CSE 546 Project - 3 Report

Arizona State University

Aditya Reddy Mali - 1224165813
Paromita Roy - 1224708511
Tejesh Reddy Sigineni - 1222183339

1. Problem Statement

In this project, we aim to develop an elastic application using resources from both Amazon Web Services (AWS) and OpenStack, a free and open-source cloud computing platform. The application will be a smart classroom assistant for educators that utilises face recognition technology to retrieve academic information about students. The challenge is to transfer the application developed in Project 2 to a hybrid cloud environment and ensure seamless integration between AWS and OpenStack resources. Additionally, we aim to optimise the application to ensure efficient use of cloud resources, minimise costs, and enhance performance.

2. Design and Implementation

2.1. Architecture

In our implementation, a user is asked to upload an MP4 file. The video is uploaded into an S3 bucket, which functions as an input bucket. Once the video is successfully uploaded, it simultaneously adds an event into an input SQS. OpenStack is continuously monitoring and polling the input queue. If OpenStack is alerted about an event in the input queue, it takes the input and invocates the Lambda function. This function is created from an image containing the face recognition functionality, also known as the Lambda handler function. It extracts a frame from the video and stores it as an image. A facial recognition module is run on this image, which creates an encoding. The encoding is matched with a known encoding file to retrieve the student's name. Finally, we use this name to query a DynamoDB, which contains all the relevant information of every student. All this processing is handled by Lambda. It then uploads the output into another S3 bucket, which serves as the output bucket. Additionally, it populates an output SQS as well. OpenStack is also monitoring this output SQS, and once there is anything in this output queue, it takes the event and loads the results in csv format into a text file, which is stored in the instance on OpenStack.

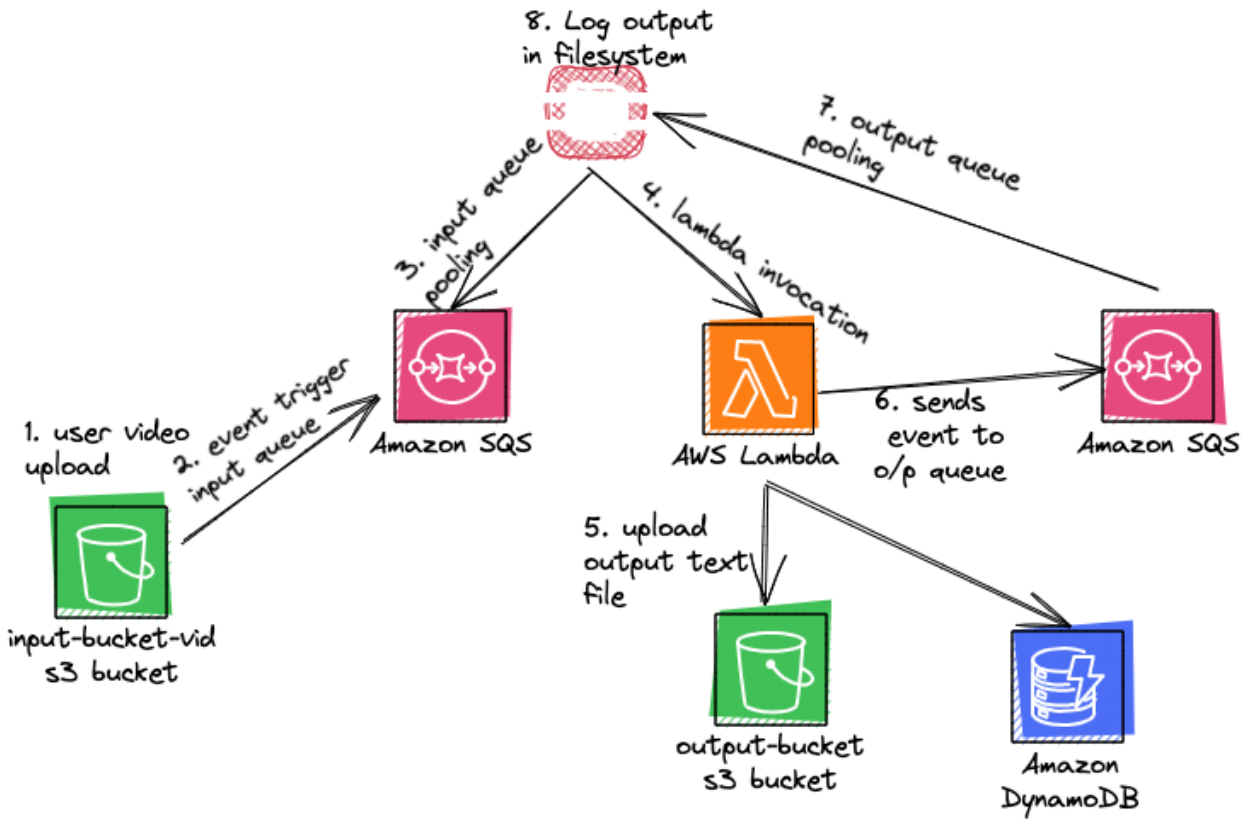

Figure 1. Architecture Diagram

2.2. AWS Services Used

1. AWS EC2

AWS EC2 is a web service that allows users to quickly deploy and scale virtual machines known as instances. EC2 instances can be deployed in several availability zones and configured with different operating systems, programs, and applications. Users can select from a range of instance types, each with different CPU, memory, storage, and networking capabilities. EC2 also provides features such as security groups, network access control, and monitoring to ensure the security and reliability of instances. EC2 was primarily used to deploy and host the devstack.

2. AWS SQS

Amazon Simple Queue Service (SQS) is a fully managed message queuing service that enables decoupling and asynchronous communication between distributed software components and microservices. It allows developers to send, store, and receive messages between different components of their application. SQS was used to store

events based on the content in the S3 input buckets. It was also used by Lambda to populate the output queue.

3. AWS S3

Amazon Simple Storage Service (S3) is a highly scalable and durable object storage service that allows users to store and retrieve any amount of data from anywhere on the web. It provides unlimited storage capacity, low latency, and high availability for various data types, including documents, images, videos, and application backups. S3 objects are organised into buckets and can be accessed via a REST API or a web console. S3 is used to store the input videos uploaded by the user and also the output csv formatted .txt files.

4. AWS Lambda

Amazon Lambda is a serverless computing service that enables developers to run code without provisioning or managing servers. It allows users to upload their code as functions and execute them in response to events, such as changes in data, user actions, or API calls. It supports a number of programming languages, including Python, Node.js, Java, and C#. A wide variety of serverless applications may be created using Lambda in combination with other AWS services, including S3, DynamoDB, SQS, and API Gateway. Lambda used for performing facial recognition and uploading the output event to SQS and output file to S3.

5. AWS DynamoDB

Amazon DynamoDB is a fully managed NoSQL database service that provides fast and flexible storage for structured and semi-structured data. Users can create and query tables with unlimited scalability, millisecond latency, and the capacity to store and retrieve any amount of data. DynamoDB supports various data models, including key-value, documents, and graphs. To create real-time and event-driven applications, DynamoDB may be used with other AWS services like Lambda, and S3. DynamoDb stored all the data related to the students and their information.

2.3. Member Tasks

1. **Aditya Reddy Mali:** was responsible for setting up the EC2 instance and installing devstack on it. They will then proceed to create and configure Openstack VMs, including network configurations. Additionally, they will upload these VMs to glance, which is an Openstack service that enables users to discover, register, and retrieve virtual machine images. They will also document the problem statement and Openstack configuration.
2. **Paromita Roy:** was responsible for ensuring that the Openstack VMs are properly configured, which includes configuring the network settings. They will

also be responsible for setting up an S3 event trigger to an SQS queue, which will require them to have a good understanding of AWS services. Furthermore, they will document the coding process thoroughly and take screenshots to provide visual aids for the team's reference.

3. **Tejesh Reddy Sigineni:** was responsible for configuring and developing output SQS event from a lambda function, this was performed using python programming language. Moreover, they will design an Openstack handler that will pool the input queue and output queue. They will also build the Openstack handler that will invoke the lambda function by passing the SQS event. Additionally, they will document the architecture and testing processes.

3. OpenStack

1. Openstack services used:
    a. Horizon: Horizon is the web-based dashboard for OpenStack, which provides a graphical user interface (GUI) for managing resources and services in an OpenStack environment. It allows administrators and users to perform various tasks such as creating virtual machines, managing storage, configuring networks, and monitoring system performance.
    b. Nova: Nova is the compute service of OpenStack, which provides the functionality to create and manage virtual machines (VMs). It allows users to launch instances with different operating systems and configurations and provides features such as automatic scaling, live migration, and high availability.
    c. Neutron: Neutron is the networking service of OpenStack, which provides the functionality to create and manage virtual networks and network services. It allows users to configure and manage network resources such as routers, subnets, and security groups and provides features such as load balancing, VPN, and firewall.
    d. Glance: Glance is the image service of OpenStack, which provides the functionality to store, retrieve, and manage disk images used to create virtual machines. It allows users to create and share custom images and provides features such as versioning, encryption, and caching.
2. Setting up OpenStack
    a. OpenStack requires a large amount of computational resources. Due to this, we decided to make use of an AWS EC2 instance - t2.large, on which we installed devstack with the help of the commands given to us [3].
    b. Install devstack on the EC2 instance. Some important code and configurations to remember are:

i.
```
$ sudo useradd -s /bin/bash -d /opt/stack -m stack
$ sudo chmod +x /opt/stack
$ echo "stack ALL=(ALL) NOPASSWD: ALL" | sudo tee /etc/sudoers.d/stack
$ sudo -u stack -i
$ git clone https://opendev.org/openstack/devstack
$ cd devstack
```

ii. Local.conf

```
[[local|localrc]]
ADMIN_PASSWORD=secret
DATABASE_PASSWORD=$ADMIN_PASSWORD
RABBIT_PASSWORD=$ADMIN_PASSWORD
SERVICE_PASSWORD=$ADMIN_PASSWORD
```

c. After installing devstack, we launched the Horizon dashboard.
d. We uploaded a CentOS image (CentOS-7-x86_64-GenericCloud.qcow2c) onto Glance.
e. We configured a security group that allows SSH requests and also all ICMP traffic.
f. Created a private network and bridged the private and public networks with a router.
g. After this, we finally created an instance using the image we uploaded. We used the t1.small flavour and attached it to the private network that we created. We also created a key pair so that we could SSH into the instance successfully.
h. To make it accessible from the outside, we attached a floating IP to the instance.

4. Testing

To test this application, we used the workload generator provided to us. These are the steps followed:

1. Verify that both S3 buckets are initially empty.
2. Run the following command to upload the MP4 files:

```
python3 workload.py
```

3. Verify that the number of files in the input bucket is equal to the number of videos uploaded by the user.
4. Check the number of events published by S3, which had to match the count of the videos uploaded to the input S3 bucket.
5. Check the logs to ensure that the Lambda function is being triggered as soon as the input input queue gets populated.

6. Check the output S3 bucket and output queue to see if results are being populated.
7. During the execution verify if there are multiple concurrent execution by lambda and the total number of invocations equals the input queue/input s3 bucket.
8. Once the execution of the program has been completed, verify that the number of objects in the output bucket is equal to the number of videos initially uploaded by the user.
9. Validate if the output queue is being constantly pooled and the openstack handler is fetching the messages from the queue.
10. Login to the VM in openstack and check if the openstack handler is writing the pooled messages to the disk. The count of the total number of messages should equal the number of input videos uploaded by the user.
11. Check each object in the output bucket to confirm that the text files contain the CSV formatted output of the pertaining input file.

5. Code

The files involved in this application are as follows:

1. **Dockerfile:** The dockerfile given to us required some changes. We included the upload of encoding in this file.
2. **Entry.sh:** This file ensures that the program runs in the appropriate directory.
3. **Handler.py:** This is the entry point when the lambda function kicks in. It contains code responsible for facial recognition on the file uploaded to the S3 bucket. It also queries the DynamoDB to fetch results. It runs an ffmpeg command which extracts a frame from the video and converts it to jpeg format. Then, an encoding is created and is compared with the encoding file given to us. This returns the name associated with the person in the image. The Lambda function processes the message and sends a response back to the output SQS queue. Finally, the output is converted to CSV format and is pushed into the output S3 bucket.
4. **Openstack_handler.py:** The code listens for messages on an input SQS queue and sends the event to the Lambda function which is specified by the lambda_arn variable. The code also checks the output queue for messages and saves them to a local folder specified by the folder_path variable. The code uses the boto3 Python library to interact with AWS services, and the mylogger module to log information to the console.
5. **Mapping:** This file contains the mapping for every MP4 file. It gives us information about every tag associated with each video.
6. **Workload.py**: This file has functions which contain code to clear both the input and output buckets when the program initially starts. It also pushes the user uplaoded MP4 files into the input S3 bucket.

7. **ecr_push.sh:** Created a script to run all commands necessary to log into, build and tag the created image, and push it to the ECR repository. This avoids running multiple commands repeatedly.

Steps to create the environment and run the program:

1. Set up docker on your system.
2. Retrieve an authentication token and authenticate your Docker client to your registry.

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --password-stdin 704676190155.dkr.ecr.us-east-1.amazonaws.com
```

3. Create an image containing using the Docker file to include files from the directory.

```
docker build -t smart-classroom .

docker tag smart-classroom:latest 704676190155.dkr.ecr.us-east-1.amazonaws.com/smart-classroom:latest
```

4. Push the image to Amazon ECR.

```
docker push 704676190155.dkr.ecr.us-east-1.amazonaws.com/smart-classroom:latest
```

5. Update the image URI on the Lambda function to reflect the uploaded *handler.py*.
6. Run workload generator with the following command:

```
python3 workload.py
```

7. Check the input bucket and verify that the videos are being uploaded successfully. Verify that the number of objects in the bucket is equal to the number of videos being uploaded.
8. Check the output bucket and verify the results are being accurately labelled.

6. Outputs and Screenshots

```
(base) ➜  smart-classroom-face-recognition git:(main) x
'Contents'
Nothing to clear in input bucket
Nothing to clear in output bucket
Running Test Case 1
Uploading to input bucket..  name: test_0.mp4
Uploading to input bucket..  name: test_1.mp4
Uploading to input bucket..  name: test_2.mp4
Uploading to input bucket..  name: test_6.mp4
Uploading to input bucket..  name: test_7.mp4
Uploading to input bucket..  name: test_5.mp4
Uploading to input bucket..  name: test_4.mp4
Uploading to input bucket..  name: test_8.mp4
Running Test Case 2
Uploading to input bucket..  name: test_36.mp4
Uploading to input bucket..  name: test_22.mp4
Uploading to input bucket..  name: test_0.mp4
Uploading to input bucket..  name: test_1.mp4
Uploading to input bucket..  name: test_23.mp4
Uploading to input bucket..  name: test_37.mp4
Uploading to input bucket..  name: test_21.mp4
Uploading to input bucket..  name: test_35.mp4
Uploading to input bucket..  name: test_2.mp4
Uploading to input bucket..  name: test_34.mp4
Uploading to input bucket..  name: test_20.mp4
Uploading to input bucket..  name: test_18.mp4
Uploading to input bucket..  name: test_24.mp4
Uploading to input bucket..  name: test_30.mp4
Uploading to input bucket..  name: test_6.mp4
Uploading to input bucket..  name: test_7.mp4
Uploading to input bucket..  name: test_31.mp4
Uploading to input bucket..  name: test_25.mp4
Uploading to input bucket..  name: test_19.mp4
Uploading to input bucket..  name: test_33.mp4
```

Figure 2. Running the workload generator

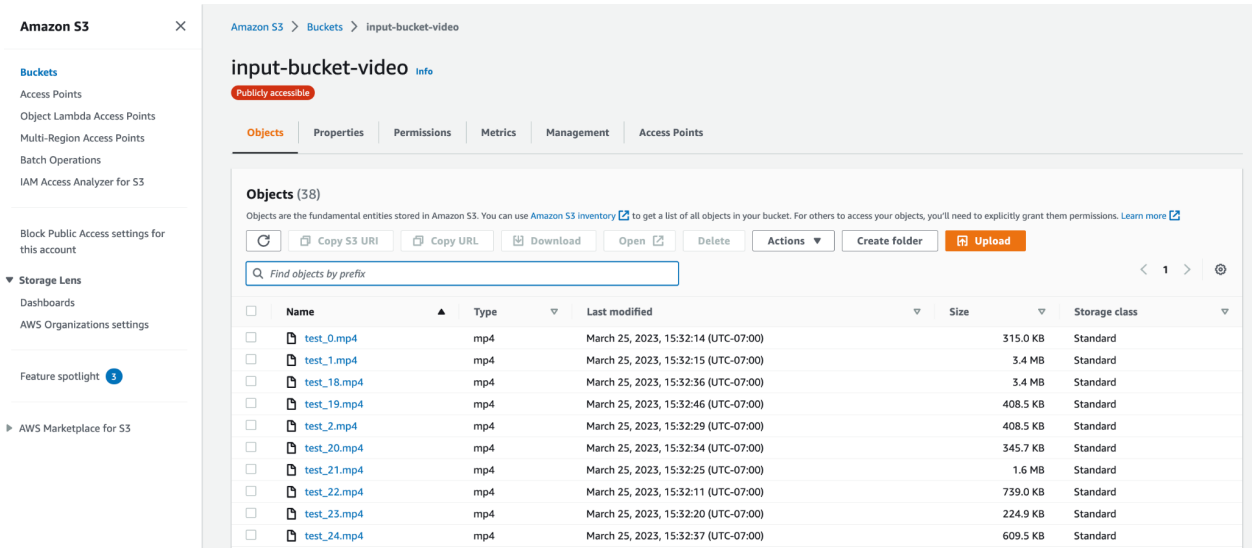

Figure 3. Objects in the input-bucket

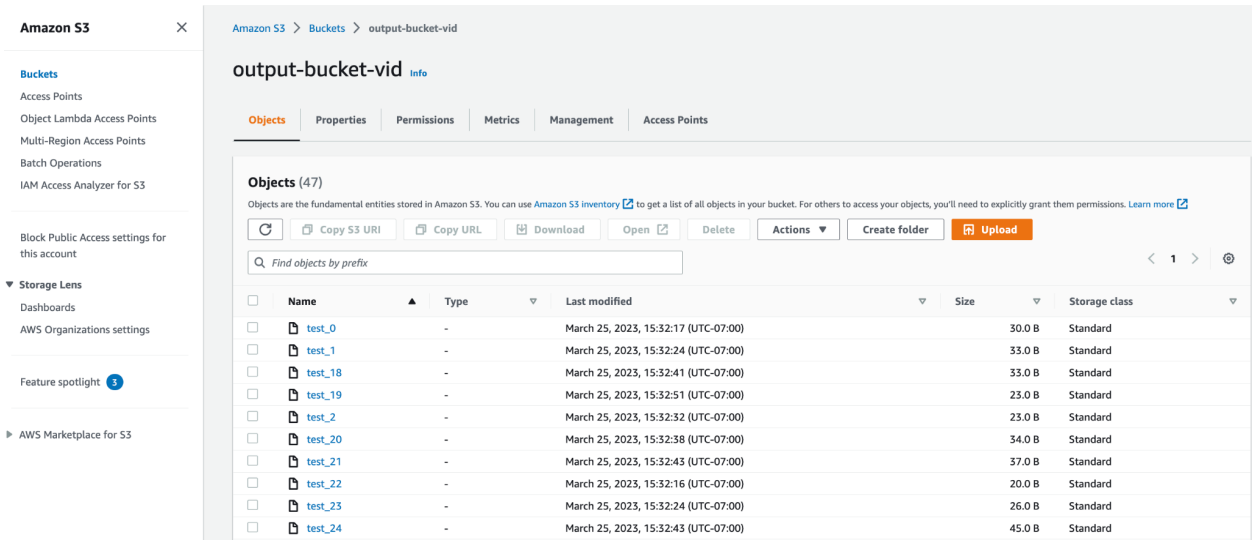

Figure 4. Objects in the output-bucket

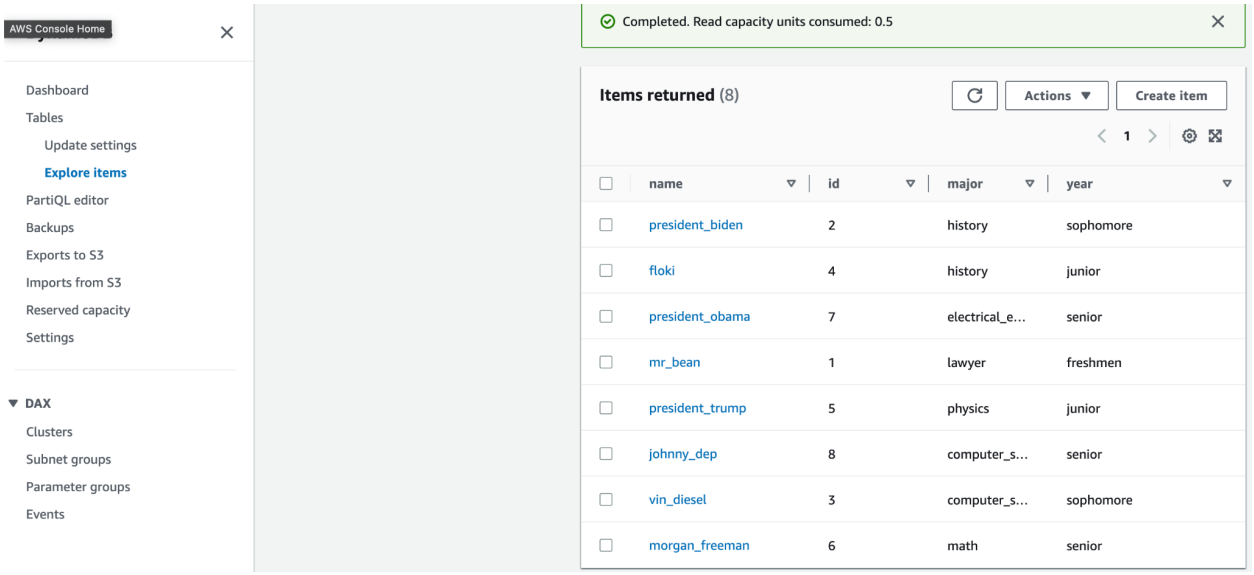

Figure 5. Returned results

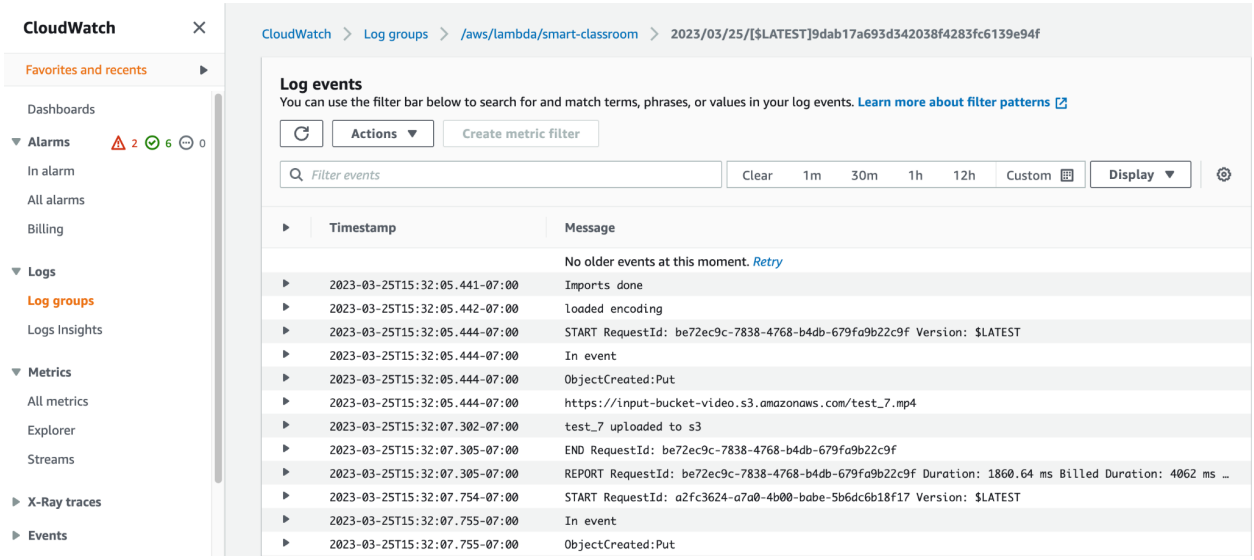

Figure 6. Logs returned by the Lambda function

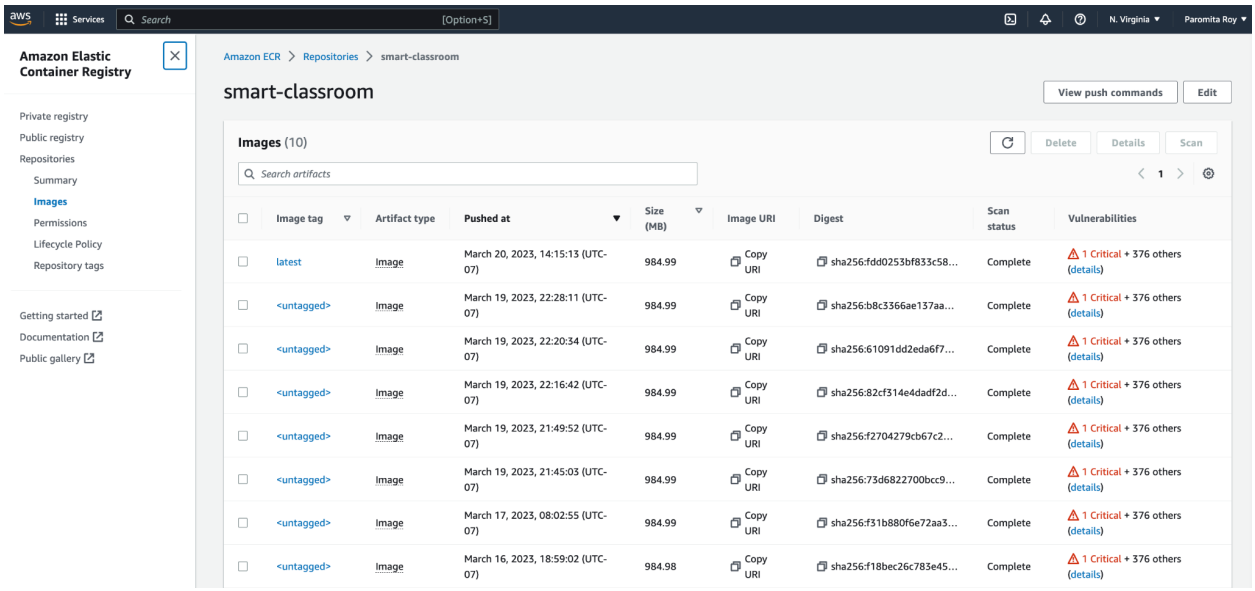

Figure 7. ECR Images

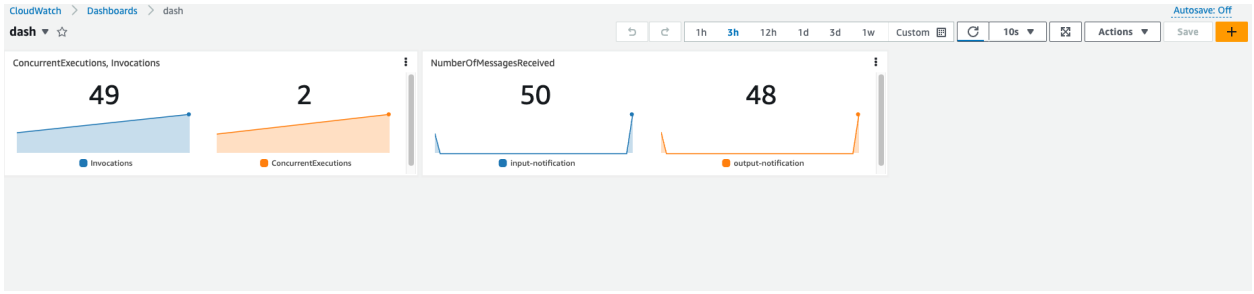

Figure 8. CloudWatch

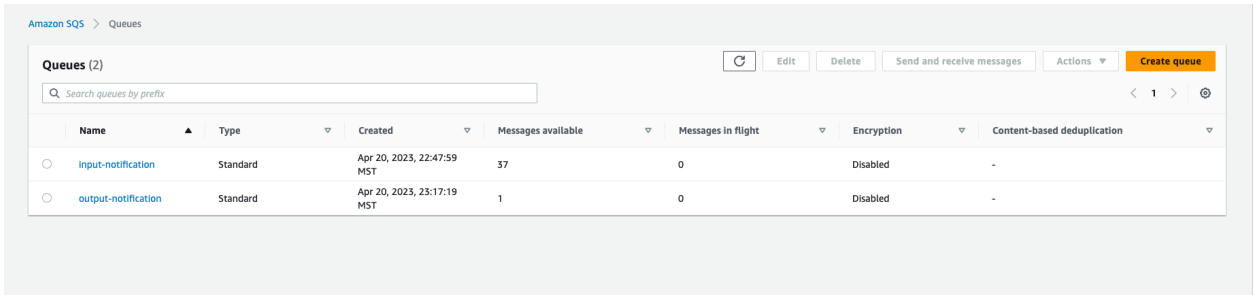

Figure 9. SQS Queues

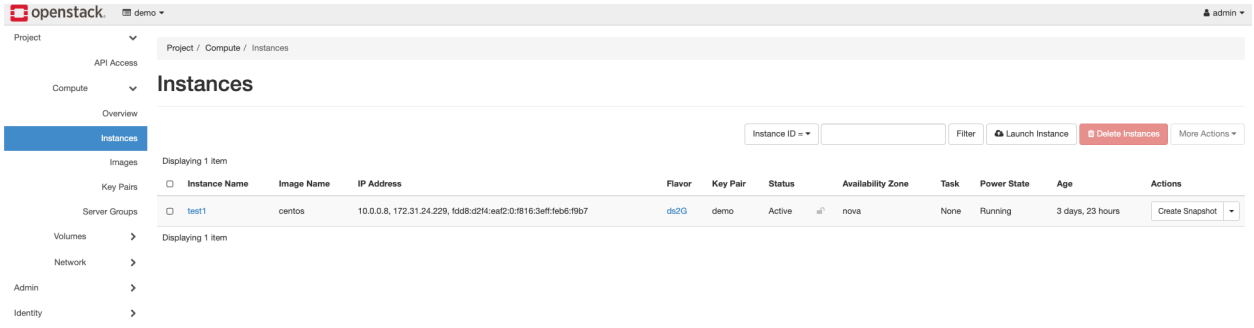

Figure 10. Instance on OpenStack

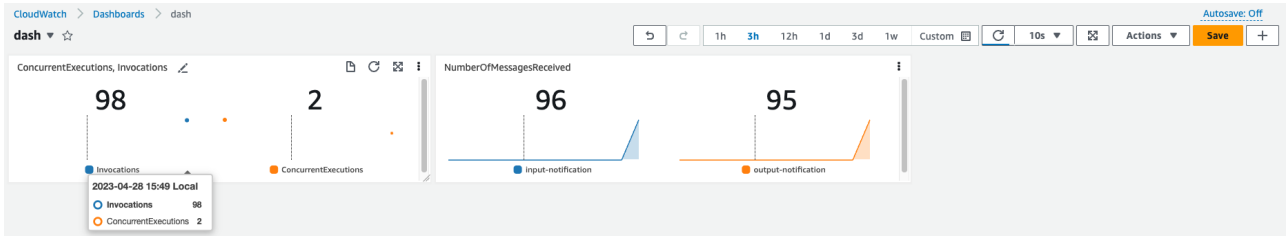

Figure 11. CloudWatch dashboard showing number of invocations

```
[centos@test1 ~]$ cd datafolder/
[centos@test1 datafolder]$ ls
test_0.txt    test_23.txt  test_4.txt   test_55.txt  test_70.txt  test_86.txt
test_1.txt    test_24.txt  test_40.txt  test_56.txt  test_71.txt  test_87.txt
test_10.txt   test_25.txt  test_41.txt  test_57.txt  test_72.txt  test_88.txt
test_100.txt  test_26.txt  test_42.txt  test_58.txt  test_73.txt  test_89.txt
test_11.txt   test_27.txt  test_43.txt  test_59.txt  test_74.txt  test_9.txt
test_12.txt   test_28.txt  test_44.txt  test_6.txt   test_75.txt  test_90.txt
test_13.txt   test_29.txt  test_45.txt  test_60.txt  test_76.txt  test_91.txt
test_14.txt   test_30.txt  test_46.txt  test_61.txt  test_77.txt  test_92.txt
test_15.txt   test_31.txt  test_47.txt  test_62.txt  test_78.txt  test_93.txt
test_16.txt   test_32.txt  test_48.txt  test_63.txt  test_79.txt  test_94.txt
test_17.txt   test_33.txt  test_49.txt  test_64.txt  test_8.txt   test_95.txt
test_18.txt   test_34.txt  test_5.txt   test_65.txt  test_80.txt  test_96.txt
test_19.txt   test_35.txt  test_50.txt  test_66.txt  test_81.txt  test_97.txt
test_2.txt    test_36.txt  test_51.txt  test_67.txt  test_82.txt  test_98.txt
test_20.txt   test_37.txt  test_52.txt  test_68.txt  test_83.txt  test_99.txt
test_21.txt   test_38.txt  test_53.txt  test_69.txt  test_84.txt
test_22.txt   test_39.txt  test_54.txt  test_7.txt   test_85.txt
[centos@test1 datafolder]$
```

Figure 12. Output files stored in the OpenStack instance

```
[centos@test1 ~]$ ls
__pycache__  awscliv2.zip  mylogger.py          server.log
aws          datafolder    openstack_handler.py
[centos@test1 ~]$
```

Figure 13. Files in the OpenStack instance

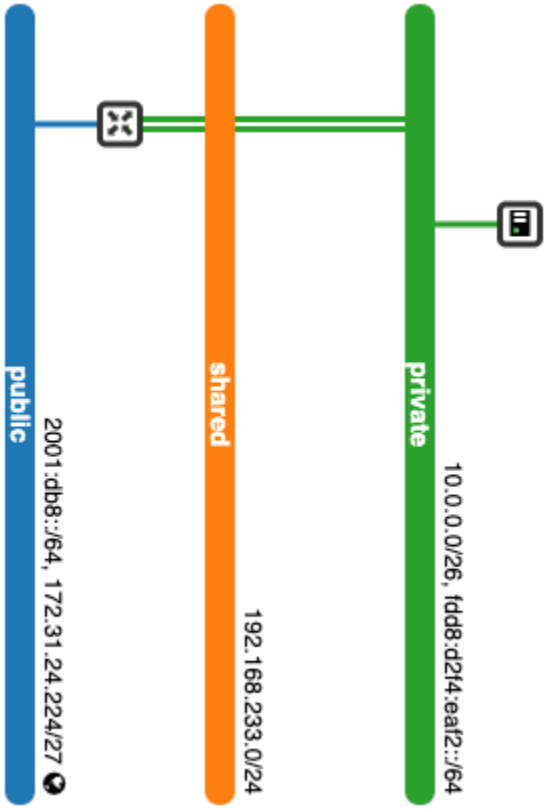

Figure 14: Network topology

7. References:

1. https://docs.aws.amazon.com/AmazonECR/latest/userguide/getting-started-cli.html
2. https://docs.aws.amazon.com/lambda/latest/dg/images-create.html
3. https://docs.openstack.org/networking-ovn/latest/contributor/testing.html
4. https://docs.openstack.org/devstack/latest/