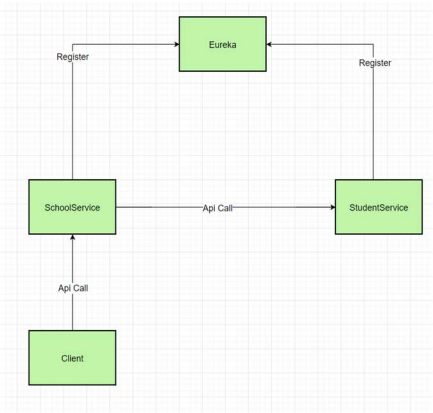
Performance Evaluation of Microservices-based Application: A Comparative Study of Deployment Locations, Communication Patterns, and Virtualization Techniques

(1) Original design:

We choose Eureka as our Microservices application tool. It is a widely used open-source service discovery tool developed by Netflix, which is used for managing microservice architecture. We choose to use Spring Boot with Eureka because they together provide a lightweight and easy-to-use framework for building microservice for service registration and discovery.

It is often difficult to maintain individual Microservice's addresses in a large distributed system, and our system intends to solve this issue. The Eureka service registry provides a lookup service where microservices can register themselves and discover other registered microservices.

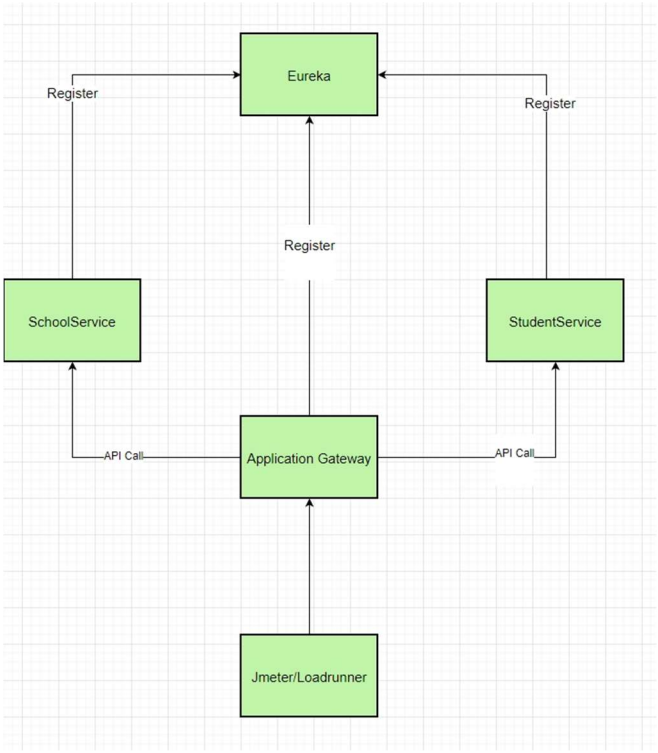
Our system design includes one server and two services. The Eureka Server is a service registry that enables microservices to locate and communicate with each other. TheStudentService provides the student list for a specific school. The SchoolService provides school detail (i.e. school name, student list) through REST API for the client. The communication between components is illustrated in the below diagram.



Refined Solution Design:

Our design still includes the Eureka Server as a service registry (name server) and two microservices: StudentService and SchoolService. In the initial design, the StudentService provides the student list for a specific school, and the SchoolService provides school details (e.g. school name, student list). A typical use case is when a client queries information about a school, which is done by calling the SchoolService, and the SchoolService calls StudentService to get the student list.

However, the initial design of the service layout creates a chaining of API calls when the SchoolService calls StudentService. The ideal design promotes loose coupling between services. To improve the design, we created an Application Gateway. The SchoolService and StudentService microservices are now called separately. The refined design diagram is illustrated below:



(2) Implementation:

Progress made in report 1:

We have accomplished below tasks:

- Refined and deployed the demo application through the local image and set up the code repository to:

https://github.com/cl456852/eureka

- Set up AWS EC2 testing environment
- Set up demo services through local deployment and ran test API calls
- Got familiarized with evaluation tools, Jmeter and Loadrunner

In the next stage, we will continue with below tasks:

- Write Dockerfiles for the server, school and student services, and push the image to ECR
- Compare two deployment methods through AWS ECS and Docker build
- Add feature functions to the demo application to assist with latency test
- Run performance test following evaluation methodology and analyze results

Progress made in report 2:

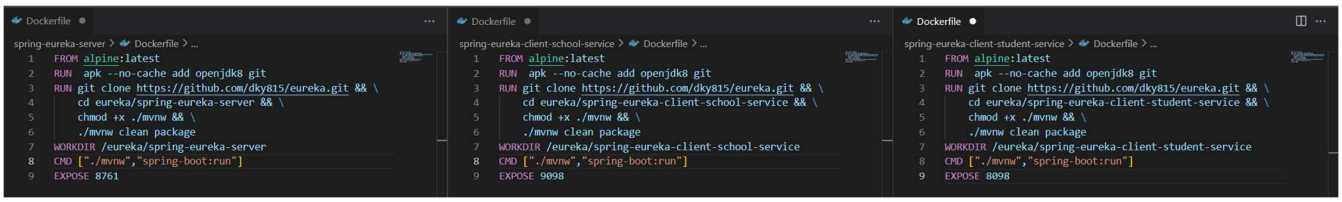
In the last two weeks, we have done below:

- Implement Application Gateway module.
- Compare two deployment methods through AWS ECS and Docker build.
- Research and implement multi-location deployment on AWS.
- Research and implement Message Queue on AWS.
- Run performance tests following evaluation methodology and analyze results.

Details are recorded below:

1.Deployment

- Write Dockerfile for the application



Dockerfile
spring-eureka-server > Dockerfile > ...

```
FROM alpine:latest
RUN  apk --no-cache add openjdk8 git
RUN git clone https://github.com/dky815/eureka.git && \
    cd eureka/spring-eureka-server && \
    chmod +x ./mvnw && \
    ./mvnw clean package
WORKDIR /eureka/spring-eureka-server
CMD ["./mvnw","spring-boot:run"]
EXPOSE 8761
```

Dockerfile
spring-eureka-client-school-service > Dockerfile > ...

```
FROM alpine:latest
RUN  apk --no-cache add openjdk8 git
RUN git clone https://github.com/dky815/eureka.git && \
    cd eureka/spring-eureka-client-school-service && \
    chmod +x ./mvnw && \
    ./mvnw clean package
WORKDIR /eureka/spring-eureka-client-school-service
CMD ["./mvnw","spring-boot:run"]
EXPOSE 9098
```

Dockerfile
spring-eureka-client-student-service > Dockerfile > ...

```
FROM alpine:latest
RUN  apk --no-cache add openjdk8 git
RUN git clone https://github.com/dky815/eureka.git && \
    cd eureka/spring-eureka-client-student-service && \
    chmod +x ./mvnw && \
    ./mvnw clean package
WORKDIR /eureka/spring-eureka-client-student-service
CMD ["./mvnw","spring-boot:run"]
EXPOSE 8098
```

Tried different deployment methods on:

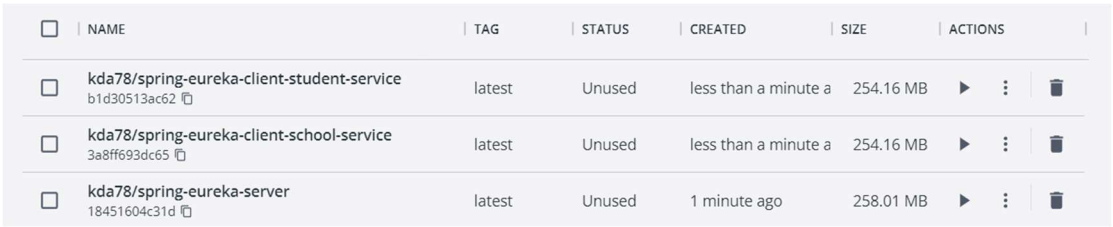
Change to EC2 host networking

Switch between using public and private IP addresses

Change to using Amazon ECS services and tasks

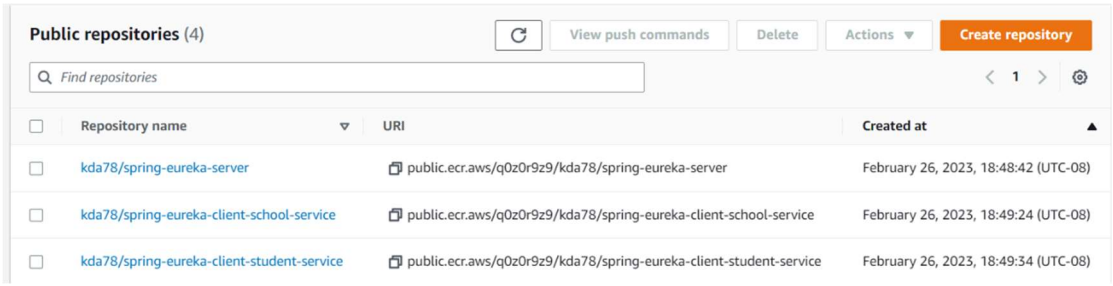
- Build docker image & push image to ECR

1. Build docker image locally



NAME	TAG	STATUS	CREATED	SIZE	ACTIONS
kda78/spring-eureka-client-student-service b1d30513ac62	latest	Unused	less than a minute a	254.16 MB	
kda78/spring-eureka-client-school-service 3a8ff693dc65	latest	Unused	less than a minute a	254.16 MB	
kda78/spring-eureka-server 18451604c31d	latest	Unused	1 minute ago	258.01 MB	

2.Create public repo in ECR



Public repositories (4)

View push commands | Delete | Actions | Create repository

Find repositories

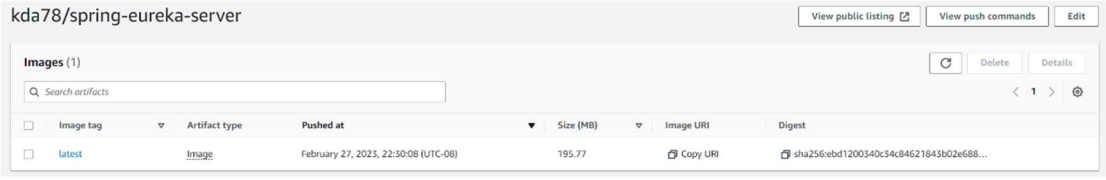
1

Repository name	URI	Created at
kda78/spring-eureka-server	public.ecr.aws/q0z0r9z9/kda78/spring-eureka-server	February 26, 2023, 18:48:42 (UTC-08)
kda78/spring-eureka-client-school-service	public.ecr.aws/q0z0r9z9/kda78/spring-eureka-client-school-service	February 26, 2023, 18:49:24 (UTC-08)
kda78/spring-eureka-client-student-service	public.ecr.aws/q0z0r9z9/kda78/spring-eureka-client-student-service	February 26, 2023, 18:49:34 (UTC-08)

3. Tag docker image

NAME	TAG	STATUS	CREATED	SIZE	ACTIONS
public.ecr.aws/q0z0r9z9/kda78/spring-eureka-serve 18451604c31d	latest	Unused	4 minutes ago	258.01 MB	
public.ecr.aws/q0z0r9z9/kda78/spring-eureka-client b1d30513ac62	latest	Unused	3 minutes ago	254.16 MB	
public.ecr.aws/q0z0r9z9/kda78/spring-eureka-client 3a8ff693dc65	latest	Unused	3 minutes ago	254.16 MB	

4. Push docker image to ECR



kda78/spring-eureka-server

View public listing | View push commands | Edit

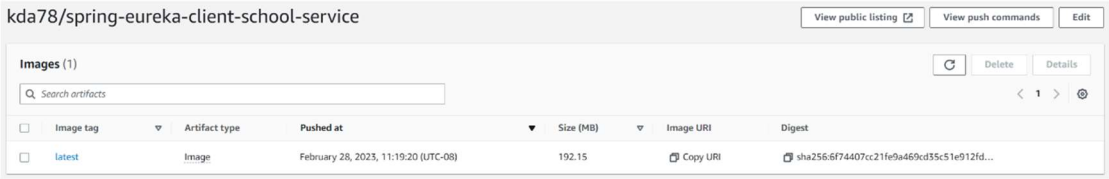
Images (1)

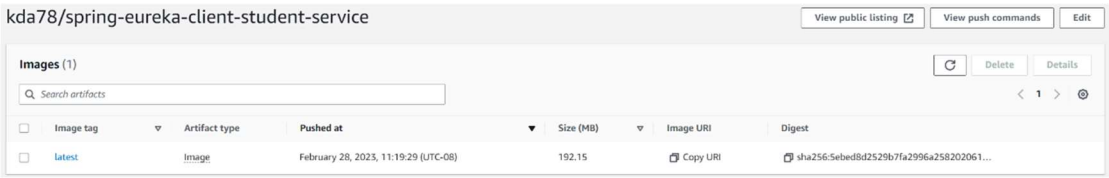
Delete | Details

Search artifacts

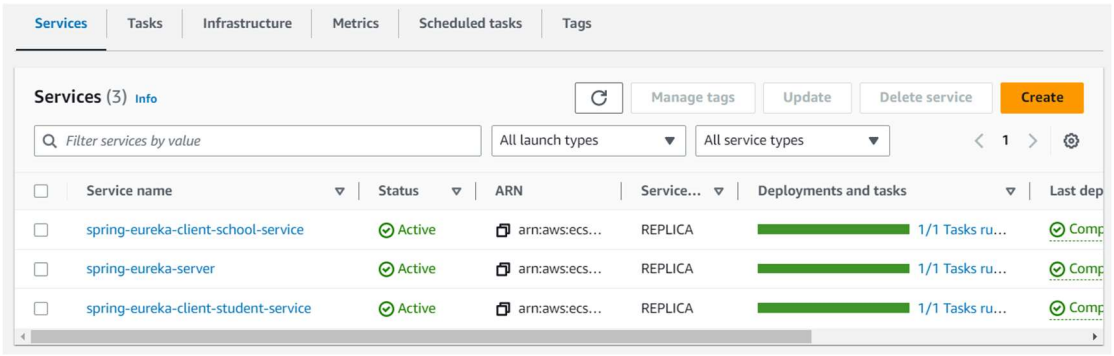
1

Image tag	Artifact type	Pushed at	Size (MB)	Image URI	Digest
latest	Image	February 27, 2023, 22:30:08 (UTC-08)	195.77	Copy URI	sha256:ebd1200340c34c84621843b02e688...

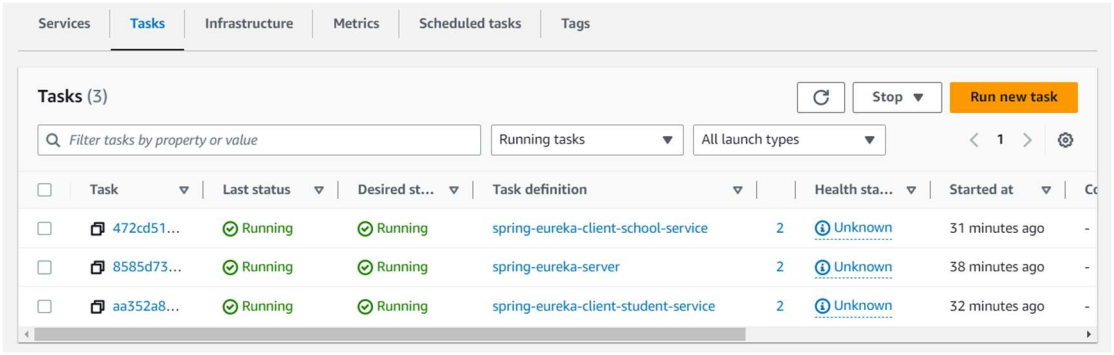




- Deploy docker image to ECS
  1. Create cluster using Amazon EC2 instances
  2. Create services in cluster



  3. Run tasks

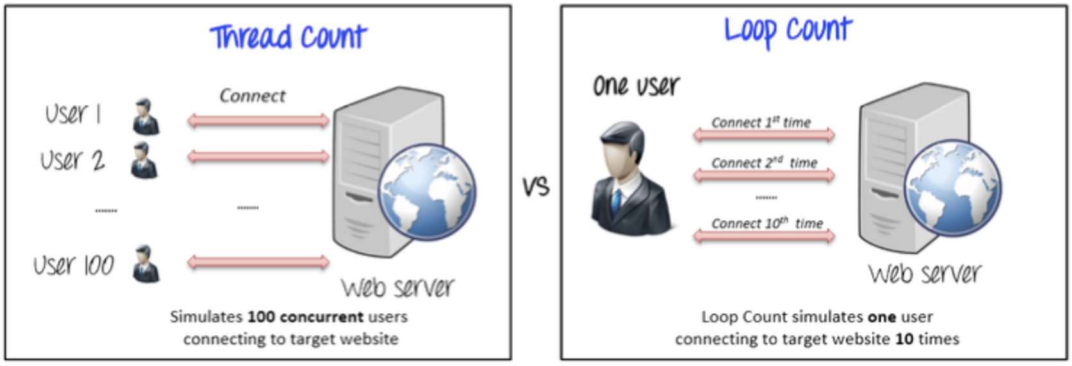


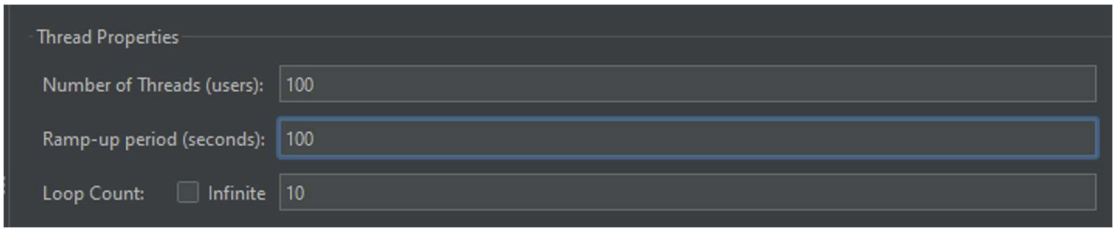
2. JMeter Research

In our test, JMeter creates 100 concurrent users. These users will be accessing the web service all at once. We have defined the Loop Count as 10, so that means every user made by JMeter would be connecting to the server 10 times.

1. Number of Threads (or Users): The number of users to model accessing the application simultaneously. Set the value to 100, because we want to model 100 users on our sample service.
2. Loop Count: The number of times JMeter simulates a user. We will set this to 10.
3. Ramp-Up Period: The time it takes in seconds for JMeter to model a new user. We'll set

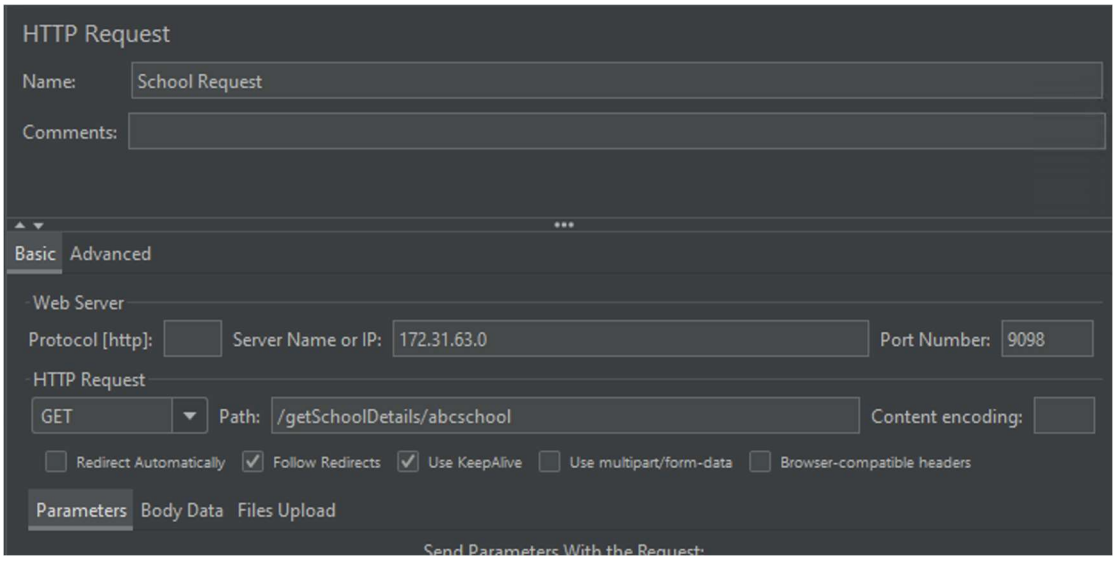
this time to 100 seconds.



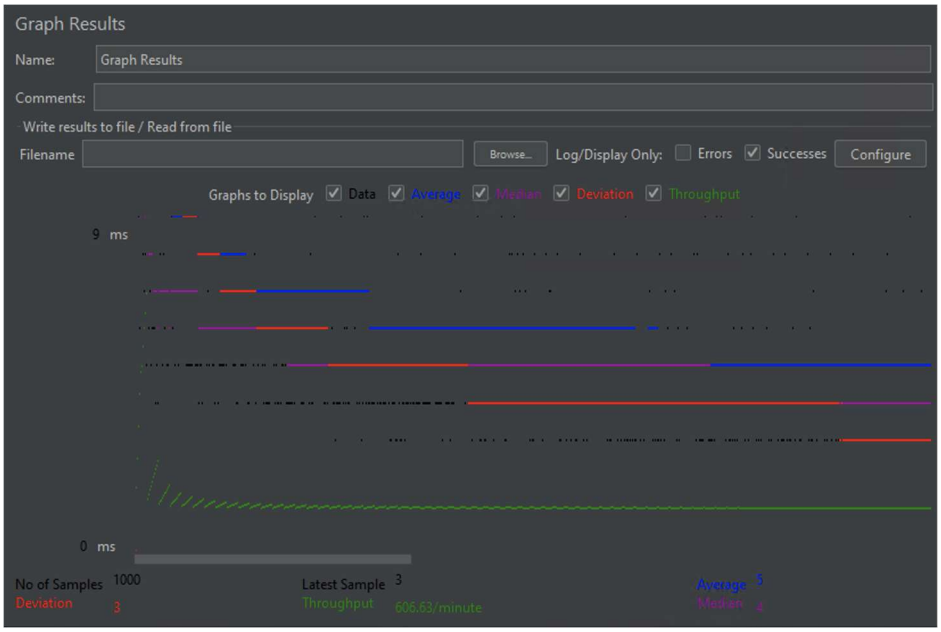


Set up HTTP request:

- IP: JMeter server and our service are located in the same subnet. So, we could directly access the service by using its private IP.
- Port number: 9098
- Request: GET
- Path: /getSchoolDetails/abcschool

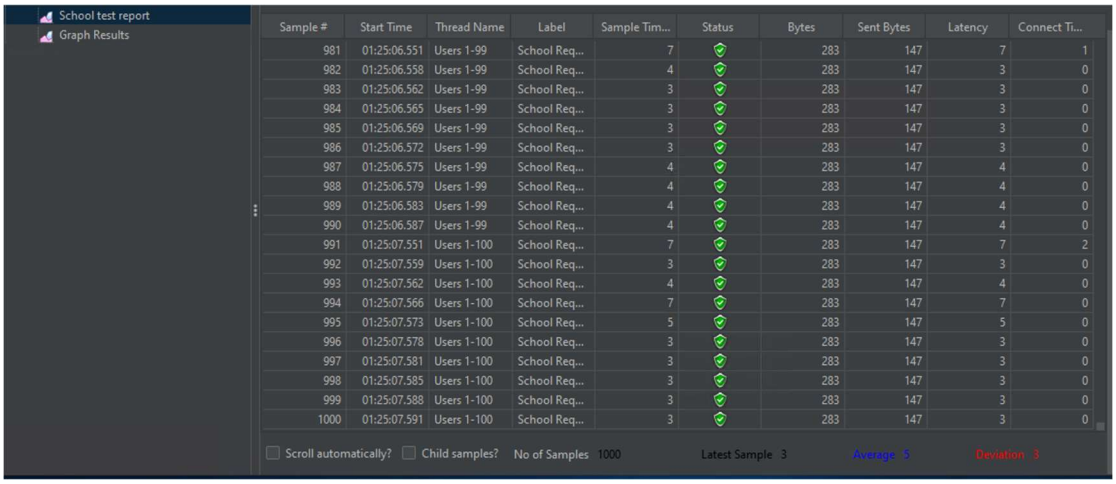


Test result:



The throughput is 606.63 requests per minute. This means that our service handled 867 requests per minute. It's important to note that throughput depends on other factors, such as internet speed, the server's current load, and CPU power. These factors continuously change, meaning you won't get the same results every time you run the test.

This could be due to JMeter, and server are in different networks; or the server is in regions of other countries, these changes will greatly affect throughput values. We will have corresponding tests in the future to approve this assumption.



In this case, the average latency for these 1000 requests is 5 milliseconds.

(3) Evaluation Plan:

- Identify test scenarios, which include the instances usage, communication pattern, and resource placement
- Set up the test environment by creating virtual or containerized instances and record

the number of instances, the CPU and memory allocations. Also, the network bandwidth will be configured based on the test scenarios

- Perform evaluation using Jmeter to test the performance of each scenario
- Analyze the result to determine the impact of resource allocation, placemen etc. on the performance of the microservices architecture. The following metrics will be used for analysis: Latency, Throughput, Error Rate, CPU Usage, Memory Usage.