CSE 546 — Project 2 Report

Student Names: Raviteja Reddy Guntaka, Sai Manoja Gadde, Karthik Chadalavada

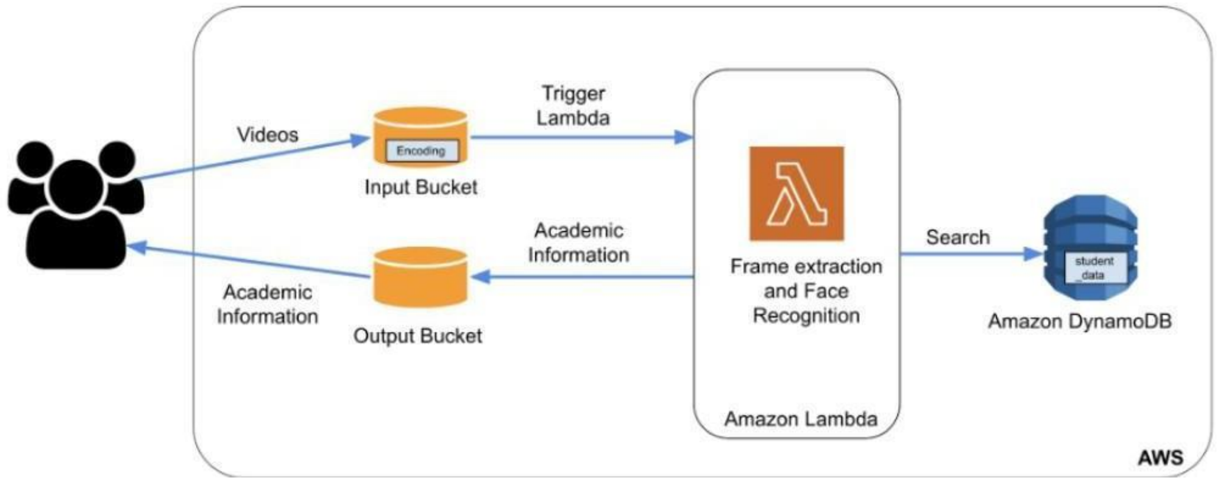
1. Problem statement

The primary goal of this project revolves around the development of an intelligent classroom assistant tailored for educators, leveraging the services offered by Amazon Web Services (AWS) Platform as a Service (PaaS). The application encompasses the following stages:

- Employ a workload generator to generate multiple requests aimed at uploading videos to the designated input bucket.
- Activate a lambda function once the input videos have been successfully uploaded to the input S3 bucket. This function is designed to extract the initial frame from the video and perform facial recognition on the captured image.
- Retrieve the name associated with the recognized face and access the corresponding details stored within the DynamoDB database.
- Compile the particulars of the input file into a CSV format and store it in the output bucket. This CSV file will encompass information such as the individual's name, year, and major.

2. Design and implementation

2.1. Architecture



1. **S3 InputBucket**: The S3 InputBucket serves as the repository for uploaded mp4 videos, from which the primary task is to classify the first individual captured in the frame. Each time a video is added to this bucket, an associated event is triggered, subsequently handled by the Lambda Function.
2. **S3 OutputBucket**: The S3 OutputBucket is designated for housing CSV files that encompass details like the person's name, their year, and major, specifically pertaining to the individual identified in the initial frame of the video.
3. **DynamoDB**: The DynamoDB database functions as the repository for student data, which includes essential information such as student ID, name, major, and year. Within this context, the Lambda function plays a crucial role in seeking out academic information related to the individual whose face has been identified. This is achieved by employing their name as the search criterion in the DynamoDB database. Notably, 'name' has been chosen as the partition key for DynamoDB, serving as the primary key responsible for data distribution across multiple nodes in the database. Consequently, the Lambda function conducts searches within DynamoDB using the 'name' field as the reference point.
4. **Lambda Function**: The Elastic Container Registry (ECR) is deployed to manage the processing of S3 input bucket events. The steps involved for each event are as follows:
    - Extract the video name from the event.
    - Utilize a Boto client to download the video from the input S3 bucket.
    - Convert the video into individual frames.
    - Employ the provided encodings to identify the name of the person in each frame.
    - Access additional attributes such as year and major from the DynamoDB database using the person's name as the identifier.
    - Upload the resulting CSV file to the output S3 bucket. This CSV file contains information such as the individual's name, year, and major. The filename of the CSV corresponds to the name of the video file.
5. **Amazon Elastic Container Registry (ECR)**: ECR, a managed Docker container registry service from AWS, is integral to AWS Lambda. It stores and manages Docker container images for Lambda functions. ECR enables the packaging of application code as a Docker container image, which AWS Lambda employs to execute the Lambda function.
6. **Workload generator**: The process involves sending 100 videos to the input S3 bucket, which subsequently triggers the Lambda function.

2.2 Auto Scaling in AWS Lambda

AWS Lambda is engineered to handle automatic scaling without the need for explicit auto-scaling algorithms or groups. When a request is made to a Lambda function, there's an initial cold start phase during which AWS prepares a container for the function's execution. If multiple requests come in, AWS automatically scales out by creating concurrent Lambdas to manage these requests. Once the request activity subsides, AWS terminates these Lambdas to scale back in.

2.3 Tasks of the members

1. Raviteja Reddy Guntaka: Implemented handler.py
    a. Parse the event payload to extract the video file name.
    b. Establish Boto clients for both S3 and dynamoDB.
    c. Download the MP4 file from the Input S3 Bucket and store it in the temporary directory.
    d. Load the given encoding data.
    e. Convert the video into frames and save them in the temporary directory. These frames are named sequentially according to their position in the video.
    f. Create encodings for each frame and compare them to the provided encodings.
    g. Retrieve the ‘name’, ‘major’ and ‘year’ attributes of the identified individual from DynamoDB.
    h. Generate a CSV file with the same name as the video, containing the person's name, year, and major.
    i. Upload the CSV file to the output bucket.
2. Sai Manoja Gadde: A Docker image was created and uploaded to the Elastic Container Registry (ECR), which was subsequently deployed as a Lambda function. AWS components, including the user role and policy, input and output buckets, were configured as part of this process.
    a. Input and Output S3 buckets were set up for storing uploaded MP4 videos and result CSV files.
    b. The Docker file was modified to copy the encoding file.
    c. A Docker image was created, incorporating 'handler.py', the encoding file, 'entry.sh', and 'requirements.txt'.
    d. The Docker image was uploaded to the Elastic Container Registry (ECR) for Lambda function deployment.
    e. A policy is created to grant required permission to execute and access aws resources like s3, dynamoDB and cloudwatch logs. A new role is created and the created policy is assigned to this role.
    f. A Lambda function was configured to trigger upon events from the S3 input bucket upload. This lambda function made use of the existing role created in earlier steps.

3. Karthik Chadalavada: DynamoDB was configured, and a script was developed to ingest data from the provided 'student_data.json' into DynamoDB. The end-to-end functionality was also validated to ensure the successful operation of the system.
    a. Amazon DynamoDB was configured to store student data with the partition key set as 'name.'
    b. A Python script was created to parse 'student_data.json' into objects and upload these objects to Amazon DynamoDB.
    c. Comprehensive end-to-end functionality testing was carried out, validating the entire process from the workload generator to the output S3 bucket.

3. Testing and evaluation

The application underwent an iterative testing process to ensure its functionality and identify and resolve any issues. Each newly developed component was subjected to individual testing, and subsequently, these components were integrated and tested collectively. To monitor progress during each instance, logging was incorporated within the 'handler.py' file.

The testing process included the following steps for 'handler.py':

Connection to Input and Output S3 Buckets:

- Verification of connection to Input and Output S3 buckets.
- Dummy files were sent to the input bucket to validate correct Access Control List configuration.
- Confirmation of writing output in the required format to the output bucket.
- Testing of the connection to input and output buckets from the workload generator.

Connection to DynamoDB:

- Configuration of the appropriate partition key to facilitate easy data access.
- Verification of DynamoDB accessibility from the Lambda function by configuring Access Control Lists.

Processing Image Based on Input Video:

- Use of a test video file to check the extraction of the first frame with the person's face.
- Confirmation of the storage of the extracted image in the temporary directory.
- Verification of the face_classification algorithm to ensure it correctly classifies the image.

Integrated Testing

- Comprehensive testing of the entire 'handler.py' script to ensure it operated as expected.

Lambda Function Integration:

- Utilizing the Docker image containing the code within the Lambda function.
- Making sure to use the correct role with assigned policy to avoid access issues.
- Configuring the trigger to the S3 input bucket.
- Validation that events were being created as expected, verified by monitoring CloudWatch logs.

End-to-End System Testing:

- Evaluation of the entire system end-to-end.
- Utilization of the workload generator to upload images to the input S3 bucket.
- Monitoring and checking the results in the output bucket to confirm the system's performance.

4. Code

Directory Structure and each file details:

── readme.md → lists group member and there tasks, the AWS credentials, S3 bucket names, DynamoDB table name, Lambda Function detail, Elastic Container Registry Detail.

── handler.py → Event processor, It handle the event generated when a file is uploaded to Input S3 bucket. It is responsible for downloading video, parsing it into frames, classifying the frame and uploading the result csv to output S3 bucket.

── .aws -> Folder containing credential file for AWS

  └── credentials -> AWS credential

── load_data_to_dynamoDB.py → Script to parse the student_data.json and upload items to DynamoDB.

── requirement.txt → list of python libraries used.

── Dockerfile → Configuration to build Docker Container image.

── mapping → As provided in the project github repo

── encoding → As provided in the project github repo

── entry.sh → As provided in the project github repo

Running the application

1. Workload Generator
    a. Make sure to copy the .aws folder from the submission.

b. Update the input and output S3 bucket names in the workload.py as given below
    i. input_bucket = "cloud-p2-input-bucket"
    ii. output_bucket = "cloud-p2-output-bucket"
c. Run the workload generator python file using below command:
    i. python workload.py

5.Individual Contributions

Raviteja Reddy Guntaka

Implementation:

Event Payload Exploration:

- Investigation of the event payload triggered upon S3 bucket file uploads and passed to the Lambda Function.
- Extraction of the file name from the event payload.

S3 Bucket Boto Client for Input:

- Creation of an S3 bucket boto client for the Input Bucket.
- Downloading the video file from the Input S3 Bucket using the extracted file name and saving it locally in the 'tmp' directory. Lambda functions have write access to this directory by default.

Video to Frames Conversion:

- Utilization of the ffmpeg tool to convert the video into frames.
- Saving the frames to the 'tmp' directory, with the frame names represented as integers to maintain their order.

Face Recognition Encoding:

- Exploration of the face_recognition Python library.
- Encoding of the previously generated frames sequentially in the order as they appear in the video.

Encoding Model Loading:

- Loading the provided encoding model, essentially a list of encodings.
- Initiating the matching process by comparing the frame's encoding to the provided encoding list.
- Upon a match, extract the person's name from the result payload.

Testing:

File Name Extraction:

- Verification of the correct extraction of the file name from the event, ensuring it was used accurately for saving the result CSV file.

Boto Clients Testing:

- Testing of the boto clients for Input and Output S3 buckets to ensure 'handler.py' could fetch the video file and perform uploads.
- Confirmation of DynamoDB boto3 client functionality and 'handler.py' access.

Downloaded Video File:

- Validation that the downloaded video file was stored in the 'tmp' directory as expected.

Frame Generation:

- Verification that the frames generated from the video were stored with names reflecting their order in the video, preserving the sequence.

Classification with Encoding:

- Verification of the classification process using the provided encoding.

Issues encountered and fixes:

- To resolve the problem of encodings not being copied into the Lambda function, a copy command was introduced in the Dockerfile. This command enabled the encodings file to be placed in the same location as 'handler.py'. Subsequently, a new Docker image was created, and it was uploaded to the Elastic Container Registry (ECR).
- Entry point is not identified when the docker image is built on windows. This occurs due to the end of file character difference in linux and windows. Fixed and made sure the entry is invoked correctly and that lambda is coming up from the docker image.
- The Lambda function encountered failures attributed to both timeout and memory limitations. The function was initially configured to wait for only 3 seconds before terminating execution. To rectify this, the timeout was extended to 5 mins, and the memory allocation was increased to 512MB to ensure smoother operation.

Sai Manoja Gadde

Implementation:

S3 Bucket Setup:

- Creation of Input and Output S3 buckets to manage uploaded MP4 videos from the workload generator and result CSV files.

Docker Image Creation:

- Learning and using Docker to generate a Docker image containing the application code and its dependencies via a custom Dockerfile.

Elastic Container Registry (ECR):

- Exploration of Amazon Elastic Container Registry and pushing the newly created Docker image to the 'face_recognition_container' repository within ECR.

AWS Lambda Configuration:

- Familiarization with AWS Lambda and its operation, along with the different methods for creating Lambda functions.
- Creation of a Lambda function utilizing the 'face_recognition_container' repository and configuring it to trigger on events from the input S3 bucket.

Access Control Lists and Roles:

- Configuration of Access Control Lists and roles to grant the Lambda function the necessary permissions for accessing the S3 bucket and DynamoDB.

CloudWatch Logging:

- Learning to use CloudWatch for monitoring and logging Lambda function activity to facilitate debugging efforts.

Testing:

CloudWatch Log Confirmation:

- Checking CloudWatch logs to ensure that the Lambda function's event was triggered when images were loaded into the S3 input bucket.

Memory and Timeout Issue Resolution:

- Verifying that changes made to Lambda function settings successfully resolved issues related to memory exhaustion and timeouts.

Encoding File Handling:

- Ensuring that the encoding file was accurately copied to the correct directory and that it was being read correctly by the Lambda function.

Issues encountered and fixes:

Encoding File Handling:

- Encodings not being copied into the Lambda function.
- Resolution: Added a copy command in the Dockerfile to place the encodings file in the same location as 'handler.py'. Built a new Docker image and uploaded it to ECR.

Lambda Function Issues:

- Failures in the Lambda function, specifically related to timeout and memory constraints.
- Resolution: Configured the Lambda function to have a timeout of 20 seconds and increased its memory allocation to 512MB.

Result Storage in Output S3 Bucket:

- Results not being stored in the output S3 bucket due to ACLs with insufficient permissions.

        - Resolution: Added another role to the ACL to grant access for all S3 bucket operations.

Karthik Chadalavada

I started my exploration of AWS by delving into the basics, understanding the core services, and their connections. I made use of the extensive resources available in the AWS documentation, which provides ample information and step-by-step guides on various AWS offerings.

Implementation:

- Established and configured a DynamoDB table named "Student-db" to store student data obtained from a JSON file. This JSON file contained various attributes, including ID, name, major, and year.
- To facilitate data retrieval, I designated the partition key as "name" since the encoder provides the name for the classified image.
- I explored multiple methods for ingesting data into DynamoDB, including options like the AWS Management Console, AWS SDKs, and AWS CLI.
- Additionally, I developed a Python script using the Boto3 library to interact with DynamoDB for data insertion. The script includes an "insert()" function designed to add data to the table using the "put_item" method. Furthermore, I implemented a "find()" function to perform searches and display the results.

Testing:

Access Control Lists (ACLs) Testing:

- Conducting multiple unit tests to ensure the accessibility of DynamoDB from the Lambda function by configuring Access Control Lists (ACLs).

Data Insertion Verification:

- Verification of whether the data was correctly inserted into the table, confirming the integrity of the data insertion process.

End-to-End Functionality Testing:

- Comprehensive end-to-end functionality testing, starting from the workload generator and extending to the output S3 bucket, to validate the entire system's performance and data flow.