CSE 546 — Project - 3 Report

Sai Varun Reddy, Mullangi

Siva Nagi Reddy, Munaganuru

Pavan Krishna Reddy, Madireddy

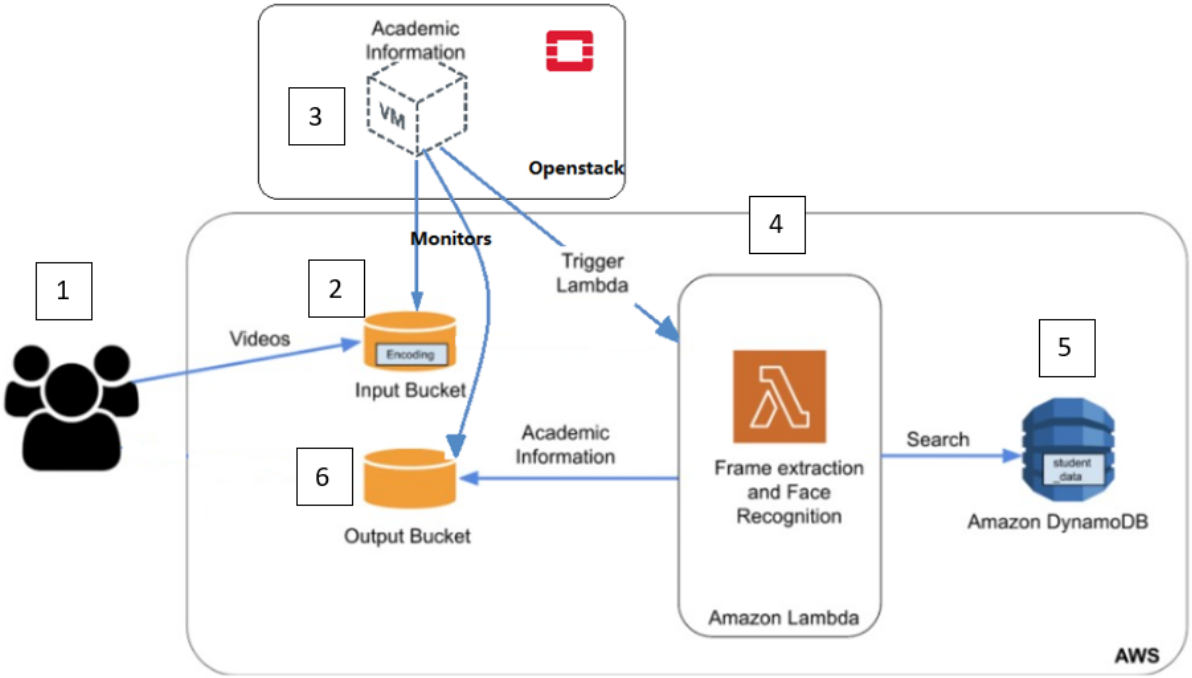
1. Problem statement

Our project aims to develop a hybrid cloud-based application that integrates advanced face-recognition techniques using the Python face-recognition library, processing collected classroom videos, extracting students' faces, and recognizing them using the face-recognition model. Once the students are identified, the assistant fetches their corresponding academic records from the DynamoDB database, providing educators with a comprehensive view of each student's performance. This integration of face recognition and database management allows for an efficient, accurate, and cost-effective solution to manage student recognition and academic record retrieval. Our project is implemented using both Amazon Web Services (AWS) and OpenStack, creating a seamless and powerful hybrid cloud environment.

The project is an extension of project 2 and the primary difference is the introduction of OpenStack as a private cloud. This new approach gives control over triggering the AWS Lambda function to the VM running in the private cloud (OpenStack). This allows the system to utilize public (AWS) and private (OpenStack) cloud resources, offering enhanced scalability and cost-effectiveness.

2. Design and implementation

2.1 Architecture



The above picture is the overall higher level with all major components of our implementation of this project. Each component's functionalities are as follows.

Component 1 - Workload generator:

In order to evaluate the performance and accuracy of the smart classroom assistant cloud service, we employ a parallel workload generator that allows us to execute multiple requests concurrently.

Component 2 - Amazon S3: Input Bucket

Classroom videos, or those used for testing, are uploaded into the designated input S3 bucket. Upon uploading a new video, an event notification is sent to the Simple Queue Service (SQS) containing the bucket name and key. This SQS message is then intercepted by the VM running in the OpenStack private cloud, which proceeds to initiate further processing.

Component 3 - OpenStack VM

In this component, we have set up a virtual machine (VM) in Azure, utilizing Ubuntu 22.04 as the base operating system. OpenStack DevStack was then installed on the VM to create a fully functional private cloud environment.

Within the OpenStack environment, we created another VM using CentOS Minimal, ensuring that all necessary network configurations were in place. This included setting up security groups, floating IPs, routers, and subnets.

A Python script was developed and deployed on the CentOS Minimal VM to monitor the Simple Queue Service (SQS) for new messages. The script is designed with multithreading capabilities to efficiently handle multiple incoming messages. When a new message is detected in SQS, the script triggers the AWS Lambda function with an event containing the S3 bucket name and key, initiating the face recognition and student information retrieval process. While waiting for the Lambda function to complete its processing, the script listens for a return notification containing the S3 output bucket details. Once the notification is received, the VM retrieves the corresponding CSV file from the output bucket and prints the student information to the terminal.

Component 4 - Lambda Function

The AWS Lambda function in this project is responsible for handling the main face recognition and student record retrieval process. Upon being triggered by the VM running in OpenStack, the Lambda function receives an event containing the S3 bucket name and key. It then retrieves the video file from the Component 2 S3 input bucket and downloads it to a temporary location. The face recognition process begins by extracting frames from the video using the FFmpeg tool, which allows for efficient video processing.

The Lambda function loads the known face encodings (stored in a file) and iterates through the extracted frames to identify student's faces. It compares the face encodings in the frames to the known encodings using the Python face-recognition library and identifies the best match for each detected face.

Once the students are recognized, the Lambda function queries the DynamoDB table using the student's name as the partition key. It retrieves the major and year information for each identified student and creates a DataFrame to store this data. This DataFrame is then converted to a CSV

file, which is uploaded to the Component 5 S3 output bucket. In addition, the Lambda function sends a notification to the VM running in OpenStack, providing the key of the newly created CSV file.By implementing the face recognition and record retrieval processes in the Lambda function and integrating it with the VM in OpenStack, this project creates a seamless and efficient workflow that successfully meets the goals of the smart classroom assistant.

Component 5 - Amzon Dynamo DB

Dynamo Db is loaded with the academic records of the students. And lambda function queries these records to find a student record.

Component 6 - Amazon S3: Output Bucket

This output bucket contains CSV files with the name, major, and year of the recognized students.

2.2 Autoscaling

Autoscaling is a crucial feature of AWS Lambda that enables our smart classroom assistant application to automatically scale in and out according to demand. When a video is uploaded to the input bucket, the Lambda function is triggered, and AWS automatically provisions the required compute resources to execute the code based on incoming requests. As the number of requests rises, AWS scales out the compute resources by launching additional instances of our function. Conversely, as the number of requests declines, AWS scales in the compute resources by terminating unnecessary instances. This capability allows the application to accommodate sudden traffic surges without manual intervention.

In our project, we have harnessed this autoscaling feature by creating a Docker image and utilizing it to establish an AWS Lambda function. This function automatically scales in and out depending on the incoming requests, ensuring that our application efficiently handles variable workloads.

2.3 Member Tasks

Munaganuru Siva Nagi Reddy - Component (3,5,6)

As part of my project responsibilities, I took charge of configuring the output S3 buckets, and also worked on installing OpenStack in an Azure VM and configuring the necessary network settings such as security groups, assigning a floating ip, and creating a new private network and a subnet. Then i have uploaded a Centos minimal to the glance and created a VM using Centos and set up the network reachability of the VM. I also set up the CentOS environment with the required tools and libraries, enabling seamless integration with Git and Python. This allowed us to efficiently monitor the Simple Queue Service (SQS) for new messages and trigger the AWS Lambda function as needed, ensuring the smooth operation of our smart classroom assistant application.

Also updated the dynamo DB with student academic records and have provided permissions for the lambda function so that it can fetch the student records and then these outputs are stored in the output bucket in the form of CSV.

Sai Varun Reddy Mullangi - Component (3, 4)

As part of my project responsibilities, I worked on Component 4 which involves frame extraction and recognition. To facilitate debugging, I set up logging to capture and analyze any errors encountered during the project setup on AWS.

I focused on the face recognition event handler, which triggers the processing of videos when there is an object create event, such as a video uploaded to S3. Using ffmpeg, I divided the video into frames and stored them in a temporary folder as Lambda has write access only to the tmp folder.

Next, I used the face recognition library to detect faces among the frames, and optimized the process by implementing a loop-breaking logic that stops the processing once the first match is found, thereby avoiding processing of the remaining frames. Finally, based on the name of the identified face, I queried the DynamoDB to retrieve the academic information of the person and wrote the information into a CSV for storage in the S3 bucket.

Further I have also worked on integrating an AWS Lambda function with an AWS SQS queue to achieve a fully-managed serverless architecture for my project. To achieve this, I wrote a Python code that uses the boto3 library to receive messages from an SQS queue and trigger a Lambda function, passing information from the message body as a payload. The Lambda function then retrieves a CSV file from S3, extracts a specific line of content from the file, and prints it along with the object key to the console.

Pavan Krishna Reddy Madireddy - Component (1, 3, 4)

As part of the project, I worked on two parts, the first one being the Dockerfile where I had to go through the given file and understanding all the details within the file. To complete the project I had to make some customizations like adding some extra files using COPY command. Then I built the docker image locally using the docker client which was saved in my macbook. Then I assigned a tag and uploaded it to Amazon Elastic Cloud Registry private repository which can be used in further steps of the projects to create a lambda function from the image.

I also worked on the workload generator part where I had run various tests and verified the sanity of the output produced by the lambda functions. The workload generator uploads videos into input S3 bucket which inturn triggers the lambda functions as configured and generated the output csv files into output buckets. Verified the output csv files data with the expected output.

I also worked on setting up the virtual machine in Openstack by uploading the centos image and installing python and all the dependencies required to run trigger.py which is the script which triggers lambda on receiving messages from SQS.

3. Testing and evaluation

From the test we conducted, we can see that the service is upscaling (Lambda inbuilt functionality) and handling the workload without any breakdowns. The whole process of the

reading the video, splitting into frames, using the face recognition library to detect faces, querying the Dynamodb for information and finally storing the results in output bucket takes less than a minute.

For the workload of 108 concurrent requests below is the graph for the number of Lambda invocations

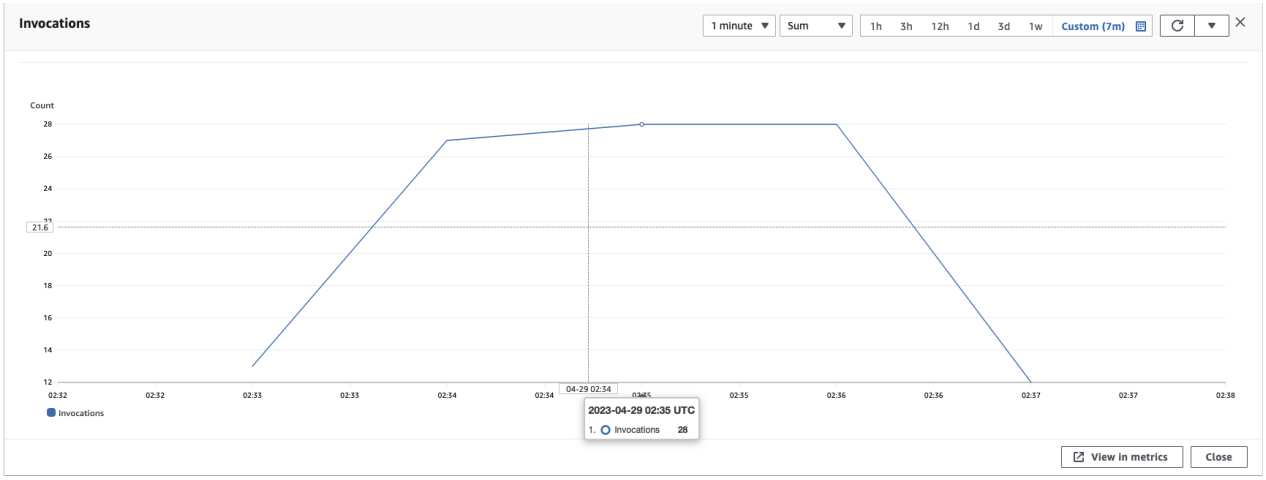


Fig: Lambda Invocations

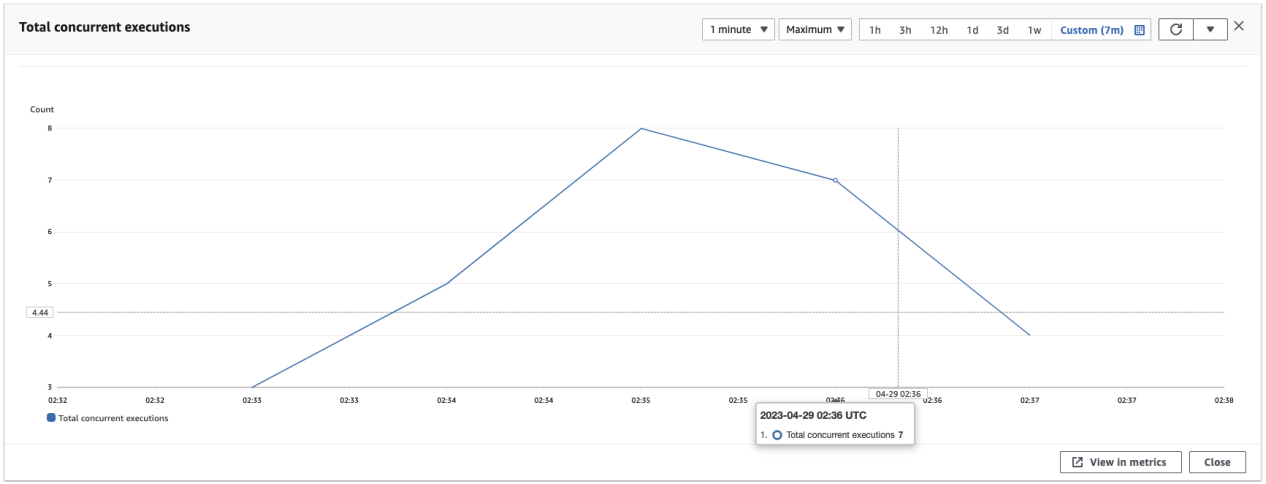
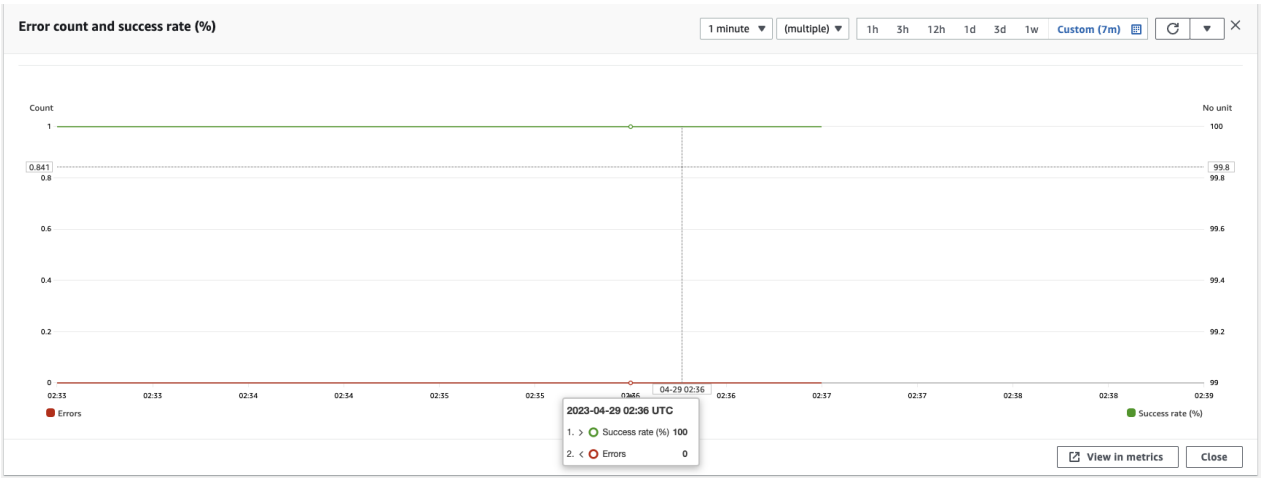


Fig: Lambda Concurrent Executions

Next we checked for the workload of 108, there are no errors or missing data datapoints, which is evident in the below graph



4. Code

handler.py

In our handler.py file, we have a function called face_recognition_handler, which gets triggered whenever an object is added to a specified S3 bucket. First, we establish two variables, input_bucket and output_bucket, representing the input and output S3 buckets, respectively. We then set up logging to gain insights into the code execution. Additionally, we create S3 and DynamoDB clients to interact with their respective services and define a function called open_encoding to read the 'encoding' file.

The face_recognition_handler function is invoked with event and context parameters. The event contains information about the S3 object, such as the bucket name and key. We download the video from the input bucket, store it in the /tmp directory, and extract the frames from the video, which are saved in the /tmp/frames directory. Using the face_recognition library, we identify the face in each frame and query the DynamoDB table with the face's name to obtain further information about the individual, such as their major and year. Finally, we write the data to a CSV file and upload it to the output bucket. We created the "tmp" folder to store the frames because Lambda does not have write access to any folder other than /tmp.

Dynamo.py

In this file, we import the boto3 and json modules. We then open a JSON file named "student_data.json" in read mode and parse the JSON data using the json.load() method. Next, we iterate through the parsed data and create a new dictionary in which we convert the id to a string and store it in a dictionary with the key "N", and store the other fields in a dictionary with the key "S". We append this new dictionary to a list called "new_data".

Afterward, we create a DynamoDB client using boto3 and specify the table name. We upload the data to the table using a for loop that iterates over each item in the "new_data" list and employs the put_item() method of the DynamoDB client to upload each item to the table. Finally, we print the response received from the put_item() method.

Docker File :

For the AWS Lambda function, we utilize a custom-built image created using the Docker File in the code. This image is based on Alpine Linux, a lightweight Linux distribution, and employs Python 3.8 as the runtime environment. The Dockerfile consists of three stages. In the first stage, GCC is installed, and pip is updated. The second stage involves building the function and its dependencies, as well as installing the Lambda Runtime Interface Client for Python. The third stage generates the final runtime image, copies the built dependencies, and installs libraries listed in requirements.txt. The entry point script /entry.sh is designed to execute the handler function handler.face_recognition_handler when the container starts.

Overall, this Dockerfile establishes a container environment suitable for running a Python function on AWS Lambda. Using Alpine Linux ensures a small and lightweight image, while the three-stage build process guarantees that only essential dependencies are included in the final image. The installation of FFmpeg enables the function to process video data, and the entry point script streamlines the procedure of running the function in a container.

Trigger.py:

This code uses the Python programming language to create an AWS Lambda function that listens for messages on an AWS SQS queue. When a message is received, the function extracts information from the message body, specifically the S3 bucket and object key. It then sends this information as a payload to another Lambda function named 'ccproj3' using the boto3 library. Once the payload is successfully delivered to 'ccproj3', the function deletes the original message from the SQS queue using the message's receipt handle. The function then retrieves a CSV file from S3 using the object key from the message, extracts a specific line of content from the file, and prints this content along with the object key to the console. This process is repeated until the function is stopped. The code also imports the threading and time libraries to handle concurrent execution and introduce a delay between SQS receive messages calls, respectively.