MICROSERVICE AND MONOLITH PERFORMANCE COMPARISON IN TRANSACTION APPLICATION

[1]Alexander Jason Lauwren, [2]Y.b Dwi Setianto
[1,2]Program Studi Teknik Informatika Fakultas Ilmu Komputer,
Universitas Katolik Soegijapranata
[2]setianto@unika.ac.id

ABSTRACT

When it comes to building or developing an online system, developers need to choose what kind of architecture will be used for the system. When facing the challenge, a developer needs the most suitable architecture that most fit the case whether uses microservices or monolithic architecture. Both architectures offer different benefits. Microservice recently become popular because many large companies start migrating from monolith to microservices but on the other hand. many organizations are still unfamiliar with microservices. Despite microservice providing many benefits, it also has challenges. With that being said, it is common that many organizations choose to stick with monolithic architecture since it was easier to maintain, develop, and deploy. To find out the better architecture performance-wise, API provided by both application need to be tested. The test was conducted by hitting the API several times with many threads concurrently. The test result is latency and request time, success rate needs to be monitored as well since the error occurred during load testing. With the data collected it could be shown which architecture performs better. The test results obtained by testing both monolith and microservice with several scenarios are quite unexpected. From the data, it turns out that for most features, average latency from the monolith is better than microservice. Meanwhile in many other scenarios, microservice edge monolith on the close gap in terms of success rate. The latency average number result for microservice is slightly worse because there are more success requests and taking more time.

Keywords: Microservice, Monolith, Latency, Performance

INTRODUCTION

Background

In this modern era, the requirement for digitalized systems increasing rapidly. With the Covid-19 Pandemic happening, many major living aspects tend to be online. Government and companies attempt to cope with the change of lifestyle by providing the online system and or application. This leads developers to create a high-performance system that is capable to serve the demand. One of the most common answers used by developers is web application. Web applications demanded to be agile, fast, and capable of handling such traffic.

When it comes to building or developing an online system, developers need to choose what kind of architecture will be used for the system. When facing the challenge, a developer needs the most suitable architecture that most fit the case whether uses microservices or monolithic architecture. Both architectures offer different benefits. Microservice recently become popular because many large companies start migrating from monolith to microservices but on the other hand. many organizations are still unfamiliar with microservices. Despite microservice providing many benefits, it also has challenges. With that being said, it is common that many organizations choose to stick with monolithic architecture since it was easier to maintain, develop, and deploy.

In this research, microservices and monolithic performance are compared in a different scenario. Both microservices and monolithic architecture build identical and contain identical services. Even though both architectures use Go-Language as the programming language, the framework used is different since the monolith use Echo as the framework and the microservice uses Go-Kit as a microservice toolkit. Thus, this research tries to answer how does architecture affect application performance, how does test design affect test result, and which architecture performs better. The main objective of this research is to compare the performance between microservices architecture and monolithic architecture in different scenarios. On the other hand, this research aims to help developers decide which architecture is most suitable to the case faced.

LITERATURE STUDY

Gos and Zabierowski [1] compare microservice and monolithic architecture by building an identical application coded using Java language. This journal compared both architectures by testing their performance. Test conducted done by make thirty thousand requests at once. The result shows that microservice edge monolithic on the performance aspect. However, both architectures have their own advantage. This journal shows that microservices could perform better when facing a massive number of requests. However, monolithic architecture is easier to deploy and build rather than microservices. In this research, the test conducted will use JMeter to test the API provided by both architectures.

Bucchiarone et al. [2] discuss migration from monolithic to microservices of Danske Bank's FX Core system. The article suggests that migrating from monolithic to microservices allows the system to be more distributed, avoid overlapping responsibilities, and more scalability. Like most of the migration processes, the monolith application spread into smaller entity then the process begins with containerization. In this case, FX Core Microservice architecture uses Docker Swarm Cluster, the step followed by automation which is done by deploying CI/CD pipeline. With Docker Swarm, it allows orchestration which means failed service can be automatically restarted. For integrating such a massive system which used to be a large monolithic, the huge number of services need to communicate with each other by using RabbitMQ as the messaging system. The difference between this journal and the project is in the number of services used, and the microservice pattern used.

Desai [3] discuss the architecture of microservices and technology and functionality. This article aims on explaining the benefit of microservice as an improvement and solution to monolithic architecture disadvantage. The research focuses on adopting the microservices architecture which offers scalability, flexibility, resilience, and autonomy. Besides that, the researcher explained the tools and step needed to migrate to microservices. Although the benefit is promising, migrating to microservices from the monolith is not as easy as it looks. The project compares both microservices and monolithic performances.

To compare the performance between both architecture, testing is the core part. Nevedrov [4] describe Apache JMeter is a tool used for testing web application which utilizes HTTP or FTP servers. Apache JMeter is java based application. JMeter testing involves creating a loop and a thread group. JMeter is capable to simulate load testing consisting of many concurrent threads hitting the server. Jmeter testing results in form of latency and response time. The data testing later could be used as a comparison or measurement of the system.

Since there are many forms of a web application, this research use REST API and JSON as the data output or input. Neumann et al. [5] analyzed technical features being used in the web services regarding REST principles. The principles consist of REST architecture, HTTP Use, Input-Output formats support, Security mechanism, Usage policies, and documentation and application use. This journal went through find 500 sites that provide web service API. These sites were analyzed to understand the level of compliance with the REST principles. It concluded that 84% already use the REST URL scheme with 55% of them using JSON as output format.

Schmager et al. [6] describe Go as object-oriented programming built with a C-like syntax. This journal evaluates Go syntax and idioms. Go syntax is an upgrade from Java and C. Not requiring semicolons, initial capital for public encapsulation scheme are some positive traits from Go Syntax making it is nice to write. But defining methods outside classes in Go is hard since it requires the specific receiver’s name and types explicitly. On the idioms aspect, this journal highlighted that go doesn’t have the equivalent of abstract classes, Go also doesn’t support constructors like Java. But in spite all the setback, comma OK idioms is nice to have since methods in go can return multiple values. Since Go is produced to code efficiency, this research use Go language to build the application.

Sulander [7] discuss implementing a microservices approach on a fully open retail interface in order to be able to sell public transport tickets. Microservices become very suitable since the objective was to build scalable and modular systems to serve all functionalities. The researcher also described microservices as the stem from the desire to build upgradeable and scalable software. This research implement Domain Driven Design as its design pattern which allows the researcher to encapsulate the business logic and capabilities throughout the system. The services communicate with each other over APIs. The data storage in this journal uses Azure Cosmos DB and uses a single database for every service. All traffic for the system using OpenMaaS happens over HTTPS with a private endpoint so that security could be provided. On the other hand, the project uses NGINX to serve the API, PostgreSQL to store the data.

Vionnet et al.[8] present Synapse, a semantic system for large-scale web service based on microservice architecture. Synapse is designed to handle large data drives and integrated as a web service. Synapse architecture uses microservice architecture in which the services have their own databases which structure very differently. Synapse build to perform data integration at object level so the system can cope with SQL and no SQL database. Synapse was developed using Ruby on Rails as the framework to develop Synapse. Meanwhile, the project uses Go-Kit to develop the microservices.

The project faced some challenges since building a microservice is quite complicated. Kalske et al.[9] discuss challenges when migrating monolith systems to microservice architecture. The journal stated that the transition from monolith architecture to microservice is not easy at all since it requires time and effort. The challenge faced include many aspects such as technical challenges and organizational challenges. The most common technical challenge is to split up the services, logging, and testing since microservice is harder to test. Usually, the integration between services to already existing technologies will be a problem as well. Besides the technical challenges, there is an organizational challenge such as structure organization, task separation. While some companies already have a big team it needs to spread the team to smaller teams while migrating from monolith to microservices.

Villamizar et al.[10] analyze microservice patterns used by Amazon, Netflix, and LinkedIn which deploy the large application on a cloud as small separated services. It is proven that microservice architecture provides businesses the scalability, capability to handle a huge number of users. Microservices allow companies to manage applications easier since every application is separated as small services that run independently. With all benefit this journal said, building or migrating to microservices have some challenges as well. The project proves that microservices handle many users better than a monolith.

Al Debagy and Peter Martinek [11] test the performance of both microservices and monolith applications. The result of the test scenario concluded as a monolith application is recommended while building a small application and on the other hand, when the application will be used by a large number of users, microservices is a perfect choice. The journal conducts the test by using JHipster to generate a web application that consists of Spring Boot and Angular JS. Unlike the journal, the project uses Go-Language to develop a web application.

Therefore, this project uses both monolith and microservices, the difference from other research in this project aims to compare the performance between microservice and monolith while given such scenario. Not only that, the programming language used, the design of microservices, and database design are different. This project use Go-Language for both monolith and microservice, the design used is based on the Go-Kit framework and Echo framework. Both were used so that the project could serve on REST API. The database used for monolith is one database for every application, and for microservice, a single database serves only one service.

ANALYSIS AND DESIGN

Analysis

When a developer starts building a web application, the process usually started with building applications with monolithic architecture that contain many services. Monolith architecture chose because it is easier to develop and maintain. While the application is still small, the monolith application works perfectly with a small number of the team. The problem with monolith starts surfacing when the application starts getting bigger, the number of developers increases, and features growth. Since monolith offers no scalability, many companies chose to migrate to microservices. When migrating to microservices, the first step done is to split the monolith services into smaller services. The next step is to divide the data from a single database into separated databases. The database separation is done so that the service could run independently. Even it seems that microservice have so many advantages, they also have drawbacks.

To find out the better architecture performance-wise, API provided by both application need to be tested. The test was conducted by hitting the API several times with many threads concurrently. The test result is latency and request time, success rate needs to be monitored as well since the error occurred during load testing. With the data collected it could be shown which architecture performs better.

Design

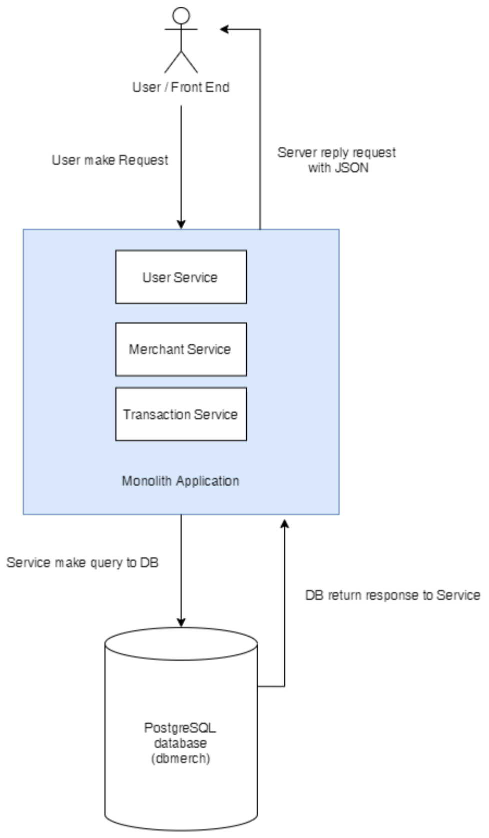


Figure 1. Monolith Architecture Design

Monolith application design based on Go Echo framework where service separated as some package. Endpoint part or usually coded on main package handle HTTP request and routing the request. Usecase package handle the business logic. Every logic that goes through on the application is handled on the use-case part. The repository package handles every communication to the database. In order to achieve smaller latency, the repository part can't have any business logic outside query to the database. The application consists of many services such as merchant service, user service, and transaction service. Every service has its own character and features. Merchant services have basic create, update and delete or so-called CRUD features. User services have login features that return JSON web tokens or are more popular as JWT for authorization purposes. Transaction services have bulk insert and transaction logic features. Every service is made different in order to compare architecture performance while handling such features. Understanding how the application work, in general, is explained in figure 2.

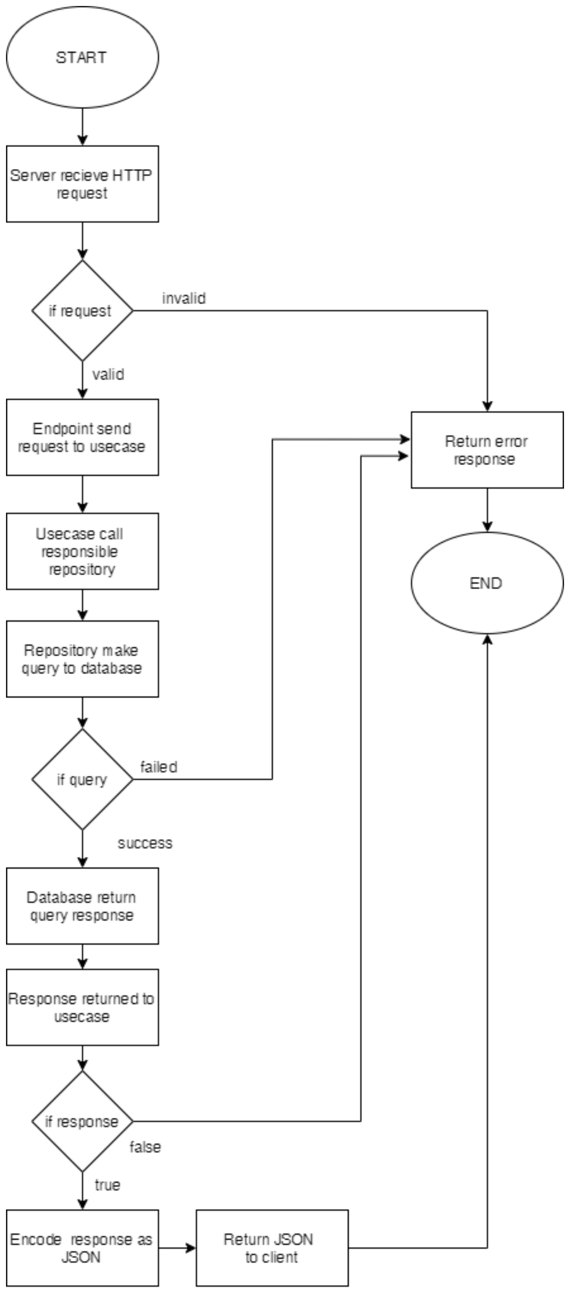

START
Server recieve HTTP request
if request
invalid
valid
Endpoint send request to usecase
Return error response
END
Usecase call responsible repository
Repository make query to database
if query
failed
success
Database return query response
Response returned to usecase
if response
false
true
Encode response as JSON
Return JSON to client

Figure 2. Flowchart Monolith Application Handling Single HTTP Request

The monolith application starts with the user, in this case, could be the front end or directly to the user making HTTP request to the server. When the user hits the API provided, the endpoint layer calls the use-case function as requested. Usecase call repository function to make a query to the shared database which consists of all the data stored by every service. Upon receiving a response from the database, the repository returns the data to the use case function, use-case function sends the data to the endpoint layer to encode data as JSON response. Meanwhile, it is possible to face errors while the program is running. When a layer face error or even the database sends error messages, the server will respond to the request with an error message. The service count is finished when the server sends a return to the client-side whether it is a success or an error.

Microservice application designed following microservices principles where an application runs independently, distributed, and scalable. As explained on figure 2, in a microservice architecture, every service act independent while having their own port and database. Even though all the services share the same relational database management system (RDBMS), all of the services have their own unique database. Since the service is distributed, to make a fair test all the services need to be accessible from one port. NGINX server used to reverse proxy services port and combined them so that the services' API listening on a single port.

On this project, a microservice was built based on Go Kit as a microservices toolkit. In the Go Kit principles, services are separated into three main layers such as transport layer, endpoint layer, and service layer. The transport layer is the part where transport processes are done since some cases need more than HTTP API to transport. In this project, the transport layer handles HTTP transport. The endpoint layer is often described as a controller where safety logic is coded. Since the logic functions need to be exposed externally, the endpoint layer is the one that receives the request and converts it to the struct needed. Not only that, but the endpoint layer is also the one that calls the service layer to get the return struct. The service layer is where the business logic lives. Monolith-wise, the service layer is the use-case where all the logic functions are coded.

The microservice application starts when the NGINX server receives a request from a client as drawn in figure 3. Upon receiving a request, NGINX forwards the request to a suitable service. While forwarding the request, NGINX errors may occur. If such scenarios happen, NGINX will return an error such as 502 Bad Gateway. On the other hand, when the request success to be forwarded, the transport layer of the service receives the request. From the transport layer, the request will be decoded as a struct. While decoding the request body, it may occur that the struct which is desired doesn't match. If such a scenario happens, the service will return an error message encoded by an endpoint as a response. When the struct match, the service layer will start the work by calling the business logic function. It consists of a logic function and repository function. The logic function will handle all the business logic while the repository handles the communication to the database. Upon getting a response from the database, if no errors are faced, the service layer returns the response to the endpoint layer so that the struct could be encoded to JSON format which later will be used by the transport layer as a response to the client.

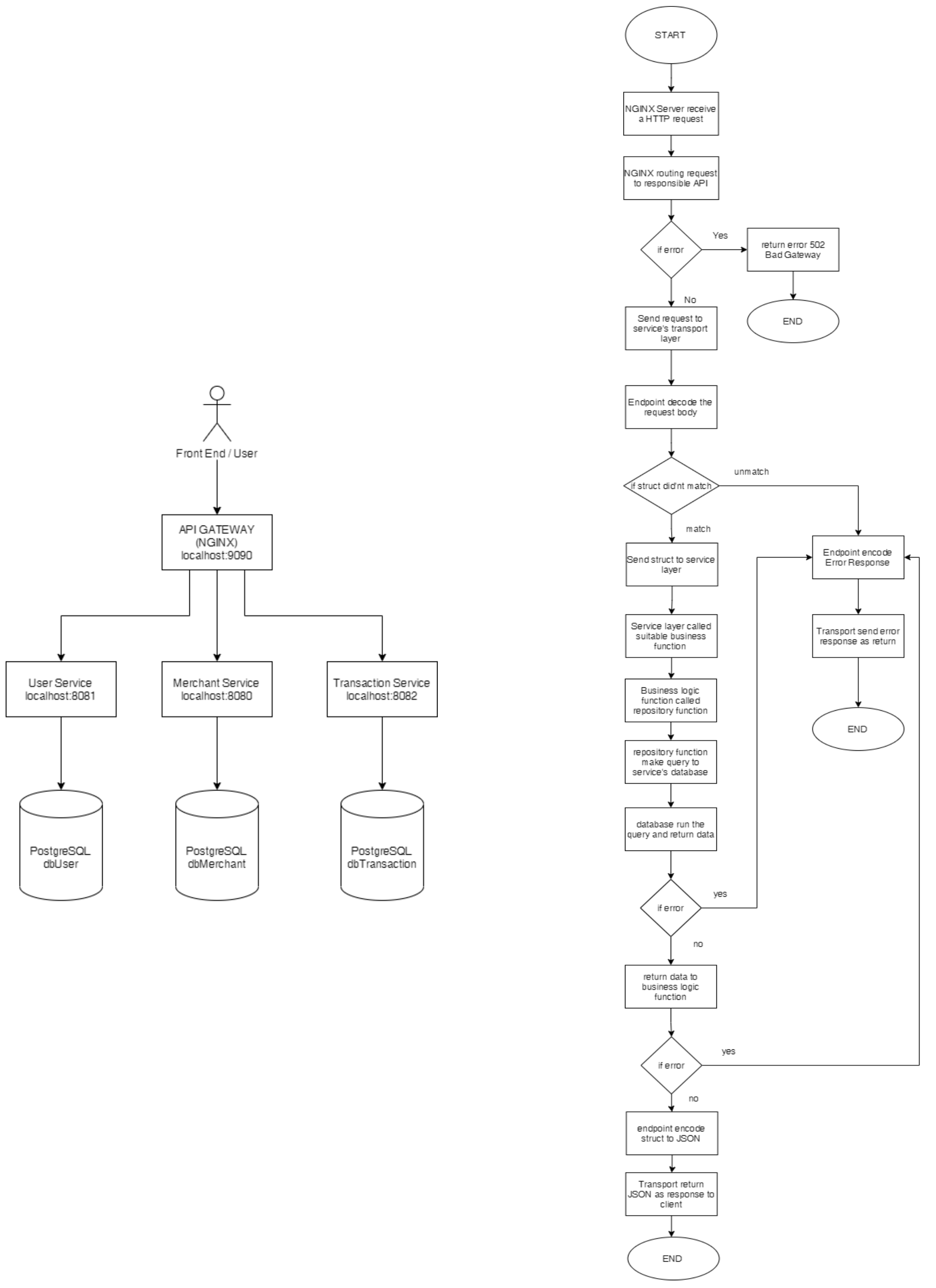

Front End / User
API GATEWAY (NGINX) localhost:9090
User Service localhost:8081
Merchant Service localhost:8080
Transaction Service localhost:8082
PostgreSQL dbUser
PostgreSQL dbMerchant
PostgreSQL dbTransaction
START
NGINX Server receive a HTTP request
NGINX routing request to responsible API
if error
Yes
return error 502 Bad Gateway
END
No
Send request to service's transport layer
Endpoint decode the request body
if struct did'nt match
unmatch
match
Endpoint encode Error Response
Transport send error response as return
END
Send struct to service layer
Service layer called suitable business function
Business logic function called repository function
repository function make query to service's database
database run the query and return data
if error
yes
no
return data to business logic function
if error
yes
no
endpoint encode struct to JSON
Transport return JSON as response to client
END

Figure 3. Left: Microservice Application Architecture, Right: Microservice Flowchart Handling Single HTTP Request

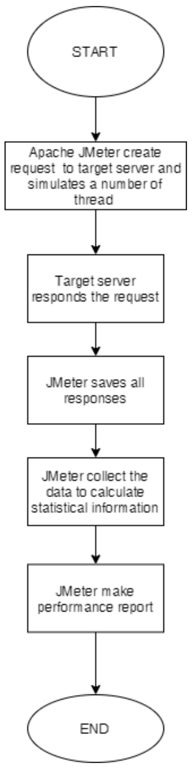

START
Apache JMeter create request to target server and simulates a number of thread
Target server responds the request
JMeter saves all responses
JMeter collect the data to calculate statistical information
JMeter make performance report
END

Figure 4. Flowchart JMeter

In order to compare both architectures, testing software is needed. Since this project tests HTTP server performance, Jmeter is the most suitable testing application. JMeter is capable to make numerous requests concurrently hitting the server's API. Upon getting a response from the server, JMeter saves all of them and collects them so that the data could be calculated and displayed as a performance report. The data used in this project are latency, success rate, and response time.

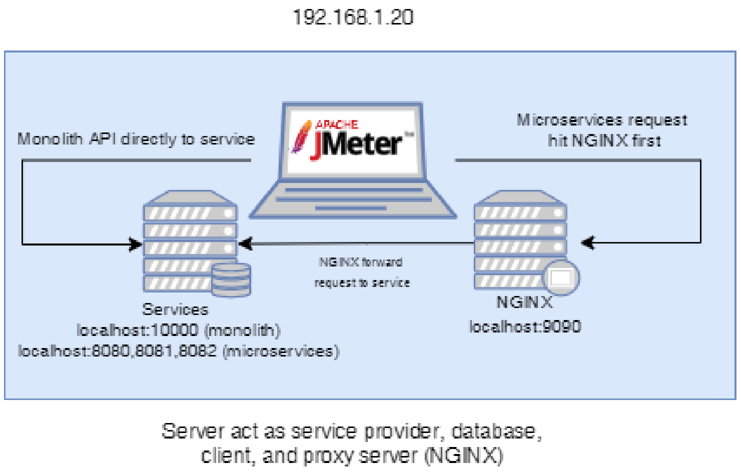

192.168.1.20
Monolith API directly to service
Apache JMeter
Microservices request hit NGINX first
NGINX forward request to service
Services
localhost:10000 (monolith)
localhost:8080,8081,8082 (microservices)
NGINX
localhost:9090
Server act as service provider, database, client, and proxy server (NGINX)

Figure 5. Testing Design Where JMeter, Services, Database, and Proxy Server (NGINX) Deployed On a Single Device.

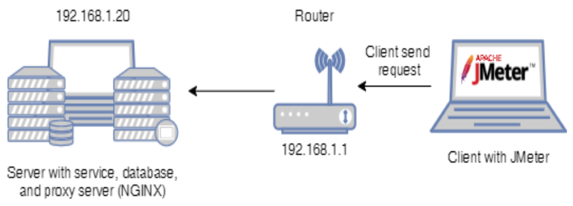


Figure 6. Testing Design Where JMeter Separated From Main Server

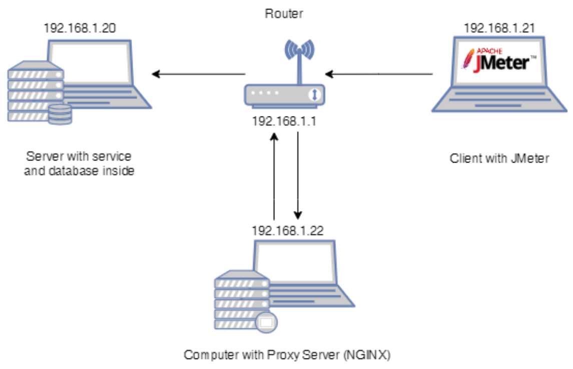


Figure 7. Testing Design Where Services Server, JMeter, NGINX Are Separated

In this project, there are several testing designs conducted. The first testing design is shown in figure 5. The design where services, database, JMeter, and NGINX all deployed together in a single device only. In the first design, every process is done locally, when JMeter tests the monolith application, it creates multiple requests, those requests are sent to localhost where services and their database are deployed. On the other hand, when testing microservice applications, to hit the API through a single port, NGINX deployed. The NGINX server deployed on localhost as well. This means that when JMeter creates a request to a microservice application the request is sent to the NGINX server and after that NGINX hits the service's API.

The second testing design is where JMeter is separated on another device as shown in figure 6. In this design, a test conducted with JMeter with IP address 192.168.1.21 hit the local server on IP address 192.168.1.20. When JMeter tests the monolith application, JMeter directly makes a request to the monolith application's API. Meanwhile, when testing microservices, JMeter makes a request to the NGINX server port first before NGINX forwards the request to the responsible service. NGINX server in the second design is deployed together with the services and database. These devices can communicate through a local network and are routed using a router.

For the final testing design, not only JMeter, the NGINX proxy server is separated onto another device as shown in figure 7. This mean, testing conducted using three devices which every device have their responsibilities. The first device contains services and a database, this device act as the service provider. Unlike the other design, the server device no longer contains an NGINX proxy server. This first device runs with IP address 192.168.1.20. This address will be the address hit by JMeter for the monolith application. The second device was deployed with the NGINX proxy server. This device serves only a single purpose as a reverse proxy. The second device registered with static IP on 192.168.1.22. The third device's sole purpose is to be the client. On this device, JMeter was deployed for testing. This device registered on static IP address 192.168.1.21. When testing the monolith application, JMeter on the third device directly sends the request to 192.168.1.20:10000. On the other hand, when testing the microservice application, JMeter sends the request to the second device first, meaning the request is sent to NGINX server port which is later forwarded to service API where located on the first device. NGINX on the second device role is to forward the request from a client to service.

IMPLEMENTATION AND RESULT

Implementation

This project uses the Go programming language. In this project, the performance of monolith architecture and microservice architecture is compared by load testing both architecture. Both architectures contain the same amount of services. Those services are built as REST API services where services communicate with the outside world using JSON format. In this project, both architectures are load-tested using JMeter. Each service has its own unique features that make it different from one another.

In monolith, all service connect to single database. This project use PostgreSQL as the database. In order to minimalize latency, every runtime service call database only once so that there won't be too much connection. In Go, like in Java programming, the executable functions are all located in the main function. Since connecting to the database should be secured and not exposed to the outside world. In order to achieve that, all functions that connect to the database are separated into a different package. Since there are 3 main services, there are 3 repositories. Package repository handles all functions responsible to make queries.

Result

To find out which architecture is better, several testing scenarios were conducted. The first testing scenario is to test all features from both architectures using the first testing design. In this design, every service, database, NGINX, and JMeter are inside a single device. In this step, JMeter hit both architecture's API.

Table 1. Detailed Get Merchant Performance Data on Both Architecture

Features	**Architecture**					
	Monolith			**Microservice**		
Total Request	***100***	***1000***	***5000***	***100***	***1000***	***5000***
Avg. Latency	2 ms	1288 ms	641 ms	4 ms	2102 ms	1233 ms
Success Rate	100%	86.30%	24.40%	100%	85.80%	26.74%
Max Latency	15 ms	2909 ms	5399 ms	9 ms	4792 ms	10164 ms

From the test evidence, while handling the smaller and medium amount of requests, the monolith application performs better while having average latency of 2ms. Microservice slightly edge the monolith only when handling 5000 requests on success rate. This poor performance by microservice could be caused by NGINX load time. This is proven from the terminal log where it shows that there is approximately a 2-millisecond delay between Jmeter latency and the terminal log latency. Meanwhile, the monolith application log shows that almost no delay between the terminal and Jmeter data. In the next step, this project test the insert feature. In this step, JMeter creates a POST HTTP request to the server in 1-second concurrently.

For POST HTTP requests, data need to be sent as a request body and written in JSON format as shown on line 302. The suffix, on the data merchant name, is given by creating a counter variable on JMeter. Below is the result of the test.

Table 2. Detailed Insert Merchant Performance Data on Both Architecture

Features	**Architecture**					
	Monolith			**Microservice**		
Total Request	***100***	***1000***	***5000***	***100***	***1000***	***5000***
Avg. Latency	6 ms	1688 ms	663 ms	5 ms	2477 ms	2101 ms
Success Rate	100%	65.20%	18.02%	100%	99.50%	25.84%
Max Latency	116 ms	2909 ms	5399 ms	101 ms	4498 ms	13004 ms

From the result above, there is an insane gap of latency between 100 hits and 1000 hits this could be caused by device limitation. This means that creating 1000 threads simultaneously takes many resources. To view more detailed results below is the data table

From the data in table 5.2, it could be seen that surprisingly microservice edge monolith on 1000 hits. The success rate is far higher on every hit. The max latency and the average on monolith seem smaller but it caused by many failed requests. On the next step, the last merchant service feature, the update feature is tested to find out which architecture perform better when handling update request. The counter value will be replaced with a loop sequence number. Below is the result of the test.

Table 3. Detailed Update Merchant Performance Data on Both Architecture

Features	**Architecture**					
	Monolith			**Microservice**		
Total Request	***100***	***1000***	***5000***	***100***	***1000***	***5000***
Avg. Latency	3 ms	1211 ms	497 ms	4 ms	2798 ms	1206 ms
Success Rate	100%	54.00%	14.56%	100%	79.90%	20.78%
Max Latency	10 ms	2909 ms	8255 ms	15 ms	5584 ms	10825 ms

From the table above, it could be concluded that even though monoliths have better performance, the number of failed requests is also higher than microservice. On the other hand, in terms of handling the higher request, microservice have a higher success rate. The max latency indicates the highest latency thread accept for a single request. In terms of max latency, microservice has a bit worse record.

To test the application with higher complexity, in this step user service is tested to find out which architecture handles authentication better. In user service, the register feature has password encryption using the bcrypt algorithm. JMeter sends the request like below to the register's API. This email will be added with a number sequence from the counter variable. Both performance tests result very poorly with an average latency of over ten thousand milliseconds. From the chart, it seems that monolith performs better while handling 100 requests. But microservice handles larger performance better. The amount of requests sent is lower than the previous test because the number of errors while handling 1000 requests is already massive. To understand the test result better, it can see in the below table.

Table 4. Detailed Register User Performance Data on Both Architecture

Features	**Architecture**			
	Monolith		**Microservice**	
Total Request	***100***	***1000***	***100***	***1000***
Avg. Latency	27492 ms	11512 ms	29362 ms	10526 ms
Success Rate	100%	23.20%	100%	13.60%
Max Latency	35241 ms	83125 ms	39607 ms	60003 ms

From the test result, unlike merchant testing results, microservice have a lower success rate during handling a larger amount of register requests.

In the next step, the login feature is tested. The login feature is the most complex because it checks the data from the database, finds the match data, compares the password, and creates JWT as a return.

Table 5. Detailed Login User Performance Data on Both Architecture

Features	**Architecture**			
	Monolith		**Microservice**	
Total Request	***100***	***1000***	***100***	***1000***
Avg. Latency	27702 ms	11742 ms	28597 ms	10041 ms
Success Rate	100%	22.50%	100%	13.10%
Max Latency	35241 ms	83125 ms	39607 ms	60003 ms

From the test result, both designs have a massive latency. This perhaps happened because the service is complex and the service needs time to complete. But, the result shows that monolith performs better with lower latency on 100 requests and a higher success rate on a larger amount of requests.

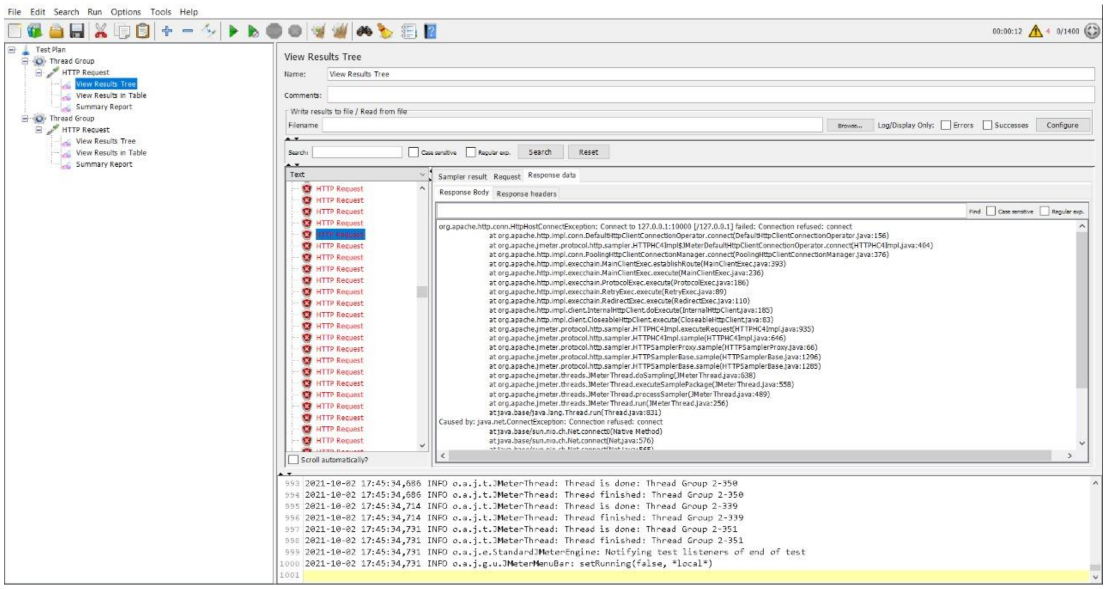
The last service that needs to be tested is the transaction service. Transaction service is a rather less complex computation than user service. This service contains bulk insert as the key feature. This request has an array attribute that will be inserted on a different table. For saving resources used, the array will only contain two structs for each request. Below is the test result.

Table 6. Detailed Insert Bulk Transaction Performance Data on Both Architecture

Features	**Architecture**					
	Monolith			**Microservice**		
Total Request	***100***	***1000***	***5000***	***100***	***1000***	***5000***
Avg. Latency	98 ms	1790 ms	2134 ms	6 ms	1849 ms	739 ms
Success Rate	100%	50.40%	30.82%	100%	70.10%	20.00%
Max Latency	402 ms	6949 ms	8255 ms	33 ms	5739 ms	9803 ms

From the test result, it could be concluded that in case handling insert transaction, microservice performs slightly better. But when it comes to handling a bigger amount of requests, the number of failures arise in microservice since it only achieves a 20 percent success rate. Although it seems that both architectures flop when handling a large number of requests, it can seem that in this particular service, microservice is slightly better.

The overall result of the test is worse than expected. There are too many errors on the test. The error message could be seen in figure 8 below



```
org.apache.http.conn.HttpHostConnectException: Connect to 127.0.0.1:10000 [/127.0.0.1] failed: Connection refused: connect
	at org.apache.http.impl.conn.DefaultHttpClientConnectionOperator.connect(DefaultHttpClientConnectionOperator.java:156)
	at org.apache.jmeter.protocol.http.sampler.HTTPHC4Impl$JMeterDefaultHttpClientConnectionOperator.connect(HTTPHC4Impl.java:404)
	at org.apache.http.impl.conn.PoolingHttpClientConnectionManager.connect(PoolingHttpClientConnectionManager.java:376)
	at org.apache.http.impl.execchain.MainClientExec.establishRoute(MainClientExec.java:393)
	at org.apache.http.impl.execchain.MainClientExec.execute(MainClientExec.java:236)
	at org.apache.http.impl.execchain.ProtocolExec.execute(ProtocolExec.java:186)
	at org.apache.http.impl.execchain.RetryExec.execute(RetryExec.java:89)
	at org.apache.http.impl.execchain.RedirectExec.execute(RedirectExec.java:110)
	at org.apache.http.impl.client.InternalHttpClient.doExecute(InternalHttpClient.java:185)
	at org.apache.http.impl.client.CloseableHttpClient.execute(CloseableHttpClient.java:83)
	at org.apache.jmeter.protocol.http.sampler.HTTPHC4Impl.executeRequest(HTTPHC4Impl.java:935)
	at org.apache.jmeter.protocol.http.sampler.HTTPHC4Impl.sample(HTTPHC4Impl.java:646)
	at org.apache.jmeter.protocol.http.sampler.HTTPSamplerProxy.sample(HTTPSamplerProxy.java:66)
	at org.apache.jmeter.protocol.http.sampler.HTTPSamplerBase.sample(HTTPSamplerBase.java:1296)
	at org.apache.jmeter.protocol.http.sampler.HTTPSamplerBase.sample(HTTPSamplerBase.java:1285)
	at org.apache.jmeter.threads.JMeterThread.doSampling(JMeterThread.java:638)
	at org.apache.jmeter.threads.JMeterThread.executeSamplePackage(JMeterThread.java:558)
	at org.apache.jmeter.threads.JMeterThread.processSampler(JMeterThread.java:489)
	at org.apache.jmeter.threads.JMeterThread.run(JMeterThread.java:256)
	at java.base/java.lang.Thread.run(Thread.java:831)
Caused by: java.net.ConnectException: Connection refused: connect
	at java.base/sun.nio.ch.Net.connect0(Native Method)
	at java.base/sun.nio.ch.Net.connect(Net.java:576)
```

```
2021-10-02 17:45:34,686 INFO o.a.j.t.JMeterThread: Thread is done: Thread Group 2-350
2021-10-02 17:45:34,686 INFO o.a.j.t.JMeterThread: Thread finished: Thread Group 2-350
2021-10-02 17:45:34,714 INFO o.a.j.t.JMeterThread: Thread is done: Thread Group 2-339
2021-10-02 17:45:34,714 INFO o.a.j.t.JMeterThread: Thread finished: Thread Group 2-339
2021-10-02 17:45:34,731 INFO o.a.j.t.JMeterThread: Thread is done: Thread Group 2-351
2021-10-02 17:45:34,731 INFO o.a.j.t.JMeterThread: Thread finished: Thread Group 2-351
2021-10-02 17:45:34,731 INFO o.a.j.e.StandardJMeterEngine: Notifying test listeners of end of test
2021-10-02 17:45:34,731 INFO o.a.j.g.u.JMeterMenuBar: setRunning(false, *local*)
```

Figure 8. Error Message From JMeter Failed Thread

When the message appears, it turns out that the server machine is still running. Also from monitoring the terminal log, it appears that the server did not face any error. Meanwhile, NGINX facing the same error as JMeter shows that the server is refusing the request.

From the observation above, moving JMeter to another device perhaps could be the solution. In the second testing design, JMeter is separated onto another device since the JMeter user interface and service already used a huge amount of RAM usage. For the second design, only the key features were tested. This is done to find out whether device limitations have an aspect of the failure rate. But moving the JMeter to another device means that the request has more steps from client to server.

The first comparison is between the Get Merchant feature. All the method are the same but, in this test, JMeter sends a request from another device and hit the IP address of the server. Below is the result of the test.

Table 7. Detailed Get Merchant Performance Data on Both Architecture

Features	**Architecture**					
	Monolith			**Microservice**		
Total Request	**100**	**1000**	**5000**	**100**	**1000**	**5000**
Avg. Latency	12 ms	4698 ms	4073 ms	15 ms	2343 ms	3823 ms
Success Rate	100%	72.70%	35.70%	100%	52.80%	10.04%
Max Latency	31 ms	6650ms	11115 ms	106 ms	5965 ms	10970 ms

From the test result above, it seems that the performance becomes poorer in the second design. In success rate wise, for this get merchant feature alose worse. But it won’t be fair to only test one feature. The heaviest service needs to be tested as well. In this step, the user login service is tested using a second design. Below is the result of the test.

Table 8. Detailed Login User Performance Data on Both Architecture

Features	**Architecture**			
	Monolith		**Microservice**	
Total Request	***100***	***1000***	***100***	***1000***
Avg. Latency	29629 ms	13115 ms	273329 ms	10126 ms
Success Rate	100%	22.30%	100%	13.50%
Max Latency	37844 ms	88281 ms	42521 ms	60151 ms

Table 9. Detailed Insert Bulk Transaction Performance Data on Both Architecture

Features	**Architecture**					
	Monolith			**Microservice**		
Total Request	***100***	***1000***	***5000***	***100***	***1000***	***5000***
Avg. Latency	13 ms	5092 ms	7649 ms	21 ms	2304 ms	6523 ms
Success Rate	100%	52.30%	23.52%	100%	34.50%	20.14%
Max Latency	41 ms	11870 ms	22260 ms	123 ms	6859 ms	29908 ms

Because it seems that the microservice doesn’t live up to the expectation, to minimize ram usage and in an attempt to improve the performance of microservice, NGINX is separated to another device with an IP address located at 192.168.1.22. In this step, only microservice features are tested since, in this third design, JMeter still hits monolith architecture directly. Just like all the tests done before, Jmeter will hit the API provided by the server. But this time, unlike the second design, JMeter will hit 192.168.1.22. The first test conducted is to see how is the performance of the third design while handling merchant gets requests.

Table 10. Detailed Get Merchant Performance Data on Design Test 2 and Design Test 3

Features	**Architecture**					
	Design 2			**Design 3**		
Total Request	***100***	***1000***	***5000***	***100***	***1000***	***5000***
Avg. Latency	15 ms	2343 ms	3823 ms	24 ms	2320 ms	7511 ms
Success Rate	100%	52.80%	10.04%	100%	99.30%	58.26%
Max Latency	106 ms	5965 ms	10970 ms	135 ms	4861 ms	17187 ms

Compared to the second design, the third design handle more requests poorly. But in terms of success rate, somehow third design is edging the second design by a huge gap. To add more weight to the comparison, the user login feature is tested, and below is the result from the test.

Table 11. Detailed Design 2 and Design 3 Login User Performance

Features	**Architecture**			
	Design 2		**Design 3**	
Total Request	***100***	***1000***	***100***	***1000***
Avg. Latency	273329 ms	10126 ms	25002 ms	11084 ms
Success Rate	100%	13.50%	100%	13.20%
Max Latency	42521 ms	60151 ms	39539 ms	60209 ms

From the test result above, it seems that on small requests, the third design has the best performance but it is the opposite with a larger amount of requests where the third design only gets a 13.20% success rate. The final test is to compare the performance while handling the insert transaction.

Table 12. Detailed Design 2 and Design 3 Insert Bulk Transaction Performance

Features	**Architecture**					
	Design 2			**Design 3**		
Total Request	***100***	***1000***	***5000***	***100***	***1000***	***5000***
Avg. Latency	21 ms	2304 ms	6523 ms	134 ms	4269 ms	9439 ms
Success Rate	100%	34.50%	20.14%	100%	57.70%	28.28%
Max Latency	123 ms	6859 ms	29908 ms	123 ms	6859 ms	29908 ms

From the test result above, the performance of the third design has become slower as the request number increases. But in terms of success rate, the third design comes out victorious. When more requests are successfully sent, the average latency will be higher. This concludes that separating NGINX from another device is not boosting the performance but the success rate. The performance is worse because the request sent by the client has to go through the router, the NGINX device, and finally to the server.

Table 13. Overall Average Services Latency

Services	**Average Latency**		**Success Rate**	
	Monolith	***Microservice***	***Monolith***	***Microservice***
Merchant	663,22 ms	1325,56 ms	62,50%	70,95%
User	19612,00 ms	19640,50 ms	61,43%	56,68%
Transaction	1340,67 ms	864,67 ms	60,41%	63,37%
TOTAL AVERAGE	7205,30 ms	7276,91 ms	61,44%	63,66%

To summarize all of the tests conducted, the average latency and success rate of every service summed up to find out the overall average as seen in table 5.13. As seen in figure 5.14, in terms of average latency for every service, the monolith has better latency with a 71.61 ms gap difference. From figure 5.15, it could be seen that in terms of success rate, microservice slightly edge monolith average success rate with 2.22%.

Table 14. Overall Average Latency and Success Rate Every Test Design

Services	**Average Latency**			**Success Rate**		
	Design 1	***Design 2***	***Design 3***	***Design 1***	***Design 2***	***Design 3***
Merchant	1113,00	2060,33	3285,00	70,85	54,28	85,85
User	19319,00	18732,50	18043,00	56,80	56,75	56,60
Transaction	898,00	2949,33	4614,00	63,37	51,55	61,99
Total Average	7110,00	7914,06	8647,33	63,67	54,19	68,15

From table 5.14, it could be seen that testing design is affecting performance for each service.

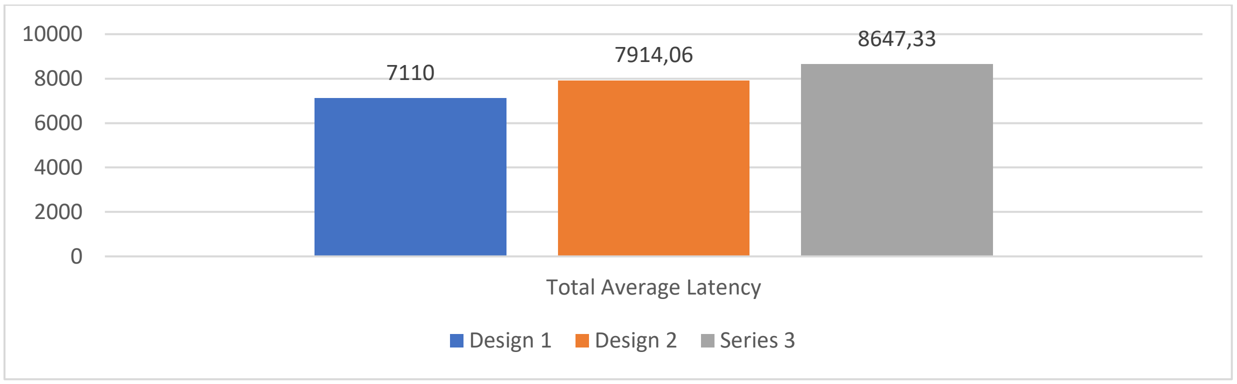


Figure 9. Testing Design Latency Comparison Chart

From figure 5.16 it could be concluded that testing design affects average latency where from the chart design 1 has the best average latency and it becomes worse on design 2 and design 3.

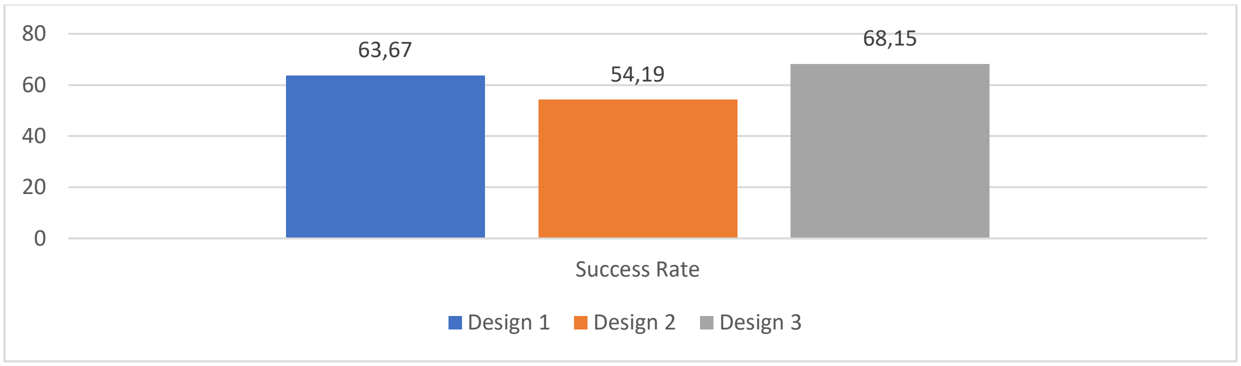


Figure 10. Testing Design Success Rate Comparison Chart

As seen in figure 5.17, the success rate is affected by the testing design. Unlike average latency, the success rate is slightly better on design 3 where NGINX is separated.

CONCLUSION

The test results obtained by testing both monolith and microservice with several scenarios are quite varied. From the testing, it could be concluded that the architecture affects the performance and it shows that:

1. In Merchant service, the average latency gap between microservice and monolith is 662,34 ms, where monolith average latency is only half of microservice's. In terms of success rate, microservice slightly edge monolith with 8,45% or 515 more successful request on microservice side. By looking at the number gap. It is clear that monolith is better for merchant service
2. In User service, monolith is better with edging microservice on both average latency and success rate with 28,5 ms lower latency and 4,75% more success rate.
3. In Transaction service, unlike user service, microservice performs better with having 494 ms lower latency and better success rate at 2,96%.

By looking at the overall testing data, the average latency from the monolith is better than the microservice by a 71.61 ms gap. Meanwhile, in terms of success rate, the data shows that microservices average success rate edge monolith with a small gap of 2.22 percentage. The latency average number result for microservice is slightly worse because there are more success requests and taking more time. The usage of the API gateway also affects the latency because the request has to go through NGINX before arriving on the server.

By changing the testing design, it could be concluded that the testing design is affecting the service's performance and it shows that:

1. Merchant service have the best latency on design 1 but the best success rate average on design 3 with 15% gap with the first one. But the latency on the third testing design is 3 times higher than first testing design
2. User service performs best on design 3 since it has lowest latency, eventhough the average success rate is worse but the gap only 0,15%.
3. Transaction service performs best on design 1 like merchant service since it has the lowest latency average and highest success rate.

From the overall data it shows that separating JMETER from another device makes the latency worse by 804.06 ms. But separating NGINX to the third device also did not make the latency better. It is shown that it has 1537,33 ms higher average latency than the first testing design. The average latency become worse because the request took longer times since the server is not in the local environment anymore. Meanwhile, in terms of success rate, from the testing data, the success rate on design 2 is slightly worse than design 1 with a 9.48% gap. On the other hand, testing design 3 has a slightly better success rate average with a 4.48% gap from design 1.

From the data it appears that there is no absolute better architecture. It could be concluded that every architecture has its own advantages. But from overall if the service is more loaded, need high chance of success rate, microservice is a better choice. If the service is light weight and need faster response, monolith is the option. In further research, the future researcher could use a better device for the server and testing. Also, the microservice could be made more complex by adding workers and increasing the number of features. It is also possible to use various database machines outside PostgreSQL with using various RDBMS for microservice.

REFERENCES

[1] K. Gos and W. Zabierowski, “The Comparison of Microservice and Monolithic Architecture,” in *2020 IEEE XVIth International Conference on the Perspective Technologies and Methods in MEMS Design (MEMSTECH)*, Lviv, Ukraine, Apr. 2020, pp. 150–153. doi: 10.1109/MEMSTECH49584.2020.9109514.

[2] A. Bucchiarone *et al.*, “From Monolithic to Microservices,” *IEEE SOFTWARE*, p. 6.

[3] V. Desai, “Microservices: Architecture and Technologies,” *IJRASET*, vol. 8, no. 10, pp. 679–686, Oct. 2020, doi: 10.22214/ijraset.2020.31979.

[4] D. Nevedrov, “Using JMeter to Performance Test Web Services,” p. 11.

[5] A. Neumann, N. Laranjeiro, and J. Bernardino, “An Analysis of Public REST Web Service APIs,” *IEEE Trans. Serv. Comput.*, vol. 14, no. 4, pp. 957–970, Jul. 2021, doi: 10.1109/TSC.2018.2847344.

[6] F. Schmager, N. Cameron, and J. Noble, “GoHotDraw: evaluating the Go programming language with design patterns,” in *Evaluation and Usability of Programming Languages and Tools on - PLATEAU ’10*, Reno, Nevada, 2010, pp. 1–6. doi: 10.1145/1937117.1937127.

[7] S. Sulander, “Microservices Architecture in Open Retail Interface for Public Transport Tickets,” *Distributed systems*, p. 60.

[8] N. Viennot, M. Lécuyer, J. Bell, R. Geambasu, and J. Nieh, “Synapse: a microservices architecture for heterogeneous-database web applications,” in *Proceedings of the Tenth European Conference on Computer Systems*, Bordeaux France, Apr. 2015, pp. 1–16. doi: 10.1145/2741948.2741975.

[9] M. Kalske, N. Mäkitalo, and T. Mikkonen, “Challenges When Moving from Monolith to Microservice Architecture,” in *Current Trends in Web Engineering*, vol. 10544, I. Garrigós and M. Wimmer, Eds. Cham: Springer International Publishing, 2018, pp. 32–47. doi: 10.1007/978-3-319-74433-9_3.

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

[11] O. Al-Debagy and P. Martinek, “A Comparative Review of Microservices and Monolithic Architectures,” in *2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI)*, Budapest, Hungary, Nov. 2018, pp. 000149–000154. doi: 10.1109/CINTI.2018.8928192.