ICIC Express Letters
Volume **16**, Number **5**, May **2022**

ICIC International ©2022 ISSN 1881-803X
pp. 545–553

EVENT-DRIVEN APPROACH IN MICROSERVICES ARCHITECTURE FOR FLIGHT BOOKING SIMULATION

Enrico Daniswara Giovanni and Ida Bagus Kerthyayana Manuaba*

Computer Science Department
Faculty of Computing and Media
Bina Nusantara University
Jl. K. H. Syahdan No. 9, Kemanggisan, Palmerah, Jakarta 11480, Indonesia
enrico.giovanni@binus.ac.id; *Corresponding author: bagus.manuaba@binus.ac.id

Received May 2021; accepted August 2021

Abstract. *Technological advancement affects several business and industry aspects and can help to increase their revenue and customer satisfaction. There are several approaches taken by software developers to solve business problems. By knowing the benefits and drawbacks and choosing the right approach, these advancements will improve overall outcome. This paper discusses an event-driven approach that could be used by businesses that have high transactions and a need for high reliability and availability for their application. In order to establish the benefits of an event-driven approach, this paper describes tests and comparisons of REST API-driven and event-driven approaches. The testing methods used stress and load tests. The results from the tests showed that an event-driven application performed better in terms of throughput, resource consumption, and average response time compared to an REST API-driven application even in higher loads.*
Keywords: Microservices, Event-driven architecture, Command query responsibility segregation (CQRS), REST API-driven

1. Introduction.

Recently, exploring product information and purchasing products online has become a habit in our lives. Based on research done by Lee and Lin [1], they found that service quality, user satisfaction and trust can be accomplished by having excellent website design, and a reliable and responsive services. For example, one of the industries that has made a huge impact in the growth of economic improvement of Indonesia is tourism. In 2017, Indonesia attained revenue of around US$ 15.2 billion with approximately 14 million foreign tourists [2].

Currently, technological developments are impacting the tourism industry by increasing the ease of use for guests in bookings. Online travel agency (OTA) is one of the most advanced products that are widely available and popular on the market today [3]. OTAs usually offer a website or mobile application to fulfill their business processes. There are two main components of a website and a mobile application, which are the user interface and the application programming interface (API).

The API is a current technology in providing reliable and responsive services in software application. APIs can be built under several software architectures such as monolithic, service-oriented architectures, serverless, and microservices [4]. The difference between these architectures is in the way they communicate and the maintainability. A number of industries are not aware about choosing the right software architecture for their website or mobile application which can impact output in terms of performance, availability, cost and reliability [5].

DOI: 10.24507/icicel.16.05.545

In order to meet service quality, user satisfaction and trust to the website or mobile application, a flawless experience through reliable and responsive services must be provided by the business owner to the user. In today's technology, the reliable and responsive services can be developed by using microservices architecture that divides the application into a smaller set of services [6, 7].

However, having multiple services that could read and write into one database only, can cause performance issues. CQRS (command query responsibility segregation) is required to address this problem by separating the way of recording and retrieving the data. CQRS is not an architectural pattern, but a code pattern instead. It separates the responsibilities of writing and reading data into conceptual and physical storage [8].

In microservices, each service could have its own database. However, because the database per service pattern has been applied, business transactions are separated into different services which may impact the data to be inconsistent. Hence, a communication mechanism is needed to ensure consistent data across multiple services, because applications cannot only use local ACID transactions (atomicity, consistency, isolation, durability). Therefore, the problem is how to maintain reliability of the communication between services with high availability, high throughput, low resource costs, and low response times.

One way to communicate between services in microservices architecture is implementing an event-based approach [9]. An event-based approach uses events to promote and move data between those loosely coupled sets of services. Adopting an event-driven approach can also benefit by providing better response times, increasing flexibility and agility, and increasing operational efficiency and resilience.

This paper provides an example of a flight booking simulation scenario for an online reservation system using microservices architecture. In maintaining the reliability of the communication between services, this paper will focus on implementing an event-driven architecture in microservices and demonstrating how event sources and CQRS lead to better performance.

Since the API service in a flight booking simulation scenario can be integrated with OTA by standard RESTful API [10, 11] conventions via the HTTP method, this paper also compares the performance between REST API-driven and event-driven based in microservices architecture. Hence, the comparison will focus on the scalability of microservice with and without an event-based approach.

This paper aims to demonstrate a reliable reservation service that is scalable, easy to maintain, and follows the latest technology trends by implementing a microservice architecture and making a comparison between REST API-driven and event-driven based approaches. Hence, by adopting the right software architecture, it could achieve a stable service with low latency and high throughput at low resource costs. In addition, it could also increase user appreciation, improve overall results, and increase revenue for OTAs and airlines industry.

The following sections discuss more details about problem analysis, solution design, implementation and testing, and also analysis with discussion, and being closed with conclusion and possible future work.

2. **Problem Analysis.** Before microservice architecture became popular, monolithic architecture was the main pattern that was used for developing cloud based applications [7]. In a traditional monolithic approach, the application is only placed in a single directory hierarchy that bundles the presentation, application logic, business logic, data access object at the same level.

Adopting new technologies will be very difficult because all modules need to be adjusted. In terms of deployment, it will be very tough to apply continuous deployment, since the entire application needs to be redeployed which will interrupt background tasks and may cause problems. Besides, scaling the application will be expensive because it needs to extend the entire application and cannot scale each component independently [5].

To avoid those problems in the development process, a microservice architecture needs to be applied. It will be easier to maintain and modify the code because each functionality is separated into a set of services. Introducing new technologies will also be easier and it does not affect other services (loosely coupled). Deployment and scaling can also be done independently based on the resource usage or server load. Furthermore, big companies such as Netflix, Amazon, and eBay have migrated from monolithic to microservices architecture [12].

Furthermore, there are factors that need to be taken as consideration in providing a reliable and responsive services, such as high availability, low response time, security, and reliability (from user perspective); and also continuous deployment, easy maintenance, scalability, and low cost (from website business owner perspective).

High availability can be achieved by separating each domain model into a set of services. Thus, microservice architecture must be used to solve this issue. Low response time can be achieved by implementing event-driven architecture because it makes the service communicate asynchronously. Reliability should be achieved by having no downtime at any given time and can be accessed from people around the world without having the need to worry about the scale of the requests. The application must also be easy to maintain because as the business goes bigger, there will be more complexity to the system. Lastly resource usage must be as low as possible to reduce the financial cost needed for the application.

Hence, this paper is discussing the implementation of microservice architecture combining with event-driven approach and demonstrating how event sources and CQRS lead to better performance. It means the communication between each service will be asynchronous and does not block the user's request.

In order to demonstrate how well event-driven approach implemented in microservices architecture, this paper will also describe testing and comparisons of REST API-driven and event-driven approaches.

3. Solution Design.

3.1. **System architecture.** The system architecture for this research is shown in Figure 1. Each request from the user will be directly loaded in balanced way into GraphQL [13], which is responsible for authorizing the request and forwarding it to another load balancer to the API gateway.

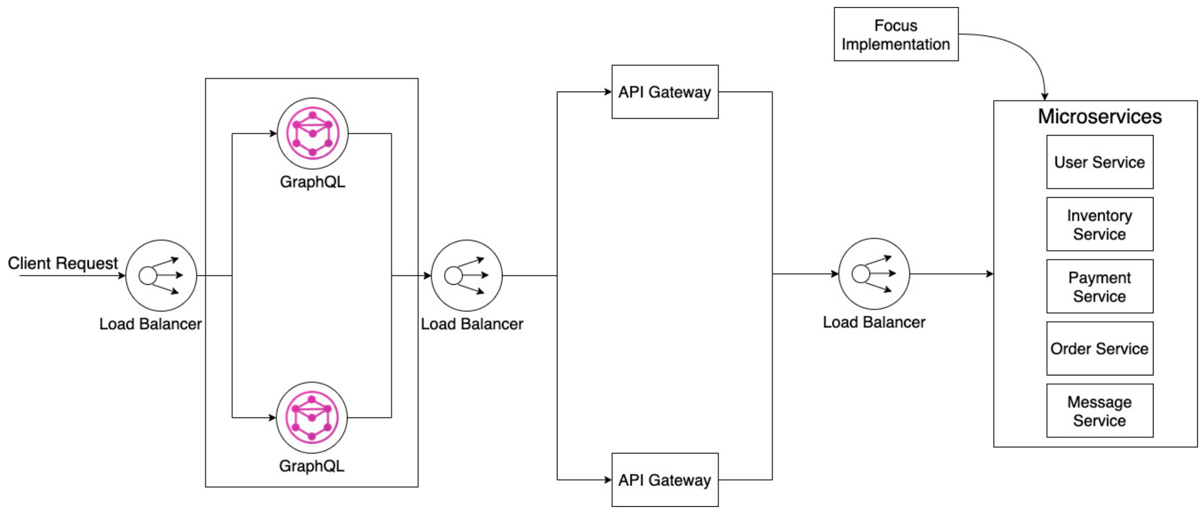

FIGURE 1. Microservices in system architecture diagram for flight booking simulation

In this architecture, there is only one instance for each service but in reality, it will have multiple instances for each service and that is where load balancers have the function to choose which instance should receive the request.

3.2. **Services flow diagram.** Every service in the whole system will use the Quarkus framework [14] and have the same software architecture as seen in Figure 2.

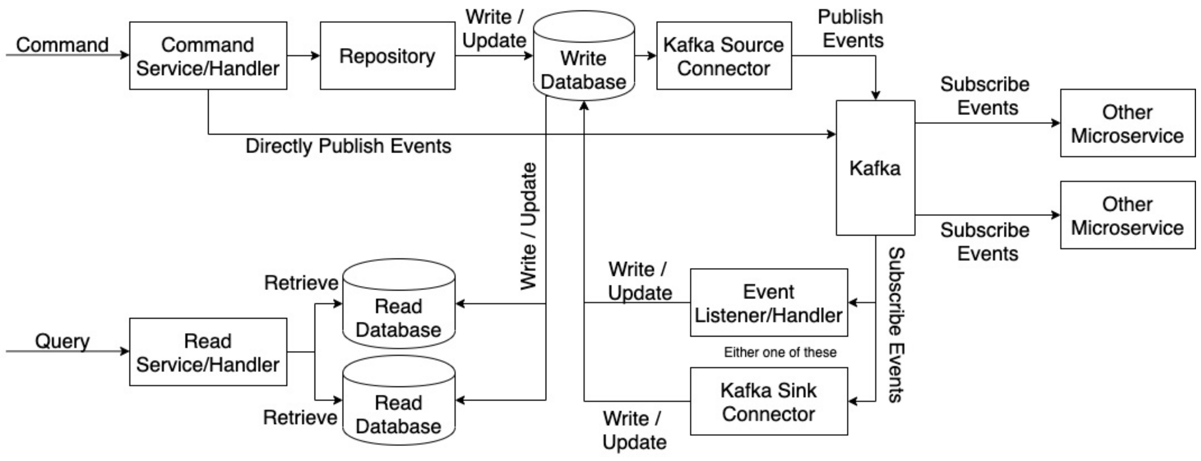

FIGURE 2. Design services flow diagram for flight booking simulation

The process of creating/updating and retrieving data is organized separately throughout the service. When a service receives a write request which is indicated by a command, it will be validated by the handler and directly written to the write (primary) database or an event will be published to Kafka [15].

Each service may consume events from a Kafka topic if required by having an event listener or by configuring a Kafka sink connector (directly write to the write database).

Database per-service pattern is also implemented, in this case MongoDB's replica set feature will be used where each database will have a write (primary) and a read (secondary) database. In regard to exchanging data information throughout services, they will communicate through Kafka by publishing or consuming events. Figure 3 shows the detail design of microservices components.

3.3. **Activity diagram.** Figure 4 shows the main business process of the application. It starts when a user requests for available inventories. The application will retrieve all available options and respond with none if there are no available inventories. The user will select an inventory and complete the booking form. The system will validate if the selected inventory is still available; if it is not, the user will get a failure response and if is still available, the user will be prompted to finish payment. After the payment is conducted by the user, the application will send a request to a third party application as its payment gateway. If the payment is confirmed by the payment gateway, user will be notified by email with the issued ticket.

4. **Implementation and Testing.**

4.1. **System requirements.** The application can run on any platform that has docker running since it is containerized into docker images. Then, it can be accessed using any API client or testing tools. There is also a built-in API documentation and testing tools using Swagger UI. In order to monitor Kafka cluster status, lists of messages, and connectors, user interfaces are provided using external libraries. We used to develop and run the application with the following hardware specifications (1st Web Server). Meanwhile,

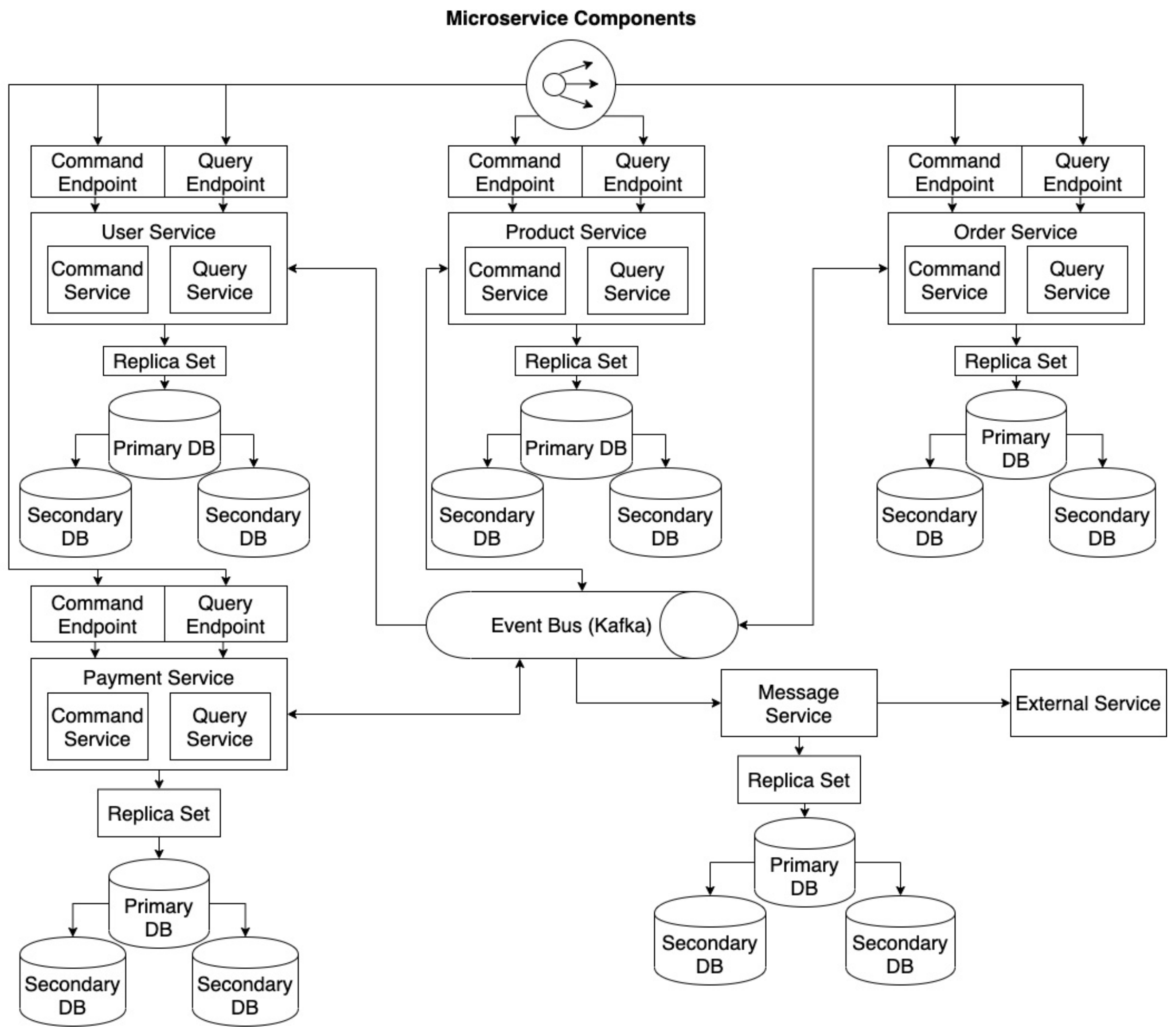

Figure 3. Microservices components design

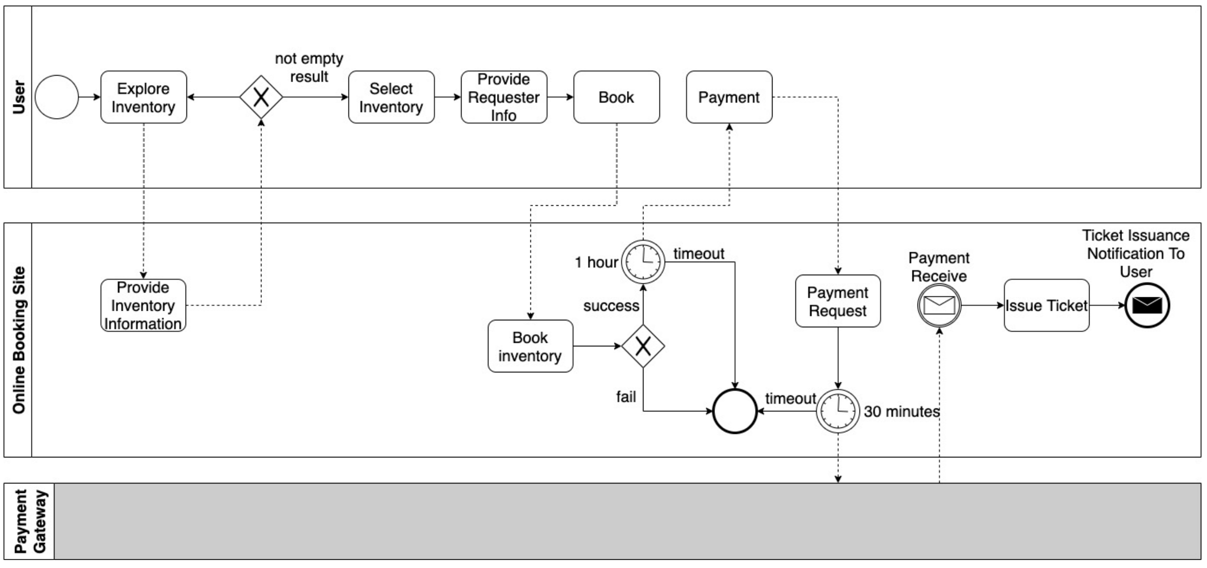

Figure 4. Order activity diagram

Table 1. Web server machine specifications

Specifications	1st Web Server	2nd Web Server
Device	MacBook Ppro (13-inch, 2017)	–
Operating system	macOS Catalina version 10.15	Windows 10 Version 10.0.18362.836
Processor	3,1 GHz Dual-Core Intel Core i5	3,8 GHz Quad-Core Intel Core i5 7600K
RAM	8 GB 2133 MHz LPDDR3	16 GB 3200 MHz DDR4
SSD	256 GB	256 GB
GPU	Intel Iris Plus Graphis 650 1536 MB	Nvidia GTX 1070 8088 MB

because of the hardware limitations of the web server the load, stress tests, and monitoring are done using another machine (2nd Web Server) with the following hardware specifications (see Table 1).

4.2. **Stress and load testing.** Load and stress testing were conducted to compare the performance of an event-driven application with the API-driven application. The scalability test was not listed here because of the time limitation in finishing this report. The stress and load test consisted of threads and one ramp-up period, where a thread defined a user and it can be concluded that if it had 10 threads and 1 second ramp-up period, each thread would start 0.1 second after the previous thread began. Table 2 shows the stress and load test properties.

Table 2. Stress and load test properties

Properties	**Value**
Taste case	Find inventory and place order
Number of threads (users)	Test 1: 10 - Test 2: 25 - Test 3: 50
Ramp-up period (seconds)	1
Loop count	100
Predefined data	
Name	**Data count**
Cancellation policy	100
Inventory	130,000
Payment methods	3
Payment transaction	110,000
Order	110,000
User	50,000
Message log	110,000
Application properties	
Name	**Value**
MongoDB min pool size	10
MongoDB max pool size	50
MongoDB max wait queue size	50
MongoDB connect timeout	10 s

The stress and load testing are done using Apache JMeter [16] which runs on another machine that also monitors the resource usages on each docker container. It is visualized in Grafana [17] dashboard by reading Prometheus' data that acts as the data source which examines exposed metrics by cAdvisor [18], and Node Exporter.

In the test case the most crucial part on an online reservation system is the order. The inventory is chosen randomly by JMeter's feature by selecting one inventory id from the API response. The place order request will also be randomly generated by the authors' JMeter script. The tests will have a different number of threads (users) to simulate how a real online reservation system is used. It increases gradually to show the results based on how the services will handle requests from users.

4.3. **Comparison test result.** In order to compare the performance of API-driven and event-driven, the authors created two exactly same applications that shared the same code base with different approaches in terms of service communication.

Based on the result in Table 3, there are three types of tests done to measure the performance of each application. Both had the same predefined data, application properties and test properties as listed earlier. Each service in both applications had the same JVM maximum memory allocation of 256 MB.

Table 3. Result of three different tests

Measurements	**Test 1**		**Test 2**		**Test 3**	
	Event-driven	**API-driven**	**Event-driven**	**API-driven**	**Event-driven**	**API-driven**
Samples	1000	1000	2500	2500	5000	5000
Average (ms)	199	317	481	644	991	647
Min (ms)	12	34	8	30	19	15
Max (ms)	9230	5251	7821	9990	10019	28658
Error (%)	0	0	0	0	0	30
Throughput	41.7/sec	29.3/sec	47.0/sec	37.9/sec	47.4/sec	62.8/sec
Average CPU/ Memory	13.24% 1022.32 MB	19.19% 958.11 MB	14.78% 967.41 MB	28.47% 1042.14 MB	19.9% 928.13 MB	12.99% 1078.97 MB

5. **Analysis and Discussion.** The predefined data was provided to simulate a real use case of the application which would have thousands of inventories and transactions. The authors found that in the idle condition most of the event-driven application services consumed more CPU usage. It was because at period one the services thread needed to reassure Kafka that they are still active and facilitates rebalancing when a new consumer joined or left the group (known as a heartbeat).

The first test consisted of ten threads and one second ramp-up period, where it could be concluded that each thread started 0.1 second after the previous thread began. The result was that the event-driven application produced higher throughput by approximately 30% compared with the API-driven application. On average, based on the JMeter results, it also produced lower response times. Also, event-driven application services mostly consumed less CPU and memory usage.

The same case also happened in the second test, although in higher load, the event-driven application produced higher throughput by almost 30% compared to the API-driven application. It also consumed less CPU and memory usage. However, the result shown in the third test is notably different between both applications. The event-driven application successfully processed all requests with 0% of errors and almost identical throughput as the second test; meanwhile, there were 30% errors and higher throughput in the API-driven application. Besides, because of the errors, the API-driven application consumed fewer resources compared with the event-driven application.

On the other hand, based on the third test, the throughput of API-driven application was significantly higher because of the error rate where it directly responded when the services were accessing MongoDB. After tracing the application logs, the product service was found throwing an exception caused by MongoDB max wait queue size was exceeded resulting in no data retrieval and data loss. This was also why in the last test the API-driven application consumed less CPU and memory compared to the event-driven application.

Based on the tests done, the problem stated in Problem Analysis section is resolved by having an event-driven approach. The event-driven application created greater output in terms of availability, as there were no processing errors throughout the tests, compared with the API-driven application which produced errors in higher loads. It also produced higher throughput and lower response timed as was seen from stress and load testing results.

Furthermore, the resource consumption on services in event-driven application mostly costs less compared with the API-driven application. Thus, this also answers the aims listed on the Introduction section where low latency and high throughput services can be created using the event-driven approach. No processing errors in any services meaning ACID transactions were guaranteed. Even though errors may occur, it can be directly resumed using Kafka's resume offset feature.

6. Conclusion and Future Work.

6.1. Conclusion.

This paper describes an implementation of microservices architecture using event-driven approach. Event-driven is a popular approach with the benefits of having loosely coupled services, asynchronous (non-blocking resources), services that can be scaled independently, replaying events, and high performance.

To show the event-driven approach in producing better outcomes, the authors compared it with REST API-driven approach. This comparison was simulated by having two applications with different approaches that shared the same code base. The full architecture design can be found in Sub-section 3.3 where we combined an event-driven approach with CQRS in order to produce better output.

To compare the performance of the event-driven and API-driven approaches, three stress and load tests were done with one scenario that had different testing properties. Based on the test result, the event-driven application produced better performance in terms of throughput, availability, reliability, latency, and resource consumption. Even in high loads, the event-driven application managed to handle all requests without any application errors.

To sum up, this project established that an event-driven approach combined with CQRS can produce better performance by approximately 30% by having higher availability, higher throughput, lower response time, better performance and easily scalable with less resource usage.

6.2. Future work.

There are several ways to improve the application. The following is a list of items that can be improved:

- Resume/replay events. The current application does not support resume or replay events automatically; instead it requires the user to resume or replay events manually from the command line interface.
- Providing real data set to benchmark the applications. To improve the accuracy of stress and load testing, a real data set can be provided.

REFERENCES

[1] G. G. Lee and H. F. Lin, Customer perceptions of e-service quality in online shopping, *International Journal of Retail & Distribution Management*, vol.33, no.2, pp.161-176, 2005.

[2] J. Sihite and A. Nugroho, H_∞ exploring the Indonesian tourism destination via Indonesia.Travel @indtravel, *Proc. of the 2nd International Conference on Tourism, Gastronomy, and Tourist Destination (ICTGTD2018)*, pp.29-32, 2018.

[3] L. Hendriyati, The influence of online travel agents on room reservations at the Mutiara Malioboro hotel in Yogyakarta, *Media Wisata*, vol.17, no.1, pp.1-10, 2019.

[4] A. Balalaie, A. Heydarnoori and P. Jamshidi, Microservices architecture enables DevOps: Migration to a cloud-native architecture, *IEEE Software*, vol.33, no.3, pp.42-52, 2016.

[5] S. Sharma, *Mastering Microservices with Java: Build Enterprise Microservices with Spring Boot 2.0, Spring Cloud, and Angular*, Packt Publishing Ltd., 2019.

[6] Microservices.io, *Command Query Responsibility Segregation (CQRS)*, https://microservices.io/patterns/data/cqrs.html, Accessed on 5-03-2021.

[7] S. Newman, *Building Microservices: Designing Fine-Grained Systems*, O'Reilly Media, Inc., 2015.

[8] B. M. Michelson, Event-driven architecture overview, *Patricia Seybold Group*, vol.2, no.12, 2006.
[9] F. Marchioni and M. Little, *Hands-on Cloud-Native Applications with Java and Quarkus: Build High Performance, Kubernetes-Native Java Serverless Applications*, Packt Publishing Ltd., 2019.
[10] L. Richardson, M. Amundsen, M. Amundsen and S. Ruby, *RESTful Web APIs: Services for a Changing World*, O'Reilly Media, Inc., 2013.
[11] H. Subramanian and P. Raj, *Hands-on RESTful API Design Patterns and Best Practices: Design, Develop, and Deploy Highly Adaptable, Scalable, and Secure RESTful Web APIs*, Packt Publishing Ltd., 2019.
[12] Carlos, *Four Companies that Migrated from Monolith to Microservices*, https://www.kambu.pl/blog/companies-that-migrated-from-monolith-to-microservices/, Accessed on 18-05-2021.
[13] S. Buna, *Learning GraphQL and Relay*, Packt Publishing Ltd., 2016.
[14] T. Koleoso, Test quarkus applications, in *Beginning Quarkus Framework, Build Cloud-Native Enterprise Java Applications and Microservices*, Springer, 2020.
[15] M. Kumar and C. Singh, *Building Data Streaming Applications with Apache Kafka*, Packt Publishing Ltd., 2017.
[16] S. Matam and J. Jain, *Pro Apache JMeter: Web Application Performance Testing*, Apress, 2017.
[17] T. T. Hoang, M. T. Tao and P. H. Au, *Research and Implementation of Monitoring Systems Prometheus and Grafana*, Ph.D. Thesis, FPTU HCM, 2020.
[18] J. Makai, *New Solutions in IT Monitoring: cAdvisor and Collectd*, Tech. Report., CERN, 2015.