Journal of Theoretical and Applied Information Technology
30th September 2021. Vol.99. No 18

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

PERFORMANCE ANALYSIS OF CHOREOGRAPHY AND ORCHESTRATION IN MICROSERVICES ARCHITECTURE

[1]HANDI KRISTIANTO, [2]AMALIA ZAHRA

[1,2] Computer Science Department, BINUS Graduate Program, Master of Computer Science, Bina Nusantara University, Jakarta, Indonesia 11480

E-mail: [1]handi.kristianto@binus.ac.id, [2]amalia.zahra@binus.edu

ABSTRACT

Microservices architecture (MSA) is a system architecture design pattern with an approach that applies applications as a collection of small services. Service composition is combining various services together to provide the solution. There are two methods for the microservice composition i.e., Choreography and Orchestration. Both techniques have pros and cons based on the use case which is being implemented. In the past some researchers have suggested the use cases for which these approaches are suitable, but a quantitative analysis has not been performed thoroughly and did not evaluate the effects of large number of concurrent users and number of service instances on performance of MSA. We perform an extensive quantitative analysis to analyze and quantitatively measure the performance of Choreography and Orchestration in a Saga which consists of several services, with a maximum number of eight services, and to determine the criteria for selecting the Choreography or Orchestration method to be used based on the measured parameters, namely response time, CPU utilization, and throughput. The factors which impact the performance are the number of services, the number of concurrent users, and the number of service instances. Both models are simulated by varying these parameters. It can be concluded that Choreography has a better response time, throughput, and CPU utilization when compared to Orchestration.

Keywords: *Microservices, Database per Service, Saga Pattern. Choreography, Orchestration*

1. INTRODUCTION

Microservices architecture (MSA) is a system architecture design pattern with an approach that implements applications as a collection of small services. MSA has similarities with Service-Oriented Architecture (SOA) in that the services are independent and stand-alone. The difference with SOA is that MSA does not use Enterprise Service Buses (ESB) as middleware that distributes tasks to each connected application component, but MSA uses a lighter technology. Each service is a unit that can run autonomously and communicate with each other with a simple mechanism, usually through an Application Programming Interface (API). In MSA, communication between services is done using synchronous protocols such as HTTP/REST or asynchronous protocols such as Advanced Message Queuing Protocol (AMQP).

MSA is expected to be a solution to the problems faced by monolithic system architecture, namely in terms of horizontal scalability, high availability, modular, and agile infrastructure. Increasing the scalability and high availability of monolithic systems must be done by scaling-up the entire application so that it requires more infrastructure resources. Monolithic architecture requires more effort and human resources in designing a modular system. To build a monolithic system requires good infrastructure capacity planning at the beginning of the project because capacity addition cannot be done only for a part of the system but must be done for the whole system. By applying MSA, the deployment process of an application becomes more independent because each service can be built by a team with a relatively small number of members so that the team can focus more on developing services using appropriate technology. Each service can be developed using different programming languages so that it becomes more appropriate because each programming language has its own advantages and can be applied according to the needs of the service being built.

One of the advantages of using MSA in building an application is the nature of the service that is independent and isolated from other services so that each service can be upgraded without

affecting other services. The consequence of this independent nature is that each service must have its own separate database [1]. When a transaction involves several services, the changes that occur in one service's database cannot be recognized by the other service's database. Applications cannot use ACID (atomicity, consistency, isolation, durability) transactions locally because the database is distributed across each service. Likewise, when a failure occurs in one of the services involved in the transaction, the rollback process cannot be done using the Two-Phase Commit (2PC) method because the database is distributed. This distributed database concept is known as the Database per Service pattern [2]. To handle the Database per Service pattern, the Saga Pattern is used [3]. Saga is a sequential transaction that involves more than one service. In Saga, every service that has finished running its process publishes an event that can trigger the next service. When a failure occurs, the services involved in a Saga must be able to rollback by generating events in reverse order. According to [3] Saga can be designed with two methods, i.e., Choreography and Orchestration.

In Choreography every service involved in a Saga can trigger a service event directly, without going through a coordinator. Communication between services can run autonomously because each service knows when to start the process and which service should be triggered next. While in Orchestration, each event is determined and triggered by a central coordinator. When a service has finished the process, that service cannot directly trigger another service, but must publish an event to the coordinator and then the coordinator who will continue to the next service. In Orchestration, communication between services must go through a coordinator, so that the coordinator's performance greatly determines the overall system performance.

It is a big challenge to identify which composition approach is better. Some researchers already performed analysis on how microservices Choreography and Orchestration techniques used for implementing MSA [3], [4], [9], [10], [12]. Previous studies did not evaluate the effects of large number of concurrent users and number of service instances on performance of MSA. In this research we performed an extensive quantitative analysis on various considerable parameters like response time, throughput, CPU utilization and study the impact of multi-instances of service (with large number of concurrent users - up to 1,000 users) on the performance of each technique. Response time needs to be measured because the faster the response time, the resources can be immediately freed so that they can be allocated to handle other processes. The higher the CPU utilization, memory utilization, and power utilization, the higher the server specifications that need to be provided, thereby increasing infrastructure costs. Throughput is also an important parameter to measure because the higher the throughput, the better the performance of an application system.

In this study, the performance of Choreography and Orchestration is evaluated based on the number of services involved in a Saga so that it will be known how effective the two techniques are and how much influence the number of services has on the performance of Choreography and Orchestration. Furthermore, experiments are conducted with different number of concurrent users (the number of users who conduct transactions simultaneously) and different number of service instances. In this research we also experimented by using two instances for each service. By using more than one instance for each service, a load balancer is needed as a proxy and distributed the workload and traffic of each service. Both methods also be evaluated based on how much effort is required to change the order of processes in a Saga.

2. MICROSERVICES

Microservices is a system architecture with an approach that implements an application as a collection of services that run independently and communicate with each other using the HTTP API [7]. Each service is built independently and deployed automatically. This is what distinguishes microservices from applications built on a monolithic architecture. Enterprise applications usually consist of three main parts, namely the user interface (using HTML and javascript), the database as a data storage medium (Relational Database Management System - RDMS), and the server-side application that handles HTTP requests, executes business logic, saves to the database, and sends the HTML script to display in the browser. This server-side application is an example of a monolithic architecture because every time there is a change in the system, development and deployment of the entire server-side application must be carried out. Scalability is also one of the

obstacles in monolithic architecture because it must be applied to the application as a whole, while in microservices each service can be scaled-up and scaled-down independently without affecting other services.

Database per Service is a pattern that requires each service to have its own database that is separate from other database services and the database can be accessed directly only by the service itself. Other services cannot access the database of a service directly but must go through the API provided by the respective service. By implementing the Database per Service pattern, each service will be completely independent of each other so that if there is a database change in a service, it will not affect the other database services. In addition, each service will also have the freedom to use a database schema that is more in line with needs, for example, for services that handle text searches, they can use elasticsearch [5] and for services that handle fraud detection, they can use graph databases such as neo4j [5].

3. CHOREOGRAPHY AND ORCHESTRATION

Saga is a sequential transaction involving more than one service [7]. In Saga, every service that has finished running the process then publishes an event that can trigger the next service. Likewise, when a service fails, the services involved in a Saga must be able to rollback by generating events in the reverse order. According to [3] Saga can be designed using two methods, namely Choreography and Orchestration. In Choreography approach, each service can communicate directly using events without going through a coordinator [5]. A service that has finished running its process can publish an event that can trigger the next service. MSA that uses the Choreography method, each service will be more decouple [6]. A service that needs to interact with other services can subscribe to the event service.

In Choreography Saga, every service that has finished running the process then publishes an event that can trigger the next service. Likewise, when a failure occurs in one of the services, the service involved in a Saga must be able to roll back by generating events in the reverse order. Fig 1 shows a Saga using Choreography starting from the service Account. Following are the advantages of Choreography:

- Changes in the flow of communication between services are easier because it can be done by rewiring the input and output queues.
- Application system becomes more autonomous because it reduces dependence on other components.

The disadvantages of Choreography:

- For workflows that involve many services, communication between services becomes more complicated and difficult to manage.
- More difficult for software developers to understand because the process of a Saga is implemented for every service involved.

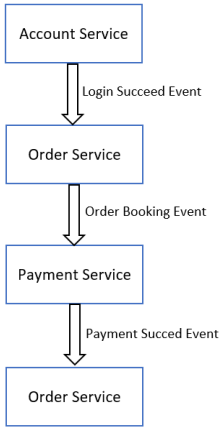

Figure 1. Choreography Flow

Service Account validates the logged-in user and then publishes an event that triggers the Service Order to place a booking order. After the order has been successfully booked, the Order service will trigger the Payment service and after the payment is successful, the Payment service will trigger the Order service to place an order.

The other technique to implement Saga is called Orchestration, in this approach each service cannot communicate directly but must go through a coordinator [5]. Each service that has finished running the process can coordinator publish an event to the coordinator and then based on the predetermined routing, then will trigger the next service. MSA which is built using the Orchestration method, several services will be connected and work together sequentially to complete a transaction [8]. Fig. 2 shows a Saga that uses the Orchestration method in communication between services. Following are the advantages of Orchestration:

- Saga implementation becomes simpler because each service works based on orders from the coordinator.
- Business processes become easier to manage because the logic is centralized in the coordinator.

Following are the disadvantages of Orchestration:

- Each service has a high dependence on the coordinator.
- Coordinator becomes a single point of failure.
- Too much business logic is implemented in the coordinator so that the service becomes less independent.

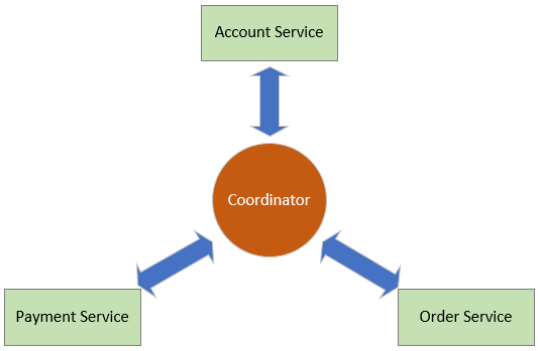

Figure 2. Orchestration Flow

4. RELATED WORK

Some researchers have conducted studies on the performance of microservices applications [9]-[12]. Most of the research conducted was to compare the performance of microservices with monolithic architectures [11], [13], [14]. Villamizar et al. [11] claim that microservice applications performance was reduced by 13% when compared to the monolithic using the Web server use case. Lloyd et al. [15] have done an extensive performance evaluation of the microservices application in server-less platforms. Hasselbring and Steinacker [16] migrated the otto.de e-commerce website from a monolithic architecture to a microservices architecture. With the number of website visitors reaching one million users every day, otto.de is one of the largest e-commerce platforms in Europe [16]. The increasing number of visitors and transactions made the company Otto decide to build the otto.de website from scratch using the microservices architecture. Communication between services is carried out by using the REST API protocol.

There are several existing researches [9], [10], [16], [18], [19], [20] that investigate the performance characteristics of microservices application in containers. Alam et al. [21] also use docker and microservices in building Edge Computing infrastructure as a supporting platform for Internet of Things (IoT) applications. Krylovskiy et al [22] designed a Smart City IoT platform based on the Microservices architecture and found MSA to have a better performance than SOA. Sun et al. [23] developed simulation models that can estimate the performance of microservice applications. Barakat [24] used Kieker framework for monitoring microservices performance during run time and Kieker's trace analysis for analysis of the application. Dai et al. [25] used Labeled Transition System (LTS) as Choreography specification language and performed analysis under synchronous and asynchronous compositions. Akbulut & Perros [26] performed a research on performance of MSA by implementing various applications using the API Gateway, Chain of Responsibility, and Asynchronous Messaging design patterns.

Rudrabhatla [3] compared the performance of the two communication methods by building a simulation model consisting of several services and each service has its own separate database and cannot be accessed directly by other services (database per service pattern). The performance of Choreography and Orchestration was measured using the time parameter required to complete a transaction. Experiments were simulated for each model with a maximum number of services of eight services, up to 10 users, and single instance. Singhal, et al. [4] compared the performance of the two methods based on three parameters, namely execution time, memory utilization and power utilization. The measurement of these three parameters was not carried out from the beginning of the transaction to the end of the transaction but on every service involved in a transaction.

In previous studies, it can be seen that the Choreography method has better performance when compared to the Orchestration method, especially for systems that are not too complicated, and the number of services is not too much. Research on the performance of these two techniques has been performed before but has not been performed thoroughly and the effect of the number of service instances and the large number of concurrent users

on the performance of the two models has not been evaluated yet, thus it is necessary to perform a quantitative analysis to measure the performance of each technique. In previous research, when developing applications with MSA, the criteria for selecting the Saga Pattern have not been determined, especially criteria related to code management. We compared the performance of Choreography and Orchestration based on parameters of response time, throughput, and CPU utilization. Furthermore, experiments were performed with various number of services, various number of concurrent users and various number of service instances. The model built will also use a load balancer which functions to distribute the workload to each service. We also evaluated both methods based on how much effort is required to change the order of processes in a Saga.

5. RESEARCH METHODOLOGY

To determine which saga implementation technique is more suitable under which scenario, we have implemented a research project and simulated various circumstances. We have developed two models of MSA application using Spring Boot Java programming language, MySQL database, and system messaging Kafka. Then the performance of the two application models is evaluated by measuring the parameters of response time, throughput, and CPU utilization. A transaction simulation is carried out involving several services and each measurement parameter is recorded. Furthermore, we perform experiments with different number of services, concurrent users, and service instances to determine how much the impact on the performance of Choreography and Orchestration. Fig 3 shows a model developed using Choreography approach.

Each service has an API which functions to accept requests from other services and an internal function that is used to trigger other services. The event of each service is published to the event broker Kafka and each service is registered as a subscriber at Kafka so that the service can find out when there is an event addressed to itself. For simulations with multi-instances and multi-users, a load balancer is implemented so that the workload can be divided and not focused on just one service instance.

Each service in the model Choreography method communicates directly with other services in the following order:

1) Service S_1 starts the Saga by publishing an event S_1.
2) Event S_1 triggers service S_2.
3) After service S_2 has finished its process, it publishes event S_2.
4) Event S_2 triggers service S_3.
5) After service S_3 has finished its process, it publishes event S_3.
6) Event S_3 re-triggers service S_1 and service S_1 ends the Saga.

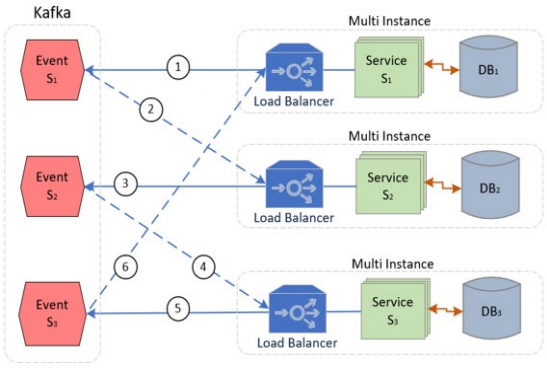

Figure 3. MSA Model using Choregraphy

Choreography makes the communication process between services faster because it does not depend on a service controller (coordinator) as used in Orchestration (Fig. 4). In this model, a service is built and has a function as a coordinator.

In Orchestration each service has an API which functions to receive requests from the coordinator and an internal function which is used to publish events to the coordinator. The event of each service and coordinator will be published to the event broker Kafka and each service and coordinator will be registered as a subscriber at Kafka so that the service and coordinator can find out when there is an event addressed to itself.

Each service in the model using the Orchestration method can communicate with other services through the coordinator in the following order:

1) Service S_1 starts the Saga by publishing an event to coordinator.
2) Coordinator receives event S_1.
3) Coordinator publishes an event to channel S_2.

Journal of Theoretical and Applied Information Technology
30th September 2021. Vol.99. No 18

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

4) Channel S_2 triggers service S_2.
5) After service S_2 has finished its process, it publishes an event to Coordinator.
6) Coordinator receives event S_2.
7) Coordinator publishes an event to channel S_3.
8) Channel S_3 triggers service S_3.
9) After service S_3 has finished its process, it publishes an event to Coordinator.
10) Coordinator receives event S_3 and ends the Saga.

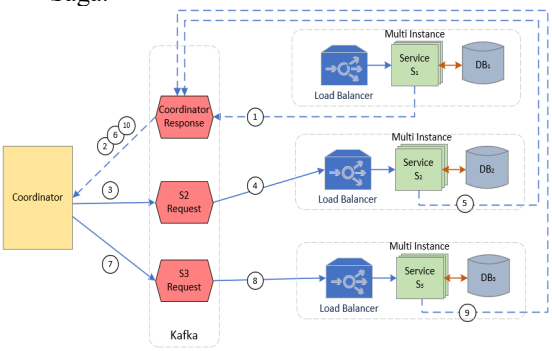

Figure 4. MSA Model using Orchestration

6. EXPERIMENTAL DESIGN

We implemented the models using Spring Boot Framework, MySQL database, and Kafka as messaging broker. Each service was deployed to Google Cloud Platform by leveraging Kubernetes. We used JMeter, which is widely adopted in workload characterization literature [13, 26], as a load testing client and compared the performance of Choreography and Orchestration based on the parameters of response time, throughput, and CPU utilization.

Data is collected by performing transaction simulations involving several services and each measurement parameter is recorded starting from the beginning of the transaction until the transaction ends as shown in Fig 5. For example, in Saga which involves two services i.e., S_1 (with database DB_1) and S_2 (with database DB_2). The timestamp is recorded when S_1 writes to DB_1 (1) and ended when S_1 writes back to DB_1 after being triggered by S_2 (3).

The simulation is done by using a different number of services, starting from 2 services, 4 services, 6 services, and 8 services. Furthermore, experiments were carried out by increasing the number of concurrent users gradually with multiples of 100 starting from 1 user, 100 users, 200 users, and up to 1,000 users. The models also simulated by using a different number of service instances (maximum two instances) to analyze which method has the better performance. Simulations with a single instance is done without a load balancer, while when the number of instances of each service is increased to two instances and in multi-user environment, the load balancer is added to distribute the workload.

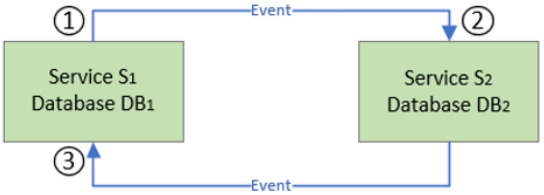

Figure 5 Saga Involving Two Services

7. EXPERIMENTAL RESULT

In this paper various comparison analyses were performed for Choreography and Orchestration methods based on the following two parameters namely average response time, throughput, and CPU utilization. Response time needs to be measured because the faster the response time, the resources can be freed immediately so that they can be allocated to handle other processes. Throughput is one of the key metrics in performance testing. It is used to check how many requests a software will be able to process per second, per minute or hour. In this paper we measure the throughput in how many transactions per second. CPU utilization needs to be measured because the higher the CPU utilization, the higher the server specifications that need to be provided, thereby increasing infrastructure costs.

6.1 Response Time

Fig. 6. and Fig. 7 show the response time difference between Choreography (shown as solid blue line) and Orchestration (shown as dotted orange line). Fig. 6 gives the two average response time as a function of number of users. The Saga consists of two services, single instance, without load balancer. Choreography represented by c2-1 while Orchestration represented by o2-1. In Fig. 7 the Saga consists of six services, single instance, and without load balancer. Choreography represented by c6-1 while Orchestration represented by o6-1.

Journal of Theoretical and Applied Information Technology
30th September 2021. Vol.99. No 18

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

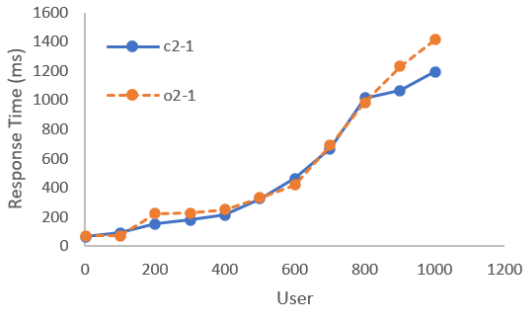

Figure 6. Average Response Time 2 Services without Load Balancer

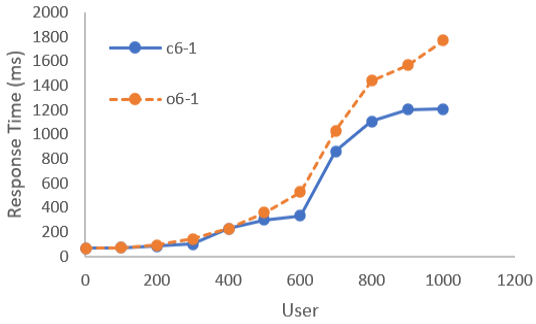

Figure 7. Average Response Time 6 Services without Load Balancer

In the environment which does not use load balancer, Choreography has a better response time than Orchestration either a small or large number of services. With the increasing number of services and the number of concurrent users, the system workload becomes heavier so that this has an impact on the higher average response time. In Orchestration the impact is more significant, this is due to the back-and-forth communication between the service and the Coordinator.

In a multi-users environment especially with heavy traffic load, system performance can be improved by adding more service instance and using a load balancer to distribute the workload. Fig. 8 gives the two average response time as a function of number of users. The Saga consists of two services, two instances, and use load balancer. Choreography represented by c2-2 while Orchestration represented by o2-2. In Fig. 9 the Saga consists of six services, two instances, and use load balancer. Choreography represented by c6-2 while Orchestration represented by o6-2. From both charts it can be seen that in the environment that uses load balancer, Orchestration is slower than Choreography (same result as in the environment that does not use load balancer.

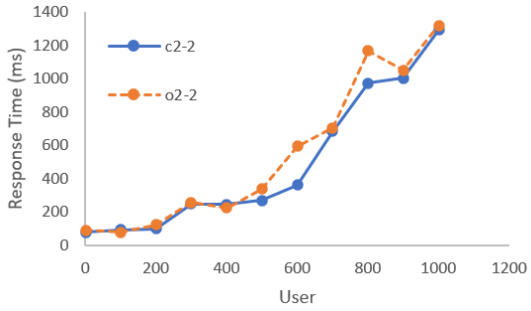

Figure 8. Average Response Time 2 Services with Load Balancer

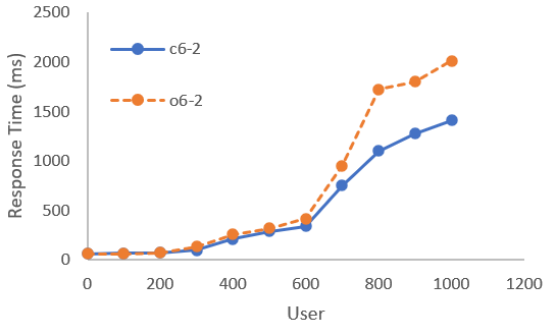

Figure 9. Average Response Time 6 Services with Load Balancer.

As the number of services and the number of concurrent users increases, the system workload becomes heavier, which causes the response time of both models to be slower. In Orchestration the impact becomes more significant because of the back-and-forth communication between the service and the coordinator.

6.2 Throughput

The throughput of both models can be seen in Fig. 10 and Fig. 11, Choreography shown as solid blue line and Orchestration shown as dotted orange line. Fig. 10 gives the two throughputs as the function of number of users. The Saga consists of eight services, single instance, does not use load balancer. Choreography represented by c8-1 while Orchestration represented by o8-1. In Fig. 11, the Saga consists of eight services, two instances, and uses load balancer. Choreography represented by c8-2 while Orchestration represented by o8-2. As the number of users and number of services increasing, in terms of throughput, Choreography (both without load balancer and with load balancer) performed better than Orchestration.

In the environment which uses load balancer (Fig. 11) has a higher throughput than

Journal of Theoretical and Applied Information Technology
30th September 2021. Vol.99. No 18

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

environment which does not use load balancer (Fig. 10). The use of load balancer can improve the throughput of both models especially in a multi-users environment with heavy traffic load.

The simulation results show that Choreography has better throughput than Orchestration for both models that use a load balancer and do not use a load balancer. This is due to the existence of a coordinator in the Orchestration model which increases the possibility of bottlenecks, especially when the number of concurrent users and the traffic load increases.

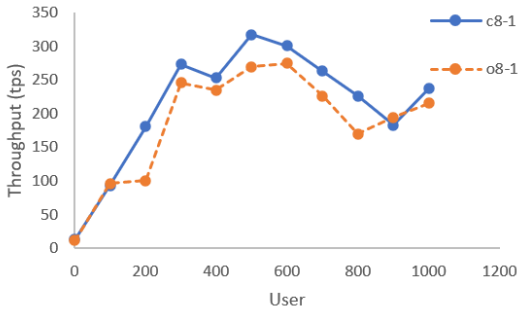

Figure 10. Throughput 8 Services without Load Balancer.

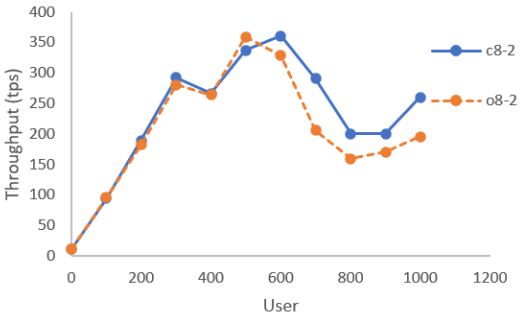

Figure 11. Throughput 8 Services with Load Balancer.

6.3 CPU Utilization

Fig. 12 and Fig. 13 shows the difference of CPU utilization between Choreography (shown as solid blue line) and Orchestration (shown as dotted orange line). Fig. 12 gives the two average CPU utilization as a function of number of users. The Saga consists of eight services, single instance, without load balancer. Choreography represented by c8-1 while Orchestration represented by o8-1. In Fig. 13 the Saga consists of eight services, two instances, and uses load balancer. Choreography represented by c8-2 while Orchestration represented by o8-2.

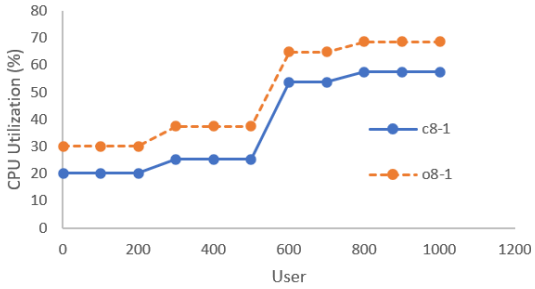

Figure 12. CPU Utilization 8 Services without Load Balancer.

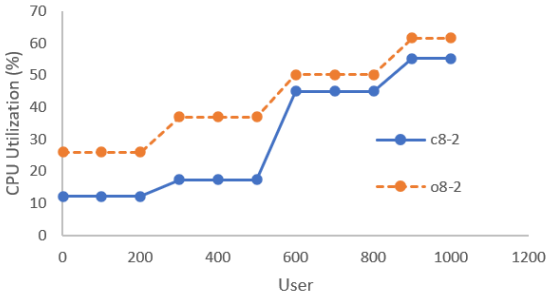

Figure 13. CPU Utilization 8 Services with Load Balancer.

The simulation results for both models show that Choreography has better CPU utilization than Orchestration. Overall, a model that uses a load balancer has better CPU utilization because the load balancer can help distribute the workload evenly across each service instance.

6.4 Changes of Process Sequence

In this paper we also simulate a situation where there is a change in business process which results in a change in the sequence of processes. This will impact to the interaction flow between services. The initial processing sequence is $S_1 - S_2 - S_3 - S_4$ and then changed to $S_1 - S_3 - S_2 - S_4$. Service inventory of Choreography before and after changes are given in the Table 1 and Table 2. Service inventory of Orchestration before and after changes are given in the Table 3 and Table 4. As the process sequence was changed to $S_1 - S_3 - S_2 - S_4$, in Choreography we need to modify 6 APIs (i.e., : API A, API B, API C, API D, API E, and API F) and the total services impacted is 4 services, while in Orchestration there is only 1 API that need to be modified and total services impacted is 1 service (i.e.,: coordinator). This shows that when there is workflow change, more effort is required to modify Choreography than Orchestration.

Table 1:
Service Inventory of Choreography Processing Sequence $S_1 – S_2 – S_3 – S_4$

Service Name	API Name	API Function
Service S_1	API A	Handle a request from User
	API B	Trigger S_2
Service S_2	API C	Handle a request from S_1
	API D	Trigger S_3
Service S_3	API E	Handle a request from S_2
	API F	Trigger S_4
Service S_4	API G	Handle a request from S_3
	API H	Trigger S_1

Table 2
Service Inventory of Choreography Processing Sequence $S_1 – S_3 – S_2 – S_4$

Service Name	API Name	API Function
Service S_1	API A	Handle a request from User
	API B	Trigger S_3
Service S_2	API C	Handle a request from S_3
	API D	Trigger S_4
Service S_3	API E	Handle a request from S_1
	API F	Trigger S_2
Service S_4	API G	Handle a request from S_2
	API H	Trigger S_1

Table 3:
Service Inventory of Orchestration Processing Sequence $S_1 – S_2 – S_3 – S_4$

Service Name	API Name	API Function
Service S_1	API A	Handle a request from User
	API B	Trigger Coordinator
Service S_2	API C	Handle a request from Coordinator
	API D	Trigger Coordinator
Service S_3	API E	Handle a request from Coordinator
	API F	Trigger Coordinator
Service S_4	API G	Handle a request from Coordinator
	API H	Trigger Coordinator
Coordinator	API I	Handle a request from S_1, S_2, S_3, S_4
	API J	Trigger S_1, S_2, S_3, S_4
	API K	Routing API call from and to service S_1-S_2-S_3-S_4

Table 4:
Service Inventory of Orchestration Processing Sequence $S_1 – S_3 – S_2 – S_4$

Service Name	API Name	API Function
Service S_1	API A	Handle a request from User
	API B	Trigger Coordinator
Service S_2	API C	Handle a request from Coordinator
	API D	Trigger Coordinator
Service S_3	API E	Handle a request from Coordinator
	API F	Trigger Coordinator
Service S_4	API G	Handle a request from Coordinator
	API H	Trigger Coordinator
Coordinator	API I	Handle a request from S_1, S_2, S_3, S_4
	API J	Trigger S_1, S_2, S_3, S_4
	API K	Routing API call from and to service S_1-S_3-S_2-S_4

8. CONCLUSION AND FUTURE WORK

In this research, we evaluated the performance of Choreography and Orchestration based on the number of services involved in a Saga so that it will be known how effective the two techniques are and how much influence the number of services has on the performance of Choreography and Orchestration. Furthermore, we performed experiments with various number of concurrent users (the number of users who conduct transactions simultaneously) and various number of service instances. By default, a service is deployed into a virtual machine (single instance), in this research we experimented using two instances for each service. By using more than one instance for each service, a load balancer is introduced and used as a proxy to divide the workload and traffic of each service. Both methods are also evaluated based on how much effort is required to change the order of processes in a Saga. We used the collected data to analyze and determine the criteria for selecting Choreography or Orchestration.

Based on the experimental results, it is found that with the increase in the number of services and the number of concurrent users involved in a Saga, the response time of the two models become slower. This is because the workload of the system becomes heavier. In addition, it was also found that Choreography has a better response time than Orchestration and the addition of the number of services and the number of concurrent users will have a more significant impact on Orchestration due to back-and-forth communication between the service and the coordinator.

The experimental results also show that Choreography has a better throughput than Orchestration for both models that do not use load balancers and models that use load balancers. The presence of an Orchestration coordinator can increase the possibility of bottlenecks at the coordinator, especially when the number of concurrent users increasing and more traffic load. Simulation results for both models show that Choreography has better CPU utilization than Orchestration due to back-and-forth communication between the service and the coordinator in the Orchestration so that increase the workload of each service. In overall, models that use load balancers have better CPU utilization than models that do not use load balancers because load balancers can help distribute workloads evenly across each service instance.

Based on the experimental results Choreography is much faster when compared to Orchestration. When both models were measured using throughput Choreography has a better performance than Orchestration. Choreography is also more efficient in CPU utilization because of the back-and-forth communication between the service and the coordinator in the Orchestration that increases the workload of each service. Choreography model becomes very difficult to code especially in managing multiple events triggered by each service. Without a central coordinator, the code will become complex, especially when there are more than one software developers in the team. We recommend using Choreography approach when there are a fewer number of microservices participating in the distributed transaction, or the number of event triggers are not too many or when the trigger actions are not too complex. Orchestration is slower than Choreography, but it is useful for handling complex transactions and easier to maintain the code. In this paper, we performed a quantitative analysis of performance of both event Choreography and Orchestration techniques used for implementing the Saga design pattern to handle the distributed transactions in microservices architecture with various scenarios such as different number of services, different number of concurrent users, and different number of service instance. In future work, a performance analysis of the hybrid model (combination of Choreography and Orchestration) can be performed.

REFRENCES:

[1] H. Kang, M. Le, and S. Tao, “Container and microservice driven design for cloud infrastructure DevOps,” Proc. - 2016 IEEE Int. Conf. Cloud Eng. IC2E 2016 Co-located with 1st IEEE Int. Conf. Internet-of-Things Des. Implementation, IoTDI 2016, pp. 202–211, 2016.

[2] A. Messina, R. Rizzo, P. Storniolo, M. Tripiciano, and A. Urso, “The database-is-the-service pattern for microservice architectures,” Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics), vol. 9832 LNCS, no. January 2018, pp. 223–233, 2016.

[3] C. K. Rudrabhatla, “Comparison of event choreography and orchestration techniques in Microservice Architecture,” Int. J. Adv. Comput. Sci. Appl., vol. 9, no. 8, pp. 18–22, 2018.

[4] N. Singhal, U. Sakthivel, and P. Raj, “Selection Mechanism of Micro-Services Orchestration Vs. ChoreographyInt. J. Web Semant. Technol., vol. 10, no. 1, pp. 01–13, 2019.

[5] C. Richardson, Microservices patterns. Shelter Island, NY: Manning Publications, 2019.

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

[7] J. Lewis and M. Fowler, "Microservices", martinfowler.com, 2014. [Online]. Available: https://www.martinfowler.com/articles/microservices.html. [Accessed: 02- Apr- 2020].

[8] R. V, *Spring 5.0 Microservices - Second Edition (2)*. Birmingham, UK: Packt Publishing, 2017.

[9] M. Amaral, J. Polo, D. Carrera, I. Mohomed, M. Unuvar and M. Steinder, "Performance Evaluation of Microservices Architectures Using Containers," *2015 IEEE 14th*

International Symposium on Network Computing and Applications, Cambridge, MA, 2015, pp. 27-34, doi: 10.1109/NCA.2015.49.

[10] D. Shadija, M. Rezai, and R. Hill, "Microservices: Granularity vs. Performance," *UCC 2017 Companion - Companion Proc. 10th Int. Conf. Util. Cloud Comput.*, pp. 215–220, 2017.

[11] M. Villamizar et al., "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud," *2015 10th Computing Colombian Conference (10CCC), Bogota, 2015*, pp. 583-590, doi: 10.1109/ColumbianCC.2015.7333476.

[12] T. F. Wenisch, "μ Suite : A Benchmark Suite for Microservices," *2018 IEEE Int. Symp. Workload Charact.*, no. 1, pp. 1–12, 2018.

[13] T. Ueda, T. Nakaike and M. Ohara, "Workload characterization for microservices," *2016 IEEE International Symposium on Workload Characterization (IISWC), Providence, RI, 2016*, pp. 1-10, doi: 10.1109/IISWC.2016.7581269.

[14] S. Stoja, S. Vukmirovic, N. Dalcekovic, D. Capko, and B. Jelacic, "Accelerating performance in critical topology analysis of distribution management system process by switching from monolithic to microservices," *Rev. Roum. des Sci. Tech. Ser. Electrotech. Energ.*, vol. 63, no. 3, pp. 338–343, 2018.

[15] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly and S. Pallickara, "Serverless Computing: An Investigation of Factors Influencing Microservice Performance," *2018 IEEE International Conference on Cloud Engineering (IC2E), Orlando, FL, 2018*, pp. 159-169, doi: 10.1109/IC2E.2018.00039.

[16] W. Hasselbring and G. Steinacker, "Microservice architectures for scalability, agility and reliability in e-commerce," *Proc. - 2017 IEEE Int. Conf. Softw. Archit. Work. ICSAW 2017 Side Track Proc.*, pp. 243–246, 2017.

[17] N. Kratzke, "About Microservices, Containers and their Underestimated Impact on Network Performance."

[18] P. Tennage, S. Perera, M. Jayasinghe, and S. Jayasena, "An analysis of holistic tail latency behaviors of java microservices," *Proc. - 21st IEEE Int. Conf. High Perform. Comput. Commun. 17th IEEE Int. Conf. Smart City 5th IEEE Int. Conf. Data Sci. Syst. HPCC/SmartCity/DSS 2019*, pp. 697–705, 2019.

[19] F. H. L. Buzato, A. Goldman, and D. Batista, "Efficient resources utilization by different microservices deployment models," *NCA 2018 - 2018 IEEE 17th Int. Symp. Netw. Comput. Appl.*, no. i, pp. 1–4, 2018.

[20] S. Mohammed, J. Fiaidhi, and M. Tang, "Towards using Microservices for Transportation Management : The New TMS Development Trend," *Proc. 10th Int. Conf. Logist. Informatics Serv. Sci.*, p. 472, 2020.

[21] M. Alam, J. Rufino, J. Ferreira, S. H. Ahmed, N. Shah, and Y. Chen, "Orchestration of Microservices for IoT Using Docker and Edge Computing," *IEEE Commun. Mag.*, vol. 56, no. 9, pp. 118–123, 2018.

[22] A. Krylovskiy, M. Jahn, and E. Patti, *"Designing a Smart City Internet of Things Platform with Microservice Architecture," Proc. - 2015 Int. Conf. Futur. Internet Things Cloud, FiCloud 2015 2015 Int. Conf. Open Big Data, OBD 2015*, pp. 25–30, 2015.

[23] Sun Y., Meng L., Liu P., Zhang Y., Chan H. (2018) Automatic Performance Simulation for Microservice Based Applications. In: *Li L., Hasegawa K., Tanaka S. (eds) Methods and Applications for Modeling and Simulation of Complex Systems. AsiaSim 2018. Communications in Computer and Information Science, vol 946. Springer, Singapore*. https://doi.org/10.1007/978-981-13-2853-4_7

[24] S. Barakat, "Monitoring and Analysis of Microservices Performance.," *J. Comput. Sci. Control Syst.*, vol. 10, no. 1, pp. 19–22, 2017.

[25] F. Dai, Q. Mo, Z. Qiang, B. Huang, W. Kou, and H. Yang, "A Choreography Analysis Approach for Microservice Composition in Cyber-Physical-Social Systems," *IEEE Access*, vol. 8, pp. 53215–53222, 2020, doi: 10.1109/ACCESS.2020.2980891.

[26] Akbulut, A., & Perros, H. G. (2019). Performance Analysis of Microservice Design Patterns. *IEEE Internet Computing*, *23*(6), 19–27. https://doi.org/10.1109/MIC.2019.2951094

[27] S. Lehrig, R. Sanders, G. Brataas, M. Cecowski, S. Ivanšek, and J. Polutnik, "CloudStore — towards scalability, elasticity, and efficiency benchmarking and analysis in Cloud computing," *Futur. Gener. Comput. Syst.*, vol. 78, pp. 115–126, 2018.