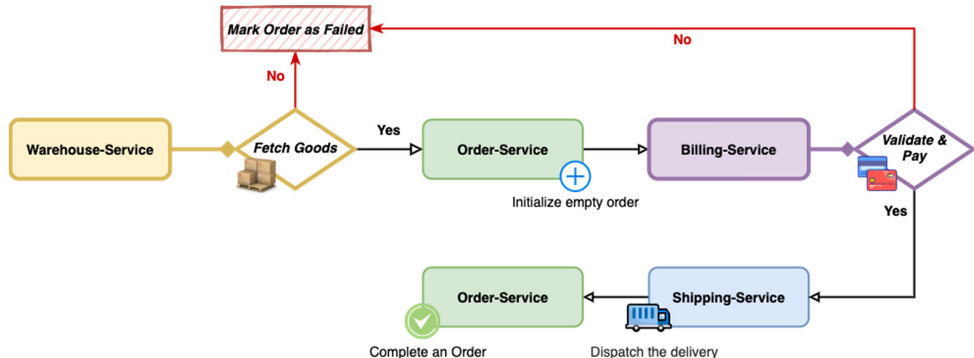


Figure 2. Saga workflow of a standard e-commerce microservices-based system.

3.2. Components of the Proposed Approach

This section demonstrates the backend components of the proposed enhanced saga pattern. Figure 3 shows the architecture and the tech stacks of the e-commerce system whose workflow is described in Section 3.1. The clean architecture approach was utilized for the implementation [16]. All the domain use cases have been defined in advance. For example, the Billing-Service has: the add-payment-use-case, the create-billing-use-case, the validate-payment-use-case, the payment-pay-use-case, the revert-payment-pay-use-case, etc.

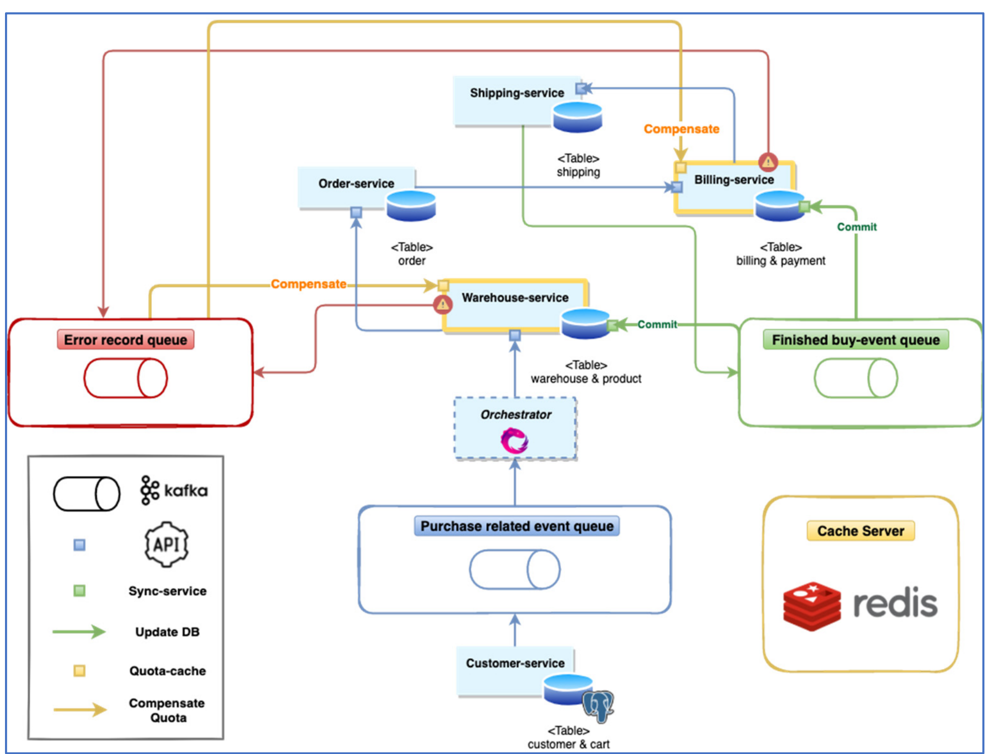


Figure 3. The architecture of a system implemented the proposed approach (enhanced saga pattern).

3.2.1. Microservices

As shown in Figure 3, five microservices are involved in the system: the Warehouse-Service, the Order-Service, the Billing-Service, the Shipping-Service, and the Customer-Service. Each microservice has its own database. The microservices were developed via the spring boot technology that has inherited the relevant use cases and implemented them [17] (see Figure 4). The microservices were exposed via the REST API [18,19] internally; thus, communication can be accomplished with the simple HTTP method.

Figure 4. The tree of the implemented e-commerce system modules.

3.2.2. Message Queue Middleware

The message queue middleware is a distributed event-store and a stream-processing platform handling both the failure and the completion events. It supports sending and receiving messages between the microservices. It preserves the order of requests from the microservices to guarantee the correctness of the eventual commitment. As shown in Figure 3, various types of message queue middleware were applied to handle the different events involved.

Apache Kafka [20] was utilized as a message queue middle. Apache Kafka is an open-source distributed event streaming platform that can be used to publish and subscribe to the streams of messages. Kafka is able to build a high throughput application as it is capable of handling thousands of messages per second. Kafka has the durability characteristic; thus, it always will store the messages on the disk for persistence [21].

3.2.3. Quota Cache

A cache in computing is generally a hardware or a software component that stores data. It is known as a high-speed data storage layer. A cache increases the performance of data retrieval by reducing main memory access. The quota cache (see Figure 5) is the quota of a specified resource from the main database. Quota is a feature that estimates the available bytes (i.e., the amount of available space) to store contents.

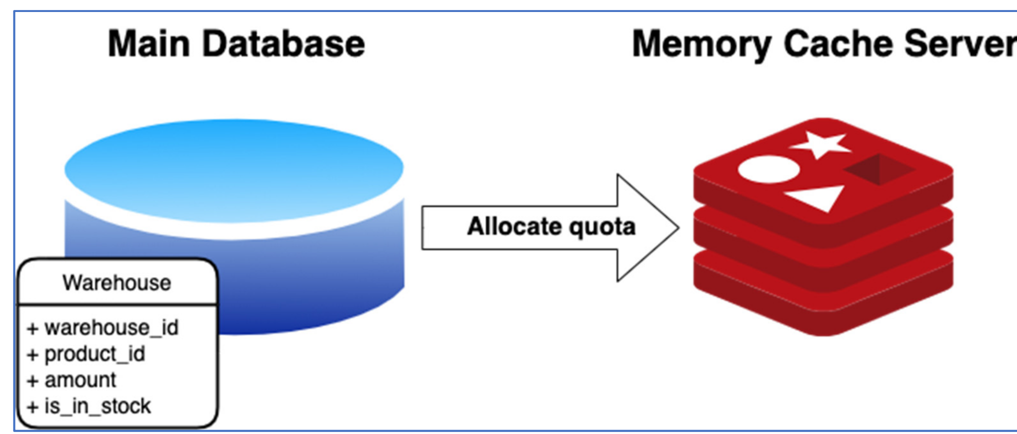


Figure 5. Quota cache example.

To resolve the read-isolation property in the saga pattern, in-memory data caching is utilized. A quota from the main database will be allocated to the memory cache server. The CRUD (create, read, update, and delete) tasks will be handled via quota cache. This in turn will never cause a wrong commit to the main database. In addition, the microservices which apply the quota cache will also benefit from the in-memory operations to ensure low latency and high throughput and thus achieve better performance in comparison to the baseline standard saga.

As shown in Figure 3, microservices that require validation apply the quota cache as exceptions may occur when the validation fails. When a microservice fails to be completed, the other microservices will run compensation transactions to rollback the changes. Assuming that an error occurs from the Warehouse-Service when fetching the goods, in the standard saga pattern, initial good fetching will cause updating of the main database; thus, a customer could see the created order. However, when the compensation transaction is run to rollback the changes, the order will be removed in the next few seconds. In contrast, with the enhanced version, the CRUD operations are moved to the cache level instead of the main database level. This will not cause a wrong database commit or update. The compensation request will be sent to the corresponding message queue middleware in the case of an error. Several message queue middleware were utilized to handle different events, including: the buy events, the completion events, and the failure events. For instance, the request will send to the warehouse-compensate queue if an error occurs in either the Warehouse-Service or the Billing-Service. Moreover, the request will be sent to the payment-related queue if an error occurs in the Billing-Service. As a result, with the enhanced saga pattern, database commit will only be performed when the transaction is completed successfully.

Redis [22] was utilized as our memory cache server. Redis is an open-source database, with an in-memory data store. Redis is suitable for building low latency in-memory cache, and also can increase the throughput. Several well-known companies, such as Twitter, GitHub, and StackOverflow, have applied Redis.

3.2.4. Eventual Commit Sync Service

This service is a synchronous commit and will be blocked until either the commit succeeds or an unrecoverable error is encountered (in which case it is thrown to the caller).

Although the CRUD operations are shifted to the memory cache server, there is still a need to commit to the main database. Therefore, the eventual commit sync service is needed.

The system that applies the proposed approach will send the "finished message" to the message queue middleware at the last step when all the events are accomplished successfully. Then, the microservices that have applied the quota cache will get the event-record and perform database commit. In other words, database commit will be done only when all the events are successful.

As shown in Figure 3, when all the purchase events are successfully accomplished, a successful message will be sent to the "finished-buy-event-queue" and both the Warehouse-Service and the Billing-Service will get the event record. Finally, these two microservices will run the eventual commit sync service to perform database commit. Therefore, the proposed approach ensures that no incorrect commit to the main database will occur.

3.2.5. Orchestrator Module

In the orchestrator module, all the microservices were configured to the corresponding web-client, and organized via the RxJava library which composes asynchronous and event-based programs by using the observer pattern [23]. Due to the orchestrator module, all the microservices can be managed easily and the workflow can be adjusted when needed.

3.2.6. Main Database

For the main database, PostgreSQL [24,25] was employed. PostgreSQL is an open-source object- relational database that is suitable for java application development.

4. System Implementation

This section illustrates the implementation of the preceding illustrated system (see the Supplementary Materials section). Two versions of the system will be implemented for later comparison. The first version employs the baseline standard saga pattern and the second version utilizes the proposed approach (i.e., enhanced saga pattern via the use of the quota cache and the eventual commit sync service).

The implementation is mainly based on the spring boot. The communication between the microservices is done via the REST API, which lets the other services obey the REST rules. Moreover, several message queues middleware were utilized to handle the different events, including: the buy events, the completion events, and the failure events. With these message queues, higher throughput can be achieved for the entire system.

For the microservices architecture, there are two approaches to coordinate sagas: the orchestration and the choreography. In our proposal, both the orchestration and the event choreography techniques were utilized for implementation.

- With the orchestration-based saga, the manager controller manages all the communications among the microservices. As shown in Figure 3, in our proposal, the orchestrator module is responsible for telling the corresponding microservice what transactions have to be executed. Thus, the order of the workflow could easily be managed without changing any microservices. When the orchestration module captures a "buy-event", it tells the Warehouse-Service to start fetching goods. Additionally, the orchestration handles both the failure and the completion events via the message queue middleware where each event happens in an asynchronous manner. When all transactions are completed successfully, the orchestration publishes a "complete-event" to the message queue middleware which enables the Warehouse-Service and the Billing-Service to consume it and perform database commit.
- The choreography service applies a decentralized approach to service composition. In our proposal, via the choreography-based saga, after a microservice finishes its local transaction, it will publish domain events that will be subscribed by the other microservices to trigger their local transactions.

4.1. Version 1: A System with the Standard Baseline Standard Saga Pattern

In this version, the back-end components excluding the memory cache server and the message queue middleware were employed for implementing the standard saga pattern. The standard baseline system (see Figure 6) is a standard saga pattern that organizes all the microservices via the orchestrator module, and each task will follow the workflow explained in Section 3.1. The logic of the revert transaction that is responsible for the rollbacks operation to the main database was implemented (i.e., to be used for the scenario with the exception). Once the "buy event" is published, the orchestrator module will start the event workflow: The Warehouse-Service fetches the goods based on the request, the Order-Service initializes the order for the customer, the Billing-Service validates the specified customer's payment and collects it, the Shipping-Service dispatches the delivery for a customer, and, finally, the Order-Service completes the created order. There may be a need to perform the revert transaction in the case of an error after fetching the goods by the Warehouse-Service.

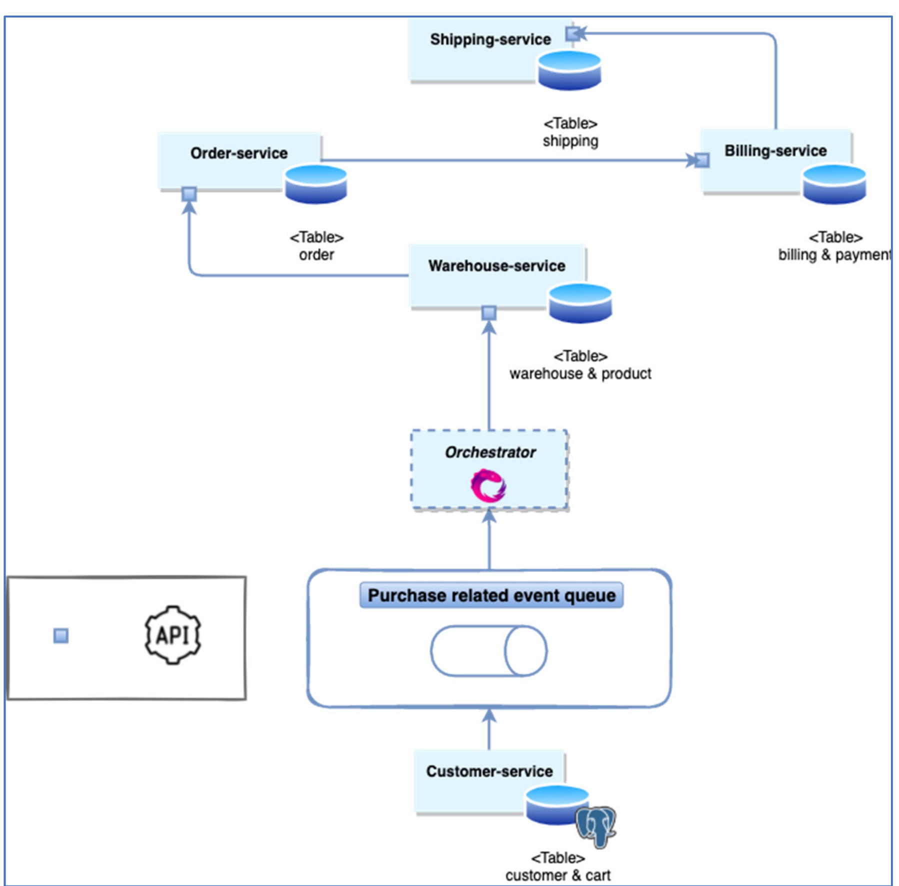


Figure 6. The architecture of a system with the standard baseline standard sage pattern.

4.2. Version 2: A System Implemented the Proposed Approach (Enhanced Saga Pattern)

This version (see Figure 3) employs all the back-end components, including: the memory cache server and the message queue middleware. As shown in the figure, the memory cache server and the message queue middleware only apply to the Warehouse-Service and the Billing-Service as only these two microservices may perform the rollbacks operation when an error occurs in the system (i.e., the fetched goods and the collected payment should be reverted in the case of exception).

The Warehouse-Service fetches goods in the memory layer using Redis, the Billing-Service validates the customer's payment and collects it in the memory layer, the Shipping-Service dispatches the delivery for a customer, and the Order-Service completes the order. Simultaneously, the system will publish the finished event to the corresponding message

middleware. The subscribed services will consume it and commit the specified update to their main database. Moreover, this system has compensation transactions that are run in the case of an error. Compensation transactions affect only the allocated quota cache instead of the main database (i.e., the memory layer instead of the disk). Moreover, the microservices, which apply the quota cache, postpone the database commit which, in turn, will be handled via the message queue middleware at the end of the workflow to achieve eventual consistency [26]. Thus, by applying the quota cache and the eventual commit sync service, the database commit will only be executed if all the involved events are completed successfully.

5. System Validation and Performance Evaluation

To verify the capability of the proposed approach in resolving the lack of read-isolation in the saga pattern, validation experiments that simulate a completion scenario with no exception and with a failure scenario were designed. Several experiments will be conducted on the two implemented versions based on both scenarios and the log of the systems will be observed. In addition, the performance of both systems was evaluated.

5.1. System Validation

For system validation, the main database and the memory cache server-related logs were exported from both versions in order to track and test the read-isolation in the saga pattern. The traced log contains information, such as the timestamp, the thread name, and the message. The experiment will show that by applying the proposed approach, there is no need to do the rollbacks operation to the main database.

5.1.1. Monitoring Tools

To export and collect the logs from each microservice, logs were structured via the logstash logback encoder, which provides log back encoder and appenders to log in logstash's JSON format [27]. In addition, Loki [28] was used for performing log aggregation. For observability, Grafana [29], which is an open-source web interface, was deployed. For the experiment, there is a need to push the logs to the log aggregation server from each microservice; thus, the Promtail tool [30] was deployed for retrieving the specified local logs to the Loki instance as it has strong integration with the Loki.

5.1.2. Experiment Results

Two scenarios were designed for validating the systems, one with no exception occurring and the other with an exception occurring during the payment.

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

Once the orchestrator module captures a "buy-event", the event workflow will be tackled by the corresponding microservices. The Warehouse-Service starts to fetch the specified number of goods at the first step, as shown in Figure 7. As shown in the "logger_name" field, this operation was handled in the main database. After that, the Order-Service initializes the order. The payment validation then will be handled via the Billing-Service. In the case of validation failure, the workflow will be terminated and the order will be marked as "FAILED". Since this scenario represents the case with zero error, the validation will be completed successfully. In the third step, the Billing-Service collects the specified payment (Figure 7 shows the steps performed from 1 to 4). The Shipping-Service then dispatches the delivery based on the customer's request. Figure 8 shows that the customer selects "boat" as a delivery means. In the last step, the Order-Service completes the order and updates the required information, including the order status, the shipping id, the amount, and the order status.

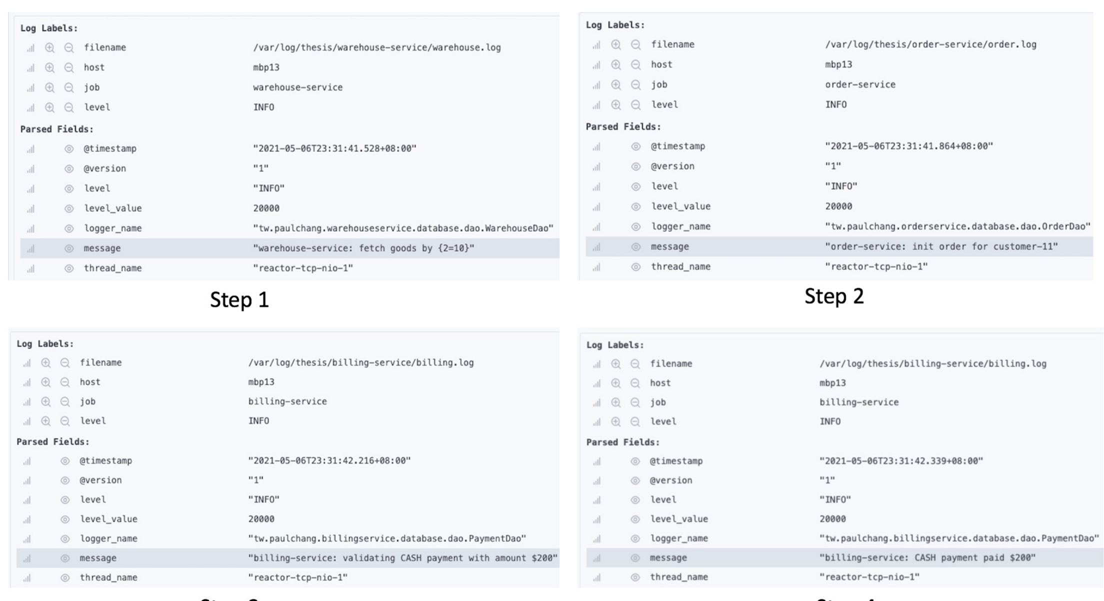
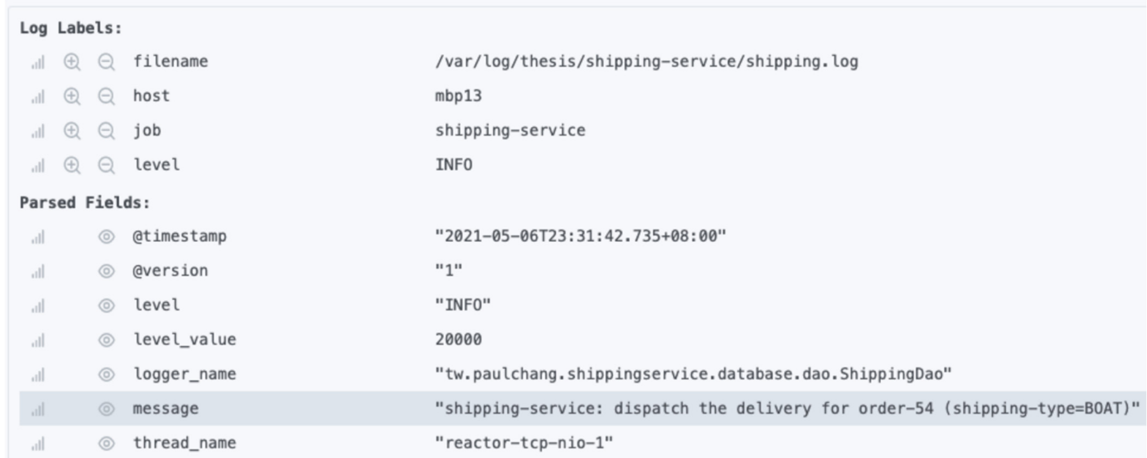


Figure 7. Steps 1–4 in version 1 (deploying standard saga pattern).



Step 5

Log Labels:
filename /var/log/thesis/order-service/order.log
host mbp13
job order-service
level INFO
Parsed Fields:
@timestamp "2021-05-06T23:31:42.878+08:00"
@version "1"
level "INFO"
level_value 20000
logger_name "tw.paulchang.orderservice.database.dao.OrderDao"
message "order-service: complete order on order-54 (shippingId=17, amount=$200, status=SHIPPED)"
thread_name "reactor-tcp-nio-1"

Step 6

Figure 8. Steps 5, 6 in version 1 (deploying standard saga pattern).

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

Step 1 and Step 2 performed by version 2 are almost the same as version 1 as only the Warehouse-Service and the Billing-Service apply the proposed approach. Note that as shown in (Figure 9), data access was handled by Redis, the memory cache server, instead of the main database. In other words, no transactions on the main database are needed at these steps. **Step 3** is shown in Figure 9, where validating the payment is transferred from the main database to the memory cache server. **Step 4:** After the validation is completed successfully, the Billing-Service collects the payment and updates the specified fields at the memory cache serve. **Steps 5 and 6** are the same as version 1 except that the orchestrator module publishes the completion event to the corresponding message queue middleware.

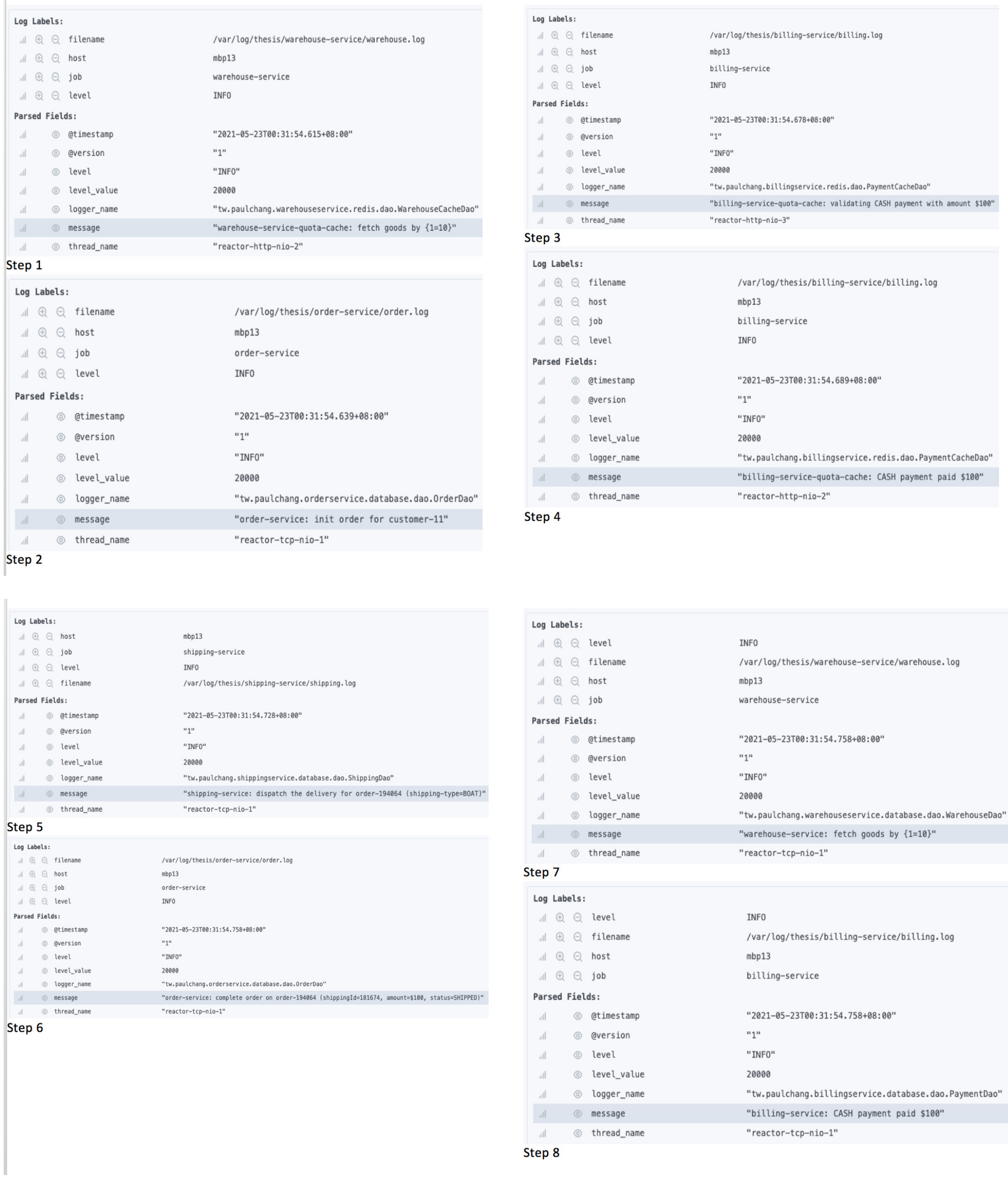


Figure 9. The Steps performed by version 2 (deploying enhanced saga pattern).

Once the Warehouse-Service and Billing-Service consume the "completion-event" message, they will perform the specified transaction to the main database, respectively. The update operation has already moved to the memory cache server in both the Warehouse-Service and the Billing-Service. It will perform the relevant transactions to the main database as long as they get the completion event message.

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

The event workflow of an e-commerce microservices-based system with a payment exception is shown in Figure 10. This workflow will be deployed to test both versions of the system.

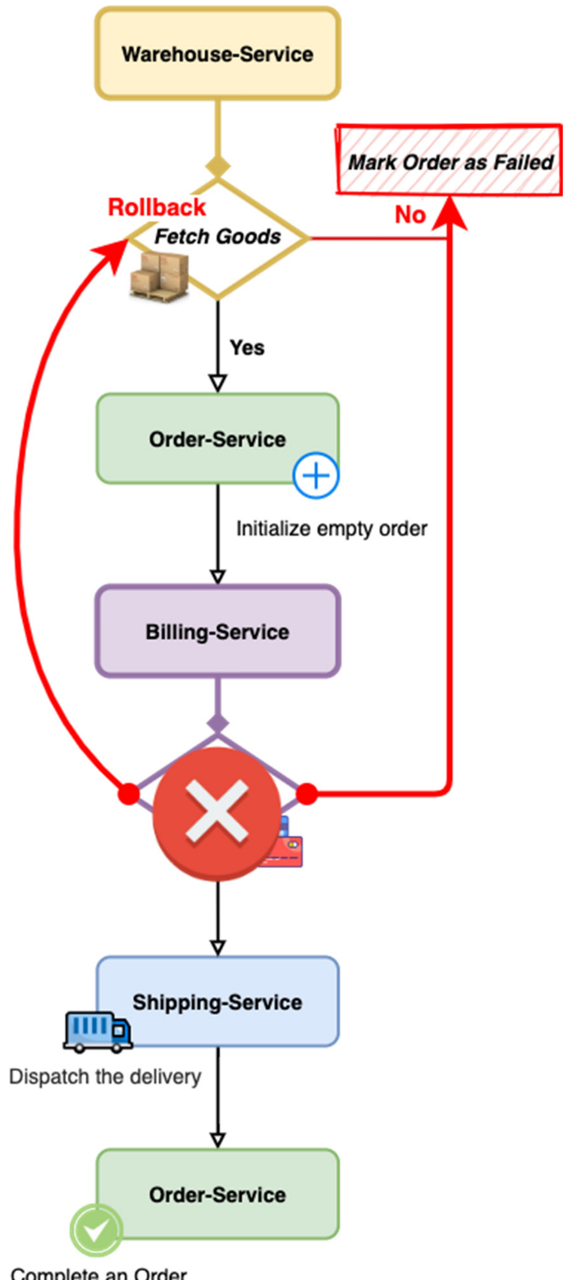


Figure 10. The event workflow of the application with a payment error scenario.

Figure 11 shows the steps performed by the standard version that applies the standard saga pattern. The first and the second steps fetch goods and initialize the order. Step 3 shows

that a large amount is given that the customer cannot afford so payment validation failure will occur. As an exception occurs, rollbacks will be executed at step 4, the Warehouse-Service reverts the fetched goods. After rolling back the fetched goods, the Order-Service completes the order and marks it as failed.

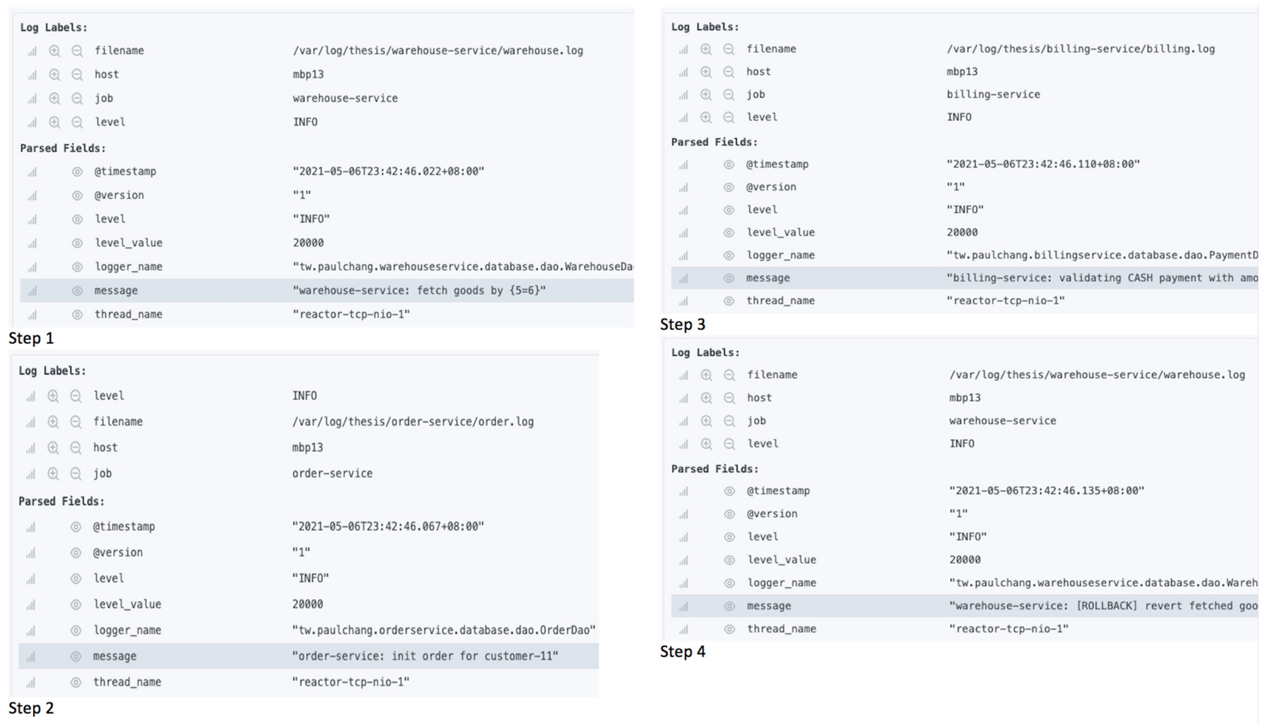
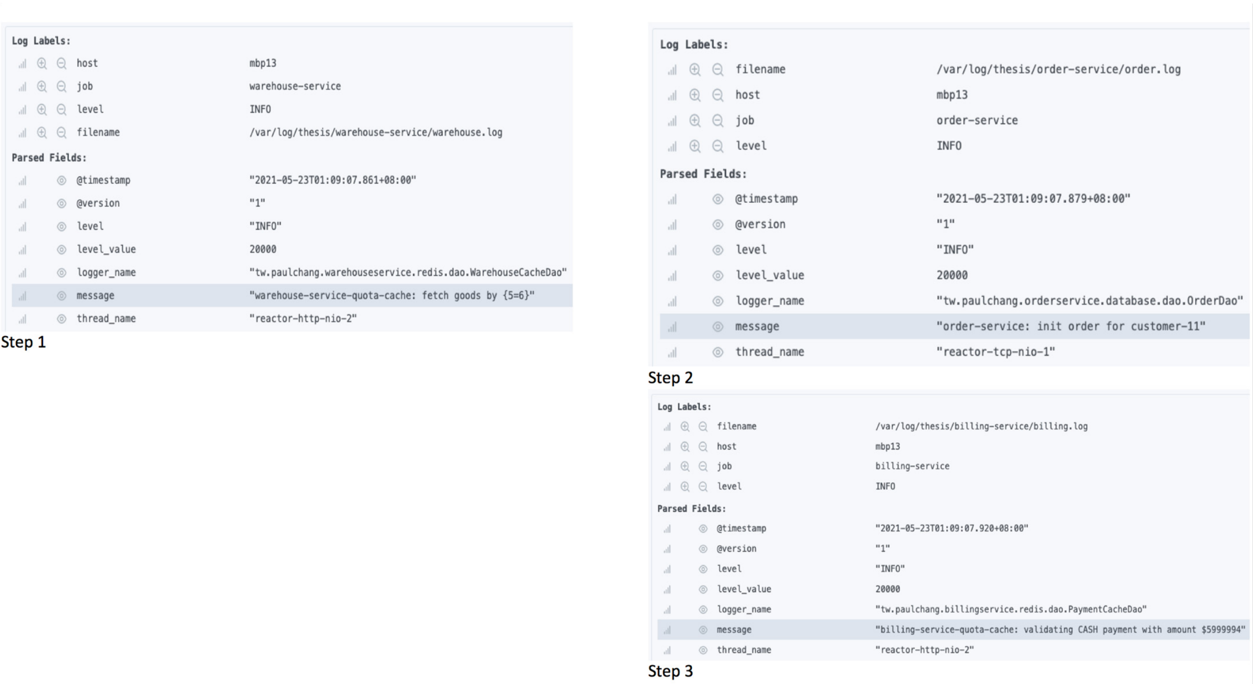


Figure 11. The Steps performed by version 1 (deploying standard saga pattern) when an exception has occurred.

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

Figure 12 shows the steps performed by version 2 that applies the proposed enhanced saga pattern. In the first and second steps, the goods will be fetched and the order will be initialized. Step 3 shows that a large amount is given that the customer cannot afford so payment validation failure will occur. A “failure-event” will be published to the message queue middleware instead of performing the rollbacks to the main database. In the last two steps, it can be observed that the timestamp of both logs is almost the same which means the system sent the failure event and completed the order at the same time. Figure 12 shows that the Warehouse-Service compensates for the fetched goods via the memory cache server, with zero change to the original main database. The logs indicate that the improved

proposed approach can handle the exception and never perform the rollbacks operation to the main database when errors occur.



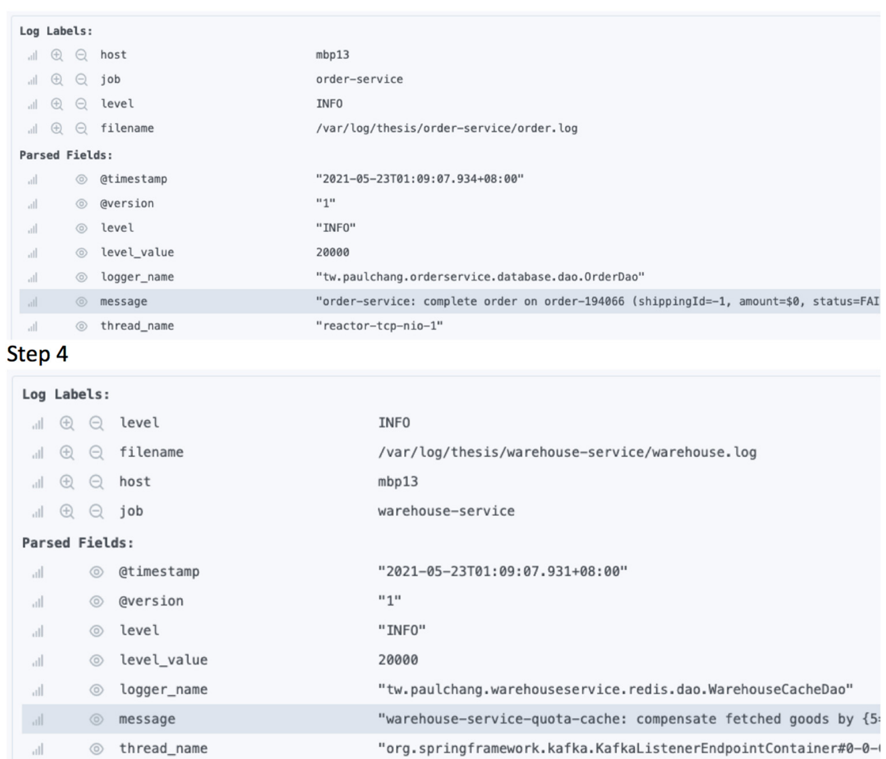


Figure 12. The Steps performed by version 2 (deploying enhanced saga pattern) when an exception has occurred.

6. System Evaluation

This section illustrates the system performance based on two-evaluation metrics: (1) the number of requests that can be handled in a specified duration in order to compare the throughput of both versions; (2) the consumed time in a scenario with errors occurring. The response time will be measured to specify the time taken by both approaches for processing. For testing the performance, the k6 [31] tool was utilized. This tool is considered a powerful load-testing tool globally as it provides an approachable scripting API and a flexible configuration. The following configurations were set in the tool.

- VUs: The number of the virtual users (VUs) to run the specified script concurrently;
- Iterations: The fixed number of iterations that specify the script, usually works together with VUs;
- Duration: The string which specifies the total duration time to run the testing script.

Performance Analysis

1. Throughput: Number of requests that are handled in a specified duration

In this experiment, the duration was set to 1 min and 3 min. The VUs were set to 1, 10, 15, 50, 100, and 150. Moreover, there is a one-second sleep between every iteration.

- Throughput Results: 1 min duration

The results of the throughput for 1 min duration are shown in Figures 13–18. As shown in the figures, the average values of the throughput for version 1 with (1 VU/10 VUs/15 VUs/ 50 VUs/100 VUs/150 VUs) are 297.5, 592.8, 781.5, 2400, 3170, and 4378 respectively, where the average of throughput for version 2 with (1 VU/10 VUs/ 15 VUs/50 VUs/100 VUs/ 150 VUs) are 299.5, 596.4, 887.3, 2928, 4442, and 6393 respectively. Results indicate that the system "version 2" that utilizes the proposed approach (enhances saga pattern) achieves better throughput. Although the duration of this experiment is short, the difference between the two versions can still be observed.

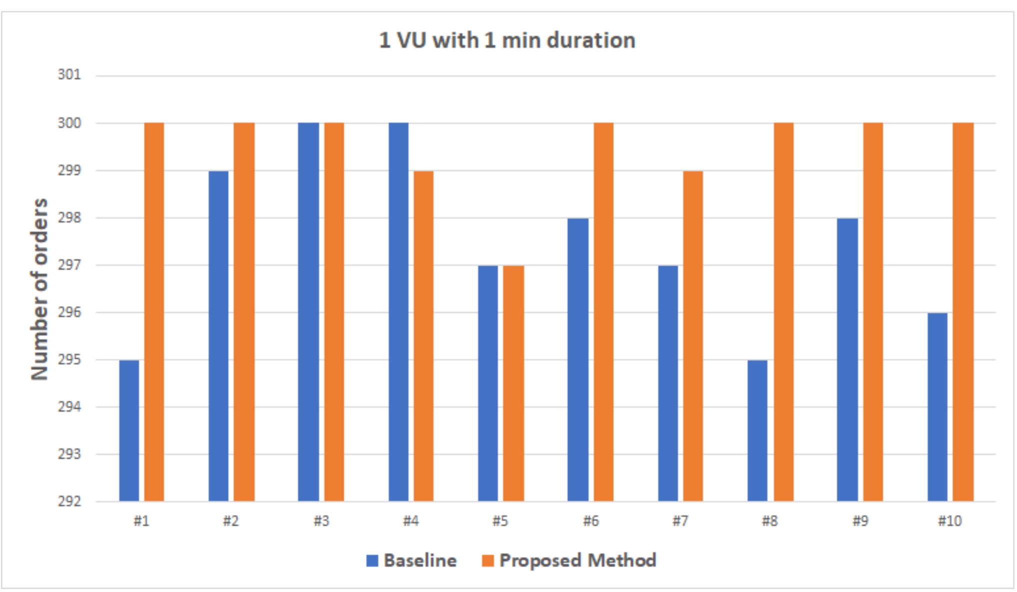


Figure 13. Load Testing—1 virtual user with 1 min duration.

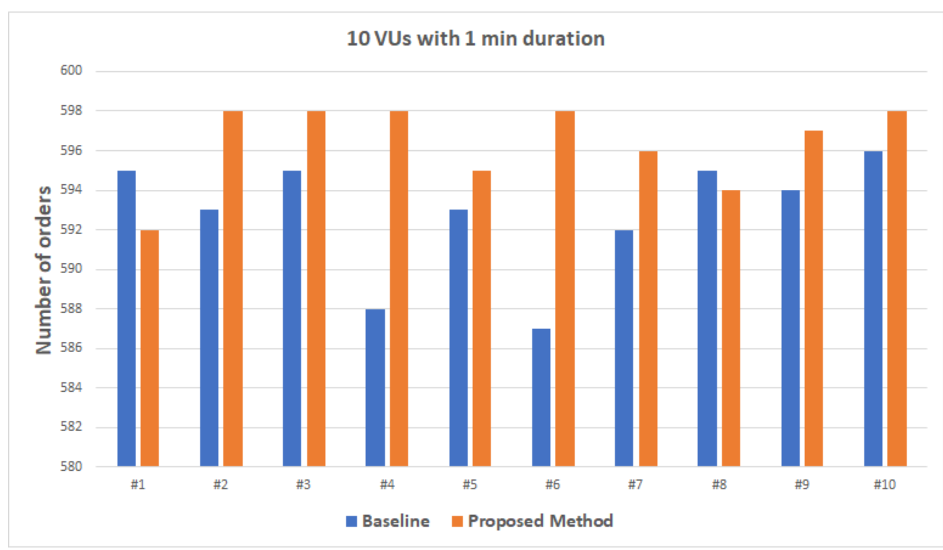


Figure 14. Load Testing—10 virtual users with 1 min duration.

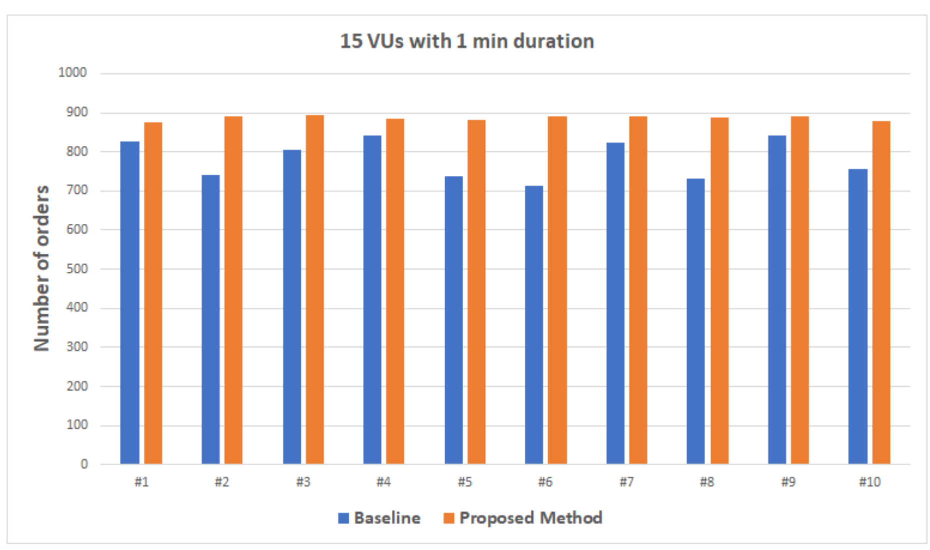


Figure 15. Load Testing—15 virtual users with 1 min duration.

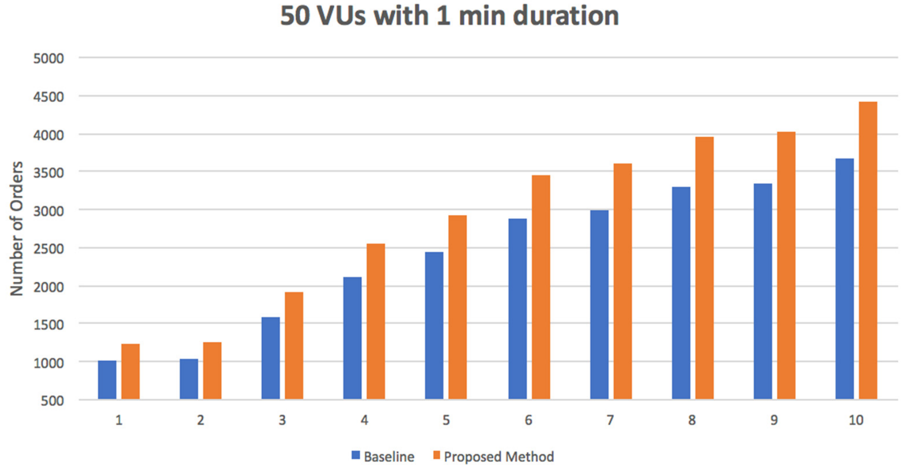


Figure 16. Load Testing—50 virtual users with 1 min duration.

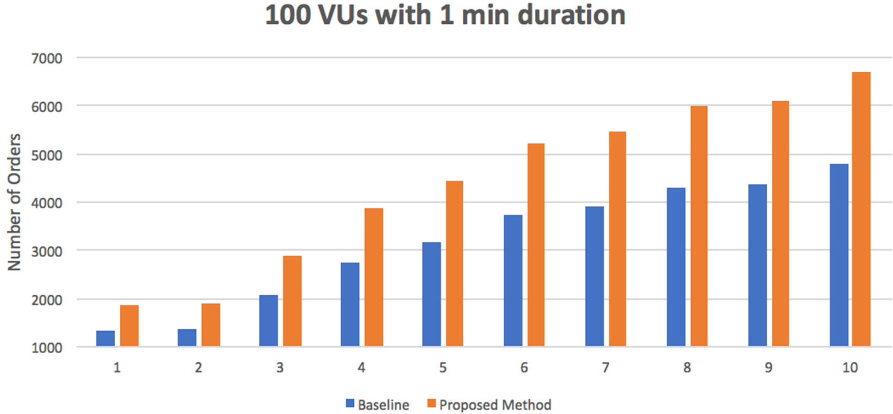


Figure 17. Load Testing—100 virtual users with 1 min duration.

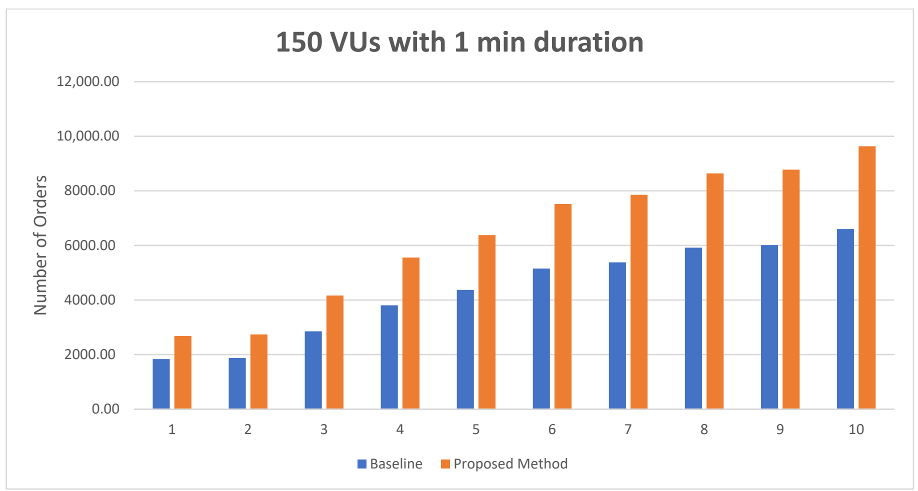


Figure 18. Load Testing—150 virtual users with 1 min duration.

- Throughput Results: 3 min duration

The results of a 3 min duration are shown in Figures 19–24. The average of the throughput for version 1 with (1 VU/10 VUs/15 VUs/50 VUs/100 VUs/150 VUs) are 889.4, 1779.3, 2526.1, 3485, 4530.5, and 6252, where the means of the throughput for version 2 with the same number of VUs are 890.7, 1780.8, 2669.5, 4182, 6342 and 9128 respectively.

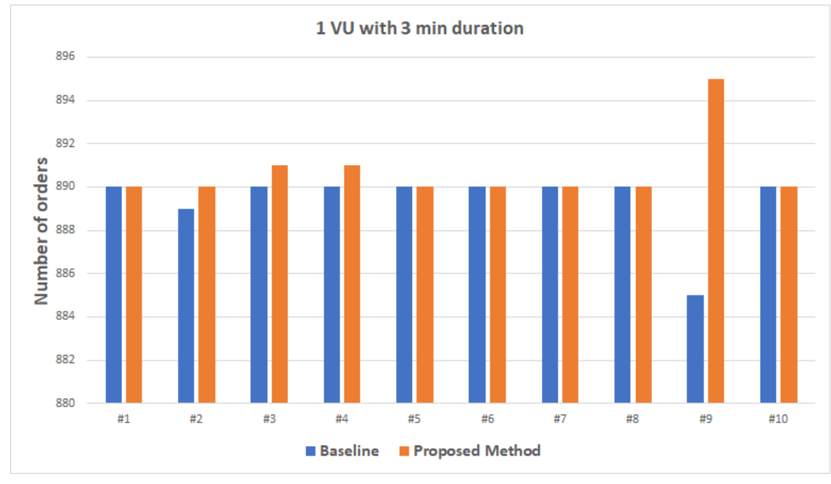


Figure 19. Load Testing—1 virtual user with 3 min duration.

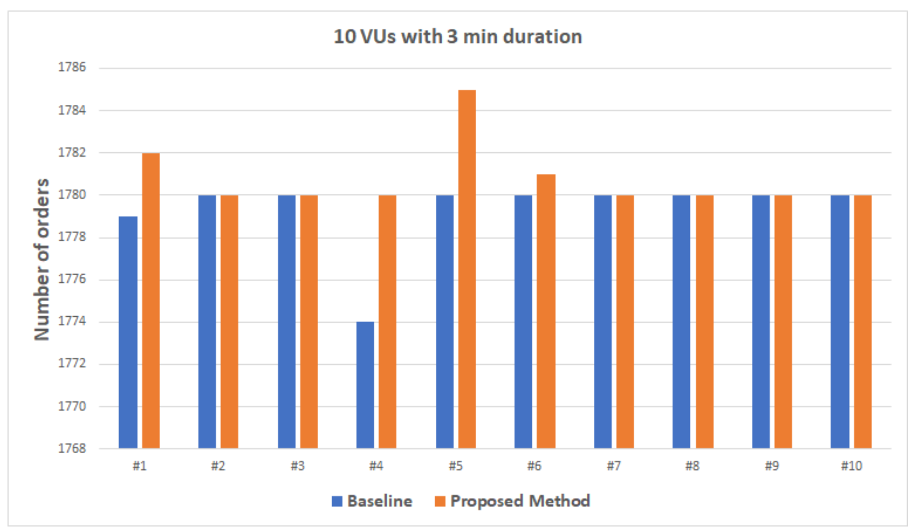


Figure 20. Load Testing—10 virtual users with 3 min duration.

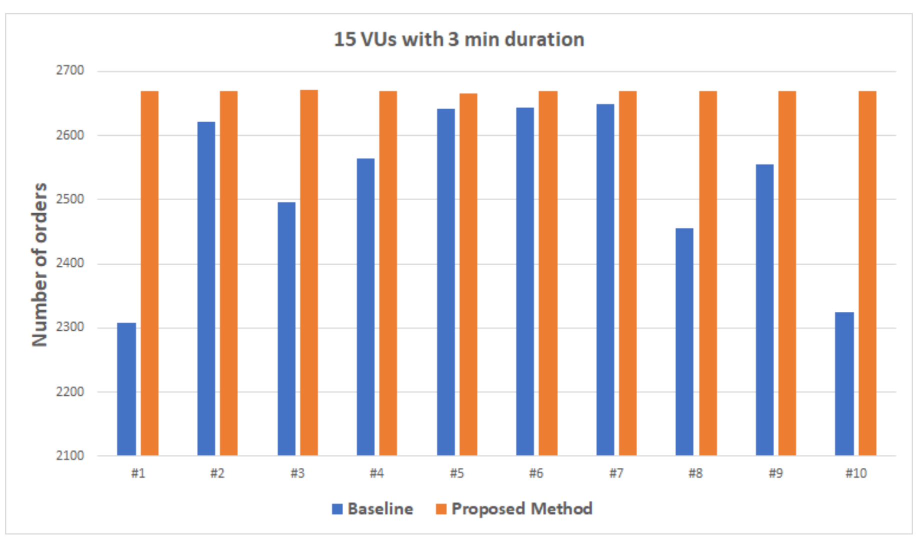


Figure 21. Load Testing—15 virtual users with 3 min duration.

50 VUs with 3 min duration

Number of orders

7000
6000
5000
4000
3000
2000
1000
0

1 2 3 4 5 6 7 8 9 10

Baseline Proposed Method

Figure 22. Load Testing—50 virtual users with 3 min duration.

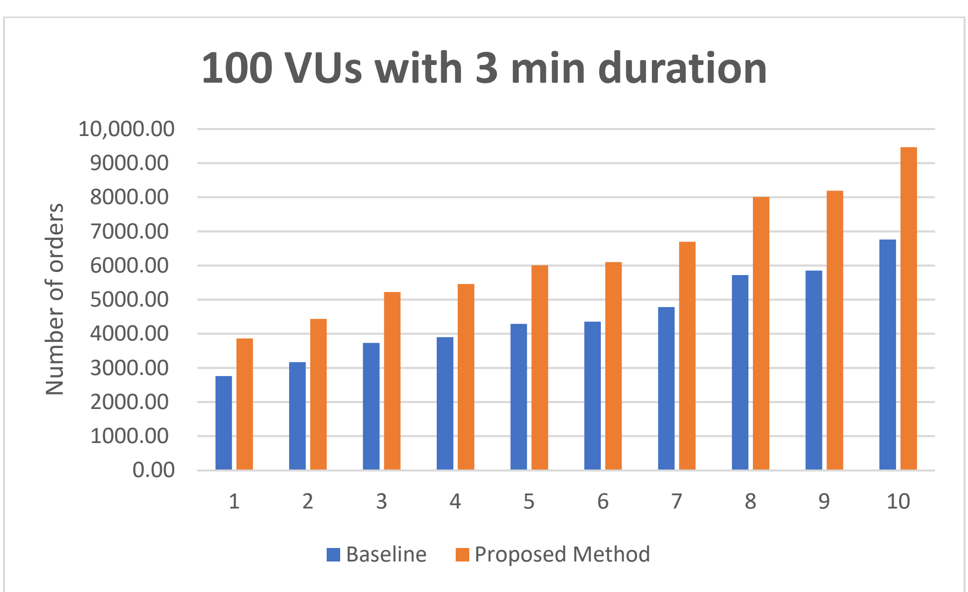


Figure 23. Load Testing—100 virtual users with 3 min duration.

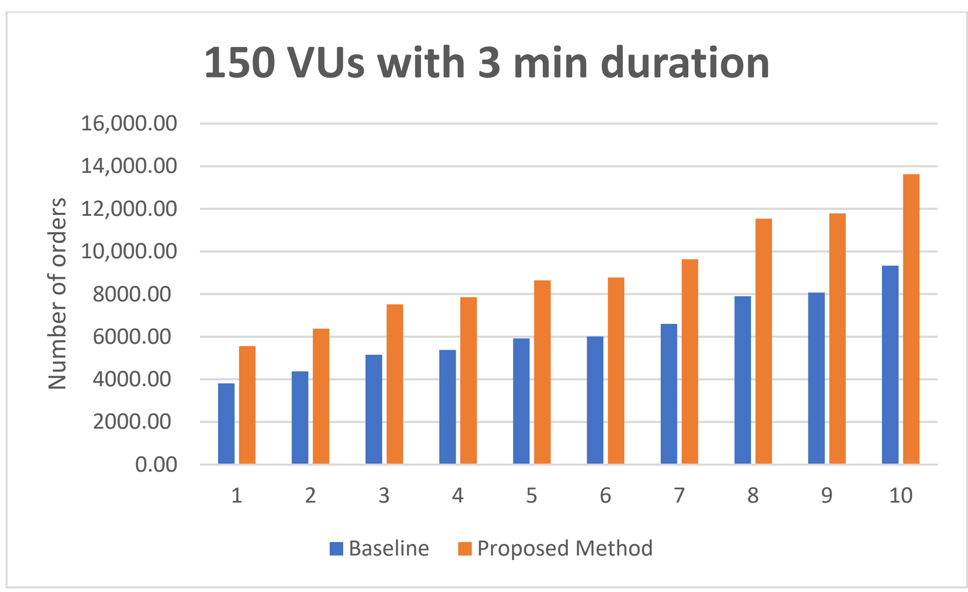


Figure 24. Load Testing—150 virtual users with 3 min duration.

Results show that both approaches are well performed with 1 virtual user and 10 virtual users. However, there is a significant difference when 15 virtual users were configured, as the mean value of the completed orders in version 1 is 2526.1 in 3 min, while version 2 completes on average 2669.5 orders in 3 min. Results show that the system that employs the proposed approach achieves better throughput. Note that, there were no failures that occurred with the two experiments performed.

The main factor behind achieving better throughput by the enhanced saga pattern is reducing the hard-disk operations. The proposed approach decreases access to the main database by utilizing the quota cache. To do that, the quota cache, which is implemented by Redis, is applied to the microservices that require validation (i.e., in the workflow above the Warehouse-Service and the Billing-Service). All CRUD operations are shifted to the memory cache server instead of the main database. Accessing the main database occurs only to perform database commit when all transactions are completed successfully.

Additionally, the proposed approach applies Redis as an in-memory cache server which plays a significant role in enhancing the performance. Redis is extremely fast as it preserves data in the primary memory (in-memory) instead of the secondary memory. Thus, data access latency is decreased. Moreover, via Redis, the read and write operations are extremely fast which in turn allows the system to deliver sub-millisecond response times that enables more requests to be handled in one second. As a result, increasing the throughput of the system. In contrast, the standard saga pattern stores data in the secondary memory; as a result, read and write operations will be much slower. Thus, it affects the throughput of the system.

Moreover, the message queue middleware, implemented by Kafka, plays a key role in improving the overall throughput. In our proposal, Kafka allows the communication between the microservices by supporting the sent and received messages between them. It also handles the failure and the completion events. Kafka enhances the performance as it is capable of handling thousands of messages per second. The amount of time it takes for a record that is produced to Kafka to be fetched by the consumer is short. Thus, more records will be fetched and the throughput will be improved.

2. Response-time "Consumed time in a scenario with failure events"

Three scenarios were tested with different error rates. Each scenario contains two kinds of error: The Warehouse-Service occurs error when goods are fetched, the Billing-Service causes an error when processing the payment. Moreover, the proportion of the Warehouse-Service error to the Billing-Service error is 9:1. Besides, 1 k requests were sent without any sleep time to the message queue middleware. Test results are listed below. Figure 25 shows the consumed time when

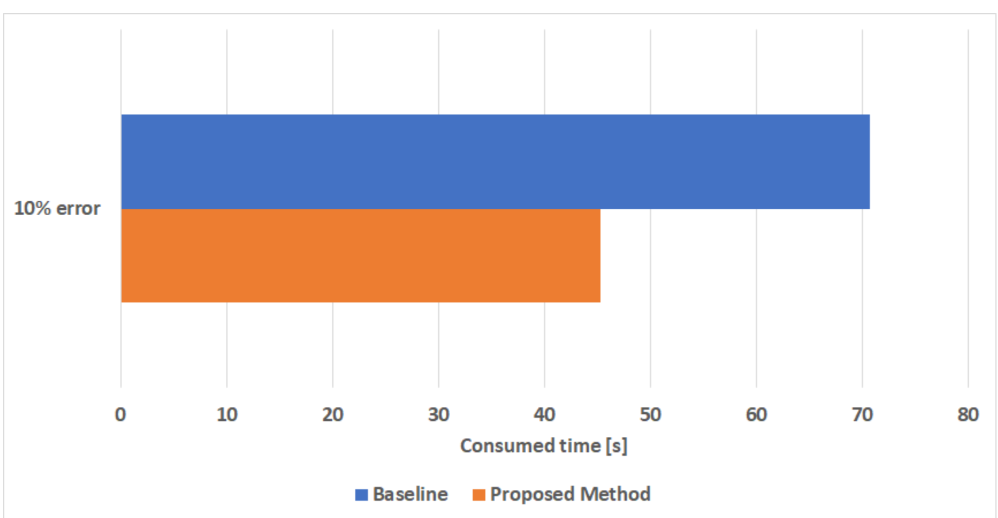


Figure 25. Load Testing—with 10% error.

- 10% error rate in 1000 requests (900 successful orders);
- 9% Warehouse-Service fetch goods occur error;
- 1% Billing-Service payment failed.

Figure 26 shows the consumed time when

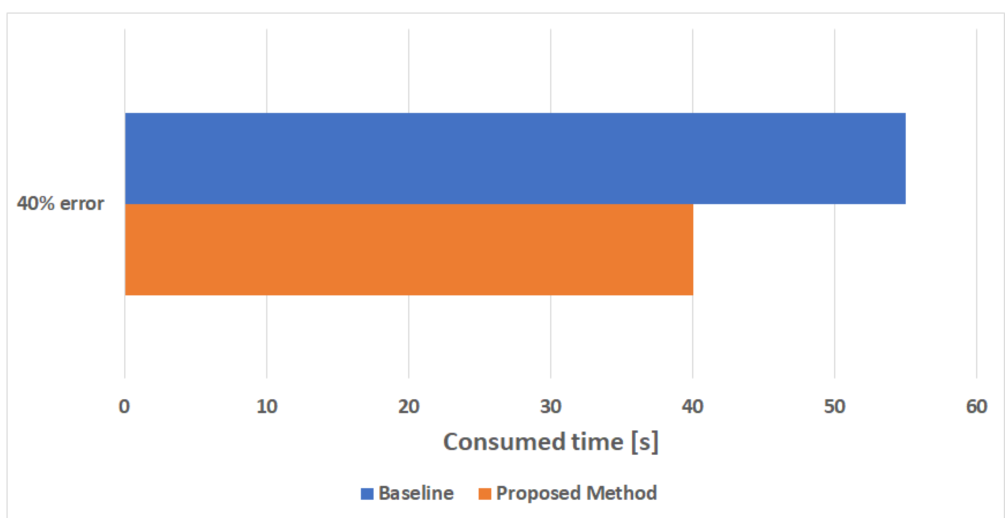


Figure 26. Load Testing—with 40% error.

- 40% error rate in 1000 requests (600 successful orders);
- 36% Warehouse-Service fetch goods occur error;
- 4% Billing-Service payment failed.

Figure 27 shows the consumed time when

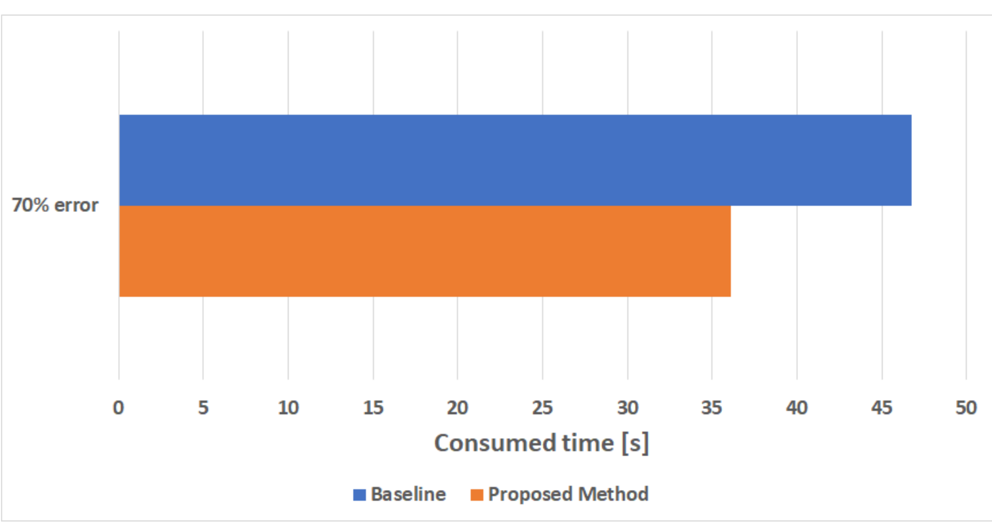


Figure 27. Load Testing—with 70% error.

- 70% error rate in 1000 requests (300 successful orders);
- 63% Warehouse-Service fetch goods occur error;
- 17% Billing-Service payment failed.

Figures 25–27 show that version 2 which applies the proposed enhanced saga pattern consumes less time than version 1 which utilizes the baseline standard saga pattern even in the scenario with errors. The reason behind this is employing the cache operations instead of the database operation generally enhances the performance and reduces the latency time. According to the proposal, CRUD operations are moved to the memory cache server to enhance the performance instead of the main database, and a commit to the main database occurs only when all requests are successful. In the case of error, no hard-disk operations are performed, there is only a need to revert the change in the memory cache. In the case where an error occurs, the compensation request will be sent to the corresponding message queue middleware.

In the standard saga pattern, updating data will be accomplished on the main database directly. In the case of an error, the compensation transaction will be run to rollback the changes, and the new update requires accessing the main database one more time. In contrast, with the enhanced version, the CRUD operations are moved to the cache level instead of the main database level. This will not cause a wrong database commit or update. The compensation request will be sent to the corresponding message queue middleware in the case of an error. As a result, with the enhanced saga pattern, database commit will only be performed when the transaction is completely successful.

Results demonstrated that transferring the main database's CRUD operations to the memory cache server would provide benefits and resolve the read isolation issue, further achieving better performance.

7. Conclusions and Future Work

This paper proposes an improved approach to resolve the missing read-isolation property in the saga pattern. The proposal integrates the standard saga with the quota cache and the eventual commit sync service. The importance of this research comes from the ability of the proposed approach in handling the lack of read-isolation of the saga pattern by moving some transactions from the database layer to the memory layer. According to the proposal, CRUD (create, read, update, and delete) tasks will be handled via quota cache (i.e., the memory cache server) instead of the main database. This will never cause a wrong commit to the main database. If a microservice fails to be completed, the other microservices will run compensation transactions to rollback the changes which affect only the cache level. Database commit will be postponed and handled via the message queue middleware at the end of the workflow when all transactions are completed successfully to achieve eventual consistency.

For demonstration, a lightweight microservices-based e-commerce system was implemented to compare the standard baseline version of the saga pattern with the proposed enhanced version. Several experiments were conducted for validation and evaluation. Results demonstrate that the proposed approach has the capability of resolving the lack of read-isolation in the saga pattern. Results also indicate that the proposed approach achieves better performance than the standard baseline version not only in typical cases but also in the scenario that needs to handle exceptions as employing the cache operations instead of the database operation enhances the performance and reduces the latency time.

Supplementary Materials: The following supporting information can be downloaded at: https://github.com/LittlePaulHi/enhanced-saga-pattern-demo.

Author Contributions: Conceptualization, C.-P.Z. and E.D.; methodology, C.-P.Z.; formal analysis, C.-P.Z.; writing—original draft preparation, E.D. and C.-P.Z.; writing—review and editing, E.D. and S.-M.Y.; supervision, S.-M.Y.; and project administration, S.-M.Y. All authors have read and agreed to the published version of the manuscript.

Funding: This research was partially supported by the Ministry of Science and Technology of Taiwan under grant number 108-2511-H-009-009-MY3.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Acknowledgments: We thank Palestine Technical University—Kadoori and National Yang Ming Chiao Tung University for their support.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Bucchiarone, A.; Dragoni, N.; Dustdar, S.; Lago, P.; Mazzara, M.; Rivera, V.; Sadovykh, A. *Microservices: Science and Engineering*; Springer International Publishing: Cham, Switzerland, 2019; ISBN 978-3-030-31646-4.
2. Hasselbring, W.; Steinacker, G. Microservice Architectures for Scalability, Agility and Reliability in E-Commerce. In Proceedings of the 2017 IEEE International Conference on Software Architecture Workshops (ICSAW), Paris, France, 21–25 May 2017; IEEE: Gothenburg, Sweden, 2017; pp. 243–246.
3. Štefanko, M.; Chaloupka, O.; Rossi, B. The Saga Pattern in a Reactive Microservices Environment. In Proceedings of the 14th International Conference on Software Technologies, Prague, Czech Republic, 26–28 July 2019; SCITEPRESS-Science and Technology Publications: Prague, Czech Republic, 2019; pp. 483–490.
4. Ahluwalia, K.S.; Jain, A. High availability design patterns. In Proceedings of the 2006 conference on Pattern Languages of Programs—PLoP '06, Portland, OR, USA, 21–23 October 2006; ACM Press: Portland, OR, USA, 2006; p. 1.
5. Richardson, C. *Microservices Patterns: With Examples in Java*; Manning Publications: Shelter Island, NY, USA, 2019; ISBN 978-1-61729-454-9.
6. Messina, A.; Rizzo, R.; Storniolo, P.; Tripiciano, M.; Urso, A. The Database-is-the-Service Pattern for Microservice Architectures. In *Information Technology in Bio- and Medical Informatics*; Renda, M.E., Bursa, M., Holzinger, A., Khuri, S., Eds.; Lecture Notes in Computer Science; Springer International Publishing: Cham, Switzerland, 2016; Volume 9832, pp. 223–233, ISBN 978-3-319-43948-8.
7. Uyanik, H.; Ovatman, T. Enhancing Two Phase-Commit Protocol for Replicated State Machines. In Proceedings of the 2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), Västerås, Sweden, 11–13 March 2020; IEEE: Västerås, Sweden, 2020; pp. 118–121.
8. Mohan, C.; Lindsay, B.; Obermarck, R. Transaction management in the R* distributed database management system. *ACM Trans. Database Syst.* **1986**, *11*, 378–396. [CrossRef]
9. Thomson, A.; Diamond, T.; Weng, S.-C.; Ren, K.; Shao, P.; Abadi, D.J. Calvin: Fast distributed transactions for partitioned database systems. In Proceedings of the 2012 International Conference on Management of Data—SIGMOD '12, Scottsdale AL, USA, 20–24 May 2012; ACM Press: Scottsdale, AL, USA, 2012; p. 1.
10. Garcia-Molina, H.; Salem, K. Sagas. In Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data—SIGMOD '87, San Francisco, CA, USA, 27–29 May 1987; ACM Press: San Francisco, CA, USA, 1987; pp. 249–259.
11. Campbell, R.H.; Richards, P.G. SAGA: A system to automate the management of software production. In Proceedings of the May 4–7, 1981, National Computer Conference on—AFIPS '81, Chicago, IL, USA, 4–7 May 1981; ACM Press: Chicago, IL, USA, 1981; p. 231.
12. Luckow, A.; Lacinski, L.; Jha, S. SAGA BigJob: An Extensible and Interoperable Pilot-Job Abstraction for Distributed Applications and Systems. In Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, Melbourne, Australia, 17–20 May 2010; IEEE: Washington, DC, USA, 2010; pp. 135–144.
13. Limón, X.; Guerra-Hernández, A.; Sánchez-García, Á.J.; Arriaga, J.C.P. SagaMAS: A Software Framework for Distributed Transactions in the Microservice Architecture. In Proceedings of the 2018 6th International Conference in Software Engineering Research and Innovation (CONISOFT), San Luis Potosí, Mexico, 24–26 October 2018; pp. 50–58.
14. Petrasch, R. Model-based engineering for microservice architectures using Enterprise Integration Patterns for inter-service communication. In Proceedings of the 2017 14th International Joint Conference on Computer Science and Software Engineering (JCSSE), Nakhon Si Thammarat, Thailand, 12–14 July 2017; IEEE: Nakhon Si Thammarat, Thailand, 2017; pp. 1–4.
15. Shopping Process. Available online: https://www.books.com.tw/web/sys_qalist/qa_1_2/0?loc=000_002 (accessed on 10 May 2022).
16. Martin, R.C.; Martin, R.C. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*; Robert C. Martin series; Prentice Hall: London, UK, 2018; ISBN 978-0-13-449416-6.
17. Walls, C. *Spring Boot in Action*; Manning Publications: Shelter Island, NY, USA, 2016; ISBN 978-1-61729-254-5.
18. Mass, M. *REST API Design Rulebook*; O'Reilly: Sebastopol, CA, USA, 2012; ISBN 978-1-4493-1790-4.
19. Rodríguez, C.; Báez, M.; Daniel, F.; Casati, F.; Trabucco, J.C.; Canali, L.; Percannella, G. REST APIs: A Large-Scale Analysis of Compliance with Principles and Best Practices. In Proceedings of the ICWE, Lugano, Switzerland, 6–9 June 2016.
20. Kreps, J. *Kafka: A Distributed Messaging System for Log Processing*; ACM Pres: Athens, Greece, 2011.
21. Narkhede, N.; Shapira, G.; Palino, T. *Kafka: The Definitive Guide: Real-Time Data and Stream Processing at Scale*, 1st ed.; O'Reilly Media: Sebastopol, CA, USA, 2017; ISBN 978-1-4919-3616-0.
22. Redis. Available online: https://redis.io/ (accessed on 11 April 2022).
23. Nurkiewicz, T.; Christensen, B. *Reactive Programming with RxJava: Creating Asynchronous, Event-Based Applications*, 1st ed.; O'Reilly Media, Inc.: Sebastopol, CA, USA, 2016; ISBN 978-1-4919-3165-3.
24. Douglas, K.; Douglas, S. *PostgreSQL: A Comprehensive Guide to Building, Programming, and Administering PostgresSQL Databases*, 1st ed.; Developer's Library; Sams: Indianapolis, IN, USA, 2003; ISBN 978-0-7357-1257-7.
25. PostgreSQL on Linux. In *DBAs Guide to Databases Under Linux*; Elsevier: Frisco, CO, USA, 2000; pp. 359–418, ISBN 978-1-928994-04-6.
26. Bailis, P.; Ghodsi, A. Eventual consistency today: Limitations, extensions, and beyond. *Commun. ACM* **2013**, *56*, 55–63. [CrossRef]
27. Logstash-Logback-Encoder. Available online: https://portal.bu.edu.sa/en/web/science-college/-17 (accessed on 11 April 2022).
28. Loki. Available online: https://github.com/grafana/loki (accessed on 11 April 2022).
29. Grafana. Available online: https://github.com/grafana/grafana (accessed on 11 April 2022).
30. Promtail. Available online: https://grafana.com/docs/loki/v2.1.0/clients/promtail/ (accessed on 11 April 2022).
31. K6. Available online: https://github.com/grafana/k6 (accessed on 11 April 2022).