CSE 546 — Project 3 Report

Karthik Aravapalli (1225611998)
Deepak Reddy Nayani (1225418259)
Nikhil Chandra Nirukonda (1223333995)

1. Problem statement

To develop an elastic application that exhibits rapid elasticity i.e scale-in and scale-out cost-effectively and on-demand in a hybrid cloud environment by leveraging AWS Services and OpenStack, a cloud OS. The usage of a hybrid cloud is to make use of the advantages provided by both public and private cloud environments. It is to be deployed as an IaaS in both public and private cloud environments where virtual servers and other resources are made available to users whilst providing a scalable and cost-effective cloud computing solution that scales as per changing demand.

2. Design and Implementation

2.1 Architecture

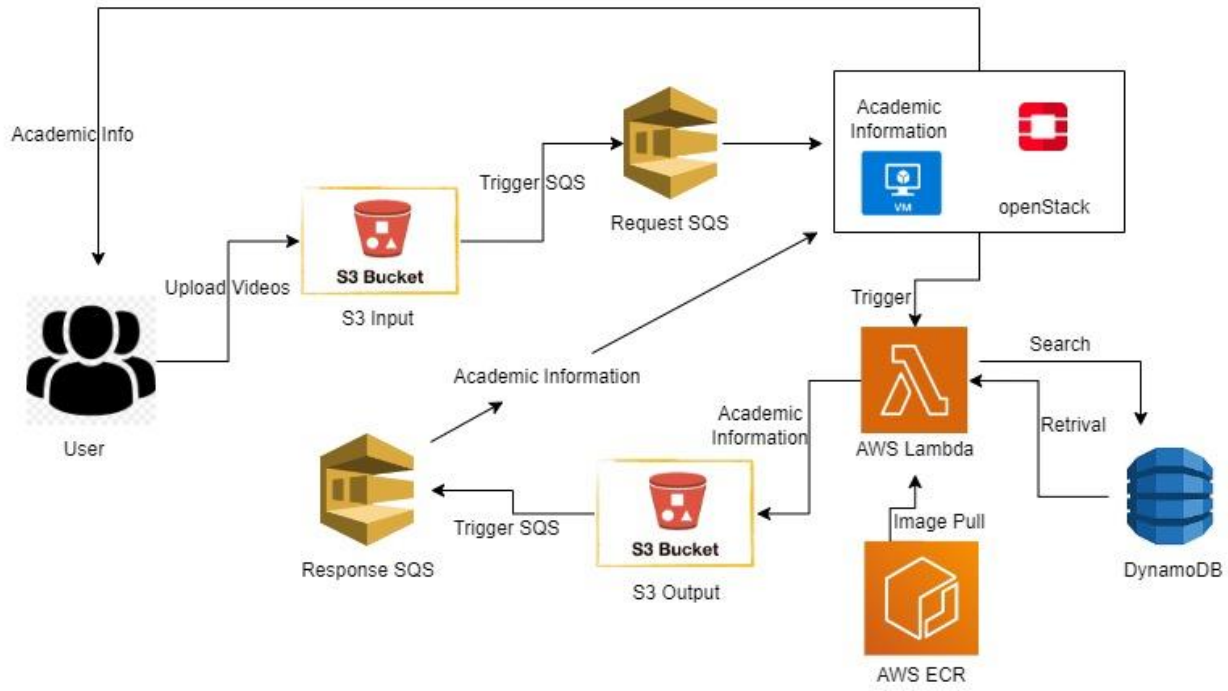

Fig. 2.1 Architecture

As shown in the figure 2.1 above, for this project we use 2 kinds of cloud environments, AWS and OpenStack which are public and private respectively. The components and services used in both of them are as follows:

Amazon Web Service:

- ➔ **Simple Storage Service (S3)** - Amazon S3 is a scalable object storage service provided by AWS, used to store the input videos. It is extremely reliable and leverages data redundancy to ensure 99.99% data reliability. It is used to store all the test case images.
- ➔ **SQS Queue Services -** It is a reliable queuing service provided by AWS. It facilitates scalable and reliable message storage and delivery. Its main application is in loosely coupled and distributed systems that require ease of use and scalability. In this program we use it to send trigger messages for OpenStack to process.
- ➔ **DynamoDB -** DynamoDB is a database management system that supports key-value storage. It can be utilized to store records of the students so that any student's academic information can be obtained using the student's name as the key. In this program we use it to store and retrieve the student data.
- ➔ **AWS Lambda -** It is a PaaS provided by AWS. It is serverless computing that offers a service for another function and also scaled in and out as per demand. For this project, it is used to run our main image classification program.
- ➔ **AWS ECR -** It is a container registry service offered by AWS to be able to deploy application images. In this case, it is used to deploy a basic image classification program using Python FFMPEG.

OpenStack:

OpenStack is an open source cloud software that is used as a private cloud for our project. It provides all the tools necessary to manage cloud resources. It is scalable, adaptable and modular and offers a high level of modification. It can be used seamlessly for this project due to its ability to integrate with the already existing infrastructure. It is a programmable infrastructure laying a common set of APIs on top of compute, networking and storage for VMs etc.
Some popular OpenStack projects:

Horizon - web based self service portal to interact with underlying services - to launch an instance, assign IP, configuring access controls.
Nova - Compute-manages life cycle of compute instances in an openstack environment responsibilities: spawning, scheduling, decommissioning of VMs on demand.
Neutron - Provides networking services. Enables network connectivity as a service for other services, such as Compute
Swift - Object storage - Stores and retrieves arbitrary unstructured data objects via a rest, http API. It is highly fault tolerant with its data replication and scale out architecture.
KeyStone - authentication/authorization service for other services - Provides a catalog of endpoints for all services

2.2 Implementation

The workflow of the project is very similar to that of the previous project 2, which uses Lambda to trigger 'On Upload of S3 Object' and run the face recognition program to output the names and then subsequently query the outputs with the DynamoDB table. The usage of S3, AWS Lambda, AWS ECR and DynamoDB remains the same even in this project, i.e the Lambda contains the docker image stored into an AWS ECR repository, which copies all the dependencies and packages to run the face recognition. The lambda pipeline begins by downloading the input videos from S3 into Lambda and processing the

input videos. The processing of the video involves creating the frame/thumbnail for each video, encoding the frame and using said encodings to run face recognition, finally query the output against a prefilled table on DynamoDB and write the output back into the S3 bucket.

For this project, the main change is made to the 'trigger' of the lambda function. We do not use an inbuilt trigger for Lambda such as 'On Upload of S3 Object', instead we leverage the use of SQS queues and S3 buckets to monitor the 'On Upload' part of the process to then manually trigger the Lambda. More formally, when an object/file is uploaded to S3, we push a trigger event message to the SQS input queue, we then use OpenStack to run a VM that monitors the SQS queue using Python boto3 module and runs a thread to poll for input queue messages. Upon polling a message, we parse the message event and invoke the Lambda function with the payload as the SQS message metadata as a JSON. Once the event is sent to Lambda, it receives the name of the input bucket and the process from hereon is similar to the previous process of running the face_recognition.py program and using the output to query to DynamoDB whose output is then subsequently written back into the output S3 bucket.
Furthermore, after receiving the output in the S3 bucket, we repeat a similar process of creating a message event into an SQS output queue. This output queue is also monitored by the OpenStack VM and the output is parsed and displayed in the OpenStack terminal.

2.3 Autoscaling

For this project, as similar to the previous, since there are no explicit changes being made to the Lambda function, there isn't a user specified auto scaling policy. The scaling is done by the AWS Lambda function. In case of concurrent requests, Lambda provides a separate execution instance, and as the number of parallel requests increases , it increases the number of execution instances till the account threshold is reached (which is usually a 1000). In some cases, we can set the concurrency limit to a specified cap, so that one particular function does not use up all the computing resources. But, in this case, we create and use only one function, so the lambda scaling automatically adjusts itself with respect to the number of requests coming in and outputs the files efficiently.

2.4 Member Tasks

Karthik Aravapalli-

I followed the given project instructions to create an Ubuntu operating system physical machine to start our project. Once I dual boot my Windows laptop with Ubuntu 24.04 then, I install the Git on my local machine to clone/ copy the given Devstack github repository using the given commands. After I cloned this repository we had all the required installation files except the local configuration file. After I created the devstack local configuration file I started installing the devstack. Once this installation process completed I received all the required features such as, Horizon-OpenStack dashboard, Nova-Compute service, Glance-Image service, Neutron-Network service, Keystone-Identity services on my physical machine. Once the openstack installation was completed, I used the Horizon Dashboard to open an OpenStack and I created a new image which is Ubuntu. After I created an Ubuntu image I added the SSH and all ICMP to default security group rules. Once the image creation and security group rules are added. Once the instance is launched I created a new SSH KeyPair in openstack to connect the virtual machine and I attach an SSH floating IP to connect the VM using a terminal to set up the machine to monitor the AWS S3 buckets to trigger the Lambda function to invoke the face recognition code. Once I connected the Ubuntu VM I installed all the required softwares such as python3, boto3 and AWS client to connect the AWS user using access key and security key. Finally, I tested all the components such as SQS queues, S3

buckets and lambda function autoscaling. Also, verified all the results of the input videos to output from the lambda.

Deepak Reddy Nayani-

Implemented code comprises two main components: The SQS message receivers for both requests and responses. The former constantly checks the sqs-request-queue for new messages while running on the main thread. It removes the received message from the request SQS queue to prevent repetition after extracting the S3 key from the message body and triggering the Lambda function asynchronously with the extracted payload. On the other hand, the Response SQS Message Receiver runs in a different thread and continuously scans the sqs-response-queue for response messages produced by S3 put events. Once a response message is received, the corresponding file is downloaded from the S3 output bucket, read, printed, and the received message is removed from the response queue.To handle errors, guarantee message existence, and manage messages properly, the code incorporates error handling.

Nikhil Chandra Nirukonda-

I created IAM roles, users, and access policies as part of the configuration process to assure secure access control. SQS queues for requests and responses were created, and access policies were configured to permit messages to be sent from S3. S3 event notifications were configured to send messages to the request queue whenever new video files were uploaded to the input bucket, and to the response queue whenever events occurred in the output bucket. AWS access credentials were configured with the proper permissions and libraries such as pip, boto3, and AWS CLI were installed on the VM to facilitate the execution of code. In addition, I actively participated in debugging and addressing difficulties throughout the OpenStack installation and code implementation, which included log analysis, configuration investigation, and troubleshooting to detect and fix any misconfigurations or mistakes found by the team.

3. Testing and Evaluation

The testing was done in an incremental fashion. First the Python face recognition was tested locally to ensure the output is as specified for the test images. Then, the program was deployed and tested in a cloud environment and made sure that no images were being lost or being left unprocessed and finally tested the correctness of the database querying.

- Tested the workload generator to make sure that all the items in the test_cases folder were being uploaded as per and also making sure that the input and output buckets are emptied before uploading any files to either of them.
- Locally tested the face_recogition module with sample images to make sure the classification is accurate and the results produced are as per mapping.
- Tested the lambda functions with various IAM roles to make sure that the reading and writing of files from and to the bucket and dynamoDB are seamless.
- Testing that all the directories are empty after each iteration, to make sure that pre-saved results from a previous iteration are not outputted again and that the entire image recognition process happens for all the relevant files only.
- Configuring OpenStack and making sure we are able to run the installation without any errors.
- Testing to make sure the S3 event messages in the input and output SQS queues are being parsed by the monitoring program and making sure no message is left unprocessed.
- Tested and verified the correctness of the output in OpenStack.

4. Code

trigger.py

This is the main script file that runs the threads to monitor the input and output SQS queues to invoke the Lambda function and parse & display the output respectively using Boto3.

The instructions on how to install and run the application are already specified in the readMe file

1. Follow the instructions in https://canvas.asu.edu/courses/141245/assignments/3780879 to install the OpenStack devstack.
2. Add triggers to the input SQS queue and input S3 input bucket. Follow the same thing for the output queue and output bucket.
3. We can either use the OpenStack dashboard (Horizon) or the OpenStack command-line interface (CLI) to connect to the OpenStack environment.
4. To create the Ubuntu VM, click on "Launch Instance" or use the "nova boot" command on the CLI.
5. Configure the VM, - name, flavor (CPU, RAM, and disk specifications), Ubuntu OS image, VNIC, and optional settings like security groups or key pairs. Now, launch instance.
6. Assign a floating public IP address to the created instance and use the key pair key to establish an SSH connection with the instance. Copy the script to the home directory and install boto3 and awscli. Execute the script to monitor the SQS messages.
7. Run workload.py on your local machine.

Individual Contribution

Karthik Aravapalli (ASU ID: 1225611998)

Design:

In the design phase all my team members collaborated with each other to understand the given architecture for the reference and shared the thoughts and identified requirements to complete the project. Once we discussed our proposed idea or thoughts we finalized to use the Amazon Simple Queue Service (SQS) for communication between the public cloud (AWS) and private cloud (OpenStack) for our project. Also, this is only the small change on our project 2 to complete our 3rd project. These SQS queues are implemented on S3 buckets to get communication on openStack to trigger the AWS Lambda function.

Implementation:

1. openStack Installation: I followed the given project instructions to create an Ubuntu operating system physical machine to start our project. Once I dual boot my Windows laptop with Ubuntu 24.04 then, I install the Git on my local machine to clone/ copy the given Devstack github repository using the given commands. After I cloned this repository we had all the required installation files except the local configuration file. After I created the devstack local configuration file I started installing the devstack. Once this installation process completed I received all the required features such as, Horizon-OpenStack dashboard, Nova-Compute service, Glance-Image service, Neutron-Network service, Keystone-Identity services on my physical machine.
2. Local File Configuration: I created a devstack configuration local file (local.conf), to provide the authentication settings to my devstack installation. In this file I provided the admin password, database password, rabbit password, service password, IPV4 address, floating range and host IP address values.
3. OpenStack setup: Once the openstack installation is completed, I used the Horizon Dashboard to open an OpenStack and I created a new image which is Ubuntu. After I created an Ubuntu image I added the SSH and all ICMP to default security group rules.
4. OpenStack VM setup: Once the image creation and security group rules are added. I launched a new instance which is ds1G Flavor Name, 1 GB of RAM, 1 VCPU VCPUs and 10 GB of Disk. Once the instance is launched I created a new SSH KeyPair in openstack to connect the virtual machine and I attach an SSH floating IP to connect the VM using a terminal to set up the machine to monitor the AWS S3 buckets to trigger the Lambda function to invoke the face recognition code. Once I connected the Ubuntu VM I installed all the required softwares such as python3, boto3 and AWS client to connect the AWS user using access key and security key.

Testing:

I verified all the outputs using the workload generator script. I verified the SQS queues to trigger the messages once the videos are uploaded to input S3 bucket and results are loaded from the Lambda to trigger response SQS queues. Also, I tested thoroughly on my python trigger script to trigger lambda based on the SQS queues and fetching academic information from output S3 using response queues. Also, I verified my OpenStack VM outputs on the terminal and both Input and Output S3 and SQS queues where 100 videos/ messages are loaded or triggered and OpenStack VM also getting all the 100 output Academic details CSV files from the Amazon AWS output S3 bucket.