CMPT 756

Distributed and Cloud Systems

Final Project Report

Team Name	Happy go go 756			
CourSys team URL	https://coursys.sfu.ca/2022sp-cmpt-756-g1/groups/g-Happy_go_go_756			
GitHub project repo URL	https://github.com/scp756-221/term-project-cloud_sprints			
Team Member Name	Wanyi Su	Wei Zhang	Yin Yu Kevani Chow	Zeyu Hu
Email (SFU)	wsa50@sfu.ca	wza105@sfu.ca	yyc15@sfu.ca	zha76@sfu.ca
Github ID	suansuan0915	joannzhang00	yyc15	zha76

Table of Content

Introduction

Parallel, heterogeneous and distributed hardware has become the mainstream for a few years. The way we write, ship, and maintain software today has evolved drastically in the last few years (Saleem, 2020). The significant transition from hardware to virtual machine is followed by the other one to containers. It makes it essential for developers/teams to be able to manage containerized workloads and services within a supportive and extensible ecosystem such as Kubernetes. Under such a background, we are going to apply and observe various ideas including Scrum methodology, containerized application, microservices distributed system architecture pattern and so on. Considering that containers are quite useful but can also be challenging to scale, we leverage the platform AWS Elastic Kubernetes Service in this project. We aim to gain a complete perspective to work on distributed application and practical experience via building, scaling, observing and reflecting on the process. The project is built and developed on the distributed system involving various operations as well as failure exploration. Our target is to practically summarize, learn and reflect on principles including scrum methodology, distributed source control system, microservices and a variety of tools.

Summary of Application

The application incorporates three services: Music, Playlist, Database. The following table shows the microservice design. The Music and Database services come from our course CMPT756. The application stores and retrieves songs and the artists who performed them. As it does originally, there are three basic operations: create, read and delete for the Music service. Each song is identified by a system-defined universal ID. We add an additional service called Playlist on the above basis. It enables the client side to create playlists based on accumulated songs to the time. There are four basic operations: create, edit, read and delete. The Database is the interface to the key-value store which we are using DynamoDB managed by Amazon.

Service	**Purpose**	**Location**
Users	Lists of user names & emails	Kubernetes cluster on AWS
Playlist	Lists of playlist & playlist name	Kubernetes cluster on AWS
Music	Lists of songs and their artist	Kubernetes cluster on AWS
Database	Interface to key-value store	Kubernetes cluster on AWS
DynamoDB	Key-value store	Service managed by Amazon

The microservice design

users							
Column Name				**Data Type**			
fname	lname	email	UUID	string	string	string	UUID

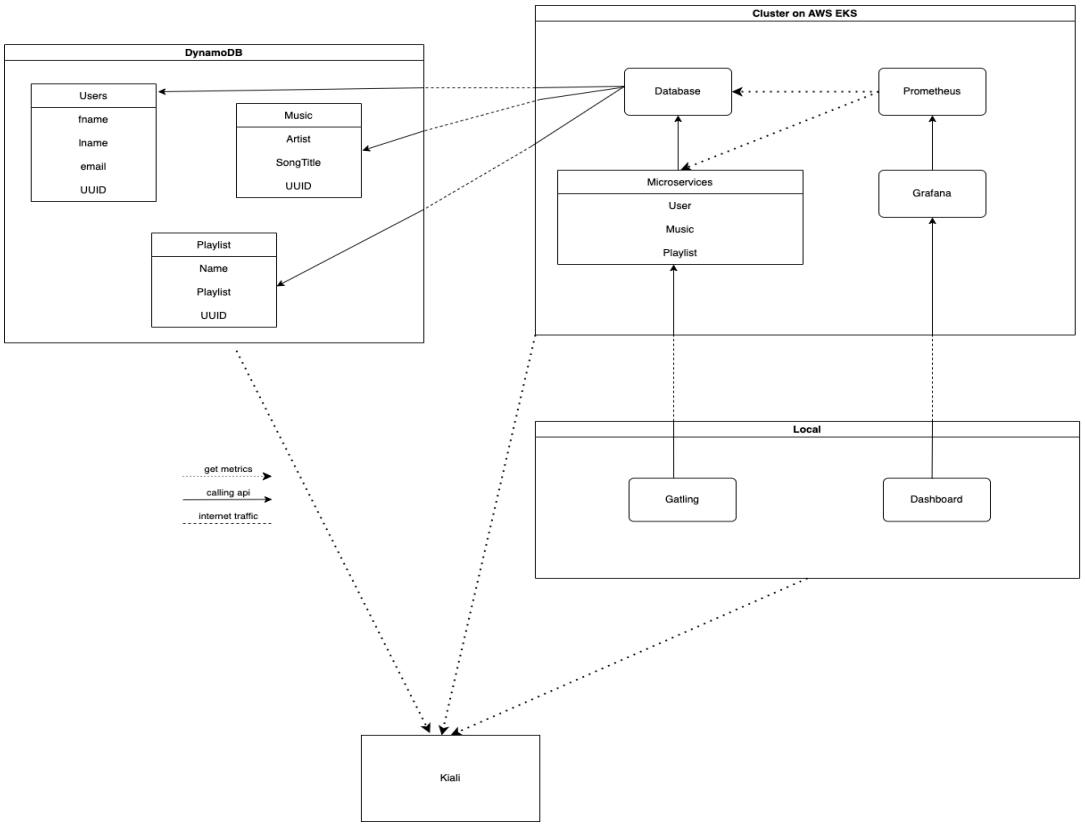
music					
Column Name			**Data Type**		
Artist	SongTitle	UUID	string	string	UUID

playlist					
Column Name			**Data Type**		
Name	Playlist	UUID	string	string	UUID

Table structure

Deployment Architecture

Technical Framework



In this project, we will run our application on a cluster. The platform used in this project is AWS EKS, which can run kubernetes to manage the containerized application. The music application we build consists of at least 3 services, including music service, user service and playlist service.

In addition to that, fast NoSQL key-value database AWS DynamoDB is used to store all the data we have at AWS side. And our 3 services will obtain and store data at the database. For example, the playlist service will get the list of songs from DynamoDB when making a request from the local machine. Also, it will store a table containing playlist ID and the corresponding list of songs.

Besides, Prometheus, a system with a dimensional data model, flexible query language, efficient time series database and modern alerting approach is utilized to monitor our application. From which, Grafana, a multi-platform open source analytics and interactive visualization web application can get the corresponding data, and then presents those analytics graphs to the dashboard on the local machine. It will display the useful visualization and give people deeper insight about the services and database in our application.

Another tool we are going to use on the local machine is gatling, which is a load testing tool for web applications, designed for DevOps and Continuous Integration. It will help us to test our services more scientifically.

Project Methodology

Agile Scrum Framework

In this project, an agile scrum framework for project development is adopted. The following scrum ceremonies have been arranged to be happened in the following days of week:

Scrum ceremonies

Day of week	Time	Ceremony
Monday to Friday	1:00pm - 1:15pm	Daily Scrum
Monday	1:15pm - 2:15 pm	Sprint Planning
Thursday	1:00pm - 2:00pm	Sprint Review
Friday	1:00pm - 2:00pm	Sprint Retrospective

Since it is a relatively tight project schedule, there are four sprints and each sprint is one week. The schedule and sprint goals are listed as the following. The sprint goal is updated accordingly in every sprint planning.

Schedule and Sprint Goals

Sprint	Date	Goal
1	14th - 18th March	Setup environment, decide micro services function, complete interim report
2	21st - 25th March	Develop micro services and testing
3	28th March - 1st April	Develop micro services and testing
4	4th -8th April	Complete Final Report, record project demo and presentation video

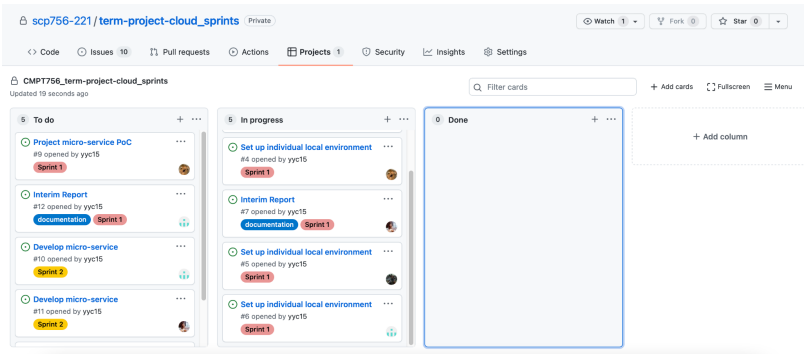
Definition of Done

The definition of done can be divided into two categories: Documentation and development.

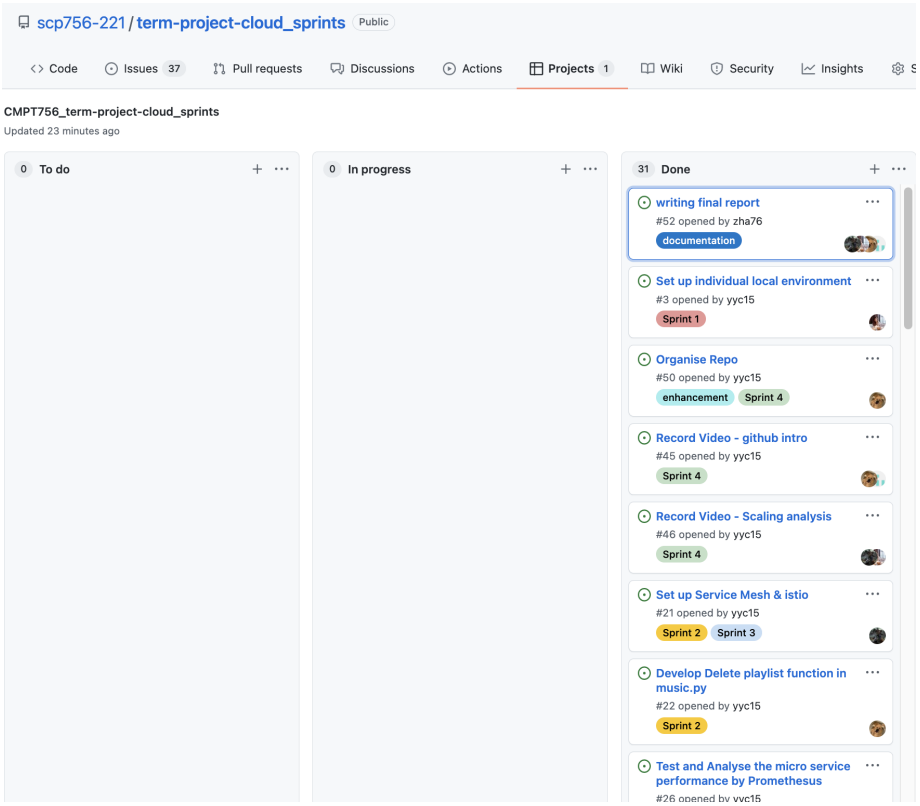
For documentation, the definition of done is that all the assigned sections are completed with grammatically-correct complete sentences, graphs and data tables. For development, the developed codes can be run locally, are integration tested with codes in the main branch as well as committed and pushed to the main branch.

Kanban Board

In order to keep the transparency of the project progress, a project in github using basic Kanban board style with statuses include “To-do”, “In-progress” and “Done” is created. Issues are created and assigned to each team member during every sprint planning. Here is the link and screen shot of the Kanban board. Progress of each issue is updated and reported during daily stand-up scrum meetings. Now at the end of the project, all tasks are done.
Kanban board link: https://github.com/scp756-221/term-project-cloud_sprints/projects/1
Here is Kanboard board of sprint 1:



After last sprint review we have completed all the tasks:



Github Guide

Structure of the repository

Developing microservices using a highly decoupled design

- ci: This directory contains the continuous integration (CI) framework. It includes the files used by GitHub Actions to accomplish CI, the files defining different versions of the tests, as well as other files useful for locally running CI tests.
- cluster: This directory is to configure the cluster, and should be treated with extra care such as not removing any lines from the .gitignore file. Otherwise the secret key might be uploaded publicly.
- db: This is the database service. The current version uses Amazon DynamoDB. This could be replaced with another service, such as MongoDB without changing the higher-level services S1 (User) and S2 (Music) and S3 (Playlist), which are insulated from the underlying storage service by this layer.
- gatling: This is for gatling simulation and tables. It contains the simulation script in scala and the csv files that stores the data of the three services.
- loader: This utility loads the DynamoDB tables, using the files users.csv, music.csv, playlist.csv from the Gatling resources directory.
- logs: This folder stores the generated logs.
- mcli: It contains the CLI for the music microservice.
- s1: The user service maintains a list of users and passwords. In a more complete version of the application, users would have to first log in to this service, authenticate with a password, be assigned a session, then present that session token to the music service for any requests.
- s2: The music service maintains a list of songs and the artists that performed them.
- s3: The playlist service maintains lists of songs, which is the playlist and the name of the playlist.
- tools: It stores scripts such as that to start docker and the services.

Run CI locally

```
test        | ============================== 2 passed in 0.90s ===============================
test        | /code/conftest.py:182: ResourceWarning: unclosed <socket.socket fd=9, family=AddressFamily.AF_INET,
type=SocketKind.SOCK_STREAM, proto=6, laddr=('172.18.0.7', 49548), raddr=('172.18.0.3', 8000)>
test        |   create_tables.create_tables(
test        | ResourceWarning: Enable tracemalloc to get the object allocation traceback
```

Command lines required:

1. The file input to docker-compose is templated. The reference version is compose-tpl.yaml. This file in the repo has already been modified, run directly

```
/home/k8s# make -f k8s-tpl.mak templates
```

to regenerate the compose.yaml file that will actually define the test.

2. run the CI tests locally via the runci-local.sh script:

```
/home/k8s # cd ci
/home/k8s/ci# ./runci-local.sh v1.1
```

Services Deployment Steps & Tools

Command lines required

```
#tools/shell.sh
/home/k8s# make -f k8s-tpl.mak templates
/home/k8s# make -f k8s.mak
/home/k8s# make -f eks.mak start
/home/k8s# aws dynamodb list-tables
/home/k8s# aws cloudformation delete-stack --stack-name db-awsID
/home/k8s#aws cloudformation create-stack --stack-name db-awsID --template-body file:///home/k8s/cluster/cloudformationdynamodb.json
/home/k8s# aws dynamodb list-tables
/home/k8s# kubectl config use-context aws756
/home/k8s# kubectl create ns c756ns
/home/k8s# kubectl config set-context aws756 --namespace=c756ns
/home/k8s# kubectl config use-context aws756
/home/k8s# istioctl install -y --set profile=demo --set hub=gcr.io/istio-release
/home/k8s# kubectl get svc --all-namespaces | cut -c -140
/home/k8s# make -f k8s.mak cri
```

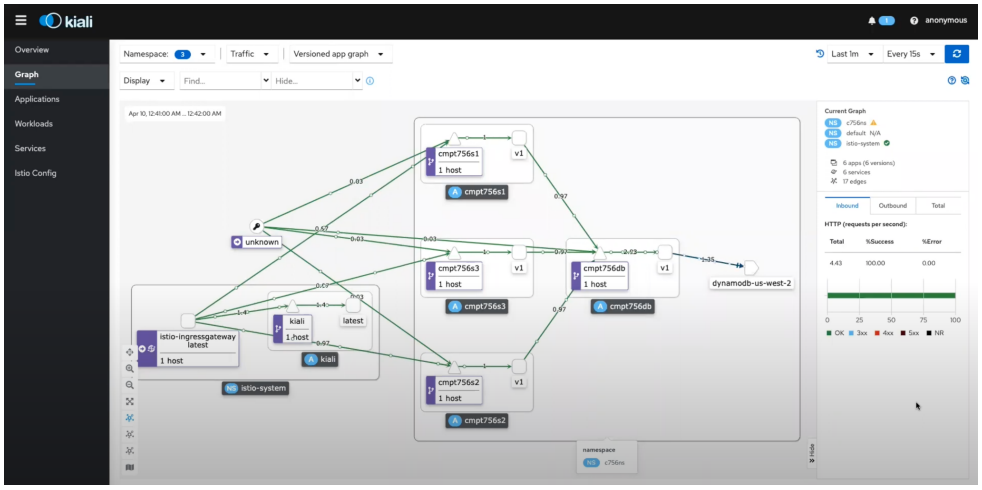
```
/home/k8s# make -f k8s.mak gw db s2
/home/k8s# make -f k8s.mak loader
/home/k8s# kubectl logs --selector app=cmpt756s1 --container cmpt756s1 --tail=-1
/home/k8s# kubectl logs --selector app=cmpt756s2 --container cmpt756s2 --tail=-1
/home/k8s# kubectl logs --selector app=cmpt756s3 --container cmpt756s3 --tail=-1
/home/k8s# kubectl logs --selector app=cmpt756db --container cmpt756db --tail=-1
/home/k8s# make -f k8s.mak provision
/home/k8s# make -f k8s.mak grafana-url
/home/k8s# make -f k8s.mak kiali-url
/home/k8s# make -f k8s.mak prometheus-url
#chmod u+x gatling-*
#./gatling-1-user.sh
#./gatling-1-music.sh
#./gatling-1-playlist.sh
/home/k8s# kubectl -n c756ns scale deployment.apps/cmpt756s1 --replicas=10
/home/k8s# ./gatling-100-user.sh
/home/k8s# kubectl -n c756ns scale deployment.apps/cmpt756s2-v1 --replicas=10
/home/k8s# ./gatling-100-music.sh
/home/k8s# kubectl -n c756ns scale deployment.apps/cmpt756s3 --replicas=10
/home/k8s# ./gatling-100-playlist.sh
/home/k8s# kubectl -n c756ns scale deployment.apps/cmpt756db --replicas=10
```

Observing a system at load using a variety of tools

We utilized Grafana, Kiali and Prometheus to observe our system. We have used grafana to observe our 4 services which are service 1 (user), service 2 (music), service 3 (playlist) and database service. Beside memory, cpu usage, we can clearly see success/error requests per second and the 4 corresponding lines for total request per minute, average/median/90th percentile response time, as well as request under 100ms. Four different colors of lines show the corresponding status.
We have used grafana to observe our 4 services which are service 1 (user), service 2 (music), service 3 (playlist) and database service. Beside memory, cpu usage, we can clearly see success/error requests per second and the 4 corresponding lines for total request per minute, average/median/90th percentile responding data.
We have used Prometheus to observe the requests of our 4 services and the status code. We can further navigate the status of each request within 5 minutes.
We have used Kiali to observe the connections between each service and the traffic flow. We can also check the error/success rates or we can see whether it is successful from colors.

Scaling and Failure Analysis

Scenario 0: Operating at a stable load



First, we add 1 user per service in Gatling simulation, so there are 3 users in total (1 user per service) running.

Metrics	Observation & Analysis
Performance of microservices	Total requests/min for each service is around 57. The line of successful requests/s shows all requests from users are successful and stable with no error shown.

Latencies of the microservices	The stable 100% values of all requests under 100ms indicate that the application's performance to meet Service Level Objective (SLO) is quite good. 90th percentile response time is minimal at 40ms for s1, s2, and s3, which indicates low latencies of the current state of application.
System utilization	Memory utilization is stable at a low level of around 140MB and CPU utilization is also quite low at 3%. The current observation indicates low utilization that there is an over-provisioning of capacity for the workload since we have only 3 users now.
Health of System Mesh (Kiali)	Traffic flows between each node is fluent and all lines are green, where the success rate of requests/s is 100% with a total request/s is 104.66.

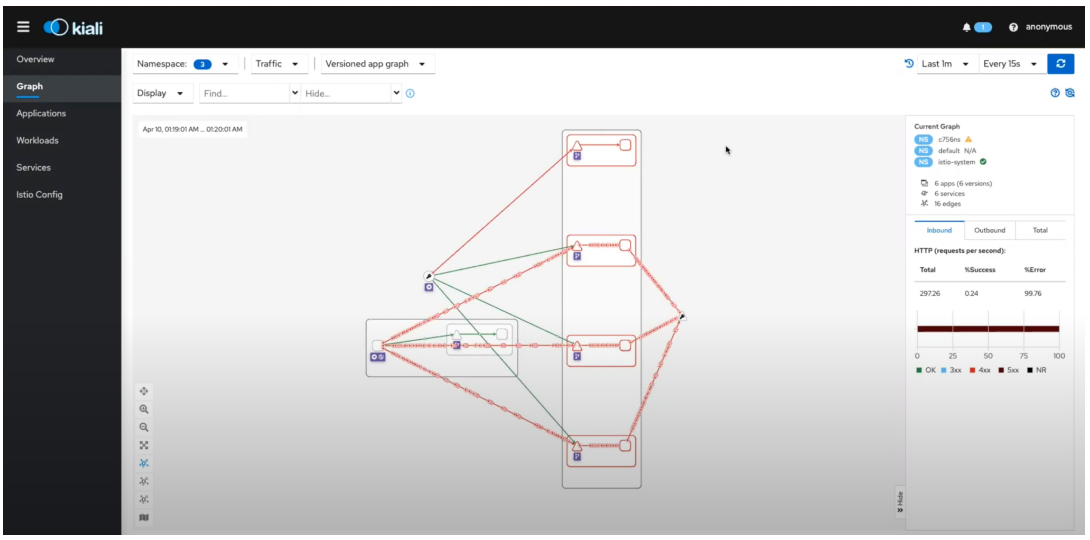
Scenario 1: scaling users to 300

We add 100 more users per service, replicas = 1, worker-node = 2, DB capacity = 5, so 300 users in total, with 100 users per service (excluding initial 3 users since the number is insignificant) are running.

Metrics	Observation & Analysis
Performance of microservices	Bars of total requests/min starts to grow higher since 300 users are added. Successful requests/s values become volatile and the error rate red line starts to climb.
Latencies of the microservices	The percentages of all requests under 100ms fall dramatically. Average and median response times of s1, s2, and s3 increase from the mini-second level to the second level. Also, the 90th %tile response time increases. This indicates the services' latencies become longer due to the increase of users and requests.
System utilization	Compared to the initial 3-user case, memory usage rises by 33%, CPU utilization increases multiple times to around 130%, and the values of process open fds also increase 15 times. This surge indicates a risk of sudden system overload and an increasing risk of system failure.
Health of System Mesh (Kiali)	traffic flows go smoothly with no error at first. But after a while some traffic jam presents.

Scenario 2: scaling users to 600

When the users have been scaled to 600 users , we can see that the errors per seconds increased obviously.



When the replicas have been changed to 2, worker-node changed to 5 and DB capacity changed to 10, the errors per second have started to drop and successful requests per second increased.

Metrics	Observation & Analysis
Performance of microservices	At first there are some errors, and as time goes by, successful request/s increases and the error rate decreases gradually to nearly 0.
Latencies of microservices	Average, median and 90th %tile response time drops. The number of requests under 100ms starts to grow. All indicate latencies become lower.

System utilization	Compared to the previous scenario, memory usage increases. Utilization of CPU increases at first before decreasing and vibrating at a certain level. Values of Linux file descriptors decrease. This indicates the state transferring from system overloading and failure to a gradually remediate state.
Health of System Mesh (Kiali)	At first, we can see that the traffic flows turn into red at the beginning, and as time goes by, traffic flows turn back to green and all errors clear.

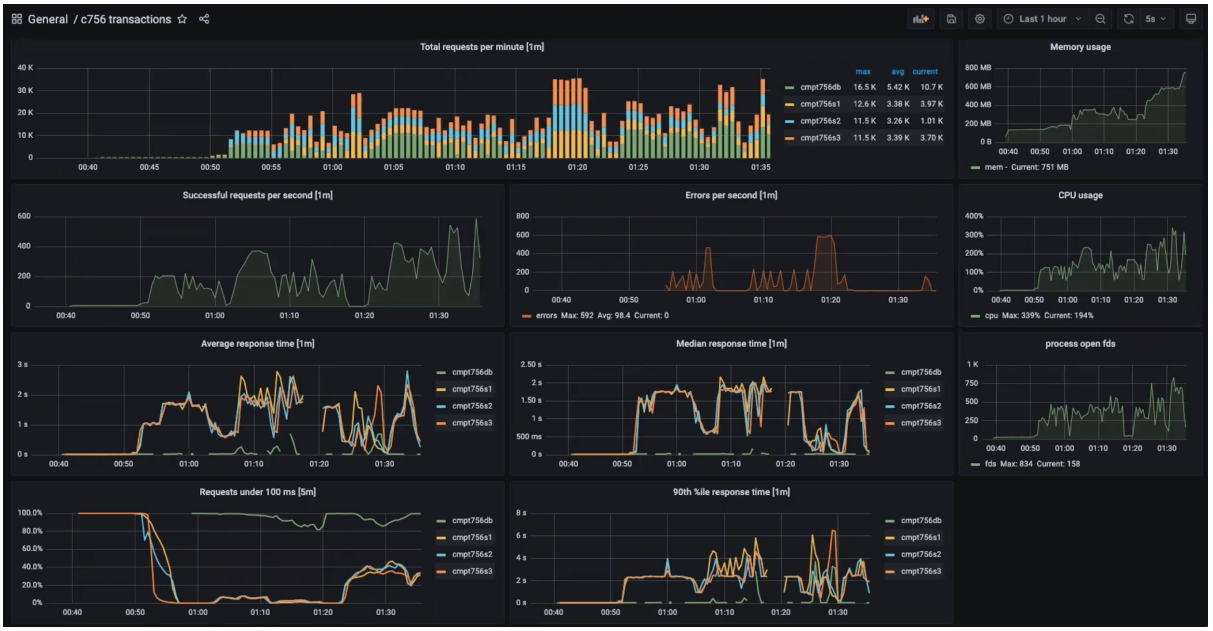
Scenario 3: scaling users to 900

In order to support 900 users, we have changed the replicas to 3, DB capacity changed to 20, the errors per second have started to drop and similarly, the successful requests per second increased.

Metrics	Observation & Analysis
Performance of microservices	Successful requests/s rise and error/s keeps stable at 0.
Latencies of microservices	Average and median response time

	decreases and requests under 100ms start to rise, which means microservices' latencies get lower and better.
System utilization	Compared to the previous 600-user scenario, memory usage increases. CPU utilization shows a decreasing trend. Process open fds utilization rises drastically.
Health of System Mesh (Kiali)	All traffic flows are green and successful requests rate are 100%.

Scenario 4: scaling users to 1200

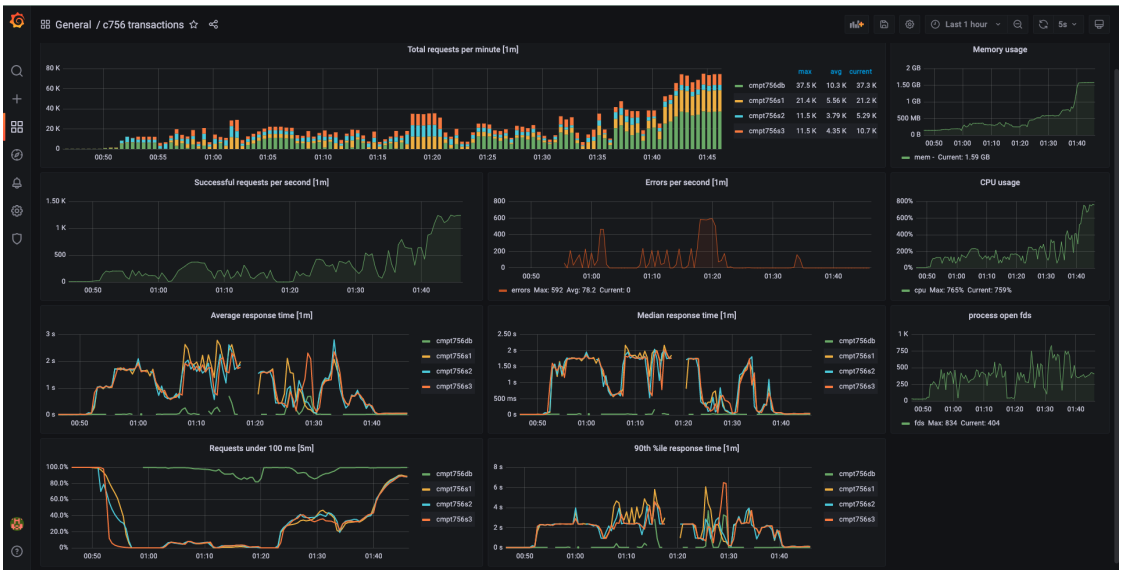


In this scenario, the user number is added to 1200. Correspondingly the replicas have been changed to 5, DB capacity changed to 100, the errors per second have started to drop after a short increase and successful requests per second increased.

Metrics	Observation & Analysis
Performance of microservices	Successful requests/s rise, but are still under 1000. Error/s is relatively stable at 0 except for some small errors.
Latencies of microservices	Average and median response time and 90th %tile response time presents a decreasing trend, while requests under 100ms start to

	rise, which means microservices' latencies get lower and better.
System utilization	Compared to the previous 900-user scenario, memory usage increases. CPU and process open fds utilization stay at roughly the same level.
Health of System Mesh (Kiali)	All traffic flows are green and successful requests rate are 100%.

Scenario 5: scaling users to 1500



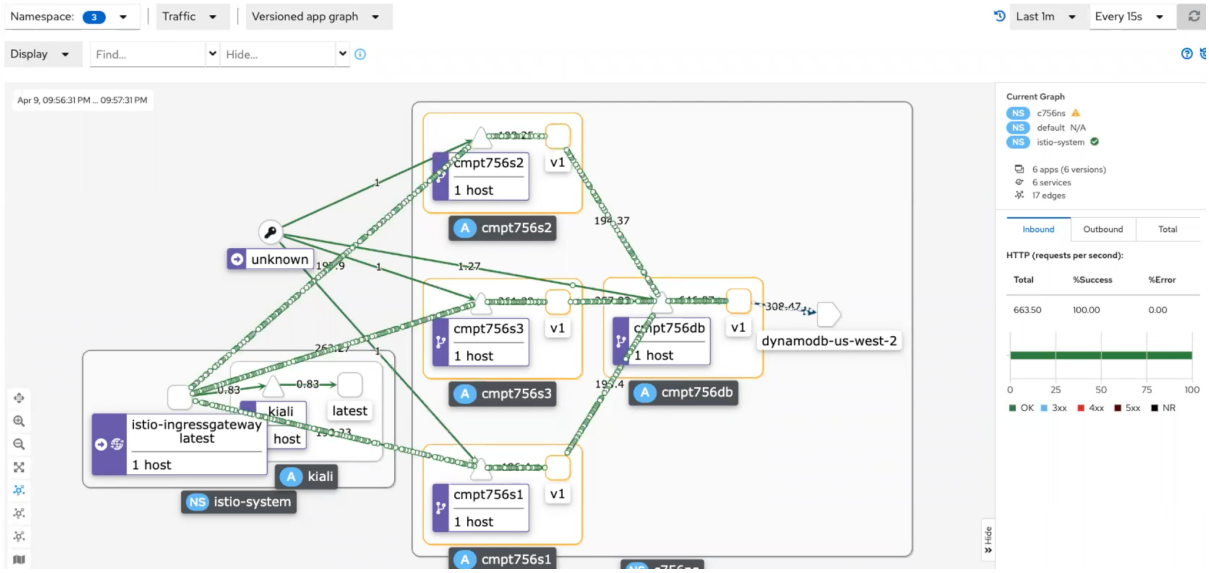
Each time we add 100 more users per service, setting replicas = 10, worker-node = 10, DB capacity = 200. Then we will have:

Total users: 1500 (exclude the beginning which the number is not significant), our success requests per second has surpassed 1000.

Metrics	Observation & Analysis
Performance of microservices	Values of successful requests/s rise, and plateau at above the 1000 level. Error/s is relatively stable at 0 after some small errors.
Latencies of microservices	Requests under 100ms rise gradually to nearly 100%. Average and median response

	time as well as 90th %tile response time decreases gradually to 0, which means our services now respond quickly and accurately, and microservices' latencies become optimal.
System utilization	Compared to the previous 1200-user scenario, memory and CPU usage increases. Process open fds utilization keeps at the similar level. Considering the current optimal performance, this indicates that system utilization is at an optimal level, neither too low nor too high, so that the system can balance costs against the risk of sudden system overload and over-provisioning of capacity for the workload.
Health of System Mesh (Kiali)	All traffic flows are green and successful requests rate are 100%.

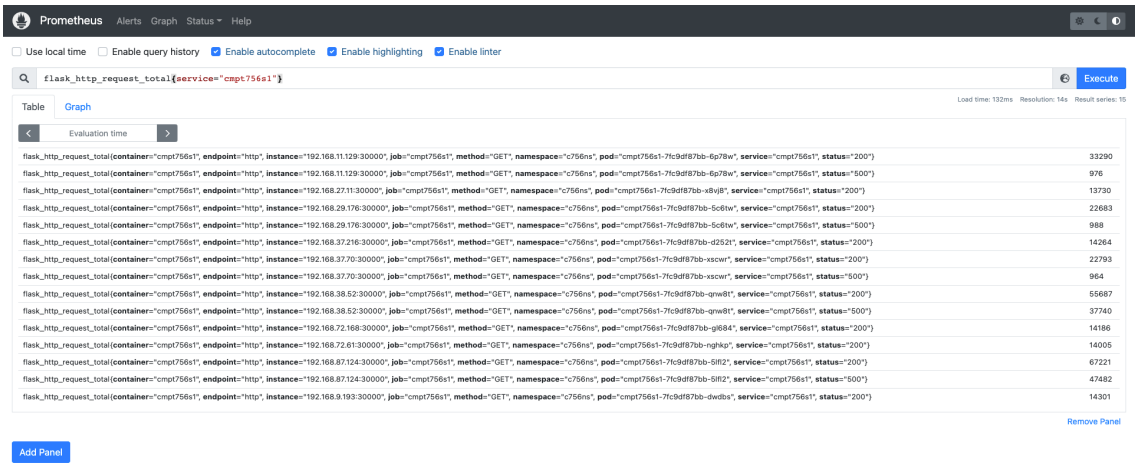
At last, we can see that all connections are green and everything goes smoothly.



Query result on Prometheus for scenario 5

We can see the status code of each request in each of our 4 services from Prometheus. For example, for service 1, we can use the command below to see the total request log for service 1.

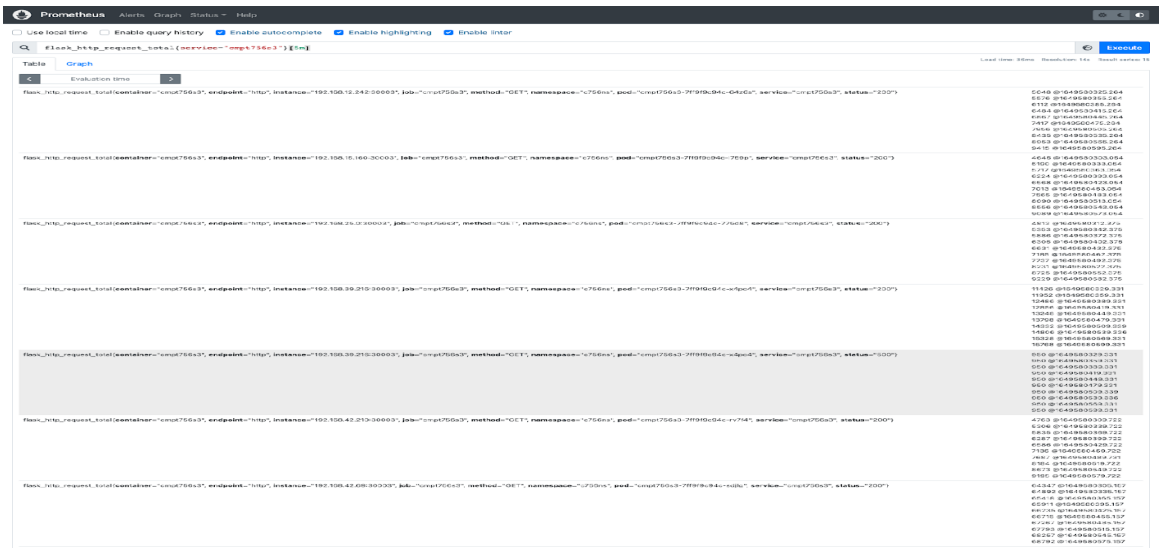
```
flask_http_request_total{service="cmpt756s1"}
```



In this log we can see that some of the status codes are “200” which means successful and some of the status codes are “500” which means error. We can also have a deeper insight into the status within a certain amount of time, such as 5 minutes for each of our 4 services. For example we can use this command:

```
flask_http_request_total{service="cmpt756s3"}[5m]
```

Also we can obtain more details from this log.



Observations

Reflection on Development

Scrum Methodology

We have arranged the sprint ceremonies and followed the schedule as much as we can. The advantage is that we understand the progress of every teammate and we can raise any difficulties in the daily scrum. Therefore, we can have quick adjustment of the project and bug fixing but not wait until the very end of the project schedule. However, as we are not only handling one project at the sametime, time management becomes very important when we adopt scrum methodology.
At the beginning of the project, we thought that the scrum methodology may be a hindrance as we have more administrative work to handle which may increase our workload. However, by adopting it, the project process has become transparent and we understand the status as well as direction. This is very important when we are working multiple projects within a tight schedule.
Compared with the previous professional experience with scrum, the team size is comparatively small in this project. Therefore, the skill set of each teammate has to be more comprehensive so that we can be more capable and flexible on the tasks assignment. Furthermore, the schedule for this project is relatively tight so we have to adjust the sprint to a reasonable length (one week) instead of using two weeks usually.

Conclusions

In this project, we first set up the project repository by using the course assignment repo as template. The following setup also includes configurations for individual environments. The newly designed microservice is playlist, which enables users to create, read, edit and delete playlist/s. We also design the various Gatling simulations, set up the Grafana dashboard and the production Kubernetes cluster to observe the system metrics via the Prometheus queries and Kiali. We run and analyze the simulation of adding small, medium and large load, and trigger the failure scenarios and test with remediation to inspect multiple system evolvings. During the whole process, we practice with the Scrum methodology for healthy schedule and progress. Overall, this project on distributed source control systems, microservices and a variety of tools helps us practically learn, observe and reflect on these principles.