



Group Work

Domain Driven Design of MaxLive

Mehmet Ustek - 2363741

Xiyuan Wang - 2475900

Jiaze Luo - 2356677

Shengzhe Shi - 2359956

Ruisi He - 2346007

MSc Cybersecurity

Faculty of Engineering

University of Bristol, 2 November 2023

Repository information

GitHub Repo: <https://github.com/Fr000g/MaxLive>

Docker Repo & Credentials:

<https://hub.docker.com/u/dropdatabase233>

Username: dropdatabase233 Password: pYn,A&diMq53qz+

Task A

1 Domain Driven Design

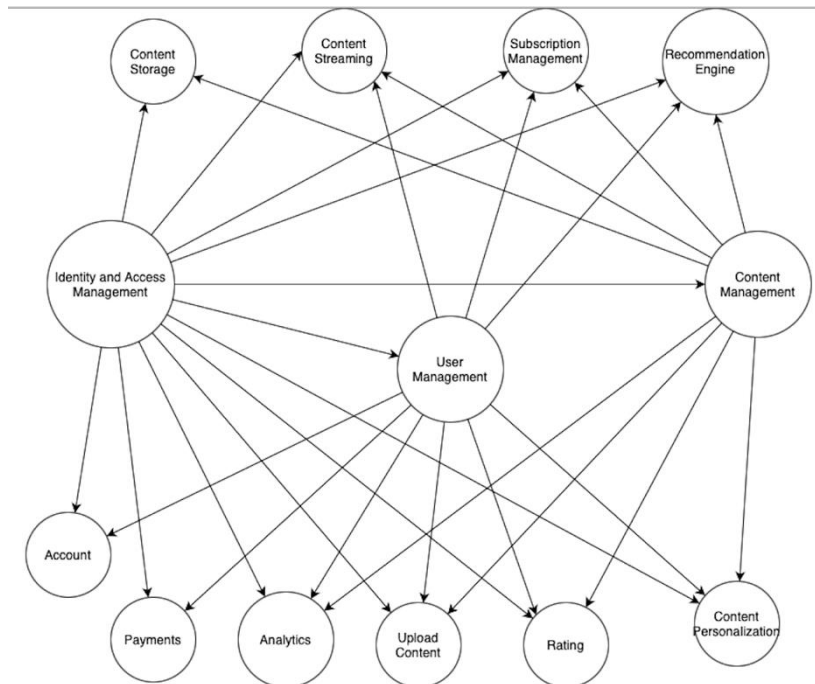


Figure 1: Domains of MaxLive

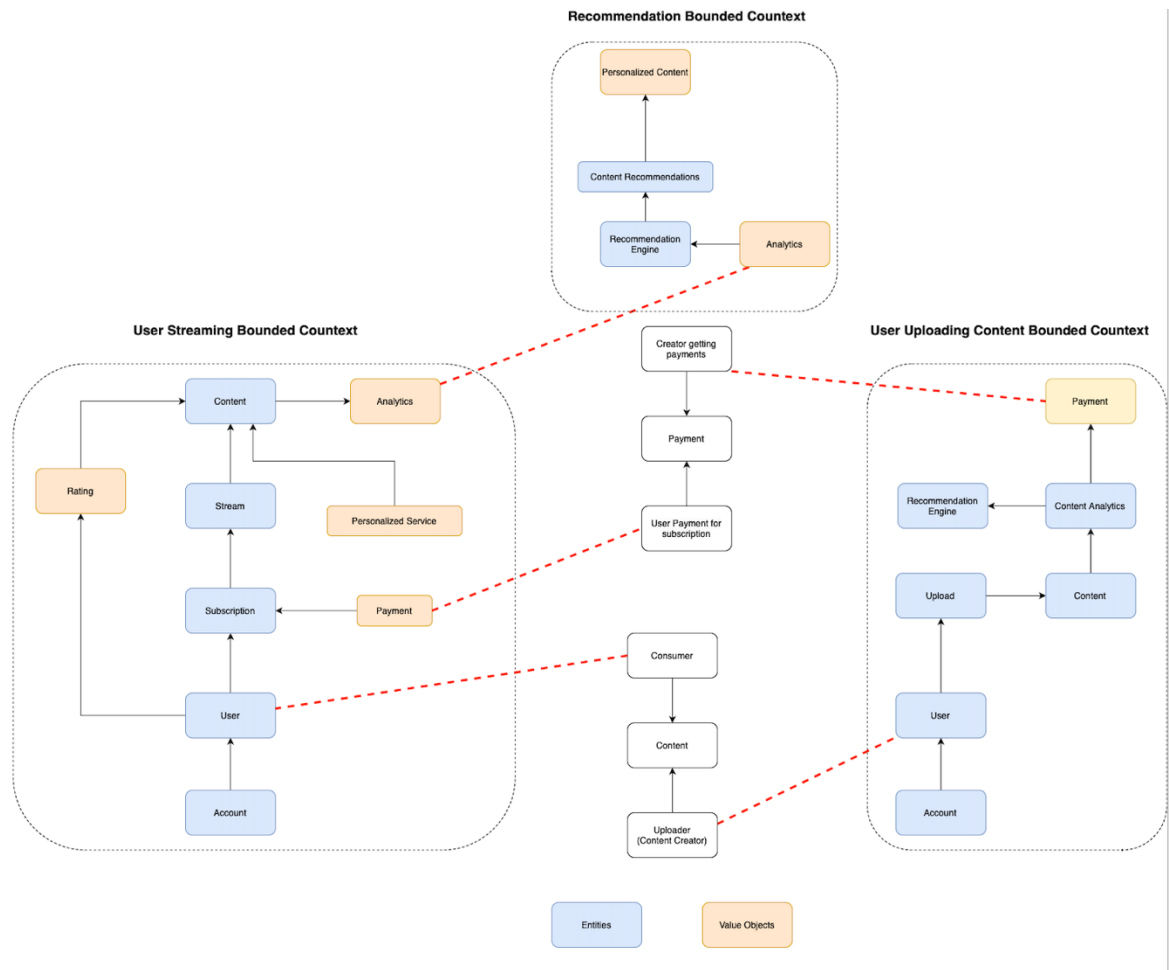


Figure 2: Bounded Context Diagram of MaxLive

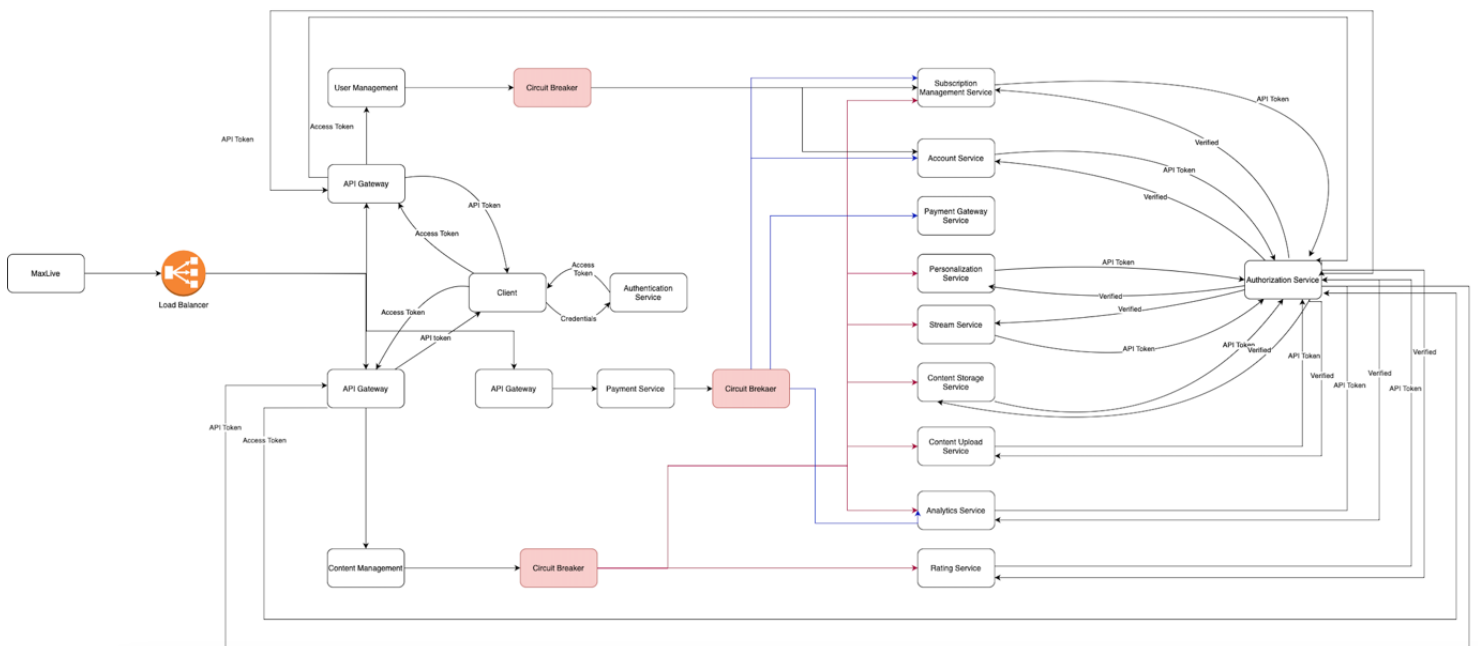


Figure 3: Domain Driven Design of MaxLive

2 Context Boundary

As shown in figure 2, user streaming bounded context encapsulates user subscription. This context handles all the user management and stores all related data in this context. For example, customer billing data is preserved in this context, and is not reachable by other contexts. Personalized service is also encapsulated in this context, which preserves the customer's preferences, for example, maturity level, genre choices. What is concerning with our design is that analytics data is sent to another context, recommendation bounded context. This context should comply with the data regulating authority standards to ensure no privacy regulation is breached. The recommendation engine and analytics gathered should be redesigned for each country to match with regulations where our system is operating. Our payment and billing information should be encrypted and secured with respect to local regulations. We should also preserve our payments to content creators and their billing data to match data protection standards of the local authority. By applying a domain driven design with a well-defined bounded context, we can preserve data privacy and security while keeping the basis of our applications same but slightly changing the smaller parts of our logic to match requirements of the data regulator.

3 Resolution of Service Level Agreement

To satisfy a 99.99% availability in its Service Level Agreement with its customers, we use a Load balancer, Replica sets, Database redundancy, Health checks, and Automatic scaling to refactor the architecture in A1. We implemented load balancers to evenly distribute incoming requests to various services, distribute traffic evenly, and ensure that there is no single point of failure. It prevents any single service instance from becoming a bottleneck and ensures uninterrupted service. Using replica sets to have multiple instances of each service ensures that even if one instance becomes unavailable, other instances can still service requests, thus ensuring high availability. Databases are the key component of any system. If one database server goes down, the other can take over without losing service, and implement health checks to monitor the status of each service and instance. Continuous monitoring ensures that

any issues are detected early and addressed before they affect users. Furthermore, the use of automatic scaling to ensure that the system can dynamically handle various loads by adjusting resources as needed, provides more stable performance.

4 Fault tolerance Architecture Implementation

As shown in figure 3, the circuit breaker pattern is applied to 3 major services: User Management, Payment Service, and Content Management. The circuit breaker pattern is used here to monitor the status of the service and decide whether to allow the requests to pass or block them.

The circuit breaker has 3 states: open, closed, and half-open. When the circuit is closed, the requests will be allowed to pass; when it is open, the requests will be blocked and a feedback response will be made; when it is half-open, some requests that meet the criteria will be allowed to pass.

The rationale for our choice is that the circuit pattern can ensure that identified services are fault tolerance. Even if some of the services fail because of network errors or other issues, the other services will still work. For example, if the Payment Service fails, the Subscription Service can still work by offering a default plan (basic plan) or a cached response. In this way, the system can handle failures efficiently and save time and energy.

5 API Token Security Pattern

Because RESTful is a stateless protocol, we use the API Token Security pattern to meet the authentication requirement. After users authenticate themselves, all the microservices that needs authorization should use the API token to authorize themselves.

The steps for user login are as follows:

- a) User Login: The user logs into the Authentication Service with his own credentials, which are username and password here.
- b) Access Token Generation: The Authentication Service will give the user the access token, including the user's identity and the validity period of the token.
- c) Token Passing: The API gateway will verify the validation of the token and generate the API token.

- d) API Token Generation: The API gateway will generate an API token for users to use the specific API. The client will carry this API token in the request in each subsequent request.

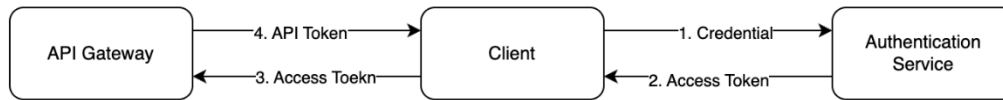


Figure 4: The flow diagram of API Token Security pattern

The function of API Gateway can be done by some code below. However, it is just a simple, the design of API is quite complex the all the process of data should be taken cautiously.

/api/* is Used to proxy API requests, verify access tokens, check request paths, and return corresponding API service responses.

```
function verifyToken(token){
  // the business code
}

app.use('/api', verifyToken, (req, res) => {
  // Used to proxy API requests,
  // verify access tokens,
  // check request paths,
  // return corresponding API service responses.

  const apiServices = {
    '/api/upload_service': 'Service1 Response',
    '/api/subscription_service': 'Service2 Response',
    // other service
  };

  // how the handle different service
  verifyToken(req.token)

});
```

Figure 5: demo of API

Task B

1 Source Code

All the code in task B have been pushed to re repository <https://github.com/Fr000g/MaxLive>, which can be downloaded by command below.

```
git clone https://github.com/Fr000g/MaxLive.git
```

Almost all services are done using node.js. Each service is structured including server.js and route.js, providing basic service example. Most service directory looks like this.

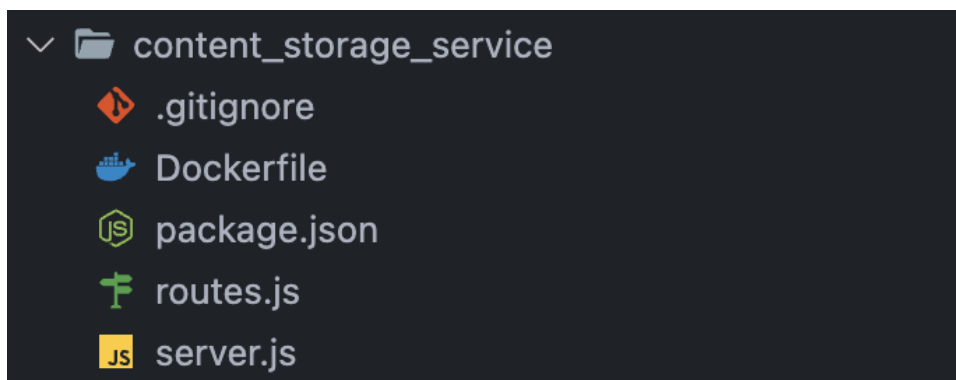


Figure 6: the structure of most microservice

Because the content of the code is not the key point in our work, we just wirte some simple code to simulate the function of a service.

```
router.get('/', (req, res) => {  
  const message = {  
    "message": "Welcome to Account server!",  
  };  
  res.send(message);  
});
```

Figure 7: the structure of most microservice

Nevertheless, we still put a lot of effort into some services to simulate a real API service as much as we can. For example, we implemented the proxy fuction in api_gateway_content_management.

```

router.get("/rating_service", async (req, res) => {
  try {
    const response = await axios.get(ratingService);
    res.json({
      message: "Response from Subscription Service",
      dataFromService2: response.data,
    });
  } catch (error) {
    console.error(error);
    res.status(500).json({ error: "Internal Server Error" });
  }
});

router.get("/stream_service", async (req, res) => {
  try {
    const response = await axios.get(streamService);
    res.json({
      message: "Response from Subscription Service",
      dataFromService2: response.data,
    });
  } catch (error) {
    console.error(error);
    res.status(500).json({ error: "Internal Server Error" });
  }
});

router.get("/upload_service", async (req, res) => {
  try {
    const response = await axios.get(uploadService);
    res.json({
      message: "Response from Subscription Service",
      dataFromService2: response.data,
    });
  } catch (error) {
    console.error(error);
    res.status(500).json({ error: "Internal Server Error" });
  }
});

module.exports = router;

```

Figure 8: the structure of most microservice

This code defines an Express.js router with multiple routes for different services (subscription, account, analytics, authorization, content storage, payment gateway, personalization, rating, stream, and upload services). Each route sends a GET request to its respective service endpoint using Axios, handles the response, and sends a JSON response back with the data received from the services.

Moreover, to reflect the Flexibility of microservices in Language and Technology Stack, we used Golang to complete the account_service service and completed some basic functions of adding, deleting, modifying, and checking users. The directory of account_service is as below.

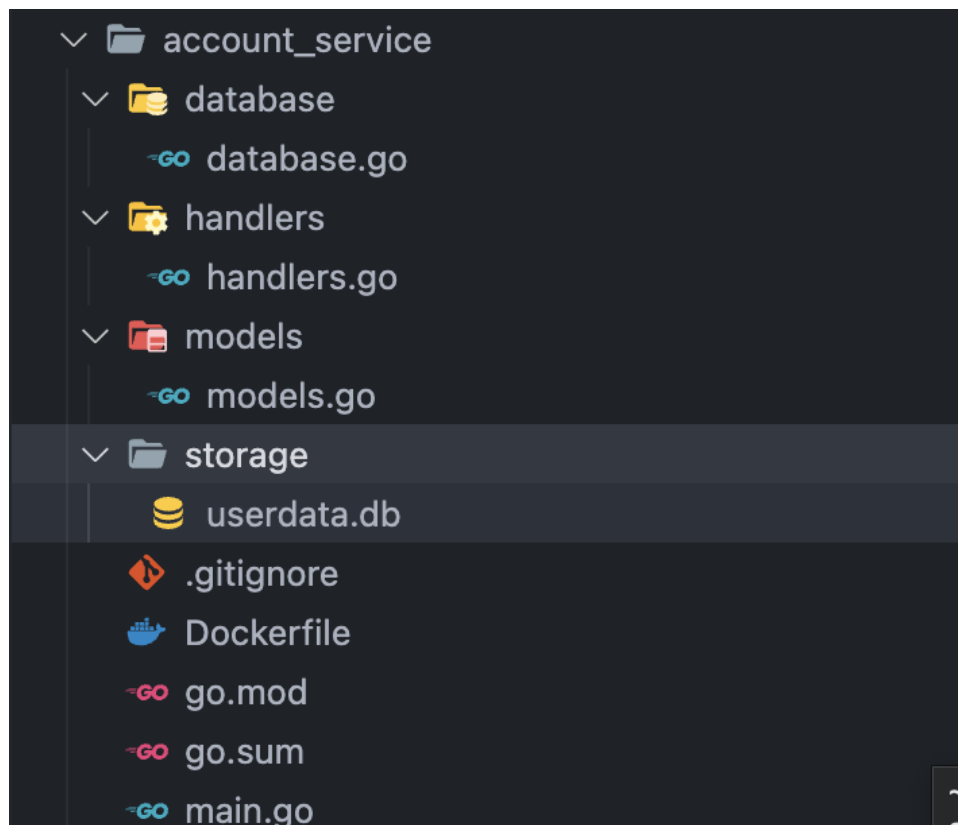


Figure 9: the structure of accoutt _service

2 Shell Script

I believe the `maxlive_commands_v1.sh` file is set to be compared with the `docker-compose.yaml`, to prove the benefits of compose file. Even though, I still want to make the shell script as elegant and clean as I can.

The `check_image` function checks if a Docker image with a service name exists. If the image does not exist, it builds the image using the `Dockerfile` located in the same directory as the script. It takes the image name as a parameter. If the image is not found, it echoes a message indicating that the image doesn't exist and builds the image. If the image already exists, it prints a message saying the image exists, and no further action is taken.

```

1  #!/bin/bash
2
3  # check if image exist
4  function check_image {
5      IMAGE_NAME=$1
6      if [[ "$(docker images -q $IMAGE_NAME 2> /dev/null)" = "" ]]; then
7          echo "image $IMAGE_NAME not exists, building..."
8          docker build -t $IMAGE_NAME ./$IMAGE_NAME
9      else
10         echo "image $IMAGE_NAME exists, skip build"
11     fi
12 }
13
14 # check if serve running
15 function check_running {
16     SERVICE_NAME=$1
17     if [[ "$(docker ps -q --filter name=$SERVICE_NAME 2> /dev/null)" = "" ]]; then
18         echo "container $SERVICE_NAME not running, starting..."
19         docker run -d -p $2:3000 $SERVICE_NAME
20     else
21         echo "container $SERVICE_NAME already running, skip"
22     fi
23 }

```

Figure 10: define and implement function.

The `check_running` function verifies if a Docker container with a service name is currently running. If the container is not running, it starts the container in detached mode, mapping a specified port to port 3000 inside the container. It takes two parameters: the container name and the port number for mapping. If the container is not running, it echoes a message stating that the container is not running and starts the container. If the container is already running, it prints a message indicating that the container is already active, and no action is performed.



```
1  check_image "dropdatabase233/account_service"
2  check_image "dropdatabase233/analytics_service"
3  check_image "dropdatabase233/authentication_service"
4  check_image "dropdatabase233/authorization_service"
5  check_image "dropdatabase233/content_storage_service"
6  check_image "dropdatabase233/payment_gateway_service"
7  check_image "dropdatabase233/payment_service"
8  check_image "dropdatabase233/personalization_service"
9  check_image "dropdatabase233/rating_service"
10 check_image "dropdatabase233/stream_service"
11 check_image "dropdatabase233/subscription_service"
12 check_image "dropdatabase233/upload_service"
13
14 check_running "account_service" 8000
15 check_running "analytics_service" 3000
16 check_running "authentication_service" 3001
17 check_running "authorization_service" 3002
18 check_running "content_storage_service" 3003
19 check_running "payment_gateway_service" 3004
20 check_running "payment_service" 3005
21 check_running "personalization_service" 3006
22 check_running "rating_service" 3007
23 check_running "stream_service" 3008
24 check_running "subscription_service" 3009
25 check_running "upload_service" 3010
```

Figure 101: function call

This script is not simple enough and has a lot of flaws, especially in some fields.

- Readability: The script is hard to read and maintain. When we need to change the configuration of the server, like increasing the containers and changing port forwarding, it will be an issue.
- Scalability: When we need to adjust the structure of the service, the script is too abstract to edit. When we want to put a docker network into it, the maintenance work will be much harder.
- Dependency Management: Different services cannot start the instance on which it depends. It is intolerable in microservice.

Despite the points we mentioned, there are still some technical issues. For example, in our script, if the source code is updated, we need to rebuild it manually. Even though, such an operation will generate an image with a repository name called <none>, wasting the disk space in our device. That is why the compose file is beneficial.

3 Compose.yml


The compose.yml file contains and image, build directory, ports, volumes, network and deploy settings. In this question, we mainly focus on deploy settings. Different services have different resource requirements, especially the service which is important and be frequently accessed.

Back to our domain design diagram in Task A.3, every access to service will go through authentication service. Almost all services heavily rely on authorization service, so it is quite essential to make it fall-saft and self-healing. So, we assign 10 replicas to it.

For the same reason, the robustness of account service should also be very strong too. So, we set the replicas to 10.

```
authorization_service:
  image: dropdatabase233/authorization_service
  build: ./authorization_service
  ports:
    - "0.0.0.0:3002:3000"
  volumes:
    - ./volumes/authorization_service:/app/database:ro
  networks:
    - backend_network
  deploy:
    replicas: 10
```

Figure 112: authentication_service



```
1  account_service:
2    image: dropdatabase233/account_service
3    build: ./account_service
4    ports:
5      - "8000:8000"
6    volumes:
7      - ./volumes/account_service:/app/database:rw
8    deploy:
9      replicas: 10
10   networks:
11     - backend_network
```

Figure 13 account_service

When we deployed the system, we found that as the number of nodes and services increased, the memory usage and consumption also increased rapidly. Therefore, if the resource consumption of individual services can be properly controlled, the unreasonable allocation of memory resources can be reduced to a certain extent. Relevant configurations can be set in deploy.

```
1  deploy:
2    replicas: 3
3    resources:
4      limits:
5        cpus: '1'
6        memory: 200M
7        pids: 10
```

Figure 14 deploy config

4 Strategic Resource Allocation in DOS and DDoS Defence

We acknowledge the validity of adding more replicas to prevent DOS and DDoS attacks. However, it is not necessary to expand all businesses simply and crudely in such way. In our system, if we assume each service consumes 7GB of memory, and there are a total of 15 services, the combined memory consumption for all services would be 105GB. With 5 replicas, this figure rises to 525GB. When the replicas increase to 10, the memory usage reaches 1050GB, which is unacceptable for the majority of systems.

CONTAINER ID	NAME	CPU %	MEM USAGE / LIMIT	MEM %	NET I/O	BLOCK I/O	PIDS
f4904965319c	maxlive-payment_service-1	0.00%	34.84MiB / 7.685GiB	0.44%	1.2kB / 0B	0B / 0B	26
3c8fdeebdac2	maxlive-api_gateway_payment_service-1	0.02%	42.45MiB / 7.685GiB	0.54%	1.42kB / 0B	0B / 0B	26
486d749af5f2	maxlive-api_gateway_user_management-1	0.02%	35.79MiB / 7.685GiB	0.45%	1.42kB / 0B	0B / 0B	26
7949f1fb6dfe	maxlive-content_storage_service-2	0.00%	33.77MiB / 7.685GiB	0.43%	1.43kB / 0B	0B / 0B	26
9a1eae7fa5ef	maxlive-rating_service-2	0.00%	33.95MiB / 7.685GiB	0.43%	1.13kB / 0B	0B / 0B	26
3003820dccc9	maxlive-stream_service-2	0.00%	36.99MiB / 7.685GiB	0.47%	866B / 0B	0B / 0B	26
a3bd33d5c3c4	maxlive-personalization_service-1	0.00%	35.86MiB / 7.685GiB	0.46%	1.54kB / 0B	0B / 0B	26
bfd733b07f47	maxlive-authorization_service-2	0.00%	33.96MiB / 7.685GiB	0.43%	1.09kB / 0B	0B / 0B	26
a8057545e1b9	maxlive-subscription_service-2	0.00%	33.71MiB / 7.685GiB	0.43%	1.31kB / 0B	0B / 0B	26
c1b126fda30f	maxlive-upload_service-3	0.00%	36.07MiB / 7.685GiB	0.46%	866B / 0B	0B / 0B	26
ae28d9446d95	maxlive-payment_gateway_service-1	0.00%	34.38MiB / 7.685GiB	0.44%	796B / 0B	0B / 0B	26
5b961059769f	maxlive-authentication_service-1	0.00%	35.71MiB / 7.685GiB	0.45%	796B / 0B	0B / 0B	26
e8610ead7773	maxlive-analytics_service-1	0.00%	36.04MiB / 7.685GiB	0.46%	866B / 0B	0B / 0B	26
0cb10afdb867	maxlive-api_gateway_content_management-1	0.03%	36.42MiB / 7.685GiB	0.46%	1.24kB / 0B	0B / 0B	26

Figure 125: The resource cost

Not only that, but some DDoS attacks are not solely caused by insufficient resources. Sometimes, logical flaws within the code itself can lead to the excessive consumption of system resources. Certain well-known middleware has faced this issue and merely responding by adding resources to prevent DDoS attacks is clearly insufficient.

Hence, there is ample reason to believe that businesses should be adjusted flexibly and dynamically, taking into account factors such as application requirements, system architecture, resource availability, and budget.

On one hand, in our compose.yml, we consider services like authentication_service, payment_gateway_service, and account_service to be essential and integral to our business. These three services are frequently requested and form an indivisible part of our system. Therefore, we need to create more replicas for these services.

On the other hand, some microservices are rarely used and do not serve as the key functions of a business. Limited resources are sufficient to meet the practical needs of such services. Examples of this are the analytics_service and rating_service, which, even if shut down, would not significantly impact the core functions of the system.

5 Implementation into Docker Swarm Cluster

(a) The Number of Managers

Due to the limitations of our personal computer's performance, we had to reduce the number of nodes created by Multipass for deployment. It is obvious that using only one manager is clearly impractical. We decided the number of managers mainly based on the factors below:

- If single manager is down for some reason, the swarm will no longer be running.
- The number of managers must be odd to avoid tie, which will lead to chaos if the system is isolated into subgroups, making conflicting decisions, influencing the instability of swarm

Therefore, we chose to have three managers to maintain the swarm's operational stability and minimize the risk of disruptions.

```
ubuntu@node1: ~  
ik2orkam0-0144ida6qeol6bsgf2kvg2qh3 192.168.65.5:2377  
  
ubuntu@node1:~$ docker node ls  
ID                                HOSTNAME    STATUS    AVAILABILITY    MANAGER STATUS  
ENGINE VERSION  
mcrbezs2yzub0l6lu16b2vbw1 *    node1      Ready     Active           Leader  
24.0.7  
rcrx6xrh81kg1bdknionetjwd      node2      Ready     Active           Reachable  
24.0.7  
v2zs1uffgndr5pg3su7rttgeb      node3      Ready     Active           Reachable  
24.0.7  
tbnobohqqyz1o57ugfu4v618u      node4      Ready     Active  
24.0.7  
bs0b6x9vr0v765kkzb0pmyfx1      node5      Ready     Active  
24.0.7  
ubuntu@node1:~$
```

Figure 113: 5 nodes in swarm

```
ubuntu@node1:~/MaxLive$ docker service ls  
ID                                NAME                                MODE                REPLICAS    IMAGE                                                    PORTS  
ue5vaycsg5b1                      maxlive_account_service            replicated          3/3          dropdatabase233/account_service:latest                *:3080->8000/tcp  
5hr0hx004y2y                      maxlive_analytics_service          replicated          1/1          dropdatabase233/analytics_service:latest                *:3000->3000/tcp  
qwxm8f6ub30b                      maxlive_api_gateway_content_management replicated          1/1          dropdatabase233/api_gateway_content_management:latest   *:3021->3000/tcp  
pw9npwuka798                      maxlive_api_gateway_payment_service replicated          1/1          dropdatabase233/api_gateway_payment_service:latest      *:3022->3000/tcp  
a6udb0b3eo08                      maxlive_api_gateway_user_management replicated          1/1          dropdatabase233/api_gateway_user_management:latest      *:3020->3000/tcp  
y8vj4gnlmw7                      maxlive_authentication_service      replicated          3/3          dropdatabase233/authentication_service:latest           *:3001->3000/tcp  
l151mqmq19ep                      maxlive_authorization_service       replicated          3/3          dropdatabase233/authorization_service:latest            *:3002->3000/tcp  
l2vhl0od4i6kc                    maxlive_content_storage_service     replicated          3/3          dropdatabase233/content_storage_service:latest          *:3003->3000/tcp  
4zdop4ci7web                      maxlive_payment_gateway_service     replicated          1/1          dropdatabase233/payment_gateway_service:latest          *:3004->3000/tcp  
zr3aadgmzhc9                      maxlive_payment_service             replicated          3/3          dropdatabase233/payment_service:latest                  *:3005->3000/tcp  
81uoyb2wxoer                      maxlive_personalization_service     replicated          1/1          dropdatabase233/personalization_service:latest          *:3006->3000/tcp  
s5rbsdy2uthc                      maxlive_rating_service              replicated          1/1          dropdatabase233/rating_service:latest                   *:3007->3000/tcp  
3ragi8rdrijw2                    maxlive_stream_service              replicated          3/3          dropdatabase233/stream_service:latest                   *:3008->3000/tcp  
wetwll07n2wz                      maxlive_subscription_service        replicated          1/1          dropdatabase233/subscription_service:latest             *:3009->3000/tcp  
sy2sxfajd58l                      maxlive_upload_service              replicated          1/1          dropdatabase233/upload_service:latest                   *:3010->3000/tcp
```

Figure 114: service list after deployment

(b) Separation of Responsibilities

To separate responsibilities among different nodes, we can assign specific labels to nodes, constraining nodes with different labels. After reading the official documentation, we learned that we could specify a node label using the following command:

```
docker node update --label-add type=queue node4
```

After that, we can find the label of node is changed. We can use placement preferences to deploy it, constraining the node which service will use

```
deploy:
  replicas: 3
  placement:
    constraints: [node.labels.type == queue]
```

Figure 18: service list after deployment

(c) Resilience Features

In our cluster, we tried our best to demonstrate robust resilience features. Firstly, by utilizing Docker Swarm's overlay network, all services are interconnected, enabling efficient communication between them. Multiple replicas of services are distributed across different nodes in the cluster, facilitating load balancing.

Moreover, we tried to specify as replicas: 1 or replicas: 3 for service to achieve fault tolerance. Although we did not set too many nodes or replicas to handle more complicated situation because of the performance limitation of our own laptop. Because of the set of replicas, even if one node fails, we can make ensure seamless operation on other available nodes. Meanwhile, we put the circuit breaker into API gateway. In that case, if some service is broken, we can make sure other unrelated service can work properly.

To make the cluster showcases self-healing capabilities, we change part of compose.yml, set the restart_policy. In the event of a service instance failure, Docker Swarm automatically detects the issue, marking the instance as unhealthy and triggering the restart process on another available node.


```
5000+0000
deploy:
  replicas: 3
  restart_policy:
    condition: any
networks:
  - backend_network
```

Figure 19: restart_policy