Impact of Instance Choice, Placement, and Communication of Microservice Architecture on Cloud Performance

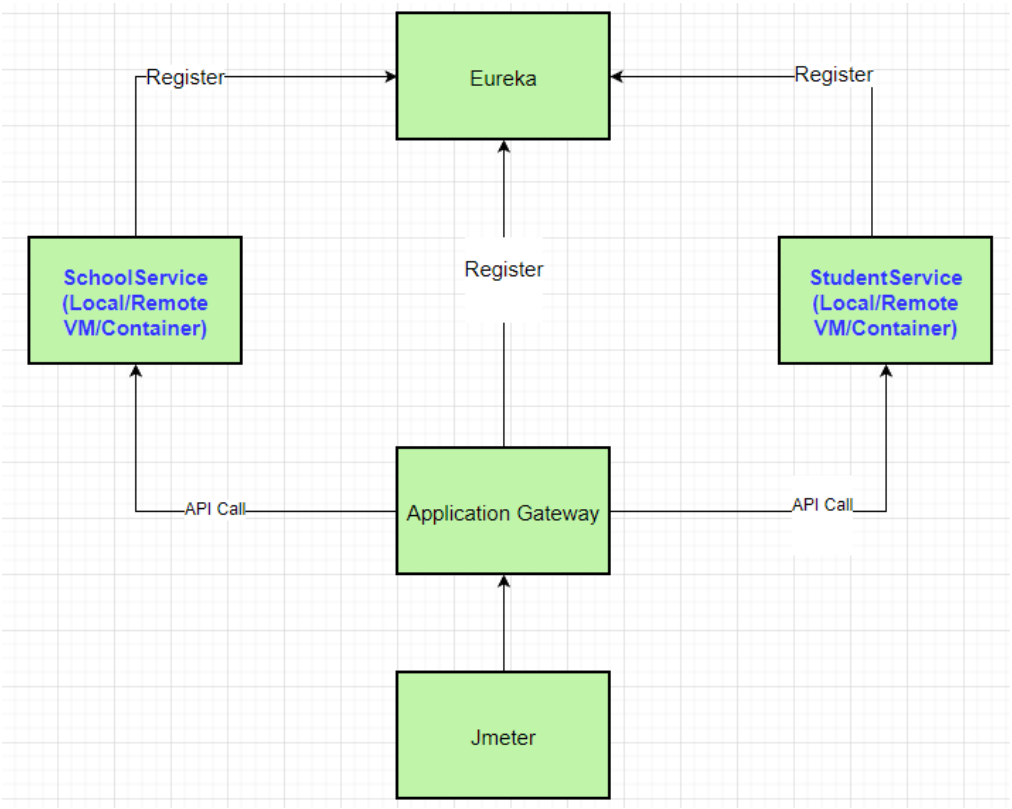
Group 19: Jason Chen, Hugh Song, Karry Dong, River Chen, Allen Liu

System design

Group 19 aims to quantify the impacts of resources regarding placements, deployment methods (VM or container), and communication patterns on their performance in microservice.

We chose a simple microservice demo to test the performance since it excludes other factors that could interfere with the result. The main functionality of the demo is to provide information on schools and students based on the requested school name.

Communication Pattern One (API call)



Eureka: a service registry and discovery tool that allows microservices to locate and communicate with each other.

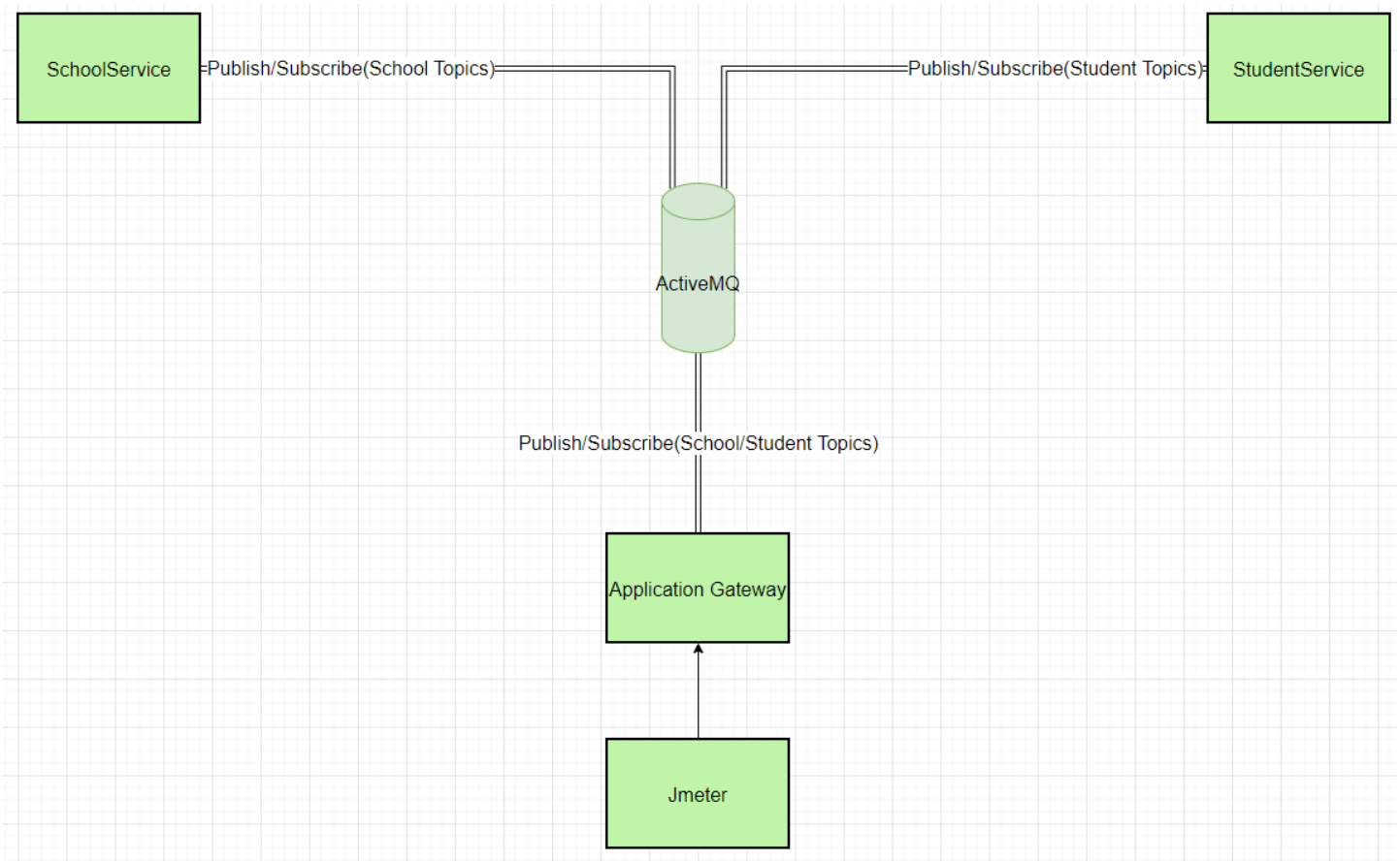
School Service: a microservice that provides school information based on the requested school name.

Student Service: a microservice that provides student information based on the requested school name.

Application Gateway: a gateway service that uses API to call School and Student Service, combine the data, and return it to the client.

Jmeter: performance test tool.

Student and School services can be deployed in the same or different zones as the Application Gateway. They also can be deployed in traditional VM or containers. We compare the performance of different deployment methods.



Communication Pattern Two (queue)

In this structure, instead of using Eureka, we use ActiveMQ to let the microservices communicate with each other. School Service, Student Service, and Application Gateway publish and subscribe to certain Topics, sending and receiving messages to communicate. We compare this communication pattern to the API call style in Communication Pattern One on performance.

Implementation

The demo is based on Spring Boot written in Java, and we deploy it on AWS EC2 and ECS(docker). We use ActiveMQ in one of the communication patterns. Also, we use Jmeter to test the performance.

In Communication Pattern One, School and Student Service register themselves to the Eureka and provide HTTP API to provide service for the Application Gateway. In Communication Pattern Two, School Service, Student Service, and Application Gateway publish SchoolResp, StudentResp, SchoolReq/StudentReq topics correspondingly and subscribe SchoolReq, StudentReq, SchoolResp/StudentResp correspondingly.

Implementation Challenge

In Communication Pattern Two, the Application Gateway needs to wait for response messages from both School and Student Service. Because the message queue communication pattern is synchronized, we need to transform it into synchronized communication. After study and research, we used **'CountDownLatch'** in Java to simulate the asynchronized communication as synchronized, waiting for both messages from School and Student Service and returning it to the client.

Results and Measurements:

The measurement method is to use JMeter to create 100 concurrent users to call the Application Gateway. We set the loop count to 10, which means every user made by JMeter sent request ten times. This resulted in a

total of 1,000 requests. Since the VM that the JMeter tool runs on and the Application Gateway server VM is located in the same subnet, we can use Application Gateway VM's private IP for access. We are making the API calls using HTTP GET requests.

We conducted tests with four different cases. The cases differ in the location of the Virtual Machines (VMs), deployment methods (through AWS ECS or Docker build), and communication patterns (API or queue). The main measurement methods are latency and throughput. The test results are shown in the below diagram:

Case	Deployment method	VM Location	Communication Pattern	Min latency (ms)	Max latency (ms)	Throughput (requests per minute)
1	AWS EC2	2	API	126	262	599.778
2	AWS EC2	1	API	4	42	605.914
3	AWS(Docker)	1	API	4	137	607.33
4	AWS EC2	1	Queue	16	145	604.906

Analysis of results:

The results presented in the table appear to be consistent with the general understanding of the factors affecting latency and throughput. The deployment method, VM location, and communication pattern can all have an impact on performance.

The VM location plays a significant role in latency. Cases 2, 3, and 4, which all use VMs in the same location (US-EAST-1), have lower latencies compared to Case 1, which uses VMs in two locations (US-EAST-1 and US-WEST-2). This suggests that the average latency decreases as the instances are located closer to each other.

The use of AWS ECS as a deployment method in Case 2 may provide advantages such as scalability and better resource management. The Docker deployment in Case 3 has slightly higher latency while the throughput remains about the same.

Lastly, the communication pattern affects the latency and throughput as well. The use of a queue-based communication pattern in Case 4 results in higher latency compared to the API-based communication in Case 2, despite both cases using AWS ECS and VMs in the same location. The maximum latency for queue communication is much higher than for API communication. The throughput for both communication patterns is similar.

The argument of correctness can be made based on the 0% error rate in all cases, which indicates the reliability of the system. Throughput remains relatively consistent across all cases, ranging from 599.778 to 607.33 requests per minute.

In conclusion, the data demonstrate the influence of instance placement, deployment method, and communication patterns on latency and throughput. These insights can help inform future optimizations and design decisions for the system.

Project on GitHub: https://github.com/cl456852/eureka/tree/master